Towards Readability-Aware Recommendations of Source Code Snippets

Athanasios Michailoudis[a], Themistoklis Diamantopoulos[b] and Andreas Symeonidis[c]
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece

Keywords: Snippet Mining, API Usage Mining, Code Readability.

Abstract: Nowadays developers search online for reusable solutions to their problems in the form of source code snippets. As this paradigm can greatly reduce the time and effort required for software development, several systems have been proposed to automate the process of finding reusable snippets. However, contemporary systems also have certain limitations; several of them do not support queries in natural language and/or they only output API calls, thus limiting their ease of use. Moreover, the retrieved snippets are often not grouped according to the APIs/libraries used, while they are only assessed for their functionality, disregarding their readability. In this work, we design a snippet mining methodology that receives queries in natural language and retrieves snippets, which are assessed not only for their functionality but also for their readability. The snippets are grouped according to their used API calls (libraries), thus enabling the developer to determine which solution is best fitted for his/her own source code, and making sure that it will be easily integrated and maintained. Upon providing a preliminary evaluation of our methodology on a set of different programming queries, we conclude that it can be effective in providing reusable and readable source code snippets.

1 INTRODUCTION

The evolution of the Internet and the open-source community has greatly influenced the way software is developed. As more and more software projects are hosted online in platforms like GitHub, developers follow a component-based software engineering paradigm, where they search for reusable snippets and integrate it into their own source code. This practice of reuse, when performed effectively, can have significant benefits both for the development time and effort and for the quality of the produced source code.

However, reusing existing code snippets is not always straightforward. The developer first has to leave the IDE, form a suitable query in a search engine, locate potential candidate snippets out of a possibly large pool and carefully assess whether they cover his/her functional criteria. After that he/she has to choose the snippet to reuse, and understand its workings before integrating it into his/her own source code.

As a result, several systems aspire to automate this process, focusing on the areas of API usage mining and/or snippet mining. API usage mining systems are tailored to specific APIs/libraries and focus on providing sequences of API calls extracted from source code examples (Xie and Pei, 2006; Wang et al., 2013; Fowkes and Sutton, 2016; Montandon et al., 2013; Kim et al., 2010; Moreno et al., 2015; Katirtzis et al., 2018), while the more generic snippet mining systems target common programming queries (e.g. opening and reading a file) and are often connected to code search engines (Wightman et al., 2012; Brandt et al., 2010; Wei et al., 2015; Diamantopoulos et al., 2018).

Both types of systems are effective in various scenarios, however they also have significant limitations. Some systems may have a limited dataset and/or methodology, not allowing them to accept queries in natural language. In addition, certain systems produce API calls instead of ready-to-use code, whereas the ones that produce snippets do not always take into account the different APIs/implementations. Finally, and more importantly, contemporary systems focus only on the functional aspect of reuse, not considering the readability of the code, thus providing solutions that may be hard to integrate and/or maintain.

In this work, we present a snippet mining system that covers the functional criteria posed by the developer, while also assessing the readability of the retrieved source code. Our system receives queries in natural language and employs the CodeSearchNet dataset (Husain et al., 2019), which offers clean pairs

[a] https://orcid.org/0009-0007-9824-0975
[b] https://orcid.org/0000-0002-0520-7225
[c] https://orcid.org/0000-0003-0235-6046

Michailoudis, A., Diamantopoulos, T. and Symeonidis, A.
Towards Readability-Aware Recommendations of Source Code Snippets.
DOI: 10.5220/0012145500003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 688-695
ISBN: 978-989-758-665-1; ISSN: 2184-2833

of documentation strings and method snippets. Moreover, we build a model that assesses snippet readability based on static analysis metrics. Finally, we extract the API calls of the snippets to detect groups of different implementations, thus allowing the developer to choose the desired source code.

The rest of this paper is organized as follows. Section 2 reviews current research in the areas of API usage mining and snippet mining. Our methodology for recommending readable snippets is presented in Section 3. Section 4 evaluates our methodology on a set of programming queries, and finally Section 5 concludes this paper and provides ideas for future work.

2 RELATED WORK

As already mentioned, our work lies in the area of recommender systems for efficient code reuse. In most cases, such systems work in one of two ways: either by identifying and mining API calls from existing source code to recommend similar methods or by mining and retrieving more generic code snippets as solutions to queries given in natural language.

Regarding API usage mining, MAPO (Xie and Pei, 2006) is a system that employs *frequent sequence mining* to extract API usage examples in the form of call sequences from client code. However, it lacks diversity-awareness of the extracted methods, which ultimately leads to the production of a substantial amount of, often irrelevant, API examples. To mitigate that, the UP-Miner (Wang et al., 2013) system was developed, which aims at mining succinct and high-coverage usage patterns of API methods from source code, using the BIDE algorithm. From a machine learning point of view, another notable implementation is PAM (Fowkes and Sutton, 2016), which extracts API calls via probabilistic machine learning. Interestingly, PAM introduces an automated evaluation framework, based on libraries that were provided with handwritten API example code by developers.

Apart from thesystems that focus on API call sequences, there are also approaches recommending reusable snippets, such as APIMiner (Montandon et al., 2013), which identifies and analyzes API calls using code slicing. Moving a step further in that direction, certain approaches cluster the retrieved snippets, according to the targeted API calls. Two such systems are eXoaDocs (Kim et al., 2010) and CLAMS (Katirtzis et al., 2018); the former applies snippet clustering using semantic features to spot duplicate code, via the DECKARD algorithm (Jiang et al., 2007), while the latter also incorporates a summarization algorithm for succinct and readable snippet presentation. Finally, MUSE (Moreno et al., 2015) also includes ranking of the resulting clusters by introducing the *ease of reuse* metric. The metric is defined as the percentage of the object types that belong to the library under analysis and is based on the assumption that custom object types may require importing other third-party libraries and thus hinder reuse.

Though certainly functional, most aforementioned methods suffer from the same limitations. First of all, the usage examples they provide are limited to a handful of API methods, while disregarding the need for a potential library change. Several of those also mainly focus on generating API call sequences, rather than practical code snippets, and, last but not least, none of these systems receive queries in natural language.

Generic snippet mining techniques have been developed to alleviate some of those limitations. Indicative examples are the Eclipse IDE plug-ins SnipMatch (Wightman et al., 2012) and Blueprint (Brandt et al., 2010), from which the first leverages code patterns and local indexes to improve snippet search and suggestions, while the second accepts queries in natural language while at the same time is connected to the Google search engine for accurate snippet retrieval and ranking. Another search engine-based recommender, developed as a Visual Studio extension, is Bing Code Search (Wei et al., 2015), which also allows for natural language questions while also implementing a multi-parameter ranking system as well as active code snippet adaptation to the developer's needs. Finally, CodeCatch (Diamantopoulos et al., 2018), is another snippet mining system, which apart from the advantages of the aforementioned plug-ins, also provides the added assets of different options for snippet selection, as well as the incorporation of reusability scoring of the mined snippets, extracted from a set of the most popular GitHub projects.

Lately, there are also several approaches using deep learning, either in the API usage mining domain or in the more generic snippet mining. Regarding API usage mining, systems like SWIM (Raghothaman et al., 2016) and T2API (Nguyen et al., 2016), translate questions given in natural language to API calls and then generate the appropriate code, containing said calls. In a similar fashion, DeepAPI (Gu et al., 2016) implements RNNs to transform queries into API sequences. On the other hand, RNNs are also utilized for code snippet reusability by the DeepCS implementation (Gu et al., 2018), where two networks are used to encode natural language and code respectively and the produced embeddings are then compared using the cosine similarity metric. CodeTransformer (Papathomas et al., 2022) follows a similar methodology, where the RNNs are replaced by Trans-

formers and the similarity metric of choice is the Triangle's Area Similarity-Sector's Area Similarity. The last two systems have the added benefit of incorporating semantics, partly due to the fact that the networks are trained on the well-curated and semantics-oriented CodeSearchNet dataset (Husain et al., 2019).

Although the aforementioned solutions usually retrieve relevant results to programming queries, they neglect to assess the readability of the provided recommendations. In this work, we present a readability-aware recommender that supports code reuse through readable code snippet selection. Our system features clustering based on API calls like API usage mining approaches, however it also allows queries in natural language and produces ready-to-use snippets instead of sequences. Moreover, using our readability model based on static analysis metrics, the system ranks these snippets using both a functional and a readability score, thus supporting developers to make the optimal decision based on their requirements.

3 METHODOLOGY

The architecture of our recommender is shown in Figure 1. As already mentioned, we use as input the CodeSearchNet dataset (Husain et al., 2019) in order to create an index of code snippets. We use the Java version of the dataset as a proof of concept, although our methodology could also be easily extended to other languages. Upon preprocessing the data (Data Preprocessor), we are able to build an index (Index Builder) for finding similar snippets. Moreover, we build a Readability Model using static analysis metrics. When a new query is given by the developer, it is initially parsed by the Query Parser (to preprocess it) and sent to the index in order to retrieve relevant snippets. The snippets are then sent to the Clusterer, which groups them according to their API calls. Finally, the Presenter is responsible for ranking the groups with respect to their functional similarity and their readability and present them to the developer.

3.1 Building Models for Snippets

3.1.1 Data Preprocessing

Before proceeding to the indexes built by our approach, we first preprocess the data. Concerning the docstrings, which are used for finding relevant methods, we create a text preprocessing pipeline that includes (1) the removal of the docstring located after the first dot symbol, effectively retaining the actual description of each snippet, (2) the removal of non-ascii and special characters, while replacing them with empty characters, (3) the removal of empty characters, (4) the conversion of every character to lowercase, (5) the removal of common stopwords of the English language, which appear frequently in text and may skew results, and (6) the lemmatization of every token, so that each word is converted to its base form.

For the code, as already mentioned, we extract the API calls of the snippets to group them, so we simply use the preprocessing pipeline by Papathomas et al. (2022), which covers our purposes. The pipeline includes steps for (1) the removal of non-ascii characters, (2) the removal of all the tokens of the code list that contain space, double quotes, or create a comment, (3) the removal of empty characters, and (4) the conversion of every character to lowercase. There are also two more steps, which do not affect our case, which are the encoding of programming symbols to unique tokens (e.g. $<$ becomes 'lessoperator', $+$ becomes 'addoperator', etc.[1]) and the removal of any tokens after first 100 for each method (used to enhance the uniformity of the dataset).

3.1.2 Index Building

Following the data preprocessing, a means of retrieving relevant data is needed. To achieve that we built a similarity scheme based on a vector space model and a metric to calculate the similarity of the two vectors.

Firstly, for the vectorization of the snippets, two vocabularies are created, corresponding to the description and code tokens, respectively. The two vocabularies consist of the 10,000 most common words of each set and are used in conjunction with a *tf-idf* vectorizer in order to create the vector representations of the respective docstring or code. The vectorizer calculates the weight of each token t within a snippet/document D according to the following formula:

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

where $tf(t,d)$ is the term frequency of term t in document d and refers to the appearances of a token 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 a specific token 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 token. By doing so, tokens that appear frequently in the snippets descriptions or code are given small weights and less common and usually more case-specific, hence useful, terms are assigned larger weights.

[1]The full list of transformations is available at (Papathomas et al., 2022).

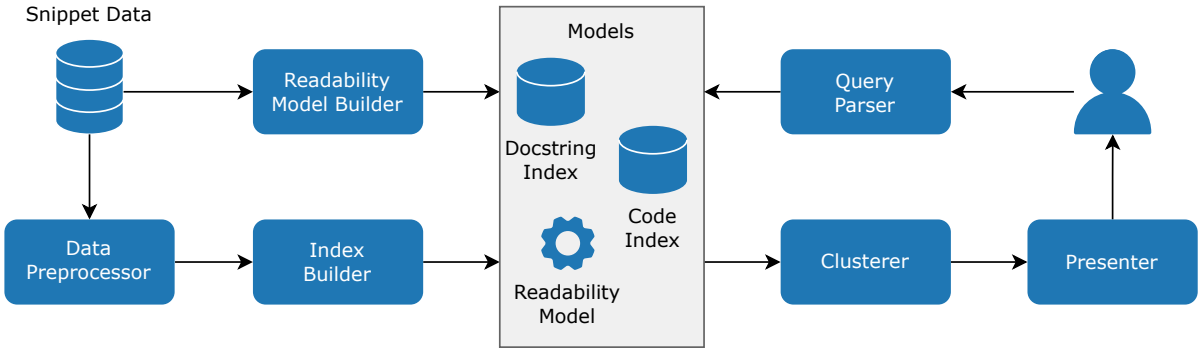

Figure 1: Methodology of our Readability-Aware Recommender of Source Code Snippets.

Finally, we use the cosine similarity metric to compare the resulting vectors. For two document vectors (i.e. query, docstring) d_1 and d_2, it is defined using the following equation:

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

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_2 respectively, and N is the total number of terms.

Any new input query in our system is vectorized through the aforementioned process, using the docstring vocabulary, and its similarity to every docstring of the dataset is computed.

To compute the similarities between snippets, we create documents by extracting the API calls of each snippet and subsequently, apply the *tf-idf* vectorizer of the code vocabulary to extract the vector representation for each document. Then, the cosine similarity metric is again computed. The similarity between API calls is preferred to the comparison of whole code snippets, as there are a lot of tokens that are shared between two snippets, thus skewing the results.

3.1.3 Readability Assessment

Regarding the readability assessment of each source code snippet, we created a machine learning model to provide a readability assessor based on static analysis metrics. A Random Forest regressor was chosen as the estimator, due to its versatility and ease of use. In order to train the model, we used the static analysis metrics and readability metrics computed by Karanikiotis et al. (2023) on the CodeSearchNet dataset. In specific, we used all metrics from the four static analysis categories relevant to snippets (complexity, coupling, documentation, and size), while we also used the readability score derived by the tool of Scalabrino et al. (2018)[2] as ground truth.

[2]https://dibt.unimol.it/report/readability/

Through an exhaustive grid search, the parameters of the model that yielded the best R^2 score were the following: $n\_estimators = 100$ (number of trees in forest), $max\_features = 0.8$ (fragment of features were considered at each split), $max\_depth = 9$ (maximum depth of each tree, $max\_samples = 100$ (subsample size used to build each tree). By the end of the training process, our model had achieved an 83.5% accuracy on a snippets readability estimation, based on the code's quality characteristics, a performance which was considered adequate for the model to be integrated in our recommendation system. The distribution of readability for all snippets of the dataset is shown in the histogram of Figure 2.

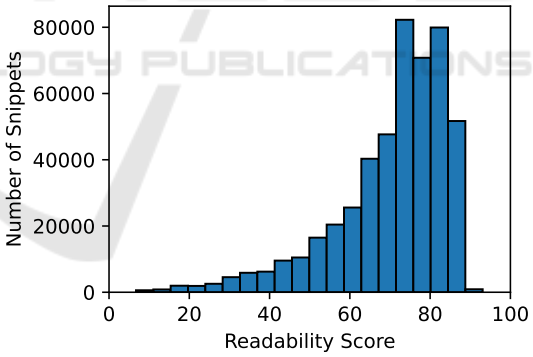

Figure 2: Histogram of the Readability of all Snippets.

3.2 Retrieving Useful Snippets

3.2.1 Query Parsing

The Query Parser receives as input the query of the user in natural language, it tokenizes and preprocesses it in the same way as described in subsection 3.1.1 and creates a tf-idf vector of the query, based on the created docstring vocabulary, as described in subsection 3.1.2. We consider a snippet as relevant if the similarity of its docstring to the developer's query is more than a similarity threshold, which we set to 0.75 as it

was enough for retrieving an adequate number of results for different queries. For each result, we retrieve the docstring, the code of snippet, as well the metrics used as input to the Readability Model in order to compute its readability.

3.2.2 Snippet Clustering

The most relevant snippets that were extracted in the previous step are forwarded to the Clusterer, which is responsible for grouping the snippets according to their API calls. To do so, we first extract the API calls of each snippet using the javalang tool[3]. The tool is used to traverse the abstract syntax tree of each snippet and retrieve two types of instructions, call instantiations (e.g. `new BufferedReader`) and method calls (e.g. `reader.readLine()`). Upon extracting the types of these instructions, we then employ the similarity scheme defined in subsection 3.1.2 (based on tf-idf) to build a distance matrix, which is in turn used for the grouping of the snippets.

The clustering algorithm of choice is Agglomerative clustering, which is a common type of hierarchical clustering used to group objects based on their similarity. The algorithm, however, has a limitation, as it requires as input the number of clusters. To automatically determine the most suitable number of clusters for a given query, we use the silhouette metric. Silhouette was selected as it encompasses both the similarity of the snippets within the cluster (cohesion) and their difference with the snippets of other clusters (separation). The clustering algorithm is executed for 2 to $snip\_num - 1$ clusters, where sip_num is the number of the retrieved snippets associated with the user's query. For each execution, we compute the total silhouette value (average over the silhouette values of all snippets) and, finally, we select the the number of clusters resulting in the highest total silhouette. The value of silhouette for each document/snippet (d) is calculated by the following equation:

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

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).

Note that the reasoning behind the clustering is that snippets with similar API calls reflect similar methods and, therefore, more than one results of the same solution are redundant. Hence, the clustering discards any duplicates, thus allowing the developer to differentiate among different implementations.

An example silhouette analysis for query "How to convert a string to integer?" is shown in Figure 3. It depicts the silhouette score for 2 to 22 clusters, where it is clear that the optimal number of clusters is 14.

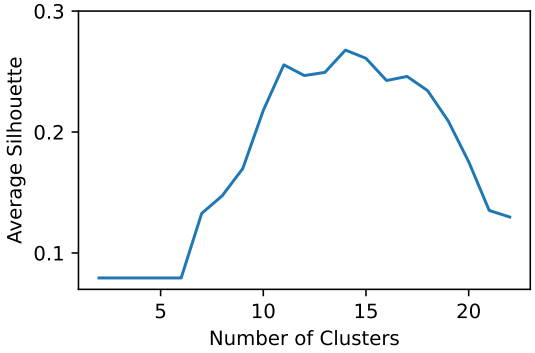

Figure 3: Example silhouette analysis for clustering the snippets of query "How to convert a string to integer?", depicting the silhouette scores for 2 to 22 clusters.

3.2.3 Results Ranking

After the clustering, the produced groups of snippets are forwarded to the Presenter, which is tasked with ranking and presenting the results. More specifically, we assume that each group/cluster consists of snippets describing the same method, thus it can be actually represented by the code snippet that has the highest similarity (or *functional score*) to the given query. Thus, the clusters are ranked according to their snippet's functional score, whereas, in cases of identical scores between snippets, higher priority is given to the snippet with the highest readability. Finally, if the readability between two snippets is also equal, the larger cluster is presented in higher order.

The results of the clustering and ranking are presented to the user along with all the necessary information, such as the original query, the optimal number of clusters, the functional and readability score of a snippet and the number of snippets in each cluster, and of course, the description, code and API calls of the retrieved snippets. To further illustrate the ranking process, we depict the two most relevant snippets for the example query "How to convert a string to integer?" in Figure 4. In this case, both results scored a perfect 1.0/1.0 similarity score with the query, while their readability scores were 86.08% and 85.89% for the first and second result, respectively.

[3]https://github.com/c2nes/javalang

```
public static Integer parse(final String s) {
    try {
        return Integer.parseInt(s);
    } catch (final NumberFormatException e) {
        return null;
    }
}
```

```
private int idToInt(String value) {
    try {
        return Integer.valueOf(value);
    } catch (NumberFormatException e) {
        throw new ConfigurationException("invalid
            type ID: " + value);
    }
}
```

Figure 4: Top two relevant snippets for the query "How to convert a string to integer?".

4 EVALUATION

4.1 Evaluation Framework

Our methodology is not compared with similar approaches, since it focuses on providing different implementations and readability scores, and not on providing the maximum number of relevant results. We assess our methodology on a dataset of 10 common programming queries shown in Table 1.

Table 1: Queries of the Evaluation Dataset.

ID	Query
Q1	How to sort an array?
Q2	How to read a csv file?
Q3	How to split a string?
Q4	How to sort list in descending order?
Q5	How to convert a string to integer?
Q6	How to get current date?
Q7	How to list all files in directory?
Q8	How to load an image?
Q9	How to move a file from directory?
Q10	How to convert date to string?

As already mentioned, the results for the queries shown in Table 1 are extracted from the CodeSearchNet dataset (Husain et al., 2019) using the vector space model for docstrings defined in Section 3. The results are then clustered, and finally the snippet groups are manually annotated to determine whether they are relevant to the given query. This annotation step was kept simple, as we marked as relevant any snippet that covers the functionality of the query, regardless of its quality and of any APIs used.

We use different metrics to assess the results of our methodology. First of all, we employ the *reciprocal rank*, which is computed as the inverse of the rank of the first relevant result (e.g. if the first relevant result is in the second position, then the reciprocal rank is $1/2 = 0.5$). This metric is the best fit if we assume that the developer most often selects the first relevant result to his/her query. However, given that in our case we assume that there may be more than one relevant implementations, which use different API calls, we also employ the *average precision* metric. The average precision for a query further takes into account the order of all relevant results, and is computed as:

$$Avg\ Precision = \frac{\sum_{k=1}^{n} P(k) \cdot rel(k)}{number\ of\ relevant\ results} \quad (1)$$

where $P(k)$ is the precision at k (fragment of relevant results at k-th position) and $rel(k)$ is equal to 1 if the result at the k-th position is relevant or 0 otherwise.

4.2 Evaluation Results

Table 2 presents certain statistics about our results. As discussed in Section 3, for each query we first retrieve any relevant snippets and then we cluster them to produce groups (third column of Table 2). We also report the number of relevant results (groups) based on the annotations (fourth column of Table 2), which are quite encouraging. In specific, most queries have at least 15 results, which are (arguably) adequate in a code search/reuse scenario. Furthermore, for 9 out of 10 queries, there are at least 3 different relevant recommendations (groups), indicating that developers could benefit from using our system.

Table 2: Statistics of the Results.

Query ID	#Retrieved Snippets	#Grouped Results	#Relevant Results
Q1	18	14	4
Q2	27	11	2
Q3	23	11	6
Q4	15	7	4
Q5	24	14	6
Q6	12	9	4
Q7	27	18	6
Q8	14	7	3
Q9	16	10	9
Q10	33	23	6

Given the results for the reciprocal rank (Figure 5, it is also clear that our methodology is effective for

retrieving useful snippets. In specific, for all queries except Q2 and Q4, a relevant result is produced in the first position, while even for Q2 and Q4 one can find a relevant snippet at least in the third position. The Mean Reciprocal Rank (MRR) is also quite high, with value 0.883, depicted with a dashed line in Figure 5.

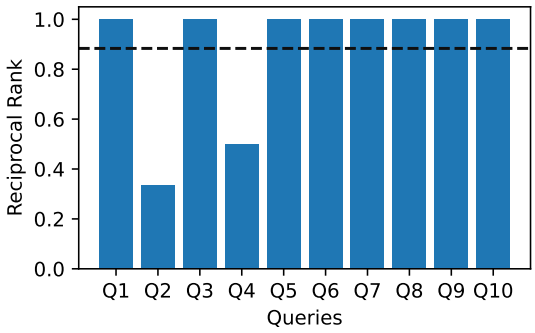

Figure 5: Reciprocal rank per query.

The results for average precision (Figure 6) are also quite encouraging. Our methodology achieves average precision close to 1 for half the queries of the dataset, with Mean Average Precision (MAP) equal to 0.826 (depicted using a dashed line). Only the results of the second query have low average precision (0.367), which is due to the fact that several snippets are incomplete (e.g. the code calls functions not present in the snippet and there are no API calls) and there are also snippets about writing csv files. An interesting note is that for Q10 there are several results about converting date to string but there are also results about converting string to date. The order of the terms is actually not captured by our model, which is a limitation that we plan to confront in future work.

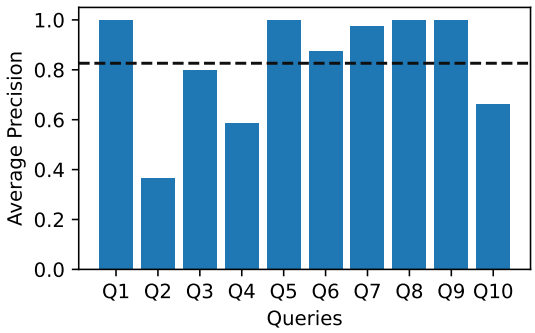

Figure 6: Average precision per query.

Finally, to assess whether our methodology retrieves snippets with high readability in the top positions, we compute the average readability of the results of each query at different levels. For each query, Figure 7 depicts the readability of the first result, the average readability of the top 3, top 5, and all results. The first result usually has quite high readability score, indicating that the developer can easily comprehend it as well as integrate it into his/her own source code. Moreover, the readability averages at top 3 and top 5 results exhibit high values (more than 0.6) indicating that on average our methodology retrieves highly readable snippets in the top positions.

5 CONCLUSION

Although finding reusable snippets is widely researched, most approaches focus only on the functional aspect. We proposed a methodology that retrieves snippets and assesses both their functionality and their readability. This way, developers can find a snippet that covers the purpose of their query and exhibits high readability, so that they can easily integrate it and, when needed, maintain it. Moreover, since we group snippets according to their API calls, we ensure that the developer reviews only the top (most readable) snippet from each possible implementation.

Concerning future work, we plan to compare our system with other systems as well as to evaluate more effective retrieval algorithms (e.g. based on code embeddings) to further enhance its results. Moreover, we could conduct a developer study to assess whether our methodology is perceived as useful by developers.

ACKNOWLEDGEMENTS

Parts of this work have been supported by the Horizon Europe project ECO-READY (Grant Agreement No 101084201), funded by the European Union.

REFERENCES

Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. (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, pages 513–522, New York, NY, USA. ACM.

Diamantopoulos, T., Karagiannopoulos, G., and Symeonidis, A. L. (2018). CodeCatch: Extracting Source Code Snippets from Online Sources. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, RAISE '18, pages 21–27, New York, NY, USA. ACM.

Fowkes, J. and Sutton, C. (2016). Parameter-free Probabilistic API Mining across GitHub. In *Proceedings*

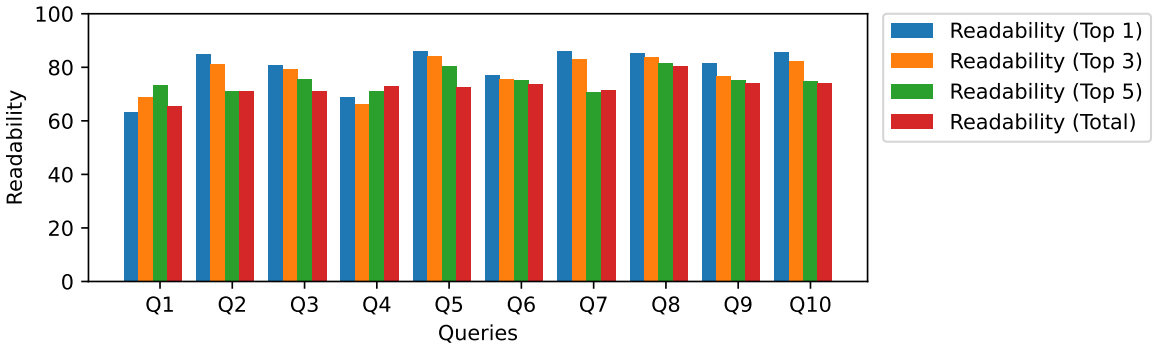

Figure 7: Readability of our methodology at the top result, the top 3 results, the top 5 results, and all the results of each query.

of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 254–265, New York, NY, USA. ACM.

Gu, X., Zhang, H., and Kim, S. (2018). Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA. ACM.

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016). Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA. ACM.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 96–105, USA. IEEE Computer Society.

Karanikiotis, T., Diamantopoulos, T., and Symeonidis, A. (2023). Source code snippets and quality analytics dataset. https://doi.org/10.5281/zenodo.7893288.

Katirtzis, N., Diamantopoulos, T., and Sutton, C. (2018). Summarizing Software API Usage Examples Using Clustering Techniques. In *21th International Conference on Fundamental Approaches to Software Engineering*, FASE 2018, pages 189–206, Boston, MA, USA. Springer International Publishing.

Kim, J., Lee, S., Hwang, S.-w., and Kim, S. (2010). Towards an Intelligent Code Search Engine. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, AAAI '10, pages 1358–1363, Palo Alto, CA, USA. AAAI Press.

Montandon, J. E., Borges, H., Felix, D., and Valente, M. T. (2013). Documenting APIs with Examples: Lessons Learned with the APIMiner Platform. In *Proceedings of the 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 401–408, Piscataway, NJ, USA. IEEE Computer Society.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., and Marcus, A. (2015). How Can I Use This Method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA. IEEE Press.

Nguyen, T., Rigby, P. C., Nguyen, A. T., Karanfil, M., and Nguyen, T. N. (2016). T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1013–1017, New York, NY, USA. ACM.

Papathomas, E., Diamantopoulos, T., and Symeonidis, A. (2022). Semantic code search in software repositories using neural machine translation. In *25th International Conference on Fundamental Approaches to Software Engineering*, pages 225–244, Munich, Germany.

Raghothaman, M., Wei, Y., and Hamadi, Y. (2016). SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 357–367, New York, NY, USA. ACM.

Scalabrino, S., Linares-Vásquez, M., Oliveto, R., and Poshyvanyk, D. (2018). A Comprehensive Model for Code Readability. *J. Softw. Evol. Process*, 30(6).

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. (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, pages 319–328, Piscataway, NJ, USA. IEEE Press.

Wei, Y., Chandrasekaran, N., Gulwani, S., and Hamadi, Y. (2015). Building Bing Developer Assistant. Technical Report MSR-TR-2015-36, Microsoft Research.

Wightman, D., Ye, Z., Brandt, J., and Vertegaal, R. (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, pages 219–228, New York, NY, USA. ACM.

Xie, T. and Pei, J. (2006). MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57, New York, NY, USA. ACM.