International Journal of Software Engineering and Computer Systems (IJSECS)
ISSN: 2289-8522, Volume 4 Issue 1, pp. 29-47, February 2018

https://doi.org/10.15282/ijsecs.4.1.2018.3.0036

LANGUAGE-AGNOSTIC SOURCE CODE RETRIEVAL USING KEYWORD & IDENTIFIER LEXICAL PATTERN

Oscar Karnalim
Faculty of Information Technology
Maranatha Christian University
Prof. Drg. Surya Sumantri Street No.65, Bandung, West Java, 40164, Indonesia
oscar.karnalim@it.maranatha.edu

ABSTRACT

Despite the fact that source code retrieval is a promising mechanism to support software reuse, it suffers an emerging issue along with programming language development. Most of them rely on programming-language-dependent features to extract source code lexicons. Thus, each time a new programming language is developed, such retrieval system should be updated manually to handle that language. Such action may take a considerable amount of time, especially when parsing mechanism of such language is uncommon (e.g. Python parsing mechanism). To handle given issue, this paper proposes a source code retrieval approach which does not rely on programming-language-dependent features. Instead, it relies on Keyword & Identifier lexical pattern which is typically similar across various programming languages. Such pattern is adapted to four components namely tokenization, retrieval model, query expansion, and document enrichment. According to our evaluation, these components are effective to retrieve relevant source codes agnostically, even though the improvement for each component varies.

Keywords: source code retrieval, language-agnostic approach, lexical pattern, domain-specific ranking;

INTRODUCTION

Software reuse is a research area which is focused on optimizing development time by reusing existing software artifacts (Chavez, et al., 1998). This activity is commonly conducted when the programmers should do repetitive tasks that have been done by other programmers or themselves. However, due to a rapid growth of software artifacts (Bajracharya, et al., 2014), retrieving relevant artifact from repositories may take a considerable amount of time, especially when targeted repositories are unstructured and contain a vast amount of software artifacts. Hence, artifact retrieval should be developed as a supportive tool for software reuse. It is expected to aid programmer for finding their relevant software artifact from local or online repository in no time.

In general, software artifact retrieval typically focuses on two major domains which are binary and source code domain. On binary code domain, artifact retrieval commonly relies on external resources such as human-defined tag since the binary code itself is not human-readable. Two example systems which applies such retrieval mechanism are Maven Repository (http://mvnrepository.com/) and NuGet Gallery (https://www.nuget.org/). To the best of our knowledge, there is only one work which does not rely on external resources. Such work has been done by Karnalim and colleagues (Karnalim & Mandala, 2014; Karnalim, 2015; Karnalim, 2016b). It relies on

binary code from Java Archive (JAR) to extract related lexicons. On source code domain, on the contrary, artifact retrieval commonly relies on more varied features due to source code high-readability. Several sample features for such retrieval system are: structural information (https://code.google.com/), program input-output (Lemos, et al., 2007), user contribution (Vanderlei, et al., 2007), external resource (Chatterjee, et al., 2009), and modified retrieval algorithm (Puppin & Silvestri, 2006).

This paper proposes a language-agnostic approach to retrieve source codes. Language-agnostic means that our approach could incorporate various programming languages automatically since it does not rely directly on programming language structure. Instead, it incorporates Keyword & Identifier lexical pattern. Keyword & Identifier lexical pattern is selected as our main concern based on twofold. First, Keyword & Identifier is the most declarative token type for representing author intention on source code. Second, Keyword & Identifier lexical patterns are typically similar in most programming language.

RELATED WORKS

Source code retrieval is a task for retrieving, classifying, and extracting information from source code (Mishne & Rijke, 2004). There are numerous reasons why such activity is so popular nowadays. Sadowski et al (2015) provides a good description about these reasons. However, regardless of the reasons, since standard Information Retrieval (IR) approach frequently yields inaccurate result on source code domain (Kim, et al., 2010; Hummel, et al., 2008), this task becomes an emerging field for research, especially for enhancing its effectiveness.

In order to enhance retrieval effectiveness, most source code retrievals rely heavily on user knowledge about target code structure. Many large-scale source code retrieval systems such as Google Code Search (https://code.google.com/), Codase (http://www.codase.com/), Krugle (http://www.krugle.com/), and searchcode (https://searchcode.com/) encourage user to provide fixed structure location of the given query (e.g. class, field, or method body components). In such fashion, a large number of false positives could be removed since not all indexed terms are taken into account. It only considers terms found on specific structure location. It is important to note that such approach is not only found on large-scale source code retrieval systems but also on various research works about source code retrieval (Sindhgatta, 2016; Keivanloo, et al,, 2010). Exploiting user knowledge further, several works even expect user to provide program pattern as a query. Such pattern is expected to draw out various target source code characteristic. It is typically represented as UML-like function specification (Hummel, et al., 2008), high-level form of programming language (Paul & Prakash, 1994), specific query language (Begel, 2007), and raw code chunk (Mishne & Rijke, 2004). Nevertheless, even though structure-based approaches are more effective than the standard IR approach, it may be unfavorable for users who only know target source code in a black-box manner. They have no clue about target source code structure.

According to the fact that black-box behavior is representable through program input-output, several works incorporate program input-output as their query. Users can provide either test cases (Lemos, et al., 2007), input-output data types (Thummalapenta & Xie, 2007; Reiss, 2009), or input-output query model (Stolee, et al., 2016) to refine their retrieval result. On the one hand, test cases are incorporated for retrieving only source codes which output is similar to given test-case output while its respective input is given. Input-output data types, on the other hand, are incorporated for retrieving only

source codes which input and output match specific object types. Even though both kinds of approaches may help user to retrieve target source code in black-box manner, it cannot help users who only know about target source code functionality in general. For example, when users only know ANTLR as a source code parser library, they cannot incorporate input-output pattern as a query for retrieving ANTLR.

Retrieving relevant source codes for users who only know target source code functionality in general is not a trivial task since it forces the retrieval system to work well even with a simple and limited query. In general, there are several approaches to achieve this goal which are: conducting pre-processing refinement, conducting post-processing refinement, incorporating user contribution, enriching source code with API documentation, and modifying retrieval mechanism.

Conducting pre-processing refinement refers to enriching either user query or indexed source codes to provide more descriptive information. On the one hand, when enriching user query, most works are focused on applying query expansion technique, that is adapted from standard IR approach. One example of such work is Lu et al's work. Lu et al (2015) enrich the query with its synonyms, that are extracted from WordNet. On the other hand, when enriching indexed source codes, most works are focused on embedding more related information. One example of such work is Vinayakarao et al's work. Vinayakarao et al (2017) enrich the indexed source codes by providing additional annotations related to syntactic representation. Nevertheless, we would argue that pre-processing refinement should be assisted with other techniques to provide more accurate result, as it is known that such refinement do not affect the retrieval process directly.

Conducting post-processing refinement refers to refining initial retrieved results with additional processes in order to yield more effective results. In such approach, initial retrieved results are commonly extracted from large-scale internet search engine due to its accessibility. It could be accessed with ease through the internet. After retrieved, initial results are then refined to accommodate specific goal through additional processes, such as re-ranking mechanism and information embedding. On the one hand, re-ranking mechanism has been applied on two works which are Kim et al's and Stylos & Myers' work. Kim et al (2010) overrides the search results of Koders, a source code search engine that have been discontinued on 2012, to retrieve source code example by re-ranking the search result. Stylos & Myers (2006) also shares similar goal with Kim et al but they override Google search result instead of Koders and utilize API documentation in their ranking mechanism. On the other hand, information embedding has been applied on one work which is Hoffmann et al (2007). They refine Google search result by embedding multiple resources such as Java Archive (JAR), code example, and code-specific snippet. Nevertheless, refining retrieved result of existing source code retrieval system may yield two additional drawbacks: 1) It takes longer processing time due to its two-fold processing mechanisms; and 2) Applied refinements are limited since initial retrieval mechanism, which is commonly defined on publicly-available retrieval system, cannot be modified directly.

Incorporating user contribution refers to embedding more information from users to enhance retrieval result effectiveness. This approach typically relies on either user behavior logs (Ye & Fischer, 2002) or collaborative information (Vanderlei, et al., 2007; Gysin & Kuhn, 2010) as its supplementary information. On the one hand, user behavior log is applied by Ye & Fischer for encouraging source code reuse (Ye & Fischer, 2002). Their retrieval mechanism is personalized under user behavior logs so that users could easily retrieve their target source code for reuse. On the other hand,

collaborative information is applied by Vanderlei et al (2007) and Gysin & Kuhn (2010). Vanderlei et al incorporates collaborative manual tagging on indexed source code whereas Gysin & Kuhn incorporates user votes and developer reputation to refine their retrieval result. Nevertheless, user contribution only affects significantly when the system is frequently used and/or it involves numerous users. Thus, its impact may be insignificant for new users and unavailing on early development stages with limited users. Even though its impact may be improved as the number of users and their interactions are larger, user contribution still relies greatly on users which might be biased due to human error.

Enriching source code with API documentation refers to utilizing API documentation to enhance retrieval effectiveness. It is inspired by the fact that source code has limited vocabulary terms and enriching the documents with external resource is proved to be effective on IR domain. In general, there are three works which fall into this category namely Chatterjee et al's, Grechanik et al's, and Lv et al's work. First, Chatterjee et al (2009) incorporates API documentation to enrich Java source code by embedding specific API documentation each time that API is used on source code. Second, Grechanik et al (2010) applies similar approach as in Chatterjee et al's work but differs in how they utilize API documentation. API documentation is used to convert query into API calls before retrieval phase and retrieval phase is conducted based on given API calls. Last, Lv et al (2015) uses API documentation to expand user query. The potential APIs will be defined based on API understanding component. Despite its promising result, enriching source code with API documentation relies greatly on the completeness and quality of the given API documentation. Thus, its impact may vary per source code dataset since not all dataset are featured with high-quality API documentation.

Modifying retrieval mechanism refers to designing domain-specific retrieval mechanism for source codes. We believe that such approach is the most promising one for enhancing retrieval effectiveness, as it is known that retrieval mechanism is the heart of information retrieval. Without proper retrieval mechanism, even the best retrieval system may yield faulty results. The simplest implementation of such approach is to consider source code as natural language text without relying on source-code-specific features (Girardi & Ibrahim, 1995). However, the result is not promising since natural language in source code domain is quite different with the real natural language. Hence, several works focus on the probabilistic approach instead. To the best of our knowledge, there are three works which use such probabilistic approach. First, Puppin & Silvestri (2006) modifies Google PageRank algorithm (Page, et al., 1998) by treating class usage as a replacement of link. Second, Spars-J (Inoue, et al., 2005) applies similar approach as in Puppin & Silvestri's work but with more fine-grained entities and relations. Last, Sourcerer (Bajracharya, et al., 2014) combines three ranking mechanisms which are graph-based, text-based, and structure-based ranking to yield the most appropriate results. Nevertheless, since these probabilistic approaches, at some extent, rely on source code structure as their retrieval features, they still require considerable efforts when incorporating new programming language(s).

In this paper, a language-agnostic source code retrieval is proposed. Language-agnostic means that our proposed source code retrieval can incorporate new programming language(s) with no effort since it ignores language-centric features such as programming language structure. To our knowledge, there are no related works that claim their works as language-agnostic. Most of them are only focused on a particular programming language. Even though some of them state that their work can be applied

to another programming language, they require a considerable effort to do so. Our work relies on Keyword & Identifier lexical pattern which rules are similar in most programming languages. Such pattern is adapted to four components namely tokenization, retrieval model, query expansion, and document enrichment.

METHODOLOGY

Similar with standard IR approach, our proposed source code retrieval system consists of three modules which are source code tokenization, retrieval. and indexing. Source code tokenization is aimed to convert query or source code into lexicons; source code retrieval is aimed to retrieve relevant source code according to given query; and source code indexing is aimed to index the source code dataset. Our contributing components, which are tokenization, retrieval model, query expansion, and document enrichment, are applied in either source code tokenization or retrieval. Tokenization is applied on source code tokenization while the other three are applied on source code retrieval.

Source Code Tokenization

The detail of our proposed source code tokenization can be seen in Figure 1. This module converts query or source code into lexicons with 4 steps namely lexicon recognition, lexicon categorization, transition-based tokenization, and standard text pre-processing (lowercasing, stopping, and stemming).

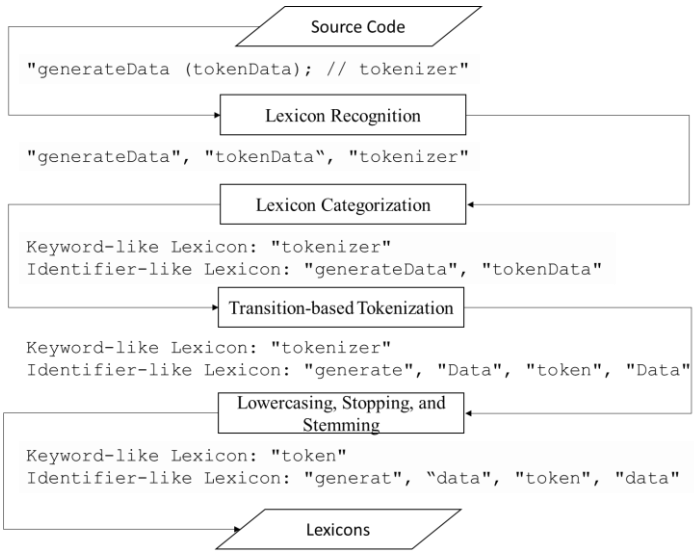

Figure 1. The Flowchart of Source Code Tokenization

Lexicon recognition is responsible to extract all possible lexicons from source code. However, since our proposed approach is aimed to be designed as language-agnostic as possible, programming language lexer and parser are not used. Instead, we utilize Keyword & Identifier lexical pattern that is generalized from naming rules of popular programming languages (Cass, 2016). Regular expression of such pattern can be seen in Eq.(1). Basically, our pattern only accepts lexicon which starts with alphabet or underscore, that is followed by zero or more alphanumeric, underscore, hyphen, and dot mark. Hyphen is included in this pattern since such character is typically used in scripting language such as PHP. Moreover, dot mark is included to differentiate

keyword and identifier. Lexicons with dot mark as its member will be considered as an identifier since most keywords only consist of alphanumeric.

$$[A\text{-}Za\text{-}z\_][A\text{-}Za\text{-}z0\text{-}9\_\backslash\backslash\text{-}\backslash\backslash.]^* \quad (1)$$

Even though comment pattern is quite similar in most programming languages, it is ignored in our work due to following reasons: 1) Comment delimiter on a particular programming language may represent another token type on other programming language. For instance, the number sign ('#'), which acts as an initial mark of comment in Python, acts as an initial mark for macro in C++; and 2) Comment lexicons are still implicitly extracted with Eq.(1), even though they are not exclusively recognized as comment lexicons.

Lexicon categorization is responsible to classify recognized lexicons based on Keyword & Identifier lexical pattern. Each lexicon is classified either as a keyword-like or identifier-like lexicon. A lexicon will be categorized as a keyword-like lexicon if such lexicon only consists either uppercase or lowercase characters and involves zero or more underscore(s). Otherwise, it will be categorized as identifier-like lexicon. Even though such heuristic may misclassify several identifiers as keywords or vice versa, we would argue that this heuristic is the best approach so far for recognizing Keyword & Identifier in language-agnostic manner.

Transition-based tokenization is responsible to parse all identifier-like lexicons based on their character transition, as it is known that most identifier lexicons are built from several sub-lexicons that are separated based on character transition. The implementation of this phase is adapted from Karnalim & Mandala's work (Karnalim & Mandala, 2014).

Standard text processing is responsible to minimalize the number of lexicons, handle character-case variation, and handle affixes variation. Firstly, minimalizing the number of lexicons is performed by stopping. It relies on Weka default stop words (http://weka.sourceforge.net/doc.stable/weka/core/Stopwords.html). Secondly, handling character-case variation is performed by lowercasing. It converts all characters to its lowercase form. Finally, handling affixes variation is performed by stemming. It is implemented based on English stemmer (Porter, 2001).

In order to get a broader view about our source code tokenization, a sample process of tokenization is also embedded on Figure 1. First of all, a particular Java source code chunk, which is *generateData (tokenData); // tokenizer*, is fed into lexicon recognition and yields three lexicons: *generateData*, *tokenData*, and *tokenizer*. Based on their respective lexical characteristic, the first two lexicons are categorized as identifier-like lexicon whereas the latter one is categorized as keyword-like lexicon. Afterwards, transition-based tokenization splits *generateData* and *tokenData* into their respective sub-lexicons and all generated lexicons from both categories is lowercased, stopped, and stemmed. As a result, these steps yield 5 lexicons which are *token*, *genera*t, *data*, *token*, and *data*.

Source Code Retrieval

Source code retrieval consists of four major sub-modules namely source code tokenization, query expansion, retrieval model, and document enrichment. Retrieval flowchart involving these sub-modules can be seen on Figure 2. First of all, source code tokenization converts query into token stream and feeds them to query expansion. After

the query has been expanded, all source codes which match to given query will be returned as its retrieval results. In this phase, each time a source code is considered as irrelevant document toward given query, the source code contents will be enriched by the most similar source code and its retrieval score will be recalculated.

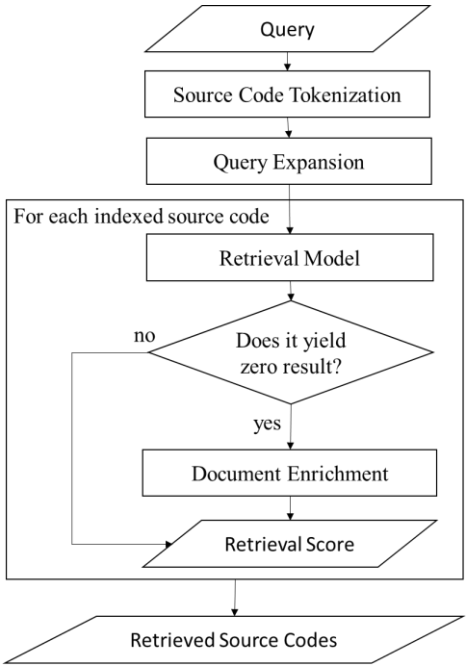

Figure 2. The Flowchart of Source Code Retrieval

Query Expansion

Query expansion is an IR technique to handle term mismatch problem by expanding query with additional terms that are mostly related to initial query terms (Carpineto & Romano, 2012). In our work, among various query expansion techniques, we apply query-specific local technique, which expands the query based on terms found on top-K retrieved source codes.

First of all, query expansion candidates are selected from top-K retrieved documents. To limit the number of candidates, a lexicon is only considered as a candidate *iff* it consists of 1 to 4 words and its category is similar to the category of query term, either keyword-like or identifier-like lexicon. After all candidates are selected, these candidates will be tokenized and merged as shortlisted query term candidates.

Shortlisted candidates are then sorted in descending order based on their importance score, that is defined based on Eq.(2). In general, Eq.(2) considers two aspects to determine candidate importance which are weighted term frequency (*weighted_tf(t)*) and one-to-many association ($\sum_{q \in Q} MI(t,q)$). A promising candidate should be frequently occurred on Top-K retrieved documents and is strongly-associated with initial query terms. Both aspects are connected in Eq.(2) through multiplication symbol where each of them has been additive-smoothed.

$$score(t) = (weighted\_tf(t) + 1) * (\sum_{q \in Q} MI(t,q) + 1) \quad (2)$$

Weighted term frequency of term t is calculated using Eq.(3), which is generally resulted from counting the occurrence of given term from top-K retrieved documents (*tf(t,di)*). For each retrieved document, the term frequency would be multiplied with their respective document retrieval score (*sd(di)*) so that the impact of term frequency on high-ranked documents are strengthened.

$$\text{weighted\_tf}(t) = \sum_{i=1}^{K}(\text{sd}(di) * \text{tf}(t,di)) \quad (3)$$

One-to-many association is calculated based on the term co-occurrence between candidate term and query terms. Term co-occurrence is preferred to natural language ontology since vocabulary used in programming may be different with standard natural language (Karnalim, 2016c) (e.g. mouse in programming context may be not related to a kind of rodent mammal). Term co-occurrence for each query term *q* and candidate term *t* is measured based on position-based mutual information defined in Eq.(4). For each document in top-K retrieved documents, term pairs are extracted and their respective inverse delta position is calculated and summed. *pos(q,i,p)* and *pos(t,i,p)* represent term position on document *i* and pair *p*. The first one is related to query term position whereas the latter one is related to candidate term position. Eq.(4) will yield higher score when both candidate and query term are frequently located in adjacent position.

$$\text{MI}(q,t) = \sum_{i=1}^{K}\sum_{p \in P}(1/ |\text{pos}(q,i,p)-\text{pos}(t,i,p)|) \quad (4)$$

However, since limited vocabulary terms in source code corpus may yield biased co-occurrence, external resource is incorporated as a replacement of source code corpus to measure mutual information. Our work utilizes noun phrases from 27.229 software-specific html pages, which are scraped from 32.000 links at the beginning of GitHub Java Corpus project list (Allamanis & Sutton, 2013) where remained 4.771 links are not accessible.

In order to assure each term position is only included in one position pair, algorithm for selecting pairwise method candidates from Karnalim's work (Karnalim, 2016a) is adapted and redirected to handle term position. It takes query and candidate term position list as its input and return selected position pairs as its result. First, all query term positions are paired with possible candidate term positions. After paired, all pairs will be sorted in ascending order based on its distance where each pair which member(s) is occurred in more-adjacent pair is removed. In such fashion, remained position pairs resulted from this algorithm are the most adjacent pairs and each position is only included once in selected pairs.

After all query expansion candidates are sorted in descending order based on its importance, top-N candidates will be selected and merged with initial query lexicons. In our work, N is defined manually by user so that it may be modified according to user necessity.

Retrieval Model

Our proposed retrieval model is extended from Okapi BM25 (Robertson, et al., 1998), a popular retrieval model from natural language IR, by incorporating Keyword & Identifier lexical pattern. Scoring function for such retrieval model can be seen in Eq.(5). *score(Q,d)* refers to our scoring function which define the relevancy between

query lexicons *Q* and an indexed source code *d*. It computes score locally per lexical pattern category and sums up both scores into its final score. *BMK* and *BMI* stand for BM25 for keyword-like and identifier-like category respectively where their equation is quite similar as standard BM25 except that they only consider lexicons with specific lexical pattern (either keyword-like or identifier-like).

$$score(Q,d) = \sum_{q \in Q}(a * BMK(q,d) + b * BMI(q,d)) \quad (5)$$

In Eq.(5), *a* and *b* are weighting constants which represent the impact of each lexical pattern category for generating final score. *a* represents keyword-like lexicon weighting whereas *b* represents the identifier ones. Such weighting mechanism is implemented based on our informal observation about user query where users typically expect the retrieved results to have given query in similar lexical pattern category. For instance, if users provide an identifier as a query, they typically expect retrieved results to have such query as an identifier, not keyword. Thus, by incorporating weighting constants, our retrieval model can enhance the impact of desired lexical pattern category. However, to provide a simple interaction, *a* and *b* are set automatically according to given query lexicon category. If given query is detected as a keyword-like lexicon, then *a* will be assigned higher than *b*. Otherwise, *b* will be assigned higher than *a*. Constant values that will be assigned as *a* and *b* are defined statically beforehand and named as *x* and *y*. *x* represents weighting constant for preferred category whereas *y* represents weighting constant for non-preferred category.

According to the fact that programming language keywords are less discriminative than other lexicons, our scoring function minimalizes its impact by computing *BMI* and *BMK* locally per file extension. In such manner, programming language keywords will generate low score since they occur frequently in a particular file extension. It is important to note that such reduction will not be achieved if *BMI* and *BMK* are computed globally. As we know, not all programming languages share similar keywords. Some of them even incorporate unique terms as their keywords.

Since source codes are frequently used as either standalone file or project, our retrieval model will generate scoring function for each representation. For retrieving standalone file (i.e. single-file source code), we will use scoring function defined in Eq.(5). For retrieving project, a post-processing mechanism is incorporated toward retrieved results. All retrieved single file source code which are from similar project will be merged and considered as one document. The retrieval score of such project will be assigned as the sum of the score of these codes.

Document Enrichment

Document enrichment is conducted based on an assumption that lexicons on similar documents (i.e. source codes in our case) should be related to each other. This mechanism is implemented by replacing initial retrieval score with the weighted retrieval score of its most similar code. Weighted retrieval score is calculated with Eq.(6) where *d2* stands for the replacement source code of *d1*, *score(d2)* stands for *d2* retrieval score toward given query, and *sim(d1,d2)* stands for the similarity degree between *d1* and *d2*. In such equation, the retrieval score of replacing source code is multiplied with its similarity degree so that enriched source code (*d1*) is assured to yield lower retrieval score than its enricher (*d2*), resulting lower rank for *d1* when a query is naturally relevant to *d2*.

$$\text{weighted\_score}(d1,d2) = \text{score}(d2)*\text{sim}(d1,d2) \quad (6)$$

Similarity degree is calculated using standard token-based approach for detecting source code plagiarism (Prechelt, et al., 2002). It converts source code into lexicons and calculate its similarity with Rabin-Karp Greedy String Tiling Algorithm (RKGST) (Wise, 1993). However, this approach is extended in our work by incorporating language-agnostic tokenization and lexicon weight. On the one hand, language-agnostic tokenization is incorporated to extract all lexicons without relying on a particular lexer. It is implemented based on our proposed source code tokenization. On the other hand, lexicon weight is incorporated to enhance the impact of all identifiers heuristically. Such mechanism is incorporated since our similarity measurement prefers identifier to keyword subsequence for determining source code similarity. As we know, two source codes with similar keyword subsequence may not share similar intention. For example, let us assume that there are two source codes: a source code for sorting numbers and a source code for accessing a matrix. Even though both source codes share similar nested-traversal keyword subsequence (i.e. a sub-sequence of two-level-nested traversal), their intentions are extremely different. Lexicon weight is applied by assigning all identifier-like lexicons with 1 and all keyword-like lexicons with their respective local inverse document frequency that is calculated locally per file extension. In such manner, programming language keywords, which have high frequency in a particular file extension, will be assigned with lower score when compared to identifiers.

The detail of our weighted RKGST can be seen in Eq.(7). It is based on the average similarity of RKGST. *A* and *B* are compared lexicon sequences; *Tiles* are RKGST output which represent similar lexicons from both sequences; and *weight(T)*, *weight(A)*, & *weight(B)* are the total weight of lexicons in *T*, *A*, and *B* respectively. In our similarity measurement, two lexicons are only considered as similar to each other *iff* their lexicon and respective weight are identical. Such mechanism is implemented to differentiate keywords between programming languages. As mentioned before, each keyword would be assigned with its respective local IDF. Thus, even though a keyword is used on two or more programming languages, such keyword would not be considered as similar to each other since each programming language would generate different IDF for given keyword. In other words, keyword subsequence is only considered when two source codes are written in similar programming language. Furthermore, this mechanism is also used to differentiate keywords which might be considered as identifier on other programming languages. By assigning unique weight to keyword per programming language, such keyword would not be considered similar with identifier on other programming languages.

$$\text{sim}(A,B) = (2\sum_{T \in \text{Tiles}} \text{weight}(T)) / (\text{weight}(A) + \text{weight}(B)) \quad (7)$$

Source Code Indexing

In general, our indexing scheme stores four major source code components: source code lexicon, source code file extension, replacement list, and project member list. All components are required for retrieving source code. First, source code lexicon is used for calculating retrieval score of each indexed source codes. Second, source code file extension is used for determining IDF of keyword-like lexicon. Third, replacement list

is used for conducting document enrichment. It stores the most similar source code for each indexed source code, including their similarity degree. In such manner, our document enrichment is not required to recalculate similarity degree at retrieval phase. It is only required to access stored similarity degree on replacement list. Last, project member list is used to accommodate project retrieval. It enlists all available projects with its members so that post-processing on our retrieval phase can be conducted just by accessing this list. Projects are recognized based on two regexes namely directory and key file regex. On the one hand, directory regex recognizes project based on directory name pattern. This mechanism is aimed to accommodate manually-created projects such as Java-based project that is developed with a standard text editor. On the other hand, key file regex recognizes project based on the name pattern of key file, an artificial file that is automatically generated by IDE to recognize its own project (e.g. ".vsproj" file on C++ project that is developed with Visual Studio). This mechanism is aimed to accommodate IDE-generated projects by considering all source codes on the same or deeper directory level with detected key file as project members.

EVALUATION

Due to unique characteristics of our approach, our evaluation will be conducted based on controlled dataset so that its impact can be measured more precisely. Our evaluation dataset is collected from source codes given on various programming books and tutorial websites. These resources are assumed to provide well-written source codes since all of them are utilized to learn programming on a particular topic. The detail of our datasets can be seen in Table 1. In general, our datasets are focused on two major topics, which are Algorithm & Data Structure and Design Pattern, wherein each topic is taken from three data sources, implemented in one or more programming languages, and represented as standalone file or project. Our dataset consists of 729 source code files where 221 of them are single-file source codes and the rests of them (508 files) are from 30 source code projects.

For each source code, its queries are generated based on keyphrases found on the most descriptive paragraph from their respective source. Keyphrases are detected in twofold. First of all, keyphrase candidates are extracted automatically through keyphrase candidate selection heuristic (Karnalim, 2016c). Afterwards, the keyphrases are manually filtered by the author to maintain query relevancy toward given source code. He discards any queries which are unrelated to given source code. Besides queries from keyphrases, our work also incorporates source code file or project name as queries since these names are frequently used for retrieving source code. As a result, our dataset yields 1066 queries: 815 queries are extracted from 187 paragraphs; 221 queries are extracted from single-file source code filename; and 30 queries are extracted from source code project name.

In order to generate more comprehensive result, our evaluation is not only measured based on the whole dataset but also measured based on sub-datasets provided on Table 1. Therefore, seven datasets are taken into our consideration. Six of them are sub-datasets from Table 1 and one of them is the merged form of these datasets. For brevity, these datasets are referred as S1, S2, S3, S4, S5, S6, and SM respectively where SM stands for merged dataset. In addition, since source codes are not always featured with comments, our evaluation also incorporates comment-excluded datasets. These datasets are generated by replicating S1-SM dataset and removing all of their respective comments. These comment-excluded datasets are then referred as CES1, CES2, CES3,

CES4, CES5, CES6, and CESM respectively where "CE" stands for "Comment-Excluded". As a result, our evaluation will be conducted on 14 datasets in total. Half of them are raw datasets whereas the others are comment-excluded datasets.

Table 1. Dataset Composition

ID	Title	Topic	Source Code Characteristic
S1	Algorithms, 4th Edition (Sedgewick & Wayne, 2011)	Algorithm & Data Structure	Java single-file source codes with intermediate comments
S2	Data Structures & Algorithms in Python (Goodrich, Tamassia, & Goldwasser, 2013)		Python single-file source codes with long comments
S3	Competitive Programming 3 (Halim & Halim, 2013)		Java and C++ single-file source codes with intermediate comments
S4	C# 3.0 Design Patterns (Bisbop, 2008)	Design Pattern	C# single-file source codes with short comments
S5	Head First Design Patterns (Freeman, Freeman, Bates, & Sierra, 2004)		Java, C++, and C# project-based source codes with short comments.
S6	Design Patterns in Java Tutorial (http://www.tutorialspoint.com/design_pattern/)		Java project-based source codes with short comments that are converted into single-file source codes.

In terms of evaluation metric, our evaluation relies on F-measure since it is frequently applied as a standard IR effectiveness measurement (Croft, et al., 2010). Yet, standard precision in F-measure is replaced with Mean Average Precision (MAP) so that proposed F-measure becomes more sensitive toward the importance of retrieved document rank.

Evaluating the Impact of Source Code Tokenization and Retrieval Model

This evaluation is intended to measure the impact of our proposed source code tokenization and retrieval model. In order to do such evaluation, two evaluation scenarios are proposed, namely Retrieval-Model-Only (RMO) and Naïve Approach (NA). On the one hand, RMO refers to our proposed approach without query expansion and document enrichment. It only consists of source code tokenization and retrieval model where the retrieval model itself is parameterized with x=2 and y=1 for weighting constants. On the other hand, NA refers to retrieval scheme which treats source codes as natural language text and retrieves its result based on standard BM25. F-measure of both scenarios toward our evaluation datasets can be seen in Figure 3. In general, RMO

outperforms NA in all cases despite its improvement varied. Thus, it is clear that RMO is more effective than NA for retrieving source code, especially for our evaluation dataset. In addition, by assuming that SM and CESM represent real-world datasets since they consist numerous source codes with various characteristics, it can also be stated that such improvement might also occur in real-world cases.

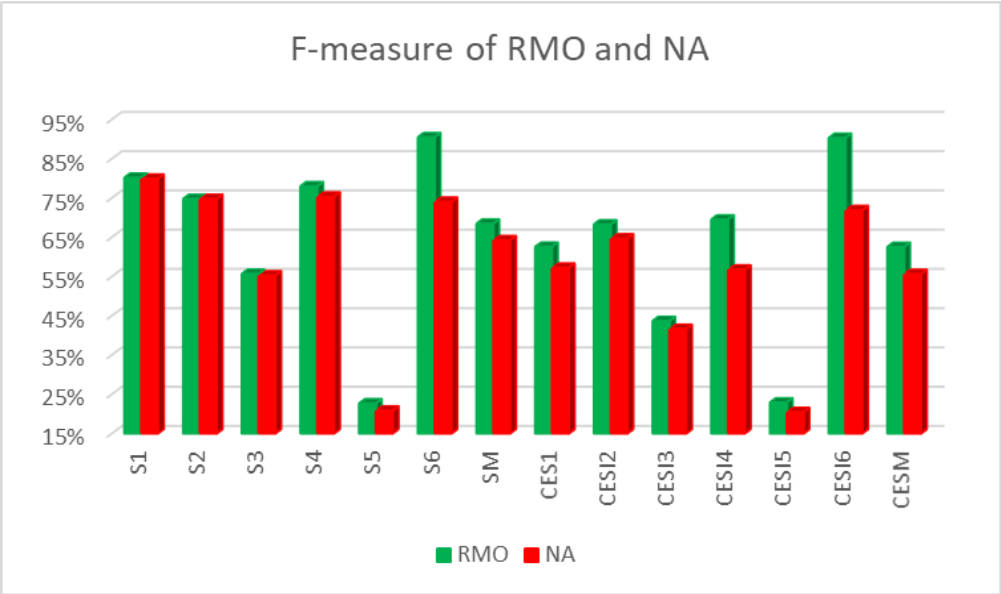

Figure 3. F-measure between RMO and NA

As seen in Figure 3, RMO generates less impact on datasets with long or intermediate comments (S1, S2, and S3). After further observation, such phenomenon is natural based on two reasons. First, in these datasets, most queries can be explicitly found on source code comments and such queries could be easily extracted by standard IR tokenization. Hence, the impact of proposed source code tokenization is unavailing when compared to standard IR approach. Second, since most comment terms are written in keyword-like manner, most of them will fall on keyword-like lexicon category, resulting imbalance proportion between keyword-like and identifier-like lexicons. Such imbalance proportion might reduce the impact of proposed tokenization that consider the distinction between keyword and identifier lexicon. When incorporated on datasets with short or no comments, our approach yields a significant improvement. It yields 6.967% averaged improvement on short-comment datasets (S4, S5, and S6) and 7.355% averaged improvement on comment-excluded dataset (CES1-CESM). According to these findings, it is clear that the impact of our source code tokenization and retrieval model are inversely proportional to the number of comments. It would be more effective when the number of comments from indexed source codes are reduced.

Evaluating Retrieval Parameters

Retrieval parameters (x and y) are incorporated to reweight retrieval score based on lexical pattern category. The larger the difference between these parameters, the higher retrieval score for preferred lexical category will be rather than the non-preferred one. This evaluation aims to find out the best parameter values so far for those parameters. For evaluation purpose, there are 10 constant pairs that will be assigned to those

parameters, resulting 10 scenarios. *x* will be assigned with an integer value from 1 to 10 for each scenario respectively while *y* will be assigned with 1 for all scenarios. For clarity, each scenario will be referred as two integer values separated by a hyphen where the first value refers to *x* and the second one refers to *y*. For example, 3-1 refers to an evaluation scenario with *x*=3 and *y*=1. In order to provide more accurate result, each scenario will be conducted without query expansion and document enrichment mechanism.

Each scenario will be evaluated with 4 datasets which are SM, CESM, SM with only identifier-like queries (SMIQ), and CESM with only identifier-like queries (CESMIQ). The last two datasets are used to evaluate the impact of retrieval parameters for handling cases where users typically expect the retrieved results to have similar lexical pattern category toward given query. Identifier-like category is preferred to the keyword one since such category is more frequently used in real cases based on our informal survey. Evaluation result toward our proposed scenarios can be seen in Figure 4. MAP is displayed as a replacement of F-measure since the modification of our retrieval parameters only affects MAP instead of the whole F-measure. It only changes the position of retrieved source codes without changing the members of retrieved results.

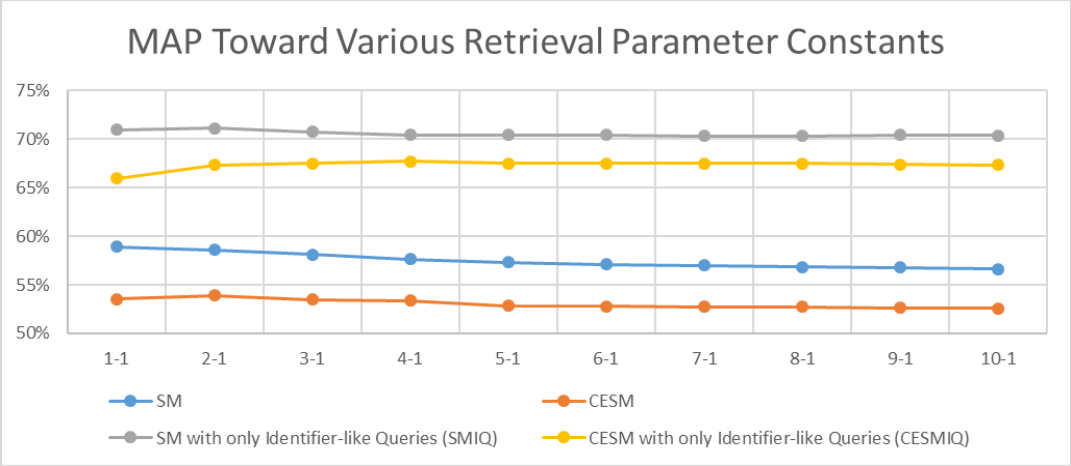

Figure 4. MAP Toward Various Retrieval Parameter Constants

As seen in Figure 4, optimal scenario for each dataset varies. It yields 1-1 scenario for SM dataset, 2-1 scenario for CESM dataset, 2-1 scenario for SMIQ dataset, and 4-1 scenario for CESMIQ dataset. There are several findings that can be deducted from this phenomenon. First, weighting mechanism is unavailing for source codes with long comments. Such finding is deducted from the fact that the highest MAP for SM dataset, a dataset where half of the source codes have long comments, is generated by 1-1 scenario, a scenario that does not favor preferred category at all. Second, weighting mechanism only affects the rank of retrieved results when the number of terms for each category is balanced. Such finding is deducted from the fact that the highest MAP for CESM, a dataset where the number of terms for each category is more balanced than SM, is generated by 2-1 scenario, a scenario which favors preferred category two times higher than the non-preferred one. Third, the impact of weighting mechanism grows higher when queries are limited to a particular category. Such finding is deducted from the fact that the most optimal scenario for both SMIQ and CESMIQ, two scenarios

which only consider identifier-like queries, generate higher *x* when compared to the most optimal scenario for their respective original dataset (SM and CESM).

Evaluating the Impact of Query Expansion and Document Enrichment

This evaluation is conducted to measure the impact of proposed query expansion and document enrichment for enhancing source code retrieval effectiveness. Four scenarios, which are generated based on the possible combination of both query expansion and document enrichment, are proposed and evaluated for each evaluation dataset. These scenarios are Retrieval-Model-Only (RMO), Retrieval-And-Query-Expansion (RAQE), Retrieval-And-Document-Enrichment (RADE), and Combined-Form (CF). RMO refers to our baseline scheme which only incorporates source code tokenization and retrieval model with x=2 and y=1; RAQE refers to RMO with query expansion; RADE refers to RMO with document enrichment; and CF refers to RMO with both query expansion and document enrichment. For evaluation purpose, the number of candidates for query expansion is limited to 20, a number that is frequently used as a candidate threshold for query expansion (Carpineto & Romano, 2012).

The impact of our proposed features can be seen in Figure 5. In order to generate more intuitive display, Figure 5 only shows delta F-measure between selected scheme and RMO. In general, both features are quite compromising since they yield positive delta improvement in most datasets. RAQE yielded 2.640% average improvement whereas RADE yielded 0.538%. When combined together (CF scenario), both features enhance retrieval effectiveness by 2.649%, which is higher than RAQE or RADE improvement. Thus, it can be stated that both features are supportive to each other in terms of enhancing effectiveness.

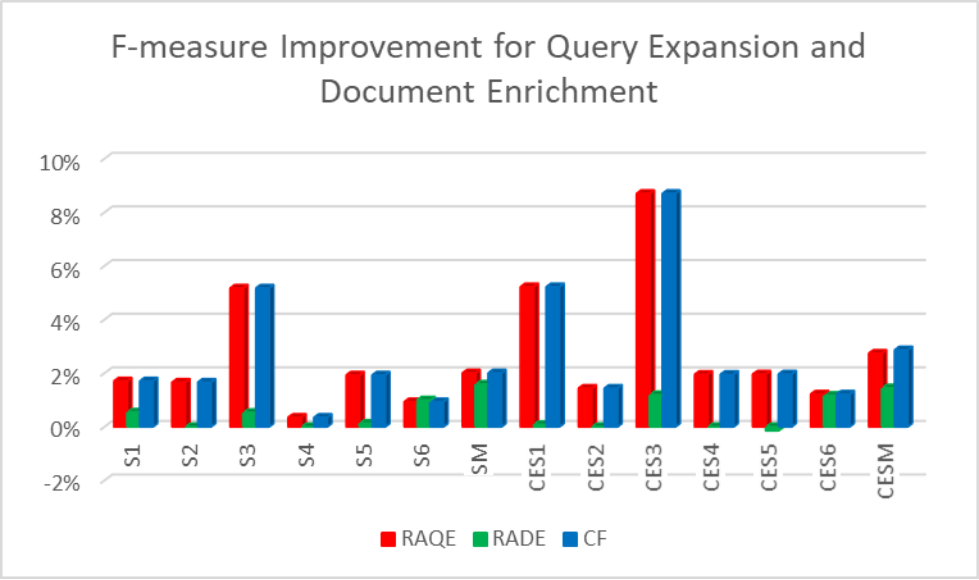

Figure 5. F-measure Improvement for Query Expansion and Document Enrichment

Query expansion yields a significant impact for our proposed approach since it reformulates given query into more-detailed form by incorporating several additional query terms. These additional query terms can either strengthen the impact of already-retrieved relevant documents or retrieve more relevant documents. Based on our manual observation through our dataset, it is clear that, for our case, query expansion is more

inclined on retrieving more relevant documents. This finding is consistent with query expansion behavior on natural language domain (Carpineto & Romano, 2012). Hence, it can be stated that query expansion on both natural language and source code domain yield similar impact, they tend to retrieve more relevant documents.

Document enrichment only yields low improvement since this mechanism assures that enriched source codes are always ranked lower than their respective enricher. Thus, even though many relevant source codes are retrieved using this approach, most of them would be placed at the end of result list due to their low score. In addition, since this mechanism might also enlarge false positive results, it may lower F-measure at some points, especially when the number of irrelevant document is large. An example of this phenomena can be seen in CES5 dataset where RADE yields negative result due to its large number of irrelevant documents.

Threats to Validity

In general, there are two threats to validity which should be considered toward the result of this work. Firstly, it is important to note that our evaluation datasets only represent a small number of programming languages. It only involves 5 programming languages, which are Java, C, C++, C#, and Python. Therefore, the result cannot be generalized for all programming languages. It might be changed when more programming languages are incorporated. Secondly, generated queries in our dataset is limited to keyphrases found on referred text. Thus, the results cannot be generalized to all kinds of queries, including human-generated queries.

CONCLUSION

In this paper, a language-agnostic source code retrieval, which relies on Keyword & Identifier lexical pattern, has been developed. Using this approach, new programming languages could be incorporated automatically since no programming-language-centric features are used. Four components are proposed as our major contribution. These components are source code tokenization, retrieval model, query expansion, and document enrichment. According to our evaluation, these components are effective to retrieve relevant source codes agnostically, even though the improvement for each component varies. For future work, we intend to incorporate large-scale dataset such as GitHub corpus (Allamanis & Sutton, 2013) for further evaluation. In addition, we also intend to measure the impact of standard IR text processing such as stemming and stopping on our language-agnostic source code domain.

REFERENCES

Allamanis, M., & Sutton, C. (2013). Mining Source Code Repositories at Massive Scale using Language Modeling. *The 10th Working Conference on Mining Software Repositories.*

Bajracharya, S., Ossher, J., & Lopes, C. (2014). Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming, 79.*

Begel, A. (2007). Codifier: a programmer-centric search user interface. *Workshop on Human-Computer Interaction and Information Retrieval.*

Bisbop, J. (2008). *C# 3.0 Design Patterns.* United States of America: O'Reilly.

Carpineto, C., & Romano, G. (2012). A Survey of Automatic Query Expansion in Information Retrieval. *ACM Computing Surveys (CSUR), 44*(1).

Cass, S. (2016, 7 26). *The 2016 Top Programming Languages - IEEE Spectrum*. Retrieved 9 28, 2016, from http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

Chatterjee, S., Juvekar, S., & Sen, K. (2009). Sniff: A search engine for java using free-form queries. *Lecture Notes in Computer Science, 5503*.

Chavez, A., Tornabene, C., & Wiederhold, G. (1998). Software component licensing issues: A primer. *IEEE Software, 15*(5).

Croft, B., Metzler, D., & Strohman, T. (2010). *Search Engine : Information Retrieval in Practice.* Boston: Pearson Education .Inc.

Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns.* O'Reilly.

Girardi, M. R., & Ibrahim, B. (1995). Using English to retrieve software. *Journal of Systems and Software, 30*(3).

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data Structures & Algorithms in Python.* United States of America: Wiley.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., & Cumby, C. (2010). A search engine for finding highly relevant applications. *ACM/IEEE 32nd International Conference on Software Engineering.*

Gysin, F. S., & Kuhn, A. (2010). A trustability metric for code search based on developer karma. *2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation.* New York.

Halim, S., & Halim, F. (2013). *Competitive Programming 3.* lulu.

Hoffmann, R., Fogarty, J., & Weld, D. S. (2007). Assieme: finding and leveraging implicit references in a web search interface for programmers. *The 20th annual ACM symposium on User interface software and technology*. New York.

Hummel, O., Janjic, W., & Atkinson, C. (2008). Code conjurer: Pulling reusable software out of thin air. *IEEE Software, 25*(5).

Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., & Kusumoto, S. (2005). Ranking significance of software components based on use relation. *IEEE Transactions on Software Engineering, 31*(3).

Karnalim, O. (2015). Extended Vector Space Model with Semantic Relatedness on Java Archive Search Engine. *Jurnal Teknik Informatika dan Sistem Informasi (JuTISI), 1*(2), 111-122.

Karnalim, O. (2016a). Detecting Source Code Plagiarism on Introductory Programming Course Assignments using a Bytecode Approach. In *2016 International Conference on Information & Communication Technology and Systems (ICTS),* (pp. 63-68). IEEE.

Karnalim, O. (2016b). Improving Scalability of Java Archive Search Engine through Recursion Conversion and Multithreading. *CommIT (Communication and Information Technology) Journal, 10*(1), 15-26.

Karnalim, O. (2016c). Software Keyphrase Extraction with Domain-Specific Features. In *2016 International Conference on Advanced Computing and Applications (ACOMP),* (pp. 43-50). IEEE.

Karnalim, O., & Mandala, R. (2014). Java Archives Search Engine Using Byte Code as Information Source. *International Conference on Data and Software Engineering (ICODSE)* (pp. 92-97). Bandung: IEEE.

Keivanloo, I., Roostapour, L., Schugerl, P., & Rilling, J. (2010). SE-CodeSearch: A scalable Semantic Web-based source code search infrastructure. *International Conference on Software Maintenance (ICSM).*

Kim, J., Lee, S., Hwang, S.-w., & Kim, S. (2010). Towards an Intelligent Code Search Engine. *The National Conference of the American Association for Artificial Intelligence.*

Lemos, O. A., Bajracharya, S. K., Ossher, J., Morla, R. S., Masiero, P. C., Baldi, P., & Lopes, C. V. (2007). CodeGenie: using test-cases to search and reuse source code. *The twenty-second IEEE/ACM international conference on Automated software engineering.* New York.

Lu, M., Sun, X., Wang, S., Lo, D., & Duan, Y. (2015). Query expansion via wordnet for effective code search. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on* (pp. 545-549). IEEE.

Lv, F., Zhang, H., Lou, J. G., Wang, S., Zhang, D., & Zhao, J. (2015). Codehow: Effective code search based on API understanding and extended boolean model. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (pp. 260-270). IEEE.

Mishne, G., & Rijke, M. D. (2004). Source code retrieval using conceptual similarity. *RIAO '04 Coupling approaches, coupling media and coupling languages for information retrieval.* Paris.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). *The PageRank Citation Ranking: Bringing Order to the Web.* PageRank Citation Ranking: Bringing Order to the Web Project.

Paul, S., & Prakash, A. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, *20*(6).

Porter, M. F. (2001). *The English (Porter2) stemming algorithm*. Retrieved 4 12, 2016, from http://snowball.tartarus.org/algorithms/english/stemmer.html

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, *8*(11).

Puppin, D., & Silvestri, F. (2006). The social network of java classes. *The 2006 ACM symposium on Applied computing.* New York.

Reiss, S. P. (2009). Semantics-based code search. *The 31st International Conference on Software Engineering.* Washington.

Robertson, S. E., Walker, S., & Hancock-Beaulieu, M. (1998). Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. *TREC.*

Sadowski, C., Stolee, K. T., & Elbaum, S. (2015). How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 191-201). ACM.

Sedgewick, R., & Wayne, K. (2011). *Algorithms, 4th Edition.* Addison-Wesley.

Sindhgatta, R. (2016). Using an information retrieval system to retrieve source code samples. *The 28th international conference on Software engineering.* New York.

Stolee, K. T., Elbaum, S., & Dwyer, M. B. (2016). Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software*, *116*, 35-48.

Stylos, J., & Myers, B. A. (2006). Mica: A web-search tool for finding API components and examples. *IEEE Symposium on Visual Languages and Human-Centric Computing.*

Thummalapenta, S., & Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. *The twenty-second IEEE/ACM international conference on Automated software engineering.* New York.

Vanderlei, T. A., Durao, F. A., Martins, A. C., Garcia, V. C., Almeida, E. S., & Meira, S. R. (2007). A cooperative classification mechanism for search and retrieval software components. *The 2007 ACM symposium on Applied computing.* New York.

Vinayakarao, V., Sarma, A., Purandare, R., Jain, S., & Jain, S. (2017). Anne: Improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (pp. 211-220). ACM.

Wise, M. J. (1993). *Running rabin-karp matching and greedy string tiling.* Basser Departement of Computer Science, Sydney University.

Ye, Y., & Fischer, G. (2002). Supporting reuse by delivering task-relevant and personalized information. *The 24th International Conference on Software Engineering.* New York.