CodeCatch: Extracting Source Code Snippets from Online Sources

Themistoklis Diamantopoulos, Georgios Karagiannopoulos, and Andreas L. Symeonidis
Electrical and Computer Engineering Dept.
Aristotle University of Thessaloniki
Thessaloniki, Greece
thdiaman@issel.ee.auth.gr, gkaragif@ece.auth.gr, asymeon@eng.auth.gr

ABSTRACT

Nowadays, developers rely on online sources to find example snippets that address the programming problems they are trying to solve. However, contemporary API usage mining methods are not suitable for locating easily reusable snippets, as they provide usage examples for specific APIs, thus requiring the developer to know which library to use beforehand. On the other hand, the approaches that retrieve snippets from online sources usually output a list of examples, without aiding the developer to distinguish among different implementations and without offering any insight on the quality and the reusability of the proposed snippets. In this work, we present CodeCatch, a system that receives queries in natural language and extracts snippets from multiple online sources. The snippets are assessed both for their quality and for their usefulness/preference by the developers, while they are also clustered according to their API calls to allow the developer to select among the different implementations. Preliminary evaluation of CodeCatch in a set of indicative programming problems indicates that it can be a useful tool for the developer.

KEYWORDS

Code Reuse, Snippet Mining, API Usage Mining

1 INTRODUCTION

Lately, the outspread of the Internet and the adoption of the open-source development paradigm have greatly influenced the way software is developed. Nowadays, the first step towards solving a coding issue, developing a component/algorithm or even integrating a library API is to search online for different possible solutions. Several tools exist to retrieve such information, including search engines, question-answering websites (e.g. Stack Overflow[1]), programming forums (e.g. Code Project[2]), etc. In this context, modern software development practices imply considerable effort in locating and integrating the solutions within these online sources of information.

In this context of software reuse, new software systems are usually built using components found in software libraries and integrating them by means of small code fragments, called *snippets*. The challenge for a developer following such a software development practice is to find the proper snippets to perform the envisioned tasks (e.g. read a CSV file, send a file over ftp, etc.) and integrate them in his/her own source code. Using the tools mentioned in the previous paragraph for this task is far from optimal, as it requires leaving one's IDE to navigate through several online pages, in an attempt to comprehend the different ways to solve the problem before selecting and integrating an implementation.

Various methodologies have been proposed to address this challenge, most of which focus on *API usage mining* and *snippet mining*. API usage mining systems extract and present examples for specific library APIs [4, 8, 11, 13, 18, 22]. Though effective, these systems are only focused on finding out how to use an API, without providing solutions in generic cases or in cases when determining which library to use is part of the question. Furthermore, several of them [8, 18, 22] return call sequences instead of ready-to-use snippets.

On the other hand, generic snippet mining systems [3, 20, 21] employ indexing mechanisms that include snippets for multiple queries. Nevertheless, they also have important limitations. Concerning systems with local indexes [21], the quality and the diversity of their results is usually confined by the size of the index. Moreover, the retrieved snippets for all systems [3, 20, 21] are presented in the form of lists that do not allow easily distinguishing among different implementations (e.g. using different libraries to perform file management). The quality and the reusability of the results are also usually not evaluated. Finally, a common limitation in certain systems is that they involve some specialized query language, which may require additional effort by the developer.

In this work, we design and develop *CodeCatch*, a system that receives queries in natural language, and employs the Google search engine to extract useful snippets from multiple online sources. Our system further evaluates the readability of the retrieved snippets, as well as their preference/acceptance by the developer community using information from online repositories. Moreover, CodeCatch performs clustering to group snippets according to their API calls, allowing the developer to first select the desired API implementation and subsequently choose which snippet to use.

2 RELATED WORK

As already mentioned, in this work we focus on source code recommendation systems and, specifically, on systems that receive queries for solving specific programming tasks and recommend source code snippets suitable for reuse in the developer's source code. Thus, in the following paragraphs, we analyze the different approaches that have been proposed for this and for any similar challenges.

Some of the first source code recommendation systems, such as Prospector [12] or PARSEWeb [17], focused on the problem of finding a path between an input and an output object in source code. For Prospector [12], such paths are called jungloids and the resulting program flow is called a jungloid graph. The tool is quite effective for certain reuse scenarios and can also generate code.

[1]https://stackoverflow.com/
[2]https://www.codeproject.com/

However, it requires maintaining a local database, which may easily become deprecated and thus its results are limited. A rather more broad solution was offered by PARSEWeb [17], which employed the Google Code Search Engine[3] and thus the resulting snippets were always up-to-date. Both systems, however, were limited to cases where the developer knows exactly which API objects to use, and he/she is only concerned with integrating them.

Another category of systems are those that generate API usage examples in the form of call sequences by mining client code (i.e. code using the API under analysis). MAPO [22] is a representative case, which employs *frequent sequence mining* to identify common usage patterns. As noted, however, by Wang et al. [18], MAPO does not account for the diversity of usage patterns, and thus outputs a large number of API examples, many of which are redundant. To improve on this aspect, the authors propose UP-Miner [18], a system that aims to achieve high coverage and succinctness. UP-Miner models client source code using graphs and mines frequent closed API call paths/sequences using the BIDE algorithm [19]. PAM [8] is another similar system that employs probabilistic machine learning to extract API call sequences, which are proven to be more representative than those of MAPO and UP-Miner. An interesting novelty of the relevant work [8] is the use of an automated evaluation framework based on handwritten usage examples by the developers of the API under analysis.

Apart from the aforementioned systems, which extract API call sequences, there are also approaches that recommend ready-to-use snippets. Indicatively, we refer to APIMiner [13], a system that performs code slicing to isolate useful API-relevant statements of snippets. Buse and Weimer [4] further employ path-sensitive data flow analysis and pattern abstraction techniques to provide more abstract snippets. Another important advantage of their implementation is that it employs clustering to group the resulting snippets to categories. A similar system in this aspect is eXoaDocs [11], as it also clusters snippets, however using a set of semantic features proposed by the DECKARD code clone detection algorithm [9]. MUSE [14] also extracts API examples from client code and employs novel heuristics to rank them. Indicatively, the system defines the *ease of reuse* for each example as the percentage of its object types that belong to the library under analysis. The intuition behind this metric is that custom object types may require importing other third-party libraries and thus hinder reuse.

Though interesting, all of the aforementioned approaches provide usage examples for specific API methods, and do not address the challenge of choosing which library to use. Furthermore, several of these approaches output API call sequences, instead of ready-to-use solutions in the form of snippets. Finally, none of the aforementioned systems accepts queries in natural language, which are certainly preferable when trying to formulate a programming task without knowing which APIs to use beforehand.

To address the above challenges, several recent systems focus on generic snippets. Some of them employ pattern-based code search and rely on local indexes [10, 21], while others connect to online search engines and further allow queries in natural language [3, 20]. Focusing on the latter, we may note BluePrint [3], a system offered as an Eclipse plugin. Blueprint employs the Google search engine to discover and rank snippets, thus ensuring that useful results are retrieved for almost any query. An even more advanced system is Bing Code Search [20], which employs the Bing search engine for finding relevant snippets, and further introduces a multi-parameter ranking system for snippets as well as a set of transformations to adapt the snippet to the source code of the developer. Finally, there are also systems that search for snippets in a single data source, such as e.g. Prompter [16], which recommends solutions from Stack Overflow. However, the results of these systems are usually limited when compared to those of the approaches that employ a full-scale web search engine.

Though useful for extracting code snippets, the aforementioned systems do not provide a choice of implementations to the developer. Furthermore, most of them do not assess the retrieved snippets both from a quality and from a reusability perspective. This is crucial, as libraries that are most often preferred by developers typically exhibit high quality and good documentation, while they are obviously supported by a larger community [2, 15]. In this work, we present CodeCatch, a snippet mining system designed to overcome the above limitations. CodeCatch employs the Google search engine in order to receive queries in natural language and at the same time extract snippets from multiple online sources. As opposed to current systems, our tool assesses not only the quality (readability) of the snippets but also their reusability/preference by the developers. Furthermore, CodeCatch employs clustering techniques in order to group the snippets according to their API calls, and thus allows the developer to easily distinguish among different implementations.

3 CODECATCH SNIPPET RECOMMENDER

The architecture of CodeCatch is shown in Figure 1. The input is a query given in natural language to the Downloader, which posts it to the Google search engine, to extract snippets from the result pages. Consequently, the Parser extracts the API calls of the snippets, while the Reusability Evaluator scores the snippets according to whether they are widely used/preferred by developers. Additionally, the readability of the snippets is assessed by the Readability Evaluator. Finally, the Clusterer groups the snippets according to their API calls, while the Presenter ranks them and presents them to the developer. These modules are analyzed in the following subsections.

3.1 Downloader

The Downloader receives as input the query of the developer in natural language and posts it in order to retrieve snippets from multiple sources. An example query used throughout this Section is "How to read a CSV file". The Downloader receives the query and augments it before issuing it in the Google search engine. Note that our methodology is programming language-agnostic; however, and without loss of generality we focus in this paper on the Java programming language. In order to ensure that the results returned by the search engine will be targeted to the Java language, the query augmentation is performed using the Java-related keywords *java*, *class*, *interface*, *public*, *protected*, *abstract*, *final*, *static*, *import*, *if*, *for*, *void*, *int*, *long*, and *double*. Similar lists of keywords can be constructed for supporting other languages.

[3]The Google Code Search Engine resided in http://www.google.com/codesearch, however the service was discontinued in 2013.

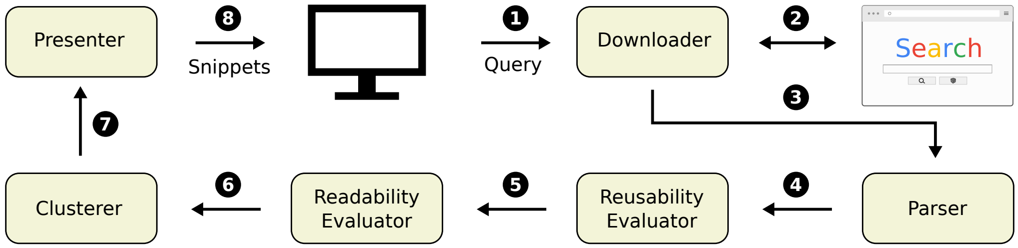


Figure 1: CodeCatch system overview.

The URLs that are returned by Google are scraped using Scrapy[4]. Upon retrieving the top 40 web pages, we extract text from HTML tags such as: *<pre>*, and *<code>*. Scraping from those tags allows us to gather the majority (around 92% as measured) of code content from web pages. Apart from the code, we collect information relevant to the webpage, including the URL, its rank at the Google results list, and the relative position of each code fragment inside the page.

3.2 Parser

The snippets are then parsed using the parser described in [6], which extracts the AST of each snippet and takes two passes over it, one to extract all (non-generic) type declarations (including fields and variables), and one to extract the method invocations (API calls). Consider, for example, the snippet of Figure 2. In the first pass, the parser extracts the declarations *line: String*, *br: BufferedReader*, *data: String[]*, and *e: Exception*. Then, upon removing the generic declarations (i.e. literals, strings and exceptions), the parser extracts the relevant method invocations, which are highlighted in Figure 2. The caller of each invocation is replaced by its type (apart from constructors for which types are already known), to finally produce the API calls *FileReader.__init__*, *BufferedReader.__init__*, *BufferedReader.readLine*, and *BufferedReader.close*.

```
String line = "";
BufferedReader br = null;
try {
  br = new BufferedReader(new FileReader("test.csv"));
  while((line = br.readLine()) != null) {
    String[] data = line.split(",");
  }
  br.close();
} catch (Exception e) {
  System.err.println("CSV file cannot be read: " + e);
}
```

Figure 2: Example snippet for "How to read a CSV file".

Note that the parser is quite robust even when the snippets are not compilable, while it also effectively isolates API calls that are not related to a type (since generic calls, such as e.g. *close*, would only add noise to the invocations). Finally, any snippets not referring to Java source code and/or not producing API calls are dropped.

[4]https://scrapy.org/

3.3 Reusability Evaluator

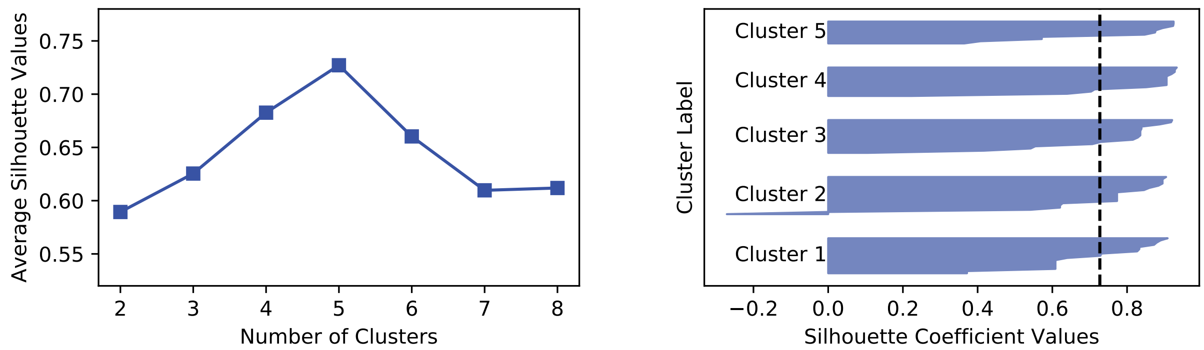
Upon gathering the snippets and extracting their API calls, the next step is to determine whether they are expected to be of use to the developer. In this context of *reusability*, we want to direct the developer towards what we call *common practice*, and, to do so, we make the assumption that snippets with API calls *commonly used* by other developers are more probable to be of (re)use. This is a reasonable assumption since answers to common programming questions are prone to appear often in the code of different projects. As a result, we designed the Reusability Evaluator by downloading a set of high-quality projects and determining the amount of reuse for the API calls of each snippet.

For this task we have downloaded the 1000 most popular Java projects of GitHub, as determined by the number of stars assigned. The rationale behind this choice of projects is highly intuitive and is also strongly supported by current research; popular projects have been found to exhibit high quality [15], while they contain reusable source code [7] and sufficient documentation [2]. As a result, we expect that these popular projects use the most effective APIs and in a good way.

Upon downloading the projects, we construct a local index where we store their API calls, which are extracted using the Parser. After that, we score each API call by dividing the number of projects in which it is present by the total number of projects. For the score of each snippet, we average over the scores of its API calls. Finally, the index also contains all qualified names so that we may easily retrieve them given a caller object (e.g. *BufferedReader: java.io.BufferedReader*).

3.4 Readability Evaluator

To construct a model for the readability of snippets, we used the publicly available dataset from [5] that contains 12,000 human judgements by 120 annotators on 100 snippets of code. We build our model as a binary classifier that assesses whether a code snippet is *more readable* or *less readable*. At first, for each snippet, we extract a set of features that are related to readability, including e.g. the average identifier length, the average number of comments, etc. (see [5] for the full list of features). After that, we train an AdaBoost classifier on the aforementioned dataset. The classifier was built with decision trees as base estimator, while the number of estimators and the learning rate were set to 160 and 0.6, respectively. We built our model using 10-fold cross-validation and the average F-measure for all folds was 85%, indicating that it is effective enough for determining whether a new snippet has high readability.



(a) **(b)**

Figure 3: Example silhouette analysis for clustering the snippets of query "How to read a CSV file", including (a) the silhouette score for different number of clusters and (b) the silhouette of each of the 5 clusters.

3.5 Clusterer

Upon scoring the snippets, the next step is to cluster them. A simple approach would be to cluster the snippets by examining them as text documents; however this approach would fail to distinguish among different implementations. Consider, for example, the snippet of Figure 2 along with that of Figure 4. If we remove any punctuation and compare the two snippets, we may find out that more than 60% of the tokens of the second snippet are also present in the first. The two snippets, however, are quite different; they have different API calls and thus refer to different implementations.

```
Scanner scanner = null;
try{
  scanner = new Scanner(new File("test.csv"));
  scanner.useDelimiter(",");
  while(scanner.hasNext()) {
    System.out.print(scanner.next() + " ");
  }
  scanner.close();
} catch (Exception e) {
  System.err.println("CSV file cannot be read: " + e);
}
```

Figure 4: Example snippet for "How to read a CSV file".

As a result, we cluster snippets based on their API calls. To do so, we employ a vector space model to represent snippets as documents and API calls as vectors (dimensions). At first, we construct a document for each snippet. For example, the document for the snippet of Figure 2 is "FileReader.__init__ BufferedReader.__init__ BufferedReader.readLine BufferedReader.close", while the document for that of Figure 4 is "File.__init__ Scanner.__init__ Scanner.hasNext Scanner.next Scanner.close". After that, we use a *tf-idf* vectorizer to extract the vector representation for each document. The weight (vector value) of each term t in a document d is computed as follows:

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D) \tag{1}$$

where $tf(t,d)$ is the term frequency of term t in document d and refers to the appearances of the API call in the snippet, while $idf(t,D)$ is the inverse document frequency of term t in the set of all documents D, referring to how common the API call is in all the snippets. In specific, $idf(t,D)$ is equal to $1 + log((1 + |D|)/(1 + d_t))$, where $|d_t|$ is the number of documents containing the term t, i.e. the number of snippets containing the relevant API call. The idf ensures that very common calls (e.g. *Exception.printStackTrace*) are given low weights, so that they do not outweigh more decisive ones.

Before clustering, we also need to define a distance metric that shall be used to measure the similarity between two vectors. Our measure of choice is the cosine similarity, which is defined for two document vectors d_1 and d_2 using the following equation:

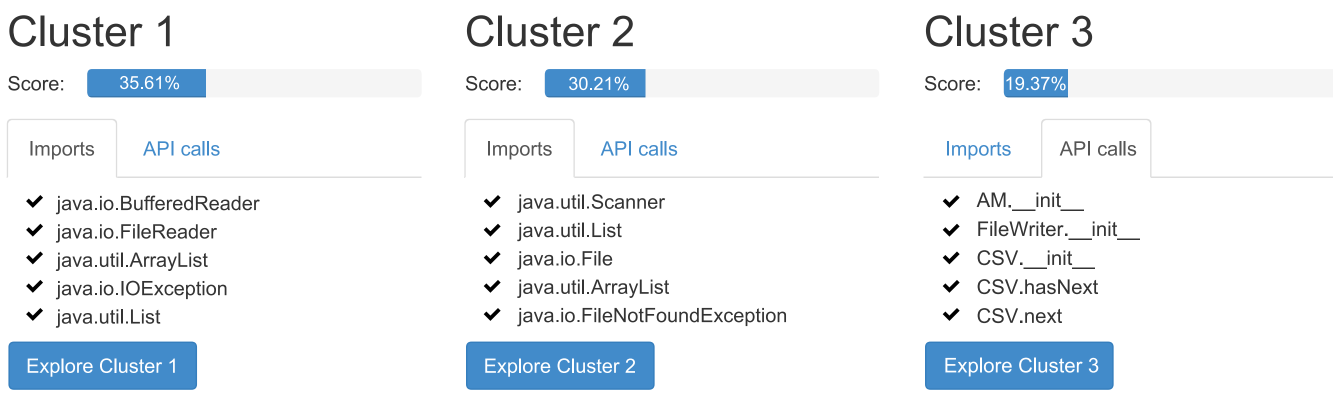
$$cos\_similarity(d_1,d_2) = \frac{d_1 \cdot d_2}{|d_1| \cdot |d_2|} = \frac{\sum_1^N w_{t_i,d_1} \cdot w_{t_i,d_2}}{\sum_1^N w_{t_i,d_1}^2 \cdot \sum_1^N w_{t_i,d_2}^2} \tag{2}$$

where w_{t_i,d_1} and w_{t_i,d_2} are the tf-idf scores of term t_i in documents d_1 and d_1 respectively, and N is the total number of terms.

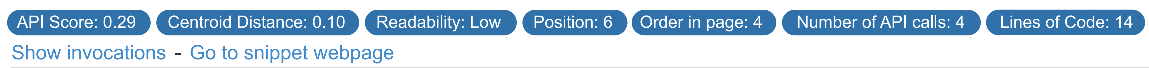
We select K-Means as our clustering algorithm, as it is known to be effective in text clustering problems similar to ours [1]. The algorithm, however, still has an important limitation as it requires as input the number of clusters. To automatically determine the best value for the number of clusters, we employ the silhouette metric. The silhouette was selected as it is a metric that encompasses both the similarity of the snippets within the cluster (cohesion) and their difference with the snippets of other clusters (separation). We execute K-Means for 2 to 8 clusters, and each time compute the value of silhouette for each document (snippet) as follows:

$$silhouette(d) = \frac{b(d) - a(d)}{max(a(d), b(d))} \tag{3}$$

where $a(d)$ is the average distance of document d from all other documents in the same cluster, while $b(d)$ is computed by measuring the average distance of d from the documents of each of the other clusters and keeping the lowest one of these values (each corresponding to a cluster). For both parameters, the distances between documents are measured using equation (2). Finally, the silhouette



(a)




```java
public class InsertValuesIntoTestDb {
    public static void main(String[] args) throws Exception {
        String splitBy = ",";
        BufferedReader br = new BufferedReader(new FileReader("test.csv"));
        while((line = br.readLine()) != null) {
            String[] b = line.split(splitBy);
            System.out.println(b[0]);
        }
        br.close();
    }
}
```

(b)

Figure 5: Screenshots of CodeCatch for "How to read a CSV file", depicting (a) the top three clusters, and (b) an example snippet.

coefficient for a cluster is given as the mean of the silhouette values of its snippets, while the total silhouette for all clusters is given by averaging over the silhouette values of all snippets.

An example silhouette analysis for the query "How to read a CSV file" is shown in Figure 3. Figure 3a depicts the silhouette score for 2 to 8 clusters, where it is clear that the optimal number of clusters is 5. Furthermore, the individual silhouette values for the documents (snippets) of the five clusters are shown in Figure 3b, and they also confirm that the clustering is effective as most samples exhibit high silhouette and only a few have marginally negative values.

3.6 Presenter

The Presenter handles the ranking and the presentation of the results. As an important aspect of this work is to present the snippets in an optimal manner, we have developed a prototype user interface as a web application that can be accessed at the URL http://codecatch.ee.auth.gr. An example screenshot of the web application for the query "How to read a CSV file" is shown in Figure 5.

When the developer inserts a query, he/she is first presented with the clusters that correspond to different implementations for the query. An indicative view of the first three clusters containing CSV file reading implementations is shown in Figure 5a. The proposed implementations include the BufferedReader API (e.g. as in Figure 2), the Scanner API (e.g. as in Figure 4), and the Java CSV reader API[5]. The clusters are ordered according to their API reusability score, which is the average of the score of each of their snippets, as defined in subsection 3.3. For each cluster, CodeCatch provides the 5 most frequent API imports and the 5 most frequent API calls, to aid the developer to distinguish among the different implementations. In cases where imports are not present in the snippets, they are extracted using the index created in subsection 3.3.

Upon selecting to explore a cluster, the developer is presented with a list of its snippets. The snippets within a cluster are ranked according to their API reusability score, and in cases of equal scores according to their distance from the cluster centroid (computed using equation (2)). This ensures that the most common usages of a specific API implementation are higher on the list. Furthermore, for each snippet, CodeCatch provides useful information, as shown in Figure 5b, including its reusability score (*API Score*), its distance from the centroid, its readability (either Low or High), the position of its URL in the results of and its order inside the URL, its number of API calls, and its number of lines of code. Finally, apart from immediately reusing the snippet, the developer has the option to isolate only the code that involves its API calls, while he/she can also check the webpage from which the snippet was retrieved.

[5]https://gist.github.com/jaysridhar/d61ea9cbede617606256933378d71751

4 EVALUATION

4.1 Evaluation Framework

Comparing CodeCatch with similar approaches has not been performed in a straightforward manner, as several of them focus on mining single APIs, while others are not maintained and/or are not publicly available. Our focus is mainly on the merit of reuse for results, and the system that is most similar to ours is Bing Code Search [20], however it targets the C# language. Hence, we have decided to perform a reusability-related evaluation against the Google search engine on a dataset of common queries shown in Table 1.

Table 1: Statistics of the Queries used as Evaluation Dataset.

ID	Query	Clusters	Snippets
1	How to read CSV file	5	76
2	How to generate MD5 hash code	5	65
3	How to send packet via UDP	5	34
4	How to split string	4	22
5	How to play audio file	6	45
6	How to upload file to FTP	4	31
7	How to initialize thread	6	51
8	How to connect to a JDBC database	5	42
9	How to read ZIP archive	6	82
10	How to send email	5	79

The purpose of our evaluation is twofold; we wish not only to assess whether the snippets are relevant, but also to determine whether the developer can indeed more easily find snippets for all relevant APIs. At first, we annotate the retrieved snippets for all the queries as relevant and non-relevant. To maintain an objective and systematic outlook, the annotation procedure was performed without any knowledge on the ranking of the snippets, while it was also kept as simple as possible; snippets were marked as relevant if and only if their code covers the functionality described by the query. That is, for the query, e.g. "How to read CSV file", any snippets used to read a CSV file were considered relevant, regardless of their size or complexity, and of any libraries involved, etc.

As already mentioned, the snippets are assigned to clusters, where each cluster involves different API usages and thus corresponds to a different implementation. As a result, we have to assess the relevance of the results per cluster, hence assuming that the developer would first select the desired implementation and then navigate into the cluster. To do so, we compare the snippets of each cluster (i.e. of each implementation) to the results of the Google search engine. CodeCatch clusters already provide lists of snippets, while for Google we construct one by assuming that the developer opens the first URL, subsequently examines the snippets of this URL from top to bottom, then he/she opens the second URL, etc.

When assessing the results, we wish to find snippets that are relevant not only to the query but also to the corresponding API usages. As a result, for the assessment of each cluster, we further annotate the results of both systems to consider them relevant when they are also part of the corresponding implementation. This, arguably, produces less effective snippet lists for the Google search engine, however note that our purpose is not to challenge the results of Google search in terms of relevance to the query, but rather to illustrate how easy or hard it is for the developer to examine the results and isolate the different ways of answering his/her query.

For each query, upon having constructed the lists of snippets for each cluster and for Google, we compare them using the *reciprocal rank* metric. This metric was selected as it is commonly used to assess information retrieval systems in general and also systems similar to ours [20]. Given a list of results, the reciprocal rank for a query is computed as the inverse of the rank of the first relevant result. For example, if the first relevant result is in the first position, then the reciprocal rank is $1/1 = 1$, if the result is in the second position, then the reciprocal rank is $1/2 = 0.5$, etc.

4.2 Evaluation Results

Figure 6 depicts the reciprocal rank of CodeCatch and Google for the snippets corresponding to the three most popular implementations for each query. At first, interpreting this graph in terms of the relevance of the results indicates that both systems are very effective. In specific, if we consider that the developer would require a relevant snippet regardless of the implementation, then for most queries, both CodeCatch and Google produce a relevant result in the first position (i.e. reciprocal rank equal to 1).

If, however, we focus on the different implementations , we can make certain interesting observations. Consider, for example the first query ("How to read a CSV file"). If the developer requires the most popular *BufferedReader* implementation (I1), both CodeCatch and Google output a relevant snippet in the first position. Similarly, if one wished to use the *Scanner* (I2) or the *Java CSV reader* (I3), our system would return a ready-to-use snippet in the top of the second cluster or in the second position of the third cluster (i.e. reciprocal rank equal to 0.5). On the other hand, using Google would require examining more results (3 and 50 results for I2 and I3 respectively, as the corresponding reciprocal ranks are equal to 0.33 and 0.02 respectively). Similar conclusions can be drawn for most queries.

Another important point of comparison of the two systems is whether they return the most popular implementations at the top of their list. CodeCatch is clearly more effective than Google in this aspect. Consider, for example, the sixth query; in this case, the most popular implementation is found in the third position of Google, while the snippet found in its first position corresponds to a less popular implementation. This is also clear in several other queries (i.e. queries 2, 3, 7, 10). Thus, one could argue that CodeCatch does not only provide the developer with all different API implementations for his/her query but also further aids him/her to select the most popular of them, which is usually the most preferable.

Finally, we refer to the mean reciprocal rank, computed as the average of all scores for each system. The mean reciprocal ranks for CodeCatch and Google are 0.754 and 0.379 respectively. Their difference is clearly significant (a paired t-test gave $t = 6.116$ and $p = 1.157 \cdot 10^{-5}$), however note that the scope of this work is to identify different implementations and thus these averages do not measure overall relevance; instead, they measure implementation-specific relevance (see subsection 4.1)[6].

[6]Several other metrics have also been possible to compute, such as the mean average precision (0.754 and 0.379 for CodeCatch and Google respectively) or the normalized discounted cumulative gain (0.843 and 0.487 for CodeCatch and Google respectively). However, these also lie outside the main scope and thus their analysis is omitted.

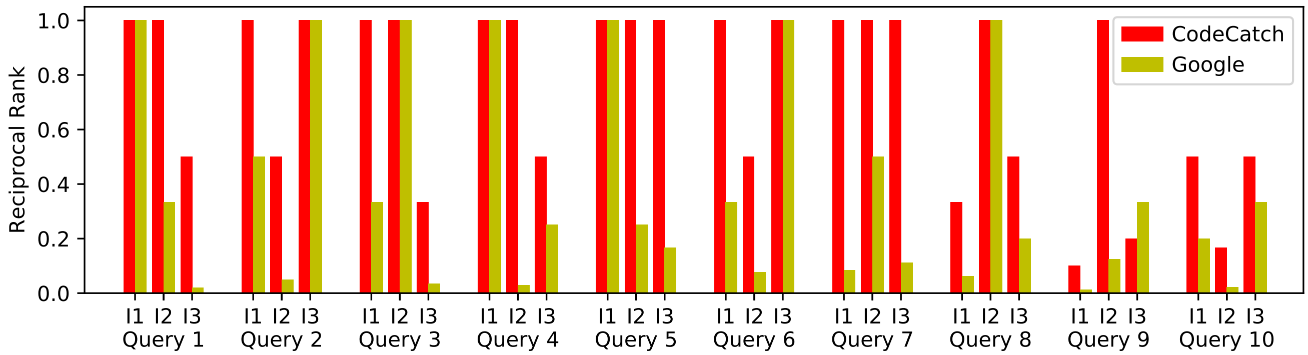


Figure 6: Reciprocal Rank of CodeCatch and Google for the three most popular implementations (I1, I2, I3) of each query.

5 CONCLUSION

Although several snippet mining systems have been developed, they typically return lists of snippets without distinguishing among the usage of different APIs and without providing information as to the reusability and readability of the snippets. In this work, we proposed a system that extracts snippets from online sources and further assesses their readability as well as their reusability based on the preference of developers. CodeCatch further provides a comprehensive view of the retrieved snippets by grouping them into clusters that correspond to different implementations.

Future work on CodeCatch lies in several directions. At first, we may extend our ranking scheme to include e.g. the position of each snippet's URL in the Google results, etc. Furthermore, different methodologies can be tested for the readability evaluator in order to assess its performance on new snippets. Finally, an interesting idea would be to conduct a developer study in order to further assess CodeCatch for its effectiveness in retrieving useful code snippets. By setting up certain development tasks, we could also assess whether the proposed snippets can be easily integrated into the source code of the developer.

REFERENCES

[1] Charu C. Aggarwal and ChengXiang Zhai. 2012. *A Survey of Text Clustering Algorithms*. Springer US, Boston, MA, 77–128.

[2] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. 2014. Co-evolution of Project Documentation and Popularity Within Github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, New York, NY, USA, 360–363.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 513–522.

[4] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 782–792.

[5] Raymond P. L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* 36, 4 (2010), 546–558.

[6] Themistoklis Diamantopoulos and Andreas L. Symeonidis. 2015. Employing Source Code Information to Improve Question-answering in Stack Overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 454–457.

[7] Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2017. Towards Modeling the User-Perceived Quality of Source Code using Static Analysis Metrics. In *Proceedings of the 12th International Joint Conference on Software Technologies (ICSOFT 2017)*. SciTePress, Setúbal, Portugal, 73–84.

[8] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 254–265.

[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105.

[10] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 664–675.

[11] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an Intelligent Code Search Engine. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*. AAAI Press, Palo Alto, CA, USA, 1358–1363.

[12] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. *SIGPLAN Not.* 40, 6 (2005), 48–61.

[13] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*. IEEE Computer Society, Piscataway, NJ, USA, 401–408.

[14] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 880–890.

[15] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas L. Symeonidis. 2016. User-Perceived Source Code Quality Estimation based on Static Analysis Metrics. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS 2016)*. IEEE Press, Piscataway, NJ, USA, 100–107.

[16] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, New York, NY, USA, 102–111.

[17] Suresh Thummalapenta and Tao Xie. 2007. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 204–213.

[18] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 319–328.

[19] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 79–90.

[20] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. 2015. *Building Bing Developer Assistant*. Technical Report MSR-TR-2015-36. Microsoft Research.

[21] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. 2012. SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 219–228.

[22] Tao Xie and Jian Pei. 2006. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 54–57.