symmetry

MDPI

Article

Identifying High-Level Concept Clones in Software Programs Using Method's Descriptive Documentation

Aditi Gupta and Rinkaj Goyal *

University School of Information, Communication and Technology,
Guru Gobind Singh (GGS) Indraprastha University, New Delhi 110078, India;
aditi.phd84.usict2016@ipu.ac.in
* Correspondence: rinkaj@ipu.ac.in or rinkajgoyal@gmail.com

Abstract: Software clones are code fragments with similar or nearly similar functionality or structures. These clones are introduced in a project either accidentally or deliberately during software development or maintenance process. The presence of clones poses a significant threat to the maintenance of software systems and is on the top of the list of code smell types. Clones can be simple (fine-grained) or high-level (coarse-grained), depending on the chosen granularity of code for the clone detection. Simple clones are generally viewed at the lines/statements level, whereas high-level clones have granularity as a block, method, class, or file. High-level clones are said to be composed of multiple simple clones. This study aims to detect high-level conceptual code clones (having granularity as java methods) in java-based projects, which is extendable to the projects developed in other languages as well. Conceptual code clones are the ones implementing a similar higher-level abstraction such as an Abstract Data Type (ADT) list. Based on the assumption that "similar documentation implies similar methods", the proposed mechanism uses "documentation" associated with methods to identify method-level concept clones. As complete documentation does not contribute to the method's semantics, we extracted only the description part of the method's documentation, which led to two benefits: increased efficiency and reduced text corpus size. Further, we used Latent Semantic Indexing (LSI) with different combinations of weight and similarity measures to identify similar descriptions in the text corpus. To show the efficacy of the proposed approach, we validated it using three java open source systems of sufficient length. The findings suggest that the proposed mechanism can detect methods implementing similar high-level concepts with improved *recall* values.

Keywords: clone detection; high-level concept clones; information retrieval; Latent Semantic Indexing; Part-of-Speech tagging

Citation: Gupta, A.; Goyal, R. Identifying High-Level Concept Clones in Software Programs Using Method's Descriptive Documentation. *Symmetry* **2021**, *13*, 447. https://doi.org/10.3390/sym13030447

Academic Editor: José Carlos R. Alcantud

Received: 7 February 2021
Accepted: 1 March 2021
Published: 10 March 2021

1. Introduction

Cloning of program fragments in any form has been identified both as a boon and bane depending upon the purpose for which it is used [1–7]. Writing code from scratch often hinders code understandability and evolvability. In such situations, reusability achieved via the use of already existing libraries, design patterns, frameworks, and templates aids structuring code well, while on the other hand, code clones formed as a result of the programmer's laziness can turn out to be harmful in the long term. These clones adversely affect software maintenance as even the smallest change made in a code fragment may have to be propagated to all the clones of that fragment; failing may lead to inconsistent code, which may result in software bugs [8–11]. Clones can also arise out of language limitations or external business forces.

Depending on the extent of similarity between code clones, they can be classified as *exact or type-1 clones*. They are the result of copy paste activity of programmers, when they find a piece of code providing exact solution to their problem. To suit their needs, programmers may further modify the pasted code leading to Type-2 or parameterized

clones (modification of identifier names) and Type-3 or near miss clones (modification in control structures, deletion or addition of statements). Further clones can also occur when the developer re-invents the wheel. These types of clones generally occur when the developer is unaware of the existing solution to the problem or wants to develop a more efficient algorithm. These clones can be termed as wide miss clones. A wide miss clone solves the same or similar problem, while having structural dissimilarities. From the experience of marcus and maletic [12], these type of clones often manifest themselves as higher-level abstractions in the problem or solution domain such as an Abstract Data Type (ADT) list. They can be termed as "concept clones". Further, on the basis of chosen granularity of code fragments (forming the basic units of clone detection), clones can be simple (fined grained) or high-level (coarse-grained). Simple clones are generally composed of few set of statements, while high level clones are clones having granularity as a block, method, class, or file. High-level clones are said to be composed of multiple simple clones. High-level clones can further be classified into behavior clones (clones having same set of input and output variables and similar functionality), concept clones (clones implementing a similar higher-level abstraction), structural clones (clones having similar structures), and domain-model clones (clones having similar domain models such as class diagram) [13]. This study aims at the detection of "high-level concept clones having granularity as java methods" in a software system.

High-level concept clones are those code fragments that implement a similar underlying concept. Their detection depends on the user's understanding of the system. Using the code's internal documentation and program's semantics could be a viable mechanism for their detection [13]. Detecting such clones is useful in selecting an efficient implementation of higher-level abstraction and propagating changes to all clones whenever underlying concept changes.

Different algorithms have been proposed in prior published studies to detect clones in source code. The algorithms can be text-based (using source code implementation or associated text, like, comments and identifiers directly or by converting to some intermediate form) [12,14–19], token-based (using source code converted to meaningful tokens) [20–29], tree-based (using parse trees) [30–35], metrics based (using metrics derived from source code) [36–41], or graph-based (using source code converted to suitable graph) [42–46].

As the use of the code's internal documentation and program semantics is recommended for detection of concept clones, several studies use text-based Information Retrieval (IR) techniques. The IR techniques enable us to extract relevant information from a collection of resources [47]. There are several techniques for information retrieval, such as Latent Semantic Indexing (LSI), Vector Space Modeling, Latent Dirichlet Allocation (LDA), etc. Researchers have used these techniques for clone detection by extracting source code text from software programs. Marcus and Maletic [12] conducted one of the earlier studies that took source code text (comments and identifiers) to identify similar high-level concept clones. They applied LSI on the text corpus formed from extracted comments and identifiers to identify such clones. Later, the authors extended this technique to locate a given concept in the source code [48] and then to determine conceptual cohesion between classes in object-oriented programs [49]. Bauer et al. [50] also compared the effectiveness of LSI with the TF-IDF approach and regarded the former as the better alternative.

One of the drawbacks of LSI is the production of a large number of false positives [51], making it difficult for developers to identify actual concept clones. Poshyvanyk et al. [52] combined LSI with Formal Concept Analysis (FCA) to reduce this difficulty. FCA was used for presenting the generated ranked results with a labeled concept lattice, thereby reducing the efforts of programmers to locate similar concepts during software maintenance. In some of the recent studies, researchers have highlighted the potential use of information retrieval with learning-based approaches to detect type-4 clones. Xie et al. [53] used the TF-IDF approach with the Siamese neural network, while Fu et al. [54] combined a Continuous Bag of Words (CBOW)/Skip Gram (SG) model with ensemble learning and reported improved accuracy measures as compared to other solo approaches. Some researchers have used LSI

to further examine the clones, which have already been detected by some clone detection tools. The findings suggest that clone detection and LSI complement each other and give better quality results [18,55]. Table 1 summarizes some of the relevant studies related to the use of information retrieval techniques in clone detection.

Table 1. Existing works using text-based Information Retrieval techniques to identify clones in software artifacts.

Author	Year	Technique Applied	Text Corpus Content	Description
Marcus and Maletic [12]	2001	LSI + CS	Comments and Identifiers	Identify similar high level concepts
Marcus and Maletic [48]	2004	LSI + CS	Comments and Identifiers	Locate a given concept in source code
Marcus and Maletic [49]	2005	LSI + CS	Comments and Identifiers	Identify conceptual cohesion between classes in object oriented programs
Kuhn et al. [56]	2007	LSI + ST + TF-IDF + CS	Identifiers	LSI was used for topic modeling of source code for detecting source artifacts that use similar vocabulary
Tairas and Grey [55]	2009	LSI + ST + CS	Identifiers	LSI improves the results obtained by using a clone detection tool (here CCFinder)
Grant and Cordy [57]	2009	ICA + BN + CS	Non-unique method tokens	Similar files are clustered
Bauer et al. [18]	2016	LSI + L(TF) + CS	Identifiers	LSI improves the results obtained by using a clone detection tool (here Conqat)
Ghosh and Kuttal [58]	2018	LDA + SWR	Comments	Achieved better precision and recall compared to PDG-based techniques
Reddivari and Khan [59]	2019	LDA + SWR + ST	Source code after identifier splitting	Achieved better results when compared to CCFinder [60] and CloneDr [61]
Abid [62]	2019	Lucene (IR engine)	Comments, keywords from function names, and API names used in function	Based on the user query, the technique finds similar functions that implement identical features using the extracted text elements.
Kuttal and Ghosh [63]	2020	LDA + SWR + TF	Comments excluding task and copyright comments	Emphasized the usage of comments for clone detection. Hybrid of both LDA (on comments) and PDG-based (on source code) approach is presented with improved accuracy metrics values.
Xie et al. [53]	2020	TF-IDF + Cosine	Source code text converted into tokens	The technique combines TF-IDF and deep learning-based (word embeddings) approaches to identify semantic clones.
Fu et al. [54]	2020	CBOW + SG	Source code text converted into tokens	The technique combines SG, CBOW word embedding model, and ensemble learning to find semantic clones. It is shown to perform well compared to other token-based and deep learning-based clone detectors.

IR: Information Retrieval, LSI: Latent Semantic Indexing, LDA: Latent Dirichlet Allocation, ICA: Independent Component Analysis, BN: Binary, TF-IDF: Term Frequency-Inverse Document Frequency, L(TF): Log (Term Frequency), TF: Term Frequency, CS: Cosine Similarity, ED: Euclidean Distance, ST: Stemming, SWR: Stop Words Removal, API: Application Program Interface, PDG: Program Dependence Graph.

Kuhn et al. [56] also illustrated the use of LSI for topic modeling of source code vocabulary. Later, Maskeri et al. [64] used a more efficient approach called Latent Dirichlet Allocation (LDA) to identify similar topics. Ghosh and Kuttal applied LDA on a text corpus formed from a combination of comments and associated source code to identify semantic clones [58]. It gave better *precision* and *recall* values as compared to PDG-based approaches. In their recent study, they emphasized the usage of source code comments to detect clones. They performed clone detection by employing a PDG-based approach to source code and LDA to source code comments (excluding task and copyright comments) for finding file level clones [63]. Their study empirically shows the relevance of good quality comments to scale such IR-based techniques well for larger projects and detecting inter-project clones. Reddivari and Khan also developed CloneTM, which uses LDA to identify code clones (up to type-3) using source code (with identifier splitting, stop-word removal, and stemming). CloneTM gave better results when compared to CCFinder [60] and CloneDr [61]. Abid [62] developed Feature-driven API usage-based Code Examples Recommender (FACER) to detect and report similar functions based on the user's query.

The tool uses comments, keywords from the function name, API class names, and API method names used in functions to find similar functions using an IR-based engine called Lucene. At the data level, data clones (replicas) can help recover lost data at runtime for quality assurance of data analysis tasks in the big data analysis [65].

The literature also provides other similar techniques for clone detection. Grant and Cordy [57] replaced LSI with a feature extraction technique, namely, Independent Component Analysis (ICA), and used tokens in the program to build a vector space model instead of terms in the comments.

Several previously published studies [12,48,49,58,63] apply IR-based techniques using source code comments (including other programming constructs) to identify clones, thereby resulting in a bulky text corpus. This study, however, omits all the unnecessary comments and uses only comments present in the method's documentation. Blending suitable NLP techniques with descriptive documentation only (excluding notes comment, explanatory comment, contextual comment, evolutionary comment, and conditional comment) makes the proposed mechanism lightweight and efficient.

In this study, high-level concept clones at method-level granularity are detected by extracting descriptive documentation of each method to form the text corpus. The descriptive part of documentation generally describes the semantics of code, thus fulfilling the criteria for detection of concept clones as stated above [13]. Latent Semantic Indexing (LSI) coupled with Part-of-Speech (POS) tagging strategy and lemmatization is then applied on the formed text corpus to identify high-level concept clones. After obtaining similar descriptions, the methods corresponding to these descriptions are regarded as concept clones.

The followings points summarize the procedural flow and factors contributing to the efficacy of the proposed approach:

1. The text corpus is formed only from the description of methods (other documentation details are removed). This is because the semantics of a particular method are mostly contained in its description. All other comments augmented with methods are also not used. Furthermore, no additional text element of the source code is used for text corpus preparation. All this makes the comparison process efficient with the reduced size of the text corpus.
2. A POS tagging strategy has been employed to assign higher weights to certain parts of speech (generally nouns and verbs) and for the removal of stop-word, filtering out all terms not tagged as nouns, adverbs, verbs, and adjectives. The use of POS tagging also reduces the size of text corpus and increases efficiency by combining two tasks in one step. Other works carry out stop-word removal as a separate task.
3. Different combinations of weight and similarity measures have been examined for clone detection. These combinations are compared based on several parameters to enable the end user to choose suitable variant. In Section 6.3, each combination is ranked based on their efficiency to find contextual clones.
4. The proposed mechanism is able to detect methods implementing similar high-level concepts with good accuracy values. The mechanism is empirically validated using 3 open source java projects (JGrapht, Collections, and JHotDraw) as subject systems.

The organization of the rest of the paper is as follows. Section 2 explains the relevant terminologies used in this study. Section 3 lays down the research questions relevant to this study. Section 4 lists the key features of documentation in java projects. Section 5 delineates the clone detection process. Section 6 outlines the empirical validation process used and addresses the listed research questions. Section 7 outlines threats to the validity of results derived in this work. The conclusion is presented in Section 8 of this exposition.

2. Terminology

This section briefly outlines the relevant terms used in this study.

2.1. High-Level Concept Clones

High-level concept clones [12,13] are defined as implementations of a similar higher-level abstraction like an ADT list. They arise when developers try to re-invent the wheel, i.e., implementing the same or similar concept using different algorithms. These clones address the same or similar problem, while having very dissimilar structures. Their detection depends on the user's understanding of the system. Using the code's internal documentation and program semantics can greatly improve detection accuracy of this type of clone.

In this research work, we have used descriptive documentation associated with methods (as it contributes to the method's semantics) to detect high-level concept clones having granularity as java methods.

2.2. Latent Semantic Indexing (LSI)

LSI [51] is an information retrieval technique used for automating the process of semantic understanding of sentences written in a particular language. It uses Singular Value Decomposition (SVD) of the term-document matrix developed using the Vector Space Model (VSM) to find the relationship among documents.

To perform LSI on a given set of documents, a term-document matrix is built by extracting terms from these documents. Every entry of the matrix indicates the weight assigned to a particular term in the corresponding document. This term-document matrix is decomposed using SVD. It decomposes the matrix (say A with dimension m,n) into three matrices as follows,

$$A = USV^T \tag{1}$$

where the dimension of U is (m,m), S (diagonal matrix) is (m,n), and V is (n,n). A random value K is chosen to form the matrix A_k (a kth approximation of matrix A). A_k is derived as

$$A_k = U(m,k).S(k,k).V(k,n) \tag{2}$$

After A_k is successfully computed, given a document, its similarity with other documents is calculated. The measure of similarity can be cosine, jaccard, dice, etc. The result of applying LSI is a list of similar documents. A schematic representation of LSI is given in Figure 1.

LSI usually finds its usage in applications like search engines to extract the most relevant documents for a given query. It can also be used for document clustering in text analysis, recommendation systems, building user profiles, etc.

2.3. Part-of-Speech (POS) Tagging

POS tagging [66] is defined as the process of mapping words in a text to their corresponding Part-of-Speech tag. The tag is assigned based on both its definition and context (relationship with other words in the sentence)*. For example, consider two sentences say "S1 = Sameera likes to eat ice cream" and "S2 = There are many varieties of ice cream like butterscotch, Kesar-Pista, etc.". Look at the word *like* in both sentences. The tag assigned to "like" changes according to the sentence: in S1 *like* is a verb, while in S2 *like* is a preposition.

POS tagging uses a predefined tag-set, such as the penn treebank tagset; transition probabilities; and word distribution. The tag-set contains a list of tags to be assigned to different words in the corpus. Transition probabilities define the probabilities of transition from one tag to another in a sentence. The word distribution denotes the probability that a particular word may be assigned to a specific tag.

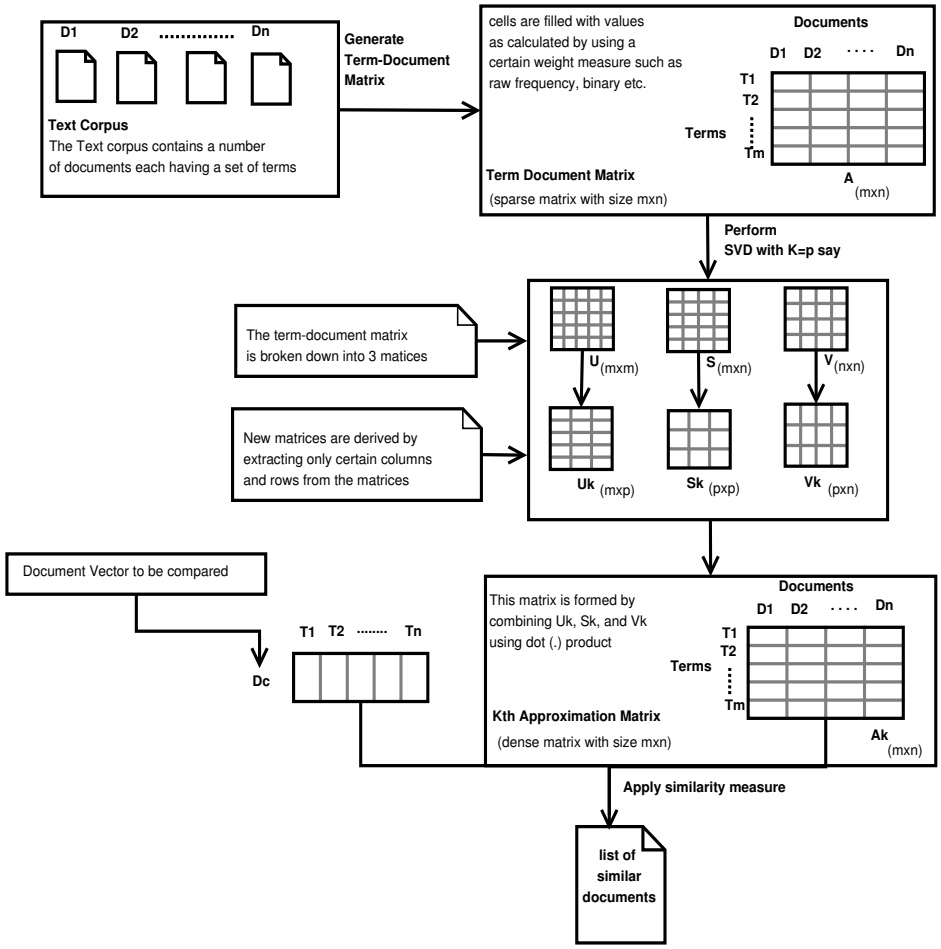

Figure 1. The illustration of working of Latent Semantic Indexing (LSI). LSI is used in this study to find similarity between documentation of methods.

Applications of POS tagging are named entity recognition (recognition of the longest chain of proper nouns), sentiment analysis, word sense disambiguation, etc.

While comparing two sentences it is sometimes required that different parts of speech be weighed according to the semantic information it conveys in a particular context, which can be done using POS tagging. In this paper, we have used POS-Tagging to give extra weight-age to nouns and verbs.

In prior studies, D. Falessi et al. [67] used this technique to find equivalent requirements. By comparing different combinations of Natural Language Processing (NLP) techniques required for building algebraic model, doing term extraction, assigning weights to each term in documents, and measuring similarity among documents, they concluded that the best outcome is observed for a combination of VSM, raw weights, extraction of relevant terms (verbs and nouns only), and cosine similarity measure. They also did an empirical validation of their results using a case study [68]. Johan [69,70] used similar NLP techniques to find related or duplicate requirements coming from different stakeholders to avoid reworking on a similar requirement.

2.4. Lemmatization

Lemmatization refers to the process of reducing different forms of a word to its root form [71]. For example, "builds", "building", or "built" gets reduced to the lemma "build".

The purpose of lemmatization is to gain homogeneity among different documents, though, another mechanism, stemming, can also be used. Stemming works by removing suffixes from different forms of a word, e.g., "studies" becomes "study", "studying" becomes "study", etc. As is evident from this example, lemmatization reduces words to a

word present in the dictionary, while stemming removes suffixes without caring whether the reduced form is available in the dictionary or not.

Stemming/lemmatization strategies have several applications in natural language processing. In the software engineering field, these are generally used to understand and analyze textual artifacts of the software development process, such as documentation provided at each level, e.g., requirement document, design document, code documentation, etc. Falessi et al. [67,68] employed stemming along with POS tagging as a word extraction strategy to find equivalent requirements.

2.5. Term Weighting

While generating a term-document matrix, each entry carries some weight according to its occurrence in the documents. These weights can be assigned in several ways as discussed below.

Raw frequency: Weights are assigned according to the number of times they appear in a particular document.

Binary: Weight value is one if a term occurs in the document; otherwise, its value would be zero.

Term Frequency (TF): Raw frequency divided by the length of the document. This helps prevent giving more preference to longer text because longer text tends to contain a particular term at a higher frequency than the shorter ones.

Inverse Document Frequency (IDF): IDF is computed as the inverse of the number of times a term occurs in all the documents. IDF score can be useful to give less importance to the terms that frequently occur in all documents, thereby making it semantically less useful for the document under review.

TF-IDF: Weight for a particular term in a document is computed by combining its TF and IDF values. This mechanism leverages the benefits of both TF and IDF scores.

Word Embeddings: Word embeddings are a way to represent words in the text corpus using distributed representations of words instead of a one-hot encoding vector. Using these vector representations, one can explore the semantic and syntactic similarity of words, the context of words, etc. in the document. Various word embedding techniques exist:

(i) *Word2Vec*: Word2Vec is a feedforward neural network with two variants. It either accepts two or more context words as input and returns the target word as output, or vice versa. The former is called the *Continuous Bag of Words (CBOW)* model and the latter is the *Skip-Gram (SG)* model.

(ii) *Glove*: Glove works by generating a high-dimensional vector representation of each word. The training is based on the surrounding words. This high-dimensional matrix (context matrix) is then reduced to a lower-dimension matrix such that maximum variance is preserved. Glove uses pretrained models that fail for text containing new words. The documentation of methods may contain many new words (such as method names), therefore *glove* was found unsuitable for this study.

(iii) *Fasttext* is similar to *glove*, with a feature that makes it suitable for random words not in the pretrained model. It breaks the word into n-grams and then generates the required context-matrix, for example, given the word "fruits" and n = 3. The n-grams generated are <fru, rui, uit, its>. Each row in the matrix represents the generated n-grams instead of whole words.

2.6. Similarity Measures

The two text documents are compared for their similarity using established similarity metrics. As this study uses the term-document matrix, we are only interested in vector similarity measures. There are three commonly used vector similarity metrics: cosine, jaccard, and dice. For two vectors, say X and Y, the three metrics are calculated as follows:

$$Dice(x, y) = \frac{2|x \cap y|}{|x| + |y|} \tag{3}$$

$$Jaccard(x, y) = \frac{|x \cap y|}{|x| + |y| - |x \cap y|} \tag{4}$$

$$Cosine(x, y) = \frac{|x \cap y|}{\sqrt{|x| \, |y|}} \tag{5}$$

2.7. Thresholds Used

Certain thresholds are used for the proper execution of the proposed mechanism:

(i) *#Method_Tokens*: This threshold is used to filter out small methods. Methods with count of tokens greater than a certain threshold are only considered in the comparison process. This helps in getting more relevant results.

(ii) *#Simterms*: #Simterms define the count of similar terms obtained by textual comparison of each pair of documentation to keep track of ordering of terms. This threshold allows only those documents to be compared which have a similar ordering of terms greater than the threshold value.

(iii) *Simmeasure*: Simmeasure defines the similarity value obtained by using similarity measures given in Section 2.6 (i.e., cosine, jaccard, or dice similarity) between the pair of documentations. This threshold ensures that the resulting clone pairs have similarity greater than the threshold value.

(iv) *K*: The value used for arriving at the kth approximation of the term-document matrix. It is used when performing LSI to ensure that maximum variance is preserved when reducing the high-dimensional matrix to a lower dimension.

2.8. Precision and Recall

In this study, *recall* refers to the ratio of the number of items that are known to be true clones and are detected successfully (number of true positives say x) to the total number of true clones say y.

$$Recall = \frac{x}{y} \tag{6}$$

Here, *precision* refers to the ratio of the number of true positives in the detected clones, say p, set to the total number of clones detected in the detected clones set, say q.

$$Precision = \frac{p}{q} \tag{7}$$

3. Research Questions

The different aspects of clone detection, which have been analyzed in this study, are abstracted into the following research questions.

- **RQ1:** *How is documentation present along with source code in a software system useful for finding concept clones?* Through this question, our aim is to investigate the effectiveness of using documentation in finding method-level concept clone pairs. As this study revolves around the fact that "similar documentation implies similar methods", it is also recommended to use internal documentation for high-level concept clone detection. Therefore, it is necessary to examine the structure of documentation followed in the software projects. Here, we do an in-depth study of documentation structures followed in java-based projects in general and validate that similar structures can also be seen in the case studies used for the empirical validation of the proposed mechanism. We then probed over the ideal documentation style of methods for the technique to work credibly without manual checking required both for java-based projects and for projects implemented in some other language. Further, we investigated the use of information retrieval techniques particularly Latent Semantic Indexing (LSI) on the text corpus formed from extracted documentation.

- **RQ2:** *What are the implications of applying different combinations of weight and similarity measures for performing LSI on the extracted method's documentation?* While applying LSI to the text corpus formed from the extracted documentation, we observed the result sets obtained using various combinations of weight and similarity measures. This question investigates which combinations gave better results and assigns a ranking to each of the combinations.
- **RQ3:** *Is the technique scalable, i.e., applicable to large-sized projects?* This research question aims to examine the scalability of the proposed mechanism. As the proposed mechanism is centered around analyzing documentation of methods, its scalability also correlates to the uniformity of style and availability of documentation.

4. Key Features of Java Documentation

The documentation accompanying the source code in a software system describes the functionality of code fragments; this motivated us to use them in the clone detection process. This is based on the assumption that "similar documentation imply similar code fragment". In this study, the code fragment is taken to be the method in the java projects.

In a java-based project, documentation contains the following parts:

(i) **Description**: Contains description of the code fragment.
(ii) **Descriptive Tags**: Several other details are presented in this part of documentation prefixed with certain tags such as @param, @see, @author, @throws, @override, @return, etc.

Here, we extract only documentation associated with methods to be used as input to LSI. Contained in the following are the relevant viewpoints of using documentation of methods for clone detection.

(i) It is not compulsory for the documentation to contain all its parts (description and descriptive tags). It can contain any or both of the parts depending upon requirements. The description of methods is extracted using both parts as described in Section 5.
(ii) Not all methods are documented in java projects. The absence of documentation can be credited to one of the following reasons:
    (a) Two or more related methods may be documented collectively. This leads to documentation being associated with only one method, while all other methods are seen undocumented by our technique.
    (b) Some methods perform a sub-function of a task. The programmer may only document the method performing the whole task, leaving the methods performing sub-functions undocumented.
    (c) Some methods are seen to be undocumented for unknown reasons, maybe the laziness of the programmer leads to these methods being left undocumented.

    We manually injected documentation for undocumented methods based on the rough understanding of methods so that they can be made suitable for input as clone candidates for our technique.
(iii) The description part may also contain other types of comments, which are as follows (the explanation of each type is derived from Howard [72]):
    (a) *Notes Comment*: Notes are used to communicate pending tasks to either the developer or the team, and sometimes used to refer to other parts of the code.
    (b) *Explanatory Comment*: Explanatory comments explain the rationale behind some part of the code.
    (c) *Contextual Comment*: These comments provide information about the context of the method, e.g., the method is "called from X", "calls Y", or "called when."
    (d) *Evolutionary Comment*: Evolutionary comments describe information about the history of the method.
    (e) *Conditional Comment*: Conditional comments specify conditions under which this method should be called.

These comments do not contribute towards the semantics of methods and need to be filtered out. Here, we also used manual ways to filter out these comments. This helps to increase the efficiency of the algorithm as the size of the text corpus used as input to LSI is reduced.

(iv) It may be possible that the whole description of the method does not contain any descriptive comments. These methods may result in false negatives. It can be considered similar to undocumented methods as they do not appear in results to form clone pairs with other methods.

5. Clone Detection Process

This section briefly describes the clone detection process used in this paper. We have taken three open source software written in java to validate our findings empirically. The steps carried out in the process are as follows.

Step 1: *Filtering out Interfaces, Abstract Methods, Small Methods, and Constructors:* Programs written in java contain a large number of interfaces and methods that need to be filtered out because they do not contribute to relevant clones. This helps in increasing the performance of the proposed mechanism by reducing the number of spurious clones and makes the overall analysis of resulting clone pairs less time-consuming (see Algorithms 1 and 2). The following programming constructs shall be filtered out:

(i) *Interfaces*: An interface contains only the names of methods without any implementation. Classes that implement the interface need to implement the required functionality for each of the methods. As interfaces do not contain any implementation, they must be filtered out as they do not contribute much to the detection of relevant clones.

(ii) *Abstract Methods*: Abstract methods are similar to the methods in interfaces and do not contain any implementation. Thus, they are also filtered out.

(iii) *Small Methods*: Methods that are smaller than a particular threshold, i.e., a specific number of tokens, are also filtered out. This threshold value is a hyperparameter. This filtering helps to remove various setter and getter methods and methods that only call other methods to perform the required functionality. These are generally single line methods whose detection as clones serves the least purpose.

(iv) *Constructors*: In this study, constructors are also not taken into account for clone detection, thus they are also filtered out.

Algorithm 1: Filter out small methods.

```
1  largeMethods ⟵ ϕ;
2  filterSmallMethods(projectDirectory)
3  paths ⟵ dirExplore(projectDirectory);
   /* paths contain a list of all the paths in the project directory */
4  javaFiles ⟵ paths.endsWith(.java);
   /* All java files are extracted */
5  while javaFile ∈ javaFiles do
       /* This loop works by comparing the size of methods with a given threshold and
          add them to the list of large methods */
6      nodes ⟵ javaFile.getAllNodes;
7      while node ∈ nodes do
8          if node instanceOf ClassOrInterfaceDeclaration then
9              methods ⟵ node.methods();
10             if methods.size() >= threshold then
11                 largeMethods.add(method);
12 return largeMethods
```

Algorithm 2: Filter out methods that least contribute to clones in software systems.

```
1 filterMethods(sourceCode)
2 methods ⟵ sourceCode.methods;
  /* Extract methods from source code                                  */
3 filtered_methods ⟵ filterSmallMethods(methods);
  /* Filter out small methods using a threshold, need parsing of code  */
4 methodDocs ⟵ filtered_methods.methodDocs;
5 while methodDoc ∈ methodDocs do
6     if methodDoc ∉ methodDoc.interfaceMethods and
       methodDoc ∉ methodDoc.constructors and
       methodDoc ∉ methodDoc.abstractMethods then
          /* Exclude interface's methods, abstract methods and constructors from
             comparison                                                */
7         filteredMethodDocs.add(methodDoc);
8 return filteredMethodDocs
```

Step 2: *Extraction of Documentation*: In this work, only documentation related to methods is used, as described in Section 4. Furthermore, we need to extract only the *description* associated with methods (see Algorithm 3). If the description is not present, the proposed method uses descriptive tags to extract a suitable description for the method using @return and @see tags.

(i) *@return tag* describes what is returned from the function. The contents of the @return tag are used when the description is absent.
(ii) *@see tag* is used when a lookup to another method/class is required (no description for the method and no @return tag is present).

Overridden methods are usually annotated with @override annotation. If documentation is not available for these methods, it is extracted from the method being overridden.

Further, to extract java documentation in a processable file format, we used built-in java library *doclet* (by default javadocs are generated as HTML pages). Steps 1 and 2 collectively describe the key features our generated javadoc should have.

Step 3: *Part-of-Speech (POS) Tagging of Documentation and Lemmatization:*

For each method's extracted documentation, POS tagging is performed (see Algorithm 5 (Step 5)). The purpose of this step is twofold:

(i) To give more weight to nouns and verbs in a sentence.
(ii) To remove stop words by extracting only words tagged as verbs, adjectives, nouns, and adverbs (see Algorithm 5 (Step 7)).

Lemmatization is done to reduce each term in documentation to its base form, to curb differences among documentation arising out of different forms of words (see Algorithm 5 (Step 6)).

Step 4: *Latent Semantic Indexing (LSI):* LSI (see Algorithm 5 (steps 10–15)) is performed as follows:

(i) Extract those terms tagged as verbs, nouns, adjectives, or adverbs.
(ii) Perform LSI on these extracted terms. The VSM built contains weights (matrix entries), as described in Section 2.5. Weights can also be assigned by giving extra weight-age to certain parts of speech (nouns and verbs).
(iii) For a pair of method's description, use similarity measures described in Section 2.6, also taking into account similarity based on term ordering to find clone pairs in a software project (see Algorithm 4).

For the complete algorithm, refer to Algorithms 1 to 5. The flowchart representation of this algorithm is given in Figure 2.

Algorithm 3: Extract comments from program's source code.

```
1  extractComment(methodDoc) /* Extract method description from method
       documentation                                                    */
2  desc ⟵ methodDoc.commentText /* commentText is the description of method
       without any tags. The actual semantic description of method is often contained
       in the commentText                                               */
3  if length(desc) > 0 then
       /* do nothing                                                   */
4  else if length(methodDoc.returnTag) > 0 then
5      desc ⟵ returnTag.commentText;
       /* If actual semantic description is not present, then we try to extract comment
          corresponding to return tag of method documentation          */
6  else if length(methodDoc.seeTag) > 0 then
7      desc ⟵ seeTag.commentText;
       /* If both the actual semantic description and return tag are not present then,
          we try to extract the corresponding comment to see the tag of method
          documentation which itself gives reference to the documentation of the other
          method                                                       */
8  else if methodDoc.isOverriden and methodDoc.isAnnotated(@Overriden) then
9      desc ⟵ OverridenSuperClassMethod.desc;
       /* Import overridden method description from the method it overrides */
10 else
11     desc ⟵ null;
12 return desc
```

Algorithm 4: Find similarity between documents.

```
1  similarityMeasure(doc_to_cmp, kth_approx_TD_Matrix)
2  clones ⟵ ϕ;
3  terms1 ⟵ doc_to_cmp.terms;
4  while column_doc ∈ kth_approx_TD_Matrix do
5      terms2 ⟵ column_doc;
6      if similarityInOrder(terms1, terms2) > 30% then
           /* Similarity in order means terms in two docs are compared in word
              similarity and order similarity                          */
7          if Similarity(term1, terms2) > 50% then
               /* Similarity can be measured using cosine, dice, or jaccard similarity
                  measures (refer to Section 2.6)                      */
8              clones.add(doc_to_cmp, column_doc);
9  return clones
```

Algorithm 5: Using POS tagging and LSI to find similar methods.

Input: Source Code
Output: Similar Method Pairs

```
descs ⟵ "" /* Find similar method pairs given source code of software */
filteredMethodDocs ⟵ filterMethods(sourceCode)
/* Filter out methods that least contributes to clones in software program, i.e.,
   interfaces, abstract methods, constructors, and small methods */
while methodDoc ∈ filteredMethodsDocs do
    desc ⟵ extractComment(methodDoc)
    t_desc ⟵ desc.POS_Tagged
    /* Tag each term in the documentation corpus with their corresponding POS tag */
    lt_desc ⟵ t_desc.Lemmatize
    /* Each term is reduced to its basic form */
    desc ⟵ lt_desc.removeStopWords
    /* Extract only nouns, adverbs, verbs, and adjectives from the description after
       POS tagging and lemmatization */
    descs ⟵ descs + desc
end
/* Execute LSI on these comments */
descs.buildTDMatrix
/* Assign weights to each extry based on word's POS tag (optional) and using one of
   the weight measures as in Section 2.5 */
TD_Matrix.singularValueDecomposition
Derive kth approximation of TD_Matrix
clone_pairs ⟵ similarityMeasure(doc_to_cmp, kth_approx_TD_Matrix)
print(clone_pairs)
```

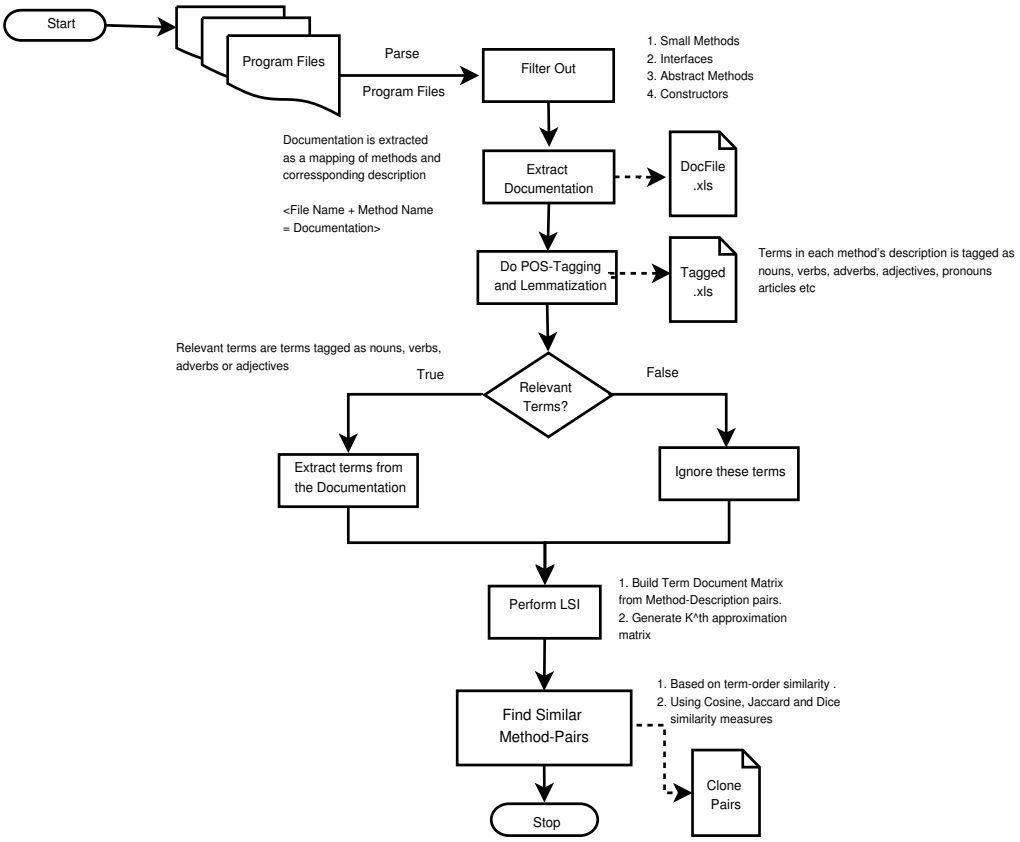

Figure 2. Flowchart representation of proposed mechanism to find method-level concept clones.

6. Empirical Validation

To empirically validate the proposed mechanism, it has been applied to three open source java projects, namely, JGrapht [73], Collections [74], and JHotDraw [75]. The results obtained were validated against the benchmark Qualitas Corpus (QCCC) [76], which was provided by Ewan Tempero [77] to present the efficacy of the proposed approach. The researcher prepared a corpus of clones for 111 open source java projects. However, the corpus only presents type-1, type-2, and type-3 clones, while our mechanism reports high-level concept clones. The validation is justified because type-1, type-2, and type-3 clones are very similar in structure and mostly implement similar concepts. The recall values obtained on validation also indicate that concept clones generally cover most clone pairs falling in these three categories. Nevertheless, the manual validation of resulting clone pairs is necessary for *accuracy* calculations.

Table 2 gives details of projects used for the empirical validation. The details include #Files analyzed, #Methods (excluding constructors, abstract methods, methods of interfaces, and small methods), Estimated Lines of Code (ELOC; excluding constructors, abstract methods, methods of interfaces, and small methods), and % of Documented Methods (i.e., % of methods documented before manual modification (excluding constructors, abstract methods, and methods of interfaces)), which shows that with increase in size, the number of documented methods decreases.

Table 2. Details about projects used in the empirical validation of mechanism.

Project Name	#Files	#Methods	ELOC (Approx.)	% of Documented Methods (Approx.)
JGrapht	175	467	6K	80%
Collections	273	1427	10k	70%
JHotDraw	589	2996	45k	53%

ELOC: Estimated Lines of Code.

After applying POS tagging, lemmatization, and stop-word removal to the description of methods, different weighing measures were used to generate term-document matrix, and thereafter to obtain a list of clone pairs, different similarity measures (to measure similarity between several term-document vectors) were used after performing LSI. Table 3 summaries a list of all the combinations of weight and similarity measures used in this study. Furthermore, with each combination POS tagging is used in two ways, i.e., with or without selective weight assignment (giving priority to certain parts of speech). Selective weight assignment cannot be used with binary weight measures because it can only hold values 1 and 0.

Table 3. Combinations of weight and similarity measures used.

S.No.	POS Weightage	Weight Measure	Similarity Measure	Acronym
1	No	Raw Frequency	Cosine	RCWOPOS
2		Binary	Cosine	BCWOPOS
3		TF	Cosine	TFCWOPOS
4		Binary	Jaccard	BJWOPOS
5		Binary	Dice	BDWOPOS
6		TF-IDF	Cosine	TFIDFCWOPOS
7		FASTTEXT	Cosine	FASTTEXT
8		CBOW	Cosine	CBOW
9		SG	Cosine	SG
10	Yes	Raw Frequency	Cosine	RCWPOS
11		TF	Cosine	TFCWPOS
12		TFIDF	Cosine	TFIDFCWPOS

POS: Part of Speech, TF: Term Frequency, TF-IDF: Term Frequency-Inverse Document Frequency, CBOW: Continuous Bag of Words model, SG: Skip-Gram model.

Owing to the small size (ELOC is approx 6K), the JGrapht project has been used to illustrate the underlying working of the proposed mechanism. The *recall* is measured by a comparison of results obtained using our mechanism with the one provided in

QCCC. *Precision* calculations are made for individual methods through augmenting manual validation of resulting clone pairs.

6.1. Relevance of Jaccard and Dice Similarity Measures with LSI

The reason behind using only binary weight measures with jaccard and dice similarity measures can be explained with the help of a counterexample.

The jaccard similarity measure is calculated as given in Equation (4). Therefore, for two vectors with raw frequency weights say x = {1,2,1,0,1,2} and y = {1,2,2,1,0,1}, then Jaccard (x,y) = 2/5 = 0.4. It will treat (1,2) and (0,1) as different numbers and consider these documents as not similar w.r.t. these terms. While, in reality, methods with vector representation x and y are highly similar, as these numbers indicate the frequency of occurrence of terms in the method's descriptions being compared. A similar example can be quoted for TF weight measure and non-applicability of LSI. Furthermore, the Dice similarity measure, as given in Equation (3), can be used only with binary weight measure and cannot be combined with LSI, based on a similar explanation, as indicated above.

6.2. Results and Discussion

The results of empirical validation of each of the case studies are presented here.

6.2.1. Empirical Validation Using JGrapht

JGraphT is a free Java class library that provides mathematical graph-theory objects and algorithms. It runs on java 2 platform (requires JDK 1.8 or later starting with JGraphT 1.0.0).

This subsection provides execution details of the proposed mechanism on JGrapht. The thresholds (see Section 2.7) used for arriving at results are as follows:

(i) *#Method_Tokens*: Small methods are filtered out using a minimum threshold of 50 on the number of tokens in a method (Evans used the same threshold on the number of Abstract Syntax Tree (AST) nodes in a method [77]).
(ii) *K*: For LSI, the value of *K* is taken as half of the total number of terms in the text corpus.
(iii) *Simmeasure* is taken as 0.5, i.e., documentation of resulting method-level clone pairs should be 50% similar.
(iv) #*Simterms* is set to 30%, i.e., documentation of resulting method-level clone pairs should be at least 30% similar in term ordering.

Table 4 shows the results obtained after applying each of the 12 combinations of weight and similarity measures to JGrapht. Column 1 gives the type of combination applied (see Table 3). Column 2 enlists the number of clone pairs observed for each applied combination. Column 3 gives the intersection of clone pairs present in both benchmark QCCC (provided by Evans [77]) and the proposed mechanism. The number of clone pairs reported for JGrapht in QCCC were 85 methods with 6 constructors (constructors have already been filtered out in this study because of their limited relevance). Column 4 gives *recall*, which is measured by comparing the obtained results against the benchmark. Last, column 5 gives *precision* metric values, which are obtained by manual analysis of resulting clone pairs.

The results are then compared (see Figure 3) to analyze the relative variation of the number of true positive clone pairs with similarity scores. Similarity scores can be Cosine, Dice, or Jaccard. An ideal plot should exhibit a large proportion of clone pairs for higher similarity values (say for similarity score > 0.7). The following observations can be made while analyzing clone pairs identified for JGrapht.

(i) A large number of clone pairs observed for each combination is attributed to the basic nature of LSI. LSI finds clone pairs by measuring the similarity between terms. In two sentences, terms can be similar, but the overall meaning of a sentence may be different, e.g., when a query search is performed on a search engine, a large number

of results are obtained with ranking given to each result. This is also the reason for less values of *precision* in the result set (see Column 5 of Table 4).

(ii) Maximum *recall* is achieved using combination number 4 (binary weight measure and jaccard similarity measure), and maximum *precision* is achieved using combination number 12 (TF-IDF weight measure and cosine similarity measure). Both of them gave better values of *precision* and *recall* metrics.

(iii) While comparing "Similarity Score" with the count of true positive clone pairs in the result set, we observed combinations 1(RCWOPOS), 2(BCWOPOS), 5(BDWOPOS), 3(TFCWOPOS), 8(CBOW), and 9(SG) as having an ideal property, as a large proportion of true positive clone pairs are encountered for higher similarity values. The rest of the combinations show a somewhat higher count of clone pairs for low values of similarity score. Combination 4(BJWOPOS) show the worst results as the high number of true positives are observed for similarity score as low as 0.3.

(iv) The graphs and tables suggested a trivial advantage of using POS selective weight-age. However, it resulted in increased *precision* values but with a decline in the *recall* values.

(v) Of all the combinations, combination 1(RCWOPOS) is preferred as it gave good values of precision and recall; it also exhibits defined idle graph property.

Table 4. Results of applying combinations 1–12 on JGrapht.

Combination Type	#Clone Pairs	#Clone Pairs Matched with Benchmark	Recall (%)	Precision (% Rough Estimate)
RCWOPOS	1599	73	85.88	60
BCWOPOS	1326	72	84.70	57
TFCWOPOS	1561	73	85.88	57
BJWOPOS	1458	74	87.05	70
BDWOPOS	1016	70	82.35	67
TFIDFCWOPOS	740	68	80.00	75
FASTTEXT	1901	72	84.70	57
CBOW	3348	71	83.52	50
SG	1847	73	84.70	55
RCWPOS	765	67	78.82	73
TFCWPOS	762	66	77.64	72
TFIDFCWPOS	573	66	77.64	80

Refer to Table 3 for all the combination types listed here.

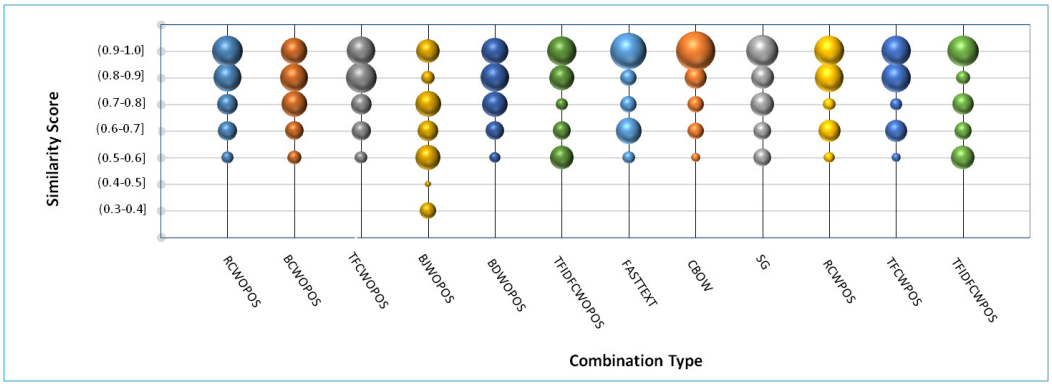

Figure 3. Comparison of count of true positive clone pairs with similarity values for JGrapht case study (refer to Table 3 for all the 12 combinations listed).

6.2.2. Empirical Validation Using Collections and JHotDraw

This section gives complete details of the execution of the proposed mechanism on Collections and JHotDraw open source software. Each of the 12 combinations of weight and similarity measures are applied to each open source java project. Small methods are filtered out using a minimum threshold of 50 on the number of tokens in the methods for both projects.

The results of the application of each of the 12 combinations on the two open source projects are given in Table 5 (for Collections case study) and Table 6 (for JHotDraw case study).

Only *recall* is measured for results obtained for Collections and JHotDraw due to time constraints. As the *precision* calculated is a manual and time-intensive process for such a large result set of clone pairs obtained, it was performed for JGrapht to show the relevance of the proposed mechanism. The number of clone pairs reported in the benchmark for Collections is 1037 methods and 3 constructors, and for JHotDraw is 2550 methods and 199 constructors.

Table 5. Results of applying combinations 1–12 on Collections.

Combination Type	#Clone Pairs	#Clone Pairs Matched with Benchmark	Recall (%)
RCWOPOS	23,586	784	75.60
BCWOPOS	21,922	784	75.60
TFCWOPOS	23,614	785	75.69
BJWOPOS	22,462	789	76.08
BDWOPOS	15,394	737	71.07
TFIDFCWOPOS	10,870	687	66.24
FASTTEXT	34,522	760	73.28
CBOW	25,683	774	74.63
SG	19,325	725	69.91
RCWPOS	12,632	678	65.38
TFCWPOS	12,646	678	65.38
TFIDFCWPOS	7478	601	57.95

Refer to Table 3 for all the combination types listed here.

Table 6. Results of applying combinations 1–12 on JHotDraw.

Combination Type	#Clone Pairs	#Clone Pairs Matched with Benchmark	Recall (%)
RCWOPOS	29,154	1851	72.58
BCWOPOS	28,698	1863	73.05
TFCWOPOS	29,116	1853	72.66
BJWOPOS	28,527	1865	73.13
BDWOPOS	20,510	1824	71.52
TFIDFCWOPOS	18,778	1788	70.11
FASTTEXT	48,106	1798	70.50
CBOW	49,235	1868	73.25
SG	36,283	1861	72.98
RCWPOS	22,428	1792	70.27
TFCWPOS	22,363	1795	70.39
TFIDFCWPOS	16,131	1697	66.54

Refer to Table 3 for all the combination types listed here.

A comparison similar to JGrapht (between similarity scores and count of true positive clone pairs) is also made for the Collections and JHotDraw case study (see Figures 4 and 5). The following observations can be made from these graphs and tables.

(i) For the Collections case study, combinations 1(RCWOPOS), 2(BCWOPOS), 3(TFC-WOPOS), 5(BDWOPOS), 7(FASTTEXT), 8(CBOW), 10(RCWPOS), and 11(TFCWPOS) show ideal curve property. Here, also similar to JGrapht, combination 4(BJWOPOS) shows the worst results with a significant number of true positives being observed for lower similarity scores. Here, combinations 1(RCWOPOS), 2(BCWOPOS), and 3(TFCWOPOS) can also be seen to show good results both in terms of recall and concentration of clone pairs.

(ii) For the JHotDraw case study, all the combinations showed a similar behavior: all of these satisfy the defined ideal property (except combination 4). Here, combination 4(BJWOPOS) also gave the worst results for similar reasons as mentioned above. The highest recall values are shown by combination 2, 4 and 8. Here, combinations 1, 3 and 9 also have good recall values with only 1 % decrease in recall.

We use a Windows 10 machine with Intel(R) Core(TM) i5-6200U CPU having 2.7 GHz frequency for the three case studies. The execution time increases exponentially with the number of documents (documents contain methods linked with its documentation).

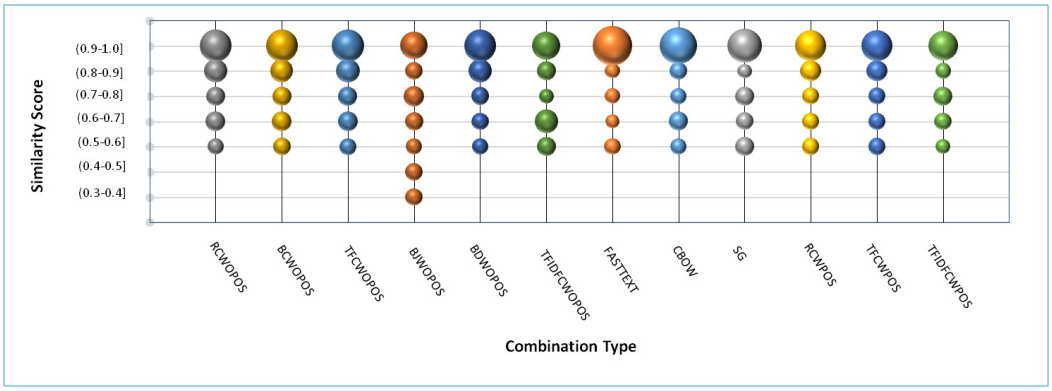

Figure 4. Comparison of count of true positive clone pairs with similarity values for Collections case study (refer to Table 3 for all the 12 combinations listed).

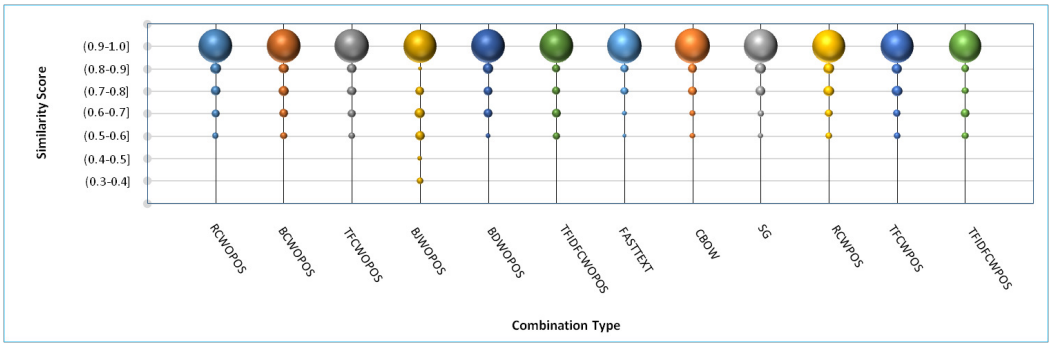

Figure 5. Comparison of count of true positive clone pairs with similarity values for JHotDraw Case Study (refer to Table 3 for all the 12 combinations listed).

6.3. Exploring the Research Questions

RQ1: How is documentation present along with source code in a software system useful for finding concept clones?

The result of the three case studies exhibited good recall values using the proposed mechanism, thereby showing the effectiveness of considering the method's documentation for clone detection purposes. This shows the strength of the LSI mechanism to link semantically related documents together. Therefore, using documentation present along with source code is beneficial for high-level concept clone detection.

Java projects follow a certain structure and style while building the documentation as given in Section 4. We saw a similar structure in the case studies used. Table 7 gives examples of each documentation characteristic that is seen in the case studies and also the manual modifications that can be applied in each situation. Looking at the availability of the method's documentation in these software projects, we observed that not all methods are documented. Table 2 shows the percentage of documented methods in JGrapht (80%), Collections (70%), and JHotDraw (53%), therefore mandating the manual modifications as described in Section 4.

For this technique to work credibly for java-based projects without the requirement of manual checking, the following points need to be taken care of:

1. The description part of documentation must only contain comments related to the semantics of the method.
2. If the description part is absent, it should contain description associated with the @return tag or a lookup to similar method documentation using @see tag (so that its documentation can be used here).
3. Only overridden methods (methods overriding some other methods present in the project) with @override annotations are allowed to be undocumented.
4. All the methods in a project should follow a uniform style of documentation.

For projects in other languages, the documentation should follow uniform style and should only contain description related to semantics of methods. These two requirements are necessary for the proposed technique to give useful results.

Most importantly methods should not be left undocumented.

Table 7. The key features of documentation as seen in case studies. The table illustrates how we manually dealt with documentation of methods that were not in the required format.

Documentation Characteristic		Case Study Examples	Manual Modification Applied
Category	**Characteristics**		
InComplete comments [78,79]	The documentation of code fragments (here methods) can contain both or any part of documentation (description or descriptive tags or both).	The method alg.BellmanFordIterator.putPrevSeenData contain only @param and @return descriptive tags. The @return tag is also accompanied by only a "."	The contents associated with @return tag are modified to contain a meaningful return description, which is later extracted to form the description of method. The description can be *"@return a set containing previously seen vertices"*
Not documented methods [80]	Two or more related methods may be documented collectively. This leads to documentation being associated with only one method, while all other methods are seen undocumented.	Methods jhotdraw.geom.Bezier.b0/b1/b2/b3 are documented collectively only for b0 while b1,b2,b3 are seen undocumented	Make separate documentation for each method.
	Some methods perform a sub-function of a task.The programmer may only document the method performing the whole task, leaving the methods performing sub-functions undocumented.	The method jhotdraw.app.OpenApplicationFileAction.openView is a sub-method of jhotdraw.app.OpenApplicationFileAction.actionPerformed method. Hence left undocumented	The method may be left undocumented or documented with description, e.g., "open the specified view of the application"
	Some methods are seen to be undocumented for unknown reasons, maybe the laziness of the programmer leads to these methods being left undocumented	The method jgrapht.graph.AbstractBaseGraph.DirectedSpecifics.addVertex is left undocumented due to developer's laziness	The method may be left undocumented or documented with description, e.g., " add vertex to the directed graph "
Not required comments [72]	Notes comment	The ending tasks are generally present in class documentation as whole	If any notes comments are present in method's documentation, it is removed.
	Explanatory comment	jgrapht.Graph.addEdge method contains a lot of explanation about how an edge is added to the graph	These kind of explanations are removed
	Contextual comment	jgrapht.Graph.removeAllVertices method contains a comment linking it to removeVertex method	All the linking comments are removed from documentation.
	Evolutionary comment	jgrapht.alg.ConnectivityInspector.pathExists contains an evolutionary comment describing its future versions	These comments are removed from documentation.
	Conditional comment	jgrapht.Graph.vertexSet method contains a comment describing the situation when the graph gets modified while it is iterated over	These comments are removed from documentation.
Low-quality comments [79–81]	It may be possible that the whole description of method do not contain any descriptive comment. These methods may result in false negatives	See documentation of the method jhotdraw.beans.WeakPropertyChangeListener.removeFromSource, which do not contain any semantic details of method	One should add some semantic details to the documentation or leave the method with the given documentation, making it similar to not documented methods. They do not come in result sets.

RQ2: What are the implications of applying different combinations of weight and similarity measures for performing LSI on the extracted method's documentation?

We examined the result obtained using all the combinations listed in Table 3. We observed that all of them gave similar results with minor variation in recall values. In all the case studies, combination 3 (TF weights and Cosine Similarity without POS Selective weight-age) outperform others, while combination 4 (binary weights and Jaccard similarity measure) performed the least. The ranking of the 12 combinations is given in Table 8. This ranking is based on the recall values and whether the combination exhibits an ideal property (a large proportion of clone pairs concentrated at higher similarity values). It can

be seen that while combination 4 gave good recall values, it is ranked lowest because it shows a large proportion of clone pairs for similarity values as low as 0.3.

Table 8. Ranking of each combination of weight and similarity measures used in this study. Ranking is based on recall values and whether the combination exhibit ideal property (a large proportion of clone pairs concentrated at higher similarity values) (see Table 3 for the full forms of acronyms used in column 2).

Rank	Combination Type	Recall Values			Follow Ideal Property		
		JGrapht	Collections	JHotDraw	JGrapht	Collections	JHotDraw
1	TFCWOPOS (3)	85.88	75.69	72.66	YES	YES	YES
2	RCWOPOS (1)	85.88	75.60	72.58	YES	YES	YES
3	BCWOPOS (2)	84.70	75.60	73.05	YES	YES	YES
4	CBOW (8)	83.52	74.63	73.25	YES	YES	YES
5	BDWOPOS (5)	82.35	71.07	71.52	YES	NO	YES
6	SG (9)	84.70	69.91	72.98	YES	NO	YES
7	FASTTEXT (7)	84.70	73.28	70.50	NO	YES	YES
8	RCWPOS (10)	78.82	65.38	70.27	NO	YES	YES
9	TFCWPOS (11)	77.64	65.38	70.39	NO	YES	YES
10	TFIDFCWOPOS (6)	80	66.24	70.11	NO	NO	YES
11	TFIDFCWPOS (12)	77.64	57.97	66.54	NO	NO	YES
12	BJWOPOS (4)	87.05	76.08	73.13	NO	NO	NO

RQ3: Is the technique scalable, i.e., applicable to large-sized projects?

In the case studies, it can be observed that with an increase in size, the number of methods documented decreases. The style of documentation becomes nonuniform, as for larger-sized projects, more developers get involved, and they follow different styles of documentation. Section 7 explains how the documentation of methods becomes nonuniform. However, if the proper documentation structures are followed, the proposed mechanism shall scale well even to the large-sized projects.

7. Threats to Validity

Documentation of software is done in the English language. A method can be described in a language in varied different styles of sentences. For example, consider a method that checks for the equality of two integers: a and b. This method can be described in the following different styles:

1. "Returns true iff a = b".
2. "Checks whether a equals b".
3. "Checks for the equality of two integers".
4. "Checks for the equality of a and b" and so on.

For LSI to work, a uniform style of documentation should be followed to find cloned methods. It is also sometimes encountered that a method is not documented with a descriptive comment or is not documented at all. To deal with this issue, a manual analysis of all the method documentation is necessary. This vagueness in documentation style and its absence poses a significant threat to results validity. Further, the absence of Javadoc comments may result in missing clone pairs, and manual injection of documentation is a subjective issue and depends on the code understanding.

8. Conclusions and Future Work

A software's documentation of methods shows an excellent expressiveness in describing "what the method does". In this study, applying information retrieval techniques to these documentations to extract functionally similar or nearly functionally similar methods exhibited impressive results. *Recall* values are found to be between 68% to 89% for the three case studies, i.e., *JGrapht*, *JHotDraw*, and *Collections*. However, the number of clone pairs identified are significant, which is attributed to LSI's superior capability to match two contextually similar terms in different documents. The proposed technique not only considers similar terms in the documentation, but also considers their ordering, e.g., two strings "Are you there?" and "there you are" shall be processed differently. For certain

combinations, we used selective POS weight assignment strategy, which is not found to exhibit significant improvement in results when compared to their counterparts that do not use selective POS weight assignment. Different weight measures (Raw, Binary, TF, and TF-IDF) and models (SG, CBOW, and FASTTEXT) are used in the study. The study shows that the combination using TF weights, cosine similarity, and without POS selective weight-age shows better results for all the three case studies performed for predicting similar methods from their documentations. The relatively low values of the "Precision" metric could be justified owing to the simplicity and lower implementation cost of the proposed mechanism.

Future works may consider tag-based comparison, i.e., the content of each tag (@param, @see, @return, etc.) may be compared separately, especially for java-based projects. Precision can also be greatly improved if along with documentation of code, the source code itself takes part in clone detection process. Various researchers also recommend the use of IR techniques combined with other clone detection tools to improve upon their results. Advanced variations of latent semantic indexing can also be used for further experimentation to analyze and compare their capabilities towards giving improved clone detection results.

Author Contributions: All authors contributed equally to this work. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Monden, A.; Nakae, D.; Kamiya, T.; Sato, S.I.; Matsumoto, K.I. Software quality analysis by code clones in industrial legacy software. In Proceedings of the Eighth IEEE Symposium on Software Metrics, Ottawa, ON, Canada, 4–7 June 2002; pp. 87–94.
2. Kapser, C.J.; Godfrey, M.W. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empir. Softw. Eng.* **2008**, *13*, 645. [CrossRef]
3. Thummalapenta, S.; Cerulo, L.; Aversano, L.; Di Penta, M. An empirical study on the maintenance of source code clones. *Empir. Softw. Eng.* **2010**, *15*, 1–34. [CrossRef]
4. Rahman, F.; Bird, C.; Devanbu, P. Clones: What is that smell? *Empir. Softw. Eng.* **2012**, *17*, 503–530. [CrossRef]
5. Chatterji, D.; Carver, J.C.; Kraft, N.A. Code clones and developer behavior: Results of two surveys of the clone research community. *Empir. Softw. Eng.* **2016**, *21*, 1476–1508. [CrossRef]
6. Mondal, M.; Rahman, M.S.; Roy, C.K.; Schneider, K.A. Is cloned code really stable? *Empir. Softw. Eng.* **2018**, *23*, 693–770. [CrossRef]
7. Saini, N.; Singh, S.; Suman, S. Code Clones: Detection and Management. *Procedia Comput. Sci.* **2018**, *132*, 718–727. [CrossRef]
8. Gabel, M.; Yang, J.; Yu, Y.; Goldszmidt, M.; Su, Z. Scalable and systematic detection of buggy inconsistencies in source code. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, New York, NY, USA, 1 October 2010; pp. 175–190.
9. Barbour, L.; Khomh, F.; Zou, Y. Late propagation in software clones. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, 25–30 September 2011; pp. 273–282.
10. Wagner, S.; Abdulkhaleq, A.; Kaya, K.; Paar, A. On the relationship of inconsistent software clones and faults: An empirical study. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 79–89.
11. Mondal, M.; Roy, C.K.; Schneider, K.A. Identifying code clones having high possibilities of containing bugs. In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 99–109.
12. Marcus, A.; Maletic, J.I. Identification of high-level concept clones in source code. In Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 26–29 November 2001; pp. 107–114.
13. Singh, M.; Sharma, V. High Level Clones Classification. *Int. J. Eng. Adv. Technol. (IJEAT)* **2013**, *2*, 2249–8958.
14. Roy, C.K.; Cordy, J.R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In Proceedings of the 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, Netherlands, 10–13 June 2008; pp. 172–181.

15. Selim, G.M.; Foo, K.C.; Zou, Y. Enhancing source-based clone detection using intermediate representation. In Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE), Beverly, MA, USA, 13–16 October 2010; pp. 227–236.
16. Cordy, J.R.; Roy, C.K. The NiCad clone detector. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, Canada, 22–24 June 2011; pp. 219–220.
17. Al-Omari, F.; Keivanloo, I.; Roy, C.K.; Rilling, J. Detecting clones across microsoft. net programming languages. In Proceedings of the 2012 19th Working Conference on Reverse Engineering, Kingston, ON, Canada, 15–18 October 2012; pp. 405–414.
18. Bauer, V.; Völke, T.; Eder, S. Combining clone detection and latent semantic indexing to detect re-implementations. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 3, pp. 23–29.
19. Caldeira, P.M.; Sakamoto, K.; Washizaki, H.; Fukazawa, Y.; Shimada, T. Improving Syntactical Clone Detection Methods through the Use of an Intermediate Representation. In Proceedings of the 2020 IEEE 14th International Workshop on Software Clones (IWSC), London, ON, Canada, 18 February 2020; pp. 8–14.
20. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* **2006**, *32*, 176–192. [CrossRef]
21. Brixtel, R.; Fontaine, M.; Lesner, B.; Bazin, C.; Robbes, R. Language-independent clone detection applied to plagiarism detection. In Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, Timisoara, Romania, 12–13 September 2010; pp. 77–86.
22. Hummel, B.; Juergens, E.; Heinemann, L.; Conradt, M. Index-based code clone detection: Incremental, distributed, scalable. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010; pp. 1–9.
23. Yuan, Y.; Guo, Y. Boreas: An accurate and scalable token-based approach to code clone detection. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 286–289.
24. Toomey, W. Ctcompare: Code clone detection using hashed token sequences. In Proceedings of the 2012 6th International Workshop on Software Clones (IWSC), Zurich, Switzerland, 4 June 2012; pp. 92–93.
25. Murakami, H.; Hotta, K.; Higo, Y.; Igaki, H.; Kusumoto, S. Folding repeated instructions for improving token-based code clone detection. In Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, Riva del Garda, Italy, 23–24 September 2012; pp. 64–73.
26. Murakami, H.; Hotta, K.; Higo, Y.; Igaki, H.; Kusumoto, S. Gapped code clone detection with lightweight source code analysis. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 93–102.
27. Shi, Q.Q.; Zhang, L.P.; Meng, F.J.; Liu, D.S. A novel detection approach for statement clones. In Proceedings of the 2013 IEEE 4th International Conference on Software Engineering and Service Science, Beijing, China, 23–25 May 2013; pp. 27–30.
28. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. SourcererCC: Scaling code clone detection to big-code. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 1157–1168.
29. Hung, Y.; Takada, S. CPPCD: A Token-Based Approach to Detecting Potential Clones. In Proceedings of the 2020 IEEE 14th International Workshop on Software Clones (IWSC), London, ON, Canada, 18 February 2020; pp. 26–32.
30. 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, 20–26 May 2007; pp. 96–105.
31. Lee, M.W.; Roh, J.W.; Hwang, S.W.; Kim, S. Instant code clone search. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, NM, USA, November 2010; pp. 167–176.
32. Biegel, B.; Diehl, S. JCCD: A flexible and extensible API for implementing custom code clone detectors. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 2010; pp. 167–168.
33. Brown, C.; Thompson, S. Clone detection and elimination for Haskell. In Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Madrid, Spain, January 2010; pp. 111–120.
34. Hotta, K.; Yang, J.; Higo, Y.; Kusumoto, S. How accurate is coarse-grained clone detection?: Comparision with fine-grained detectors. *Electron. Commun. EASST* **2014**, *63*. [CrossRef]
35. Yang, Y.; Ren, Z.; Chen, X.; Jiang, H. Structural function based code clone detection using a new hybrid technique. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 23–27 July 2018; Volume 1, pp. 286–291.
36. Patenaude, J.F.; Merlo, E.; Dagenais, M.; Laguë, B. Extending software quality assessment techniques to java systems. In Proceedings of the Seventh International Workshop on Program Comprehension, Pittsburgh, PA, USA, 5–7 May 1999; pp. 49–56.
37. Balazinska, M.; Merlo, E.; Dagenais, M.; Lague, B.; Kontogiannis, K. Measuring clone based reengineering opportunities. In Proceedings of the Sixth International Software Metrics Symposium (Cat. No.PR00403), Boca Raton, FL, USA, 4–6 November 1999; pp. 292–303. [CrossRef]
38. Perumal, A.; Kanmani, S.; Kodhai, E. Extracting the similarity in detected software clones using metrics. In Proceedings of the 2010 International Conference on Computer and Communication Technology (ICCCT), Allahabad, India, 17–19 September 2010; pp. 575–579.

39. Li, Z.O.; Sun, J. A metric space based software clone detection approach. In Proceedings of the 2010 2nd IEEE International Conference on Information Management and Engineering, Chengdu, China, 16–18 April 2010; pp. 393–397.
40. Yuan, Y.; Guo, Y. CMCD: Count matrix based code clone detection. In Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference, Ho Chi Minh City, Vietnam, 5–8 December 2011; pp. 250–257.
41. Abd-El-Hafiz, S.K. A metrics-based data mining approach for software clone detection. In Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, Izmir, Turkey, 16–20 July 2012; pp. 35–41.
42. Komondoor, R.; Horwitz, S. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 40–56.
43. Krinke, J. Identifying similar code with program dependence graphs. In Proceedings of the Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, 2–5 October 2001; pp. 301–309.
44. Gabel, M.; Jiang, L.; Su, Z. Scalable detection of semantic clones. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 2008; pp. 321–330.
45. Higo, Y.; Kusumoto, S. Code clone detection on specialized PDGs with heuristics. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, 1–4 March 2011; pp. 75–84.
46. Su, F.H.; Bell, J.; Harvey, K.; Sethumadhavan, S.; Kaiser, G.; Jebara, T. Code relatives: Detecting similarly behaving software. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, November 2016; pp. 702–714.
47. Frakes, W.B.; Baeza-Yates, R. (Eds.) *Information Retrieval: Data Structures and Algorithms*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1992.
48. Marcus, A.; Sergeyev, A.; Rajlich, V.; Maletic, J.I. An information retrieval approach to concept location in source code. In Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, 8–12 November 2004; pp. 214–223.
49. Marcus, A.; Poshyvanyk, D. The conceptual cohesion of classes. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 26–29 September 2005; pp. 133–142.
50. Bauer, V.; Völke, T.; Eder, S. Comparing TF-IDF and LSI as IR Technique in an Approach for Detecting Semantic Re-Implementations in Source Code. 2015. Available online: https://mediatum.ub.tum.de/doc/1281127/file.pdf (accessed on 20 August 2020).
51. Deerwester, S.; Dumais, S.T.; Furnas, G.W.; Landauer, T.K.; Harshman, R. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci.* **1990**, *41*, 391–407. [CrossRef]
52. Poshyvanyk, D.; Gethers, M.; Marcus, A. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2012**, *21*, 23. [CrossRef]
53. Xie, C.; Wang, X.; Qian, C.; Wang, M. A Source Code Similarity Based on Siamese Neural Network. *Appl. Sci.* **2020**, *10*, 7519. [CrossRef]
54. Fu, M.; Luo, G.; Zheng, X.; Zhang, T.; Yu, D.; Kim, M. An ensemble learning approach for software semantic clone detection. *arXiv* **2020**, arXiv:2010.04336.
55. Tairas, R.; Gray, J. An information retrieval process to aid in the analysis of code clones. *Empir. Softw. Eng.* **2009**, *14*, 33–56. [CrossRef]
56. Kuhn, A.; Ducasse, S.; Gírba, T. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.* **2007**, *49*, 230–243. [CrossRef]
57. Grant, S.; Cordy, J.R. Vector space analysis of software clones. In Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension, Vancouver, BC, Canada, 17–19 May 2009; pp. 233–237.
58. Ghosh, A.; Kuttal, S.K. Semantic Clone Detection: Can Source Code Comments Help? In Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Lisbon, Portugal, 1–4 October 2018; pp. 315–317.
59. Reddivari, S.; Khan, M.S. CloneTM: A Code Clone Detection Tool Based on Latent Dirichlet Allocation. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; Volume 1, pp. 930–931.
60. 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]
61. Clone Doctor: Software Clone Detection and Reporting. Available online: http://www.semdesigns.com/Products/Clone/ (accessed on 5 February 2021).
62. Abid, S. Recommending related functions from API usage-based function clone structures. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, August 2019; pp. 1193–1195.
63. Kuttal, S.K.; Ghosh, A. Source Code Comments: Overlooked in the Realm of Code Clone Detection. *arXiv* **2020**, arXiv:2006.14505.
64. Maskeri, G.; Sarkar, S.; Heafield, K. Mining business topics in source code using latent dirichlet allocation. In Proceedings of the 1st India Software Engineering Conference, Hyderabad, India, February 2008; pp. 113–120.
65. Wang, S.; Yuan, J.; Li, X.; Qian, Z.; Arena, F.; You, I. Active data replica recovery for quality-assurance Big Data analysis in IC-IoT. *IEEE Access* **2019**, *7*, 106997–107005. [CrossRef]
66. Kupiec, J. Robust part-of-speech tagging using a hidden Markov model. *Comput. Speech Lang.* **1992**, *6*, 225–242. [CrossRef]

67. Falessi, D.; Cantone, G.; Canfora, G. A comprehensive characterization of NLP techniques for identifying equivalent requirements. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Bolzano-Bozen, Italy, September 2010; p. 18.
68. Falessi, D.; Cantone, G.; Canfora, G. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. *IEEE Trans. Softw. Eng.* **2013**, *39*, 18–44. [CrossRef]
69. Och Dag, J.N. A Feasibility Study of Automated Support for Similarity Analysis of Natural Language Requirements in Market-Driven Development. *Manag. Nat. Lang. Requir. Large-Scale Softw. Dev.* **2002**, *7*, 85.
70. Och Dag, J.N.; Thelin, T.; Regnell, B. An experiment on linguistic tool support for consolidation of requirements from multiple sources in market-driven product development. *Empir. Softw. Eng.* **2006**, *11*, 303–329. [CrossRef]
71. Plisson, J.; Lavrac, N.; Mladenic, D. A Rule Based Approach to Word Lemmatization. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.9308&rep=rep1&type=pdf (accessed on 15 March 2020).
72. Howard, M.J.; Gupta, S.; Pollock, L.; Vijay-Shanker, K. Automatically mining software-based, semantically-similar words from comment-code mappings. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 377–386.
73. JGraphT. Available online: https://sourceforge.net/projects/jgrapht/files/JGraphT/Version%200.8.1/ (accessed on 19 January 2021).
74. Java. Available online: http://www.java2s.com/code/jar/c/downloadcommonscollections321100sourcesjar.htm (accessed on 15 January 2021).
75. JHotDraw 7. Available online: https://www.randelshofer.ch/oop/jhotdraw/ (accessed on 22 January 2021).
76. Qualitas Corpus Clone Collection. Available online: http://www.qualitascorpus.com/clones/ (accessed on 3 February 2020).
77. Tempero, E. Towards a curated collection of code clones. In Proceedings of the 7th International Workshop on Software Clones, San Francisco, CA, USA, 19 May 2013; pp. 53–59.
78. Pascarella, L.; Bruntink, M.; Bacchelli, A. Classifying code comments in Java software systems. *Empir. Softw. Eng.* **2019**, *24*, 1499–1537. [CrossRef]
79. Khamis, N.; Witte, R.; Rilling, J. Automatic quality assessment of source code comments: The JavadocMiner. In Proceedings of the International Conference on Application of Natural Language to Information Systems, Cardiff, UK, 23–25 June 2010; pp. 68–79.
80. Steidl, D.; Hummel, B.; Juergens, E. Quality analysis of source code comments. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 83–92.
81. Haouari, D.; Sahraoui, H.; Langlais, P. How good is your comment? A study of comments in java programs. In Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement, Banff, AB, Canada, 22–23 September 2011; pp. 137–146.