data

MDPI

Article

Employing Source Code Quality Analytics for Enriching Code Snippets Data

Thomas Karanikiotis *, Themistoklis Diamantopoulos * and Andreas Symeonidis

Electrical and Computer Engineering Department, Aristotle University of Thessaloniki, 541 24 Thessaloniki, Greece; symeonid@ece.auth.gr

* Correspondence: karanikio@ece.auth.gr (T.K.); thdiaman@issel.ee.auth.gr (T.D.)

Abstract: The availability of code snippets in online repositories like GitHub has led to an uptick in code reuse, this way further supporting an open-source component-based development paradigm. The likelihood of code reuse rises when the code components or snippets are of high quality, especially in terms of readability, making their integration and upkeep simpler. Toward this direction, we have developed a dataset of code snippets that takes into account both the functional and the quality characteristics of the snippets. The dataset is based on the CodeSearchNet corpus and comprises additional information, including static analysis metrics, code violations, readability assessments, and source code similarity metrics. Thus, using this dataset, both software researchers and practitioners can conveniently find and employ code snippets that satisfy diverse functional needs while also demonstrating excellent readability and maintainability.

Keywords: mining software repositories; source code mining; readability; static analysis metrics; code snippets

check for updates

Citation: Karanikiotis, T.; Diamantopoulos, T.; Symeonidis, A. Employing Source Code Quality Analytics for Enriching Code Snippets Data. *Data* **2023**, *8*, 140. https://doi.org/10.3390/data8090140

Academic Editor: Jamal Jokar Arsanjani

Received: 27 July 2023
Revised: 28August 2023
Accepted: 29 August 2023
Published: 31 August 2023

1. Introduction

Recently, the open-source model has facilitated a substantial surge in code reuse through the availability of code snippets in online repositories, such as GitHub, GitLab, or SourceForge. This approach, sometimes referred to as "component-based reuse" or "opportunistic programming", allows developers to easily find and reuse existing code in their own projects, considerably expediting the development process [1,2]. By leveraging this paradigm, they not only reduce development time and effort but also encourage cooperation and contribution within a vast network of projects. Additionally, the relevant metadata, such as statistics from code hosting, enables easy tracking and analysis of popular open-source projects, assisting in decisions about which projects to use or contribute to.

While code reuse presents significant advantages, it also poses challenges. When practicing reuse, developers must clearly understand the workings of the source code to ensure its proper integration and maintenance. Thus, it becomes paramount to assess the quality of these code components or snippets, especially in terms of readability. A readable code snippet is not only integrated more seamlessly but is also easier to modify, fitting the unique requirements of diverse projects. Moreover, comprehending the code's logic and structure simplifies its long-term maintenance, aiding in the swift identification and rectification of bugs or necessary updates.

There are various systems that mine code snippets and can recommend API calls [3–9] or even reusable code [10–14]. However, these systems primarily focus on the functional aspect of the developer query, answering questions like "how to read a csv file?", and sometimes neglect quality characteristics. More specifically, out of the characteristics defined in the ISO/IEC 25010 for software quality (including, e.g., performance, reliability, security, etc.), maintainability and readability are of paramount importance for code reuse [15]. Maintainability ensures that software can be efficiently modified to address new requirements, fix bugs, or improve

performance, making it adaptable and long-lived [16]. On the other hand, readability pertains to the clarity of the code. Easily understandable code simplifies maintenance tasks, reduces the introduction of bugs, and streamlines the onboarding process for new team members [17]. Emphasizing these two characteristics is pivotal for the sustainable development and evolution of software projects [18]. To facilitate reuse and produce readable and maintainable code, it is important to consider these characteristics when recommending source code snippets.

In this context, we have developed a dataset of code snippets that comprises a comprehensive analysis of both functional and quality characteristics. The dataset is based on the CodeSearchNet [19] corpus, and includes additional information in the form of static analysis metrics and violations, as well as readability assessments of the snippets. Furthermore, considering the functional aspect, our dataset reports the similarity between the code snippets to facilitate the process of finding similar code. The dataset is provided in MongoDB dump format, which facilitates reproducibility and provides advanced querying capabilities. As a result, software researchers and practitioners are enabled to easily retrieve snippets that are relevant to different challenges (e.g., code clone detection [20–23], code snippet recommendation [10–14], code synthesis [24,25], code summarization [26,27]), while ensuring a standard of high quality.

Our work presents a distinct contribution in the space of open-source code repositories and their subsequent reuse. As already mentioned, several approaches mine and recommend code snippets or API calls for reuse purposes [3–14]. However, most of these systems mainly focus on the functional aspects of code reuse, and do not emphasize the imperative non-functional characteristics such as readability and maintainability. This paper's novelty emerges from the synthesis of both functional and quality characteristics in one cohesive dataset. Unlike previous works that may focus predominantly on the functional domain [28,29], we incorporate static analysis metrics, violations, and readability assessments into our dataset. Moreover, by introducing a similarity measure between code snippets, we further facilitate the seamless identification and integration of reusable code, ensuring developers not only find functionally useful code but also high-quality pieces.

The rest of this paper is organized as follows. In Section 2, we delve into the detailed methodology adopted to create, analyze, and curate the dataset of code snippets. Following that, Section 3 provides a comprehensive view of the dataset generated, encompassing its various statistics and shedding light on how it can be used to yield meaningful information. Section 4 explores the numerous ways in which this dataset can be used in answering pertinent research questions, while, lastly, Section 5 encapsulates the primary takeaways from our study and underlines its significance in the larger context.

2. Materials and Methods

Figure 1 depicts the architecture of our platform, which comprises six main components, the Metrics and Violations Analyzer, the Readability Analyzer, the Source Code Parser, the Abstract Syntax Tree Analyzer, the Tree Distance Extractor, and the Hierarchical Clusterer. The source code of our system is available online (https://github.com/AuthEceSoftEng/code-snippets-dataset, accessed on 28 August 2023) to allow for full reproducibility.

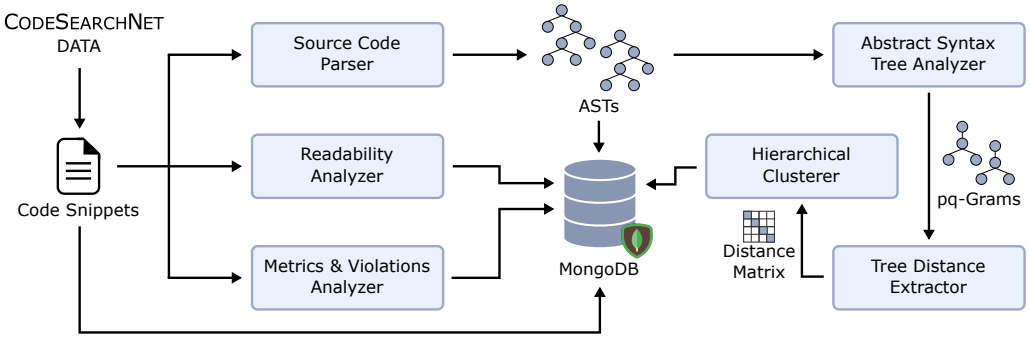

Figure 1. Architecture overview of the system.

As already mentioned, our system uses the data from CodeSearchNet [19], as it is a well-crafted dataset of source code snippets and docstrings. The code snippets are initially analyzed using three different components. The Metrics and Violations Analyzer computes the values of static analysis metrics for each snippet and further detects violations in their source code. The Readability Analyzer is used to extract metrics relevant to code readability. Finally, the Source Code Parser extracts the *Abstract Syntax Trees (ASTs)* of the snippets. All these outputs are stored in MongoDB, along with the source code and the metadata of the snippets. The ASTs extracted from the Source Code Parser are further forwarded to the Abstract Syntax Tree Analyzer that extracts an intermediate representation for each tree (pq-Grams profile, discussed in detail in Section 2.4), which is used by the Tree Distance Extractor to compute the distance between any pair of trees (snippets). The resulting distance matrix is processed by the Hierarchical Clusterer component, which forms clusters and stores them in the database, allowing for easy retrieval of functionally equivalent snippets.

Note that our methodology is mostly language-agnostic, especially regarding the similarity between snippets, since it only requires an appropriate AST extractor from code, while the rest of the components of our architecture remain as they are. However, in this paper we provide a proof-of-concept for components written in the Java programming language due to its suitability for managing large codebases, the availability of robust libraries, and a rich ecosystem that promotes snippet reuse, further underpinning our approach. The components outlined in Figure 1 are analyzed in detail in the following paragraphs.

2.1. Metrics and Violations Analyzer

The first part of our analysis concerns the extraction of useful metrics from the code. To do so, the Metrics and Violations Analyzer employs the SourceMeter analysis tool (https://www.sourcemeter.com/ accessed on 31 August 2023) and extracts a set of different method-level metrics belonging to different categories. These categories correspond to Complexity (with metrics such as McCabe Cyclomatic Complexity, Nesting Level, etc.), Coupling (with metrics for the Number of Incoming/Outgoing Invocations), Documentation (with metrics such as the Comment Density, etc.), and Size (with metrics such as the Lines of Code, etc.). Furthermore, we employ PMD (https://pmd.github.io/ accessed on 31 August 2023) to identify source code violations belonging to different categories (Best Practice Rules, Code Style Rules, Design Rules, Documentation Rules, Error-Prone Rules, Multithreading Rules, Performance Rules, and Security Rules). We also keep track of the number of violations per category of rules as well as the number of violations per priority level (minor, major, or critical).

2.2. Readability Analyzer

The second part of our analysis concerns extracting the readability of source code snippets. The concept of code readability has been studied by several researchers, and different methods have been proposed to assess it. In an effort to provide a holistic view of readability, we extracted different sets of metrics using the tool of Scalabrino et al. [30]. More specifically, the metrics extracted involve (a) the structural metrics identified by Buse and Weimer [31] (including, e.g., the number of identifiers, the number of loops, etc.), (b) the structural metrics defined by Posnett et al. [32] (lines of code, entropy, and Halstead's Volume), (c) the visual metrics of Dorn [33] (e.g., indentation length) (The Dorn metrics also include certain metrics that are relevant to spatial perception or natural language; however, we may also assume that these are relevant to the visual aspect of code.), and (d) the textual metrics of Scalabrino et al. [30] (e.g., Comments and Identifiers Consistency, Textual Coherence, etc.).

2.3. Source Code Parser

To further examine code reuse, we performed source code analysis to extract functionally equivalent snippets. This type of analysis can be combined with code readability to

allow developers to understand and navigate different solutions for a query and select the most readable one. Furthermore, by analyzing the structure and organization of reuse candidate snippets, developers can identify patterns and potential issues, such as code duplication or complex control flow. In this context, we parsed and transformed the source code of the snippets into ASTs, i.e., code representations in tree-like structure, with each node representing a different element of the code, such as, e.g., a function or a variable. To accomplish that, we made use of the ASTExtractor tool (https://github.com/thdiaman/ASTExtractor/ accessed on 31 August 2023). It should be noted that, in our methodology, the source code parser, which translates code snippets into their corresponding ASTs, is the sole component tailored to specific programming languages. Once the code has been transformed into an AST, the next phases of our approach are language-independent, leveraging the generated trees without requiring any modifications.

As an interesting alternative, one could even modify our system to integrate a parser generator, like ANTLR [34] or Bison [35]. This way, it would be possible to parse different languages into ASTs, by providing the generator with the corresponding grammar, and thus achieve language independence.

2.4. Abstract Syntax Tree Analyzer

Given the ASTs of the snippets, the next step is to compute their similarity, which can be easily performed using a *Tree Edit Distance (TED)* algorithm [36]. TED is a method for comparing two trees and measuring their similarity by counting the minimum number of operations (insertions, deletions, and label changes) required to transform one tree into another. Various methods have been developed over the years to improve the complexity of the TED algorithm, but they all have a complexity of $O(n^2)$ or greater. To avoid such computational complexity, we used the pq-Grams algorithm [37] to approximate TED. This algorithm constructs a pq-Extended tree by adding null (dummy) nodes to the tree (noted using the symbol "*"). Specifically, $p-1$ ancestors are added to the root of the tree, $q-1$ children are added before the first and after the last child of each non-leaf node and q children are inserted to each leaf of T. The pseudocode for constructing the extended tree is depicted in Algorithm 1.

Algorithm 1 pq-Grams algorithm pseudocode for building the pq-Extended tree.

```
procedure PQEXTENDEDTREECONSTRUCTION(tree, p, q)
    Add p-1 ancestors to the root of the tree
    For each non-leaf node in tree:
        Add q-1 children before the first child
        Add q-1 children after the last child
    For each leaf node in tree:
        Insert q children
    Return the pq-Extended tree
end procedure
```

The parameters p and q were set to values 2 and 3, respectively. For each pq-Extended tree, we first calculated a list of all the pq-Grams patterns it contains. A pq-Grams pattern is defined as a subtree of the extended tree T that consists of an anchor node with $p-1$ ancestors and q children. Each list of pq-Grams patterns is called a *pq-Grams profile*. Algorithm 2 depicts the pseudocode for calculating the pq-Grams profile given an pq-Extended tree. Therefore, practically, each AST of a code snippet is first extended using the procedure of Algorithm 1, and then the extended tree is given as input into the procedure of Algorithm 2 to build the final pq-Grams profile for the AST.

An example pq-Extended tree construction and extraction of pq-Grams profile for p equal to 2 and q equal to 3 is shown in Figure 2 (initially presented in [38]).

Algorithm 2 pq-Grams algorithm pseudocode for computing pq-Grams profile.

```
procedure COMPUTEPQGRAMSPROFILE(tree, p, q)
    List pqGramsProfile = empty list
    For each node in tree:
        If node has p-1 ancestors and q children:
            Extract pq-Gram pattern consisting of the node, p-1 ancestors, and q children
            Add the extracted pq-Gram pattern to pqGramsProfile
    Return pqGramsProfile
end procedure
```

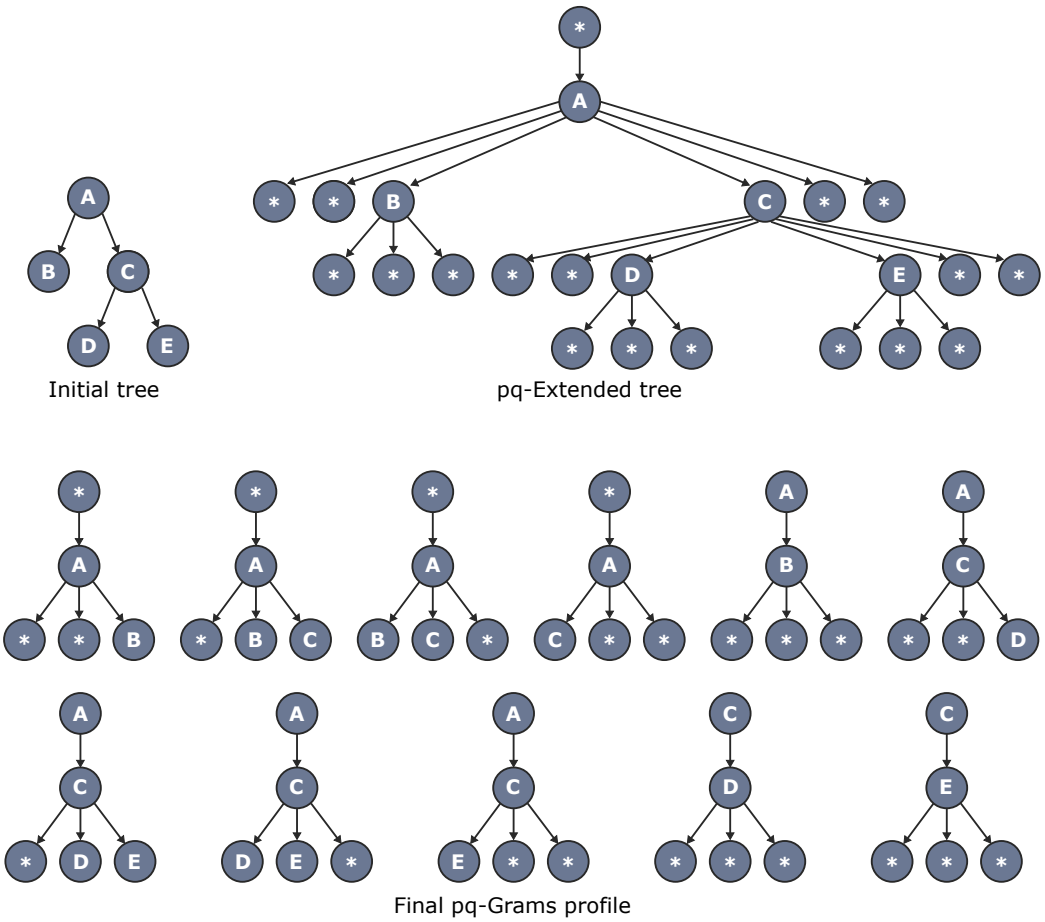

Figure 2. pq-Grams example for $p = 2$ and $q = 3$ [38]. The initial tree includes the nodes A, B, C, D, and E, while "*" represents dummy nodes that are used to produce the pq-Extended tree.

2.5. Tree Distance Extractor

Upon having extracted the pq-Grams profiles, one for each AST/snippet, we are now able to apply the TED metric of the pq-Grams algorithm [37]. Thus, the pq-Grams distance between two trees T_1 and T_2 is defined as follows:

$$distance(T_1, T_2) = 1 - 2 * \frac{|P(T_1) \cap P(T_2)|}{|P(T_1) \cup P(T_2)|} \quad (1)$$

where $P(T_1)$ and $P(T_2)$ are the pq-Grams profiles for trees T_1 and T_2, respectively. As shown by this equation, the distance between two trees depends on the number of mutual pq-Grams patterns contained in both their profiles divided by the union of the two lists, which results in a value between 0 and 0.5 [37]. It is obvious that the similarity between two trees can be easily calculated using the formula $1 - distance(T_1, T_2)$.

2.6. Hierarchical Clusterer

We used Agglomerative Hierarchical Clustering, a bottom-up approach, to group code snippets, based on their distances calculated in the previous step. The optimal number of clusters was determined using average silhouette score, defined for each point i as

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))} \tag{2}$$

where the calculation of the silhouette coefficient uses the mean intracluster distance and the minimum nearest cluster distance. The mean intracluster distance for the data point i is defined as the average distance of i to all other points in the same cluster as i (C_i)

$$a(i) = \frac{1}{|C_i| - 1} \sum_{\substack{j \in C_i \\ j \neq i}} d(i, j) \tag{3}$$

The minimum nearest cluster distance is defined as the average distance from the points of the nearest cluster of data point i, except for cluster C_i:

$$b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i, j) \tag{4}$$

After performing the cluster analysis, we examined the generated clusters (maximum silhouette equal to 0.78) and maintained the clusters that contained a minimum number of data points (100 snippets) to ensure that the snippet implementations are generic enough. Listings 1 and 2 depict two snippets that have been put together in the same cluster. Both snippets check for null values and call the *.get()* function on an object.

Listing 1. Example snippet that is in the same cluster with the snippet of Listing 2.

```
public static EarthEllipsoid getType(String name){
  if (name == null)
      return null;
  return hash.get(name);
}
```

Listing 2. Example snippet that is in the same cluster with the snippet of Listing 1.

```
public Object getUserProperty(Object key){
   if (userMap == null)
      return null;
   return userMap.get(key);
}
```

3. Results

Figure 3 depicts the schema of our database, which comprises four different collections, the *snippets*, the *analysisMetrics*, the *violations*, and the *readabilityMetrics*. The data are available as a MongoDB database (in *.bson* format, which is a binary-encoded serialization of JSON documents).

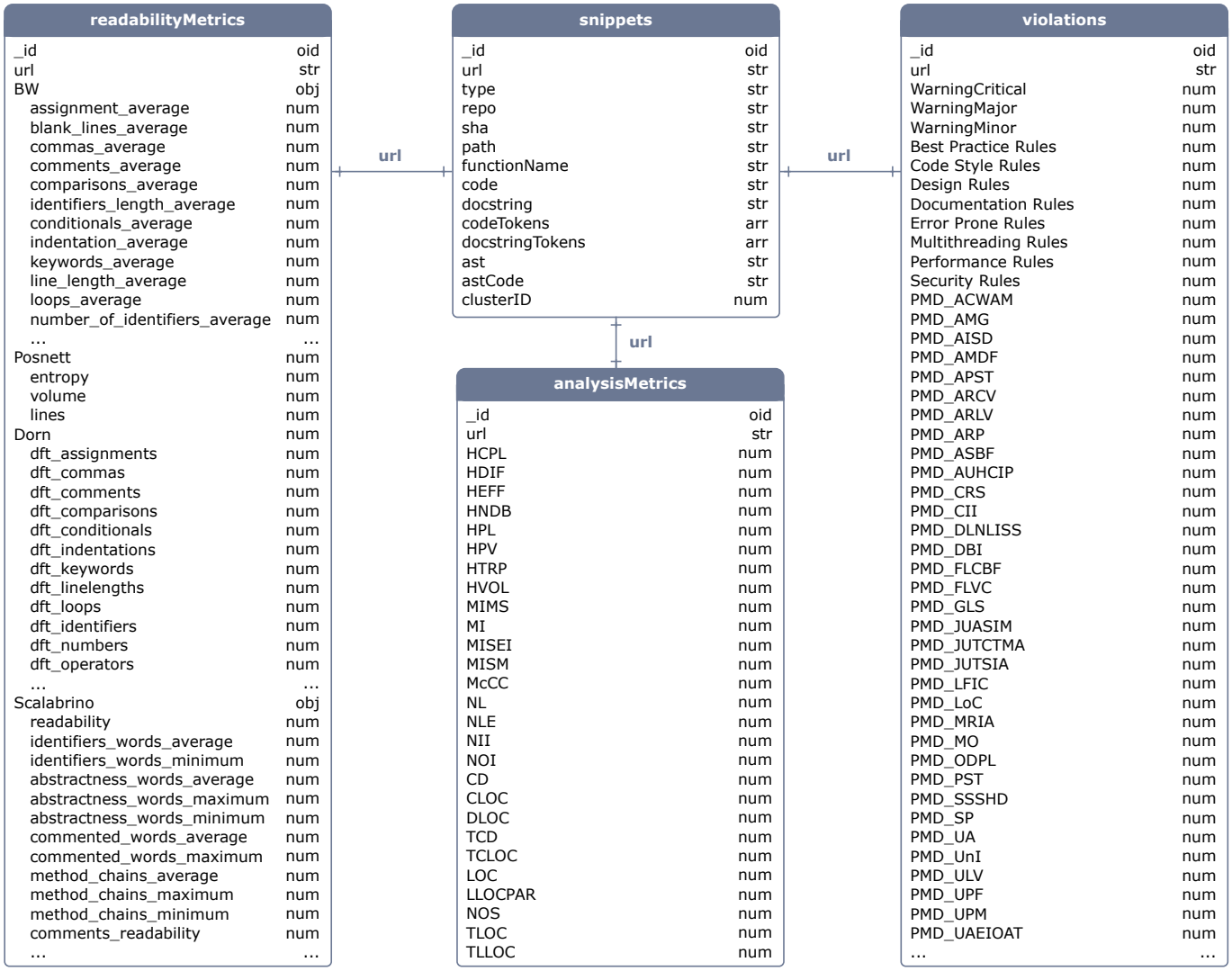

Figure 3. The schema of our database.

Our decision to provide the dataset in MongoDB dump format was based on the inherent advantages of MongoDB for handling large-scale datasets. MongoDB, as a NoSQL database, offers flexibility in storing diverse data types and is renowned for its scalability and performance. The dump format ensures that the entire database, including its structure and contents, can be easily shared, restored, and queried, promoting reproducibility. The capabilities of MongoDB in managing and querying vast datasets are well-documented in the database and big data literature [39].

It should be noted that all the collections use "url" as a "primary key" to identify the code snippet they refer to. As already mentioned, our dataset consists of four collections:

- *Snippets*: This collection contains the code and the docstring of each snippet, information about its origin, the AST and the id of the cluster it belongs to, which can be used to group snippets into clusters (i.e., similar snippets).
- *AnalysisMetrics*: This collection includes the static analysis metrics calculated by the SourceMeter analysis tool.
- *Violations*: This collection contains the source code violations identified by the PMD tool.
- *ReadabilityMetrics*: This collection includes the extracted readability metrics, which are split into four categories, with respect to the research approach they refer to (Buse and Weimer—BW, Posnett, Dorn, and Scalabrino).

Table 1 depicts certain statistics about the collected dataset. The statistics presented have been chosen to provide a comprehensive overview of the scale and organization

of the dataset. They practically indicate the range of code samples in our dataset, their distribution across different projects, and how often certain patterns or functionalities are repeated.

Table 1. Dataset statistics.

Metric	Value
Number of Documents/Snippets	496,685
Number of Repositories	500
Number of Clusters	893
Data Size	11.3 GB (940.3 MB compressed)

Additionally, Listing 3 illustrates a sample query that can be used for data extraction purposes; it retrieves the static analysis metrics and the readability metrics of the cluster of Listings 1 and 2. Queries such as this can be issued from multiple environments. For example, one could use MongoDB Compass (https://www.mongodb.com/products/compass accessed on 31 August 2023) to explore the data or form and issue queries. And, of course, it is also possible to issue requests using a programming language, e.g., for Python, one can use the pymongo library (https://pymongo.readthedocs.io/en/stable/ accessed on 31 August 2023).

Listing 3. Example query that retrieves static analysis metrics and readability metrics of a cluster.

```
db.getCollection("snippets").aggregate([
    {
        "$match": {
            "clusterID": 1
        }
    },
    {
        "$lookup": {
            "from": "analysismetrics",
            "localField": "url",
            "foreignField": "url",
            "as": "satmetrics"
        }
    },
    {
        "$lookup": {
            "from": "readabilitymetrics",
            "localField": "url",
            "foreignField": "url",
            "as": "readmetrics"
        }
    }
])
```

Thus, using our dataset, one can extract useful insight for the relations between metrics. For instance, given the query of Listing 3, we can plot the diagram of Figure 4, depicting the Scalabrino readability value versus Comment Density (CD). If we further highlight their relation using linear regression (dashed line in diagram), we see that the two metrics are positively correlated, which is expected if we consider that comments typically enhance the readability of the code.

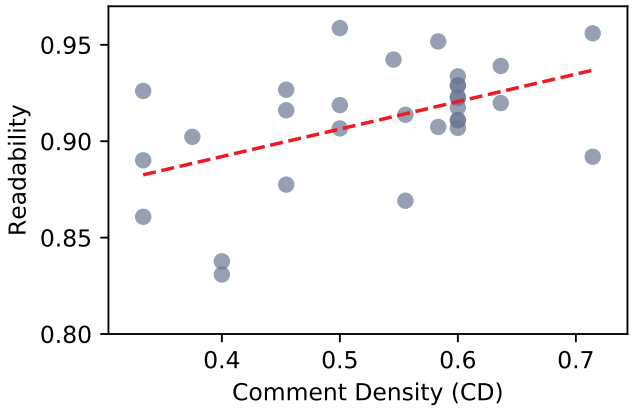

Figure 4. Example plot depicting the Scalabrino Readability metric versus Comment Density (CD) for a cluster.

4. Discussion

Concerning the structure and the design of the dataset, retrieving and reproducing it is straightforward, as it only requires a MongoDB instance. As already mentioned, MongoDB ensures scalability and offers advanced storing, retrieving, and querying capabilities; thus, it is ideal for managing the varied and dynamic nature of source code snippets and the associated metrics [39]. Our dataset can be used to confront several challenges in current research. First of all, in examining source code readability, researchers can use our dataset to empirically study the various aspects that contribute to or detract from code understandability. Consider a real-world scenario wherein a developer has to choose between multiple similar implementations of a function or algorithm. With our dataset, one could identify similar code snippets and run user experiments. For example, comparing the readability of iterative versus recursive implementations of a factorial function can help determine which version is deemed more readable by a broader audience. Moreover, the data facilitate the design and development of readability models, either by using the source code directly [40,41] or the associated static analysis metrics [42–44]. When doing so, comparison with the state of the art is straightforward, as our dataset includes different metrics for readability [30–33].

The similarity data in our dataset make it an invaluable resource for code clone detection [20–22]. For instance, researchers are able to detect whether two different open-source projects inadvertently employ the same code implementation, indicating potential for reuse. By utilizing the tokens and ASTs of the provided snippets, researchers can compare different clone detection methods. Furthermore, our dataset paves the way for studies investigating correlations between code clones and metrics like static analysis or readability. For instance, a pertinent question may be whether code clones inherently suffer from reduced readability due to repeated patterns. Finally, one could even try to determine whether it is possible to identify code clones based on these metrics [23].

Concerning code similarity challenges, consider the potential application in code search engine scenarios. For example, when a developer submits a textual query looking for a "quicksort algorithm implementation", our dataset can not only help in retrieving relevant snippets, [10–14,19,45], but it can also rank them based on their readability or quality score [46,47]. Moreover, it can be used for pattern matching tasks, by detecting whether certain snippets follow nano-patterns, study how these nano-snippets are implemented, and even examine their readability [48].

Venturing into the realm of code synthesis [24,25], imagine an automated tool that can take a cluster of similar code snippets and synthesize a new version by amalgamating the best parts of each snippet. For instance, if the cluster contains various ways to implement a database connection [49], the tool could create a single, clean, and readable version that incorporates the most efficient and readable elements from the cluster. Lastly, our

dataset offers a fertile ground for advancements in code summarization [26,27]. With the burgeoning complexity of today's software projects, being able to automatically summarize a lengthy function into a few descriptive lines or even comments can significantly expedite code reviews and maintenance.

All in all, our dataset can be useful for confronting several research challenges. Considering its limitations, and thus identifying possible future research directions, we may note that the data include method snippets, while complete project information is omitted. As a result, the dataset cannot be used for challenges at the project and/or library level (e.g., extracting the semantics of software libraries and using them for code comprehension [50]). Given also that the data are only drawn from GitHub, they may not always include generic solutions for all possible developer queries. As future work, we plan to link the method snippets to their original source repositories, which would offer interesting metadata, and even integrate other sources, such as Stack Overflow, which includes snippets that are generally regarded as high-quality code [51].

5. Conclusions

In this work, we have aimed to improve the research and practice on code reuse and readability by creating a dataset of code snippets that involves both functional and quality characteristics. The dataset is based on the CodeSearchNet corpus and includes additional information in the form of static analysis metrics and violations, readability assessments, and code similarity metrics. Thus, using our dataset, one can confront different research challenges in the areas of code reuse/readability, as well as combine these areas with the aim of optimizing the process of identifying, integrating, and maintaining reusable code. Future work could include incorporating additional representations of the code, such as control flow graphs, and even augmenting the dataset, e.g., by using also data from code-hosting repositories.

Author Contributions: Conceptualization, T.K., T.D. and A.S.; methodology, T.K., T.D. and A.S.; software, T.K., T.D. and A.S.; validation, T.K., T.D. and A.S.; formal analysis, T.K., T.D. and A.S.; investigation, T.K., T.D. and A.S.; resources, T.K., T.D. and A.S.; data curation, T.K., T.D. and A.S.; writing—original draft preparation, T.K., T.D. and A.S.; writing—review and editing, T.K., T.D. and A.S.; visualization, T.K., T.D. and A.S.; supervision, T.K., T.D. and A.S.; project administration, T.K., T.D. and A.S.; funding acquisition, T.K., T.D. and A.S. All authors have read and agreed to the published version of the manuscript.

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

Data Availability Statement: The data presented in this study are openly available in Zenodo at https://zenodo.org/record/7893288.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Crnkovic, I.; Larssom, M. Challenges of Component-Based Development. *J. Syst. Softw.* **2002**, *61*, 201–212. [CrossRef]
2. Brandt, J.; Guo, P.J.; Lewenstein, J.; Klemmer, S.R. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In Proceedings of the 4th International Workshop on End-User Software Engineering, New York, NY, USA, 10–18 May 2008; pp. 1–5.
3. Nguyen, T.; Rigby, P.C.; Nguyen, A.T.; Karanfil, M.; Nguyen, T.N. 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, New York, NY, USA, 13–18 November 2016; pp. 1013–1017.
4. Xu, C.; Sun, X.; Li, B.; Lu, X.; Guo, H. MULAPI: Improving API method recommendation with API usage location. *J. Syst. Softw.* **2018**, *142*, 195–205. [CrossRef]
5. Nguyen, P.T.; Di Rocco, J.; Di Ruscio, D.; Ochoa, L.; Degueule, T.; Di Penta, M. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In Proceedings of the 41st International Conference on Software Engineering, IEEE Press, Montréal, QC, Canada, 25–31 May 2019; pp. 1050–1060.
6. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API Learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 13–18 November 2016; pp. 631–642.

7. Cai, L.; Wang, H.; Huang, Q.; Xia, X.; Xing, Z.; Lo, D. BIKER: A Tool for Bi-Information Source Based API Method Recommendation. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 3–9 December 2019; pp. 1075–1079.
8. Li, X.; Jiang, H.; Kamei, Y.; Chen, X. Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. *IEEE Trans. Softw. Eng.* **2018**, *46*, 1–17. [CrossRef]
9. Chen, C.; Peng, X.; Sun, J.; Xing, Z.; Wang, X.; Zhao, Y.; Zhang, H.; Zhao, W. Generative API Usage Code Recommendation with Parameter Concretization. *Sci. China Inf. Sci.* **2019**, *62*, 192103. [CrossRef]
10. Ponzanelli, L.; Bacchelli, A.; Lanza, M. Seahawk: Stack Overflow in the IDE. In Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, 18–26 May 2013; pp. 1295–1298.
11. Campbell, B.A.; Treude, C. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution, Los Alamitos, CA, USA, 17–24 September 2017; pp. 628–632.
12. Diamantopoulos, T.; Oikonomou, N.; Symeonidis, A. Extracting Semantics from Question-Answering Services for Snippet Reuse. In Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering, Dublin, Ireland, 25–30 April 2020; pp. 119–139.
13. Gu, X.; Zhang, H.; Kim, S. Deep Code Search. In Proceedings of the 40th International Conference on Software Engineering, New York, NY, USA, 26–27 May 2018; pp. 933–944.
14. Papathomas, E.; Diamantopoulos, T.; Symeonidis, A. Semantic Code Search in Software Repositories using Neural Machine Translation. In Proceedings of the 25th International Conference on Fundamental Approaches to Software Engineering, Munich, Germany, 2–7 April 2022; pp. 225–244.
15. ISO/IEC 25010:2011. 2011. Available online: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (accessed on 28 August 2023).
16. Spinellis, D. *Code Quality: The Open Source Perspective*; Adobe Press: San Jose, CA, USA, 2006.
17. Sedano, T. Code Readability Testing, an Empirical Study. In Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET), 5–6 April 2016; pp. 111–117.
18. Pfleeger, S.L.; Atlee, J.M. *Software Engineering: Theory and Practice*; Pearson Education India: Noida, India, 1998.
19. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv* **2019**, arXiv:1909.09436.
20. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
21. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 19–27 May 2007; pp. 96–105.
22. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep Learning Code Fragments for Code Clone Detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 3–7 September 2016; pp. 87–98.
23. Aktas, M.S.; Kapdan, M. Structural Code Clone Detection Methodology Using Software Metrics. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26*, 307–332. [CrossRef]
24. Terragni, V.; Liu, Y.; Cheung, S.C. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In Proceedings of the 25th International Symposium on Software Testing and Analysis, New York, NY, USA, 18–20 July 2016; pp. 118–129.
25. Raghothaman, M.; Wei, Y.; Hamadi, Y. SWIM: Synthesizing What i Mean: Code Search and Idiomatic Snippet Synthesis. In Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA, 18–20 May 2016; pp. 357–367.
26. Haiduc, S.; Aponte, J.; Marcus, A. Supporting Program Comprehension with Source Code Summarization. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2, New York, NY, USA, 1–8 May 2010; pp. 223–226.
27. Katirtzis, N.; Diamantopoulos, T.; Sutton, C. Summarizing Software API Usage Examples using Clustering Techniques. In Proceedings of the 21th International Conference on Fundamental Approaches to Software Engineering, Thessaloniki, Greece, 14–21 April 2018; pp. 189–206.
28. Janjic, W.; Hummel, O.; Schumacher, M.; Atkinson, C. An Unabridged Source Code Dataset for Research in Software Reuse. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 339–342.
29. Gelman, B.; Obayomi, B.; Moore, J.; Slater, D. Source code analysis dataset. *Data Brief* **2019**, *27*, 104712. [CrossRef] [PubMed]
30. Scalabrino, S.; Linares Vasquez, M.; Oliveto, R.; Poshyvanyk, D. A Comprehensive Model for Code Readability. *J. Softw. Evol. Process* **2018**, *30*, e1958. [CrossRef]
31. Buse, R.P.L.; Weimer, W.R. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* **2010**, *36*, 546–558. [CrossRef]
32. Posnett, D.; Hindle, A.; Devanbu, P. A Simpler Model of Software Readability. In Proceedings of the 8th Working Conference on Mining Software Repositories, New York, NY, USA, 21–22 May 2011; pp. 73–82.
33. Dorn, J. A General Software Readability Model. Master's Thesis, The University of Virginia, Charlottesville, VA, USA, 2012.
34. Parr, T.J.; Quong, R.W. ANTLR: A Predicated-LL(k) Parser Generator. *Softw. Pract. Exper.* **1995**, *25*, 789–810. [CrossRef]
35. Donnelly, C.; Stallman, R. *Bison: The Yacc-Compatible Parser Generator*; Free Software Foundation: Boston, MA, USA, 2015.
36. Tai, K.C. The Tree-to-Tree Correction Problem. *J. ACM* **1979**, *26*, 422–433. [CrossRef]

37. Augsten, N.; Böhlen, M.; Gamper, J. The Pq-Gram Distance between Ordered Labeled Trees. *ACM Trans. Database Syst.* **2008**, *35*, 1–36. [CrossRef]
38. Diamantopoulos, T.; Symeonidis, A. Localizing Software Bugs using the Edit Distance of Call Traces. *Int. J. Adv. Softw.* **2014**, *7*, 277–288.
39. Parker, Z.; Poe, S.; Vrbsky, S. Comparing nosql mongodb to an sql db. In Proceedings of the 51st ACM Southeast Conference, Savannah, Georgia, 4–6 April 2013; pp. 1–6.
40. Mi, Q.; Keung, J.; Xiao, Y.; Mensah, S.; Gao, Y. Improving code readability classification using convolutional neural networks. *Inf. Softw. Technol.* **2018**, *104*, 60–71. [CrossRef]
41. Choi, S.; Kim, S.; Kim, J.; Park, S. Metric and Tool Support for Instant Feedback of Source Code Readability. *Inf. Softw. Technol.* **2020**, *15*, 221–228.
42. Karanikiotis, T.; Papamichail, M.D.; Gonidelis, I.; Karatza, D.; Symeonidis, A.L. A Data-driven Methodology towards Interpreting Readability against Software Properties. In Proceedings of the 15th International Conference on Software Technologies, Held Online, 7–9 July 2020; pp. 61–72.
43. Fakhoury, S.; Roy, D.; Hassan, S.A.; Arnaoudova, V. Improving Source Code Readability: Theory and Practice. In Proceedings of the 27th International Conference on Program Comprehension, Montreal, QC, Canada, 25–26 May 2019; pp. 2–12.
44. Roy, D.; Fakhoury, S.; Lee, J.; Arnaoudova, V. A Model to Detect Readability Improvements in Incremental Changes. In Proceedings of the 28th International Conference on Program Comprehension, New York, NY, USA, 13–15 July 2020; pp. 25–36.
45. Papoudakis, A.; Karanikiotis, T.; Symeonidis, A. A Mechanism for Automatically Extracting Reusable and Maintainable Code Idioms from Software Repositories. In Proceedings of the 17th International Conference on Software Technologies (ICSOFT), Lisbon, Portugal, 11–13 July 2022; pp. 79–90.
46. Diamantopoulos, T.; Thomopoulos, K.; Symeonidis, A.L. QualBoa: Reusability-aware Recommendations of Source Code Components. In Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, USA, 14–16 May 2016; pp. 488–491.
47. Michailoudis, A.; Diamantopoulos, T.; Symeonidis, A. Towards Readability-aware Recommendations of Source Code Snippets. In Proceedings of the 18th International Conference on Software Technologies (ICSOFT), Rome, Italy, 10–12 July 2023; pp. 688–695.
48. Gil, Y.; Marcovitch, O.; Orrú, M. A Nano-Pattern Language for Java. *J. Comput. Lang.* **2019**, *54*, 100905. [CrossRef]
49. Diamantopoulos, T.; Karagiannopoulos, G.; Symeonidis, A. CodeCatch: Extracting Source Code Snippets from Online Sources. In Proceedings of the IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 21–27.
50. Kuhn, A.; Ducasse, S.; Gírba, T. Semantic Clustering: Identifying Topics in Source Code. *Inf. Softw. Technol.* **2007**, *49*, 230–243. [CrossRef]
51. Sillito, J.; Maurer, F.; Nasehi, S.M.; Burns, C. What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow. In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 25–34.