IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 17, 2020, accepted November 12, 2020, date of publication November 24, 2020,
date of current version December 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3040024

Identifying Semantic Outliers of Source Code Artifacts and Their Application to Software Architecture Recovery

KI-SEONG LEE[1] AND CHAN-GUN LEE[2]
[1]Da Vinci College of General Education, Chung-Ang University, Seoul 06974, South Korea
[2]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Chan-Gun Lee (cglee@cau.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information and Communications Technology (ICT) under Grant NRF-2017R1E1A1A01075803.

ABSTRACT Understanding software architecture is essential to software maintenance. There has been much effort to derive software architecture views from source code artifacts. Typically, along with structural information, the semantic information derived from an identifier name and comments are helpful. However, because code vocabulary choice depends on a developer's subjective decision, some source code may have semantically low text quality, leading to an inaccurate architecture recovery. This paper aims to improve the architecture recovery of a software system by identifying and removing the semantic outliers of source code artifacts. Accordingly, we propose a novel measure *Conceptual Conformity* (CC), which computes the similarity between two latent topic distributions obtained from both the source code and its package. We use CC to identify source code that is not relevant to the package's semantic context and define it as a semantic outlier. Because the semantic outliers may cause inaccurate architecture recovery, we remove them during the recovery process. We apply our approach to three open-source projects. The results demonstrate that, for projects with low recovery performance, removing outliers leads to higher recovery accuracy.

INDEX TERMS Software architecture recovery, software quality, semantic outlier.

I. INTRODUCTION

The foundation of software maintenance is an understanding of the software. Understanding software is so laborious and time-consuming that it may be the most significant burden in the maintenance stage. Automated efforts to understand software have continued steadily in an attempt to reduce this burden [1]. Software architecture is a beneficial product for maintenance engineers because it provides a high-level view of a software system. However, in practice, the architecture document is prone to becoming outdated or unavailable over time. Accordingly, various methods for automatically recovering software architecture have been studied [2].

Architecture recovery is a technique for deriving a module view of a system from software artifacts. The recovered architecture can be used in several ways: to identify architectural changes [3], to find architectural decay [4], or to understand the overall structure of the software.

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

A static software structure helps re-modularize a software system because structural dependencies, such as inheritance, method calls, or object references, provide intuitive and direct relationships between the software entities. In contrast, there have been several attempts to consider the semantic similarity between software entities [5]–[10]. In this approach, textual information in the source code and comments is analyzed to determine the semantic similarity between software entities. Many studies have revealed the usefulness of a semantic approach in architecture recovery because semantic information can complement the simplicity of structural information. Furthermore, evolutionary information—a method that uses software development history—is used for recovery [11].

Several studies have considered the quality of software entities and their relationships for architecture recovery [12], [13]. The common argument is that low-quality software dependency information and structural noise harm architecture recovery. However, these studies focus only on structural information. Although semantic information is a beneficial factor in software analysis, little attention has been paid to its quality.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Semantics-based studies in software analysis consider a source code artifact as a text document (called a ''source code document'' in this paper). The premise is that terms that appear in a particular source code document are chosen to represent its domain and implementation concepts accurately. Most software developers try to follow the code-naming convention of the organization to which they belong. Nevertheless, because the source code is written manually, the identifier names and comments are dependent on the developer's writing ability. Semantic analysis may produce a completely different output if someone's term choice is not desirable or consistent with the software component's design concept. Therefore, we must identify and filter noisy entities in semantic-information-based software analysis.

In this paper, we propose a novel measure to identify the semantic outliers of source code artifacts and improve architecture recovery accuracy. We identify the semantic outliers by defining a new software measure called *Conceptual Conformity* (CC), which quantifies the extent to which the source code's topic matches the package topic. We identify and remove noisy entities from a software representation model and reconstruct the software system's architecture module-view using this measure. We clarify our work through the following research questions:

RQ1 - Can we identify source code that is semantically far from the software design decision?

RQ2 - Does removing outliers affect software architecture recovery?

RQ3 - How does the choice of the outlier affect recovery accuracy?

The contributions of this paper are as follows.

- We propose a software quality measure to identify the semantic outliers of source code.
- We reveal that outlier detection and filtering are effective in semantic-information-based architecture recovery.

The remainder of this paper is organized as follows. Section 2 presents the background and related work on architecture recovery and semantic analysis. Section 3 explains how to identify semantic outliers. Section 4 presents the proposed recovery method, and Section 5 analyzes the experimental results. Section 6 presents threats to validity. Finally, Section 7 concludes the study.

II. BACKGROUND AND RELATED WORK

A. SOFTWARE ARCHITECTURE RECOVERY

In software architecture recovery, an easily understandable module-view is created to represent the overall system. In the literature, software architecture recovery has been referred to as decomposition, re-modularization, partitioning, clustering, and reconstruction. Architecture recovery follows four steps: (1) identification of software entities, (2) calculation of the similarity between the entities, (3) clustering, and (4) evaluation [14]. Many studies have evolved within this research area.

Early studies focused on clustering methods. Mancoridis *et al.* [15], Mitchell and Mancoridis [16] proposed Bunch, an architecture recovery method based on optimization. Bunch finds the optimal modularization solution for decomposing a software system. Its objective function is called Modularization Quality (MQ), which measures the cohesion and coupling of modules with intra- and inter-dependencies among the software entities. Bunch provides two optimization algorithms: a hill-climbing algorithm and a genetic algorithm. Shokoufandeh *et al.* [17] found that, although the genetic Bunch algorithm finds a solution more quickly, the hill-climbing Bunch algorithm produces higher-quality clustering results.

Tzerpos and Holt [18] proposed the Algorithm for Comprehension-Driven Clustering (ACDC), which recovers the software architecture using patterns. The authors found that specific patterns occur while grouping software entities. For example, (a) procedures/variables in the same file are grouped, (b) directories may correspond to subsystems, and (c) the body (.c) and header (.h) are grouped. The main pattern is a subgraph dominator pattern, which detects a dominator node and its dominated set. ACDC finds the node and its subgraph as a cluster.

Maqbool and Babri [19], [20] proposed a Weighted Combined Algorithm (WCA), a hierarchical approach to architecture recovery. It measures the distance between software entities and merges the closest pair into a cluster. WCA also measures the distance between clusters and merges them into a high-level cluster. WCA uses the Jaccard coefficient and Unbiased Ellenberg to measure the inter-cluster distance.

Andritsos and Tzerpos [21] proposed the scaLable Information Bottleneck (LIMBO) algorithm. It is also based on hierarchical clustering but differs from WCA. LIMBO aims to enable the software system to have minimum information loss (IL), which is based on the mutual information concept from information theory.

Researchers began considering entity relationships from different perspectives—typically a semantic relationship. Corazza *et al.* [5] proposed architecture recovery using zone-based lexical information. They extracted terms from the software's source code and classified them into six types: class names, attribute names, method names, parameter names, comments, and code statements. The authors used the expectation-maximization (EM) algorithm to assign a weight to each type and then performed hierarchical clustering.

Garcia *et al.* [8] proposed Architecture Recovery using Concerns (ARC). They recovered software entity concerns using the Latent Dirichlet Allocation (LDA) statistical language model [22]. ARC is based on lexical information, similar to the method proposed by Corazza *et al.* [5], but it manages additional conceptual information using a latent topic. Garcia *et al.* combined structural and conceptual information and clustered it with the hierarchical algorithm.

Recently, attention has been shifted to the quality of the entity. Lutellier *et al.* [13] described how previous studies did not consider the impact of software dependency quality

on architecture recovery. They experimentally demonstrated that using an accurate dependency could improve the quality of existing architecture recovery techniques. Furthermore, Constantinou *et al.* [12] emphasized that structurally-noisy classes must be identified in a preparatory procedure for architecture recovery. They computed a class's significance value using graph theory techniques and classified omnipresent classes as noises according to their significance values. Their experiments demonstrated that a noise-filtering technique provides superior MQ [15] and Mojo [23] values for architecture recovery.

Against this research background, we propose incorporating semantic-information quality into architecture recovery. Our recovery is based on semantic information derived from LDA, as in [8]. Furthermore, the proposed recovery applies an outlier removal strategy, which has not been addressed in previous semantic-information-based recovery studies.

B. SOFTWARE ANALYSIS WITH SEMANTIC INFORMATION

The semantic information in source code contains domain-specific software concepts—resulting from developers reflecting their knowledge in identifier names and comments. Thus, there have been many studies on the application of information retrieval techniques to software analysis.

Kuhn *et al.* [6] proposed applying latent semantic indexing (LSI) [24] to compute the linguistic similarity between source code documents written in Java. LSI is an information retrieval technique based on the vector space model. In a term-document matrix, LSI computes latent structures to reduce various noises such as synonymy and polysemy. It then yields the vector space of the latent structures with relatively small dimensions. Based on LSI results, the authors estimated the semantic similarity between source code based on the cosine similarity, and they clustered codes from similar topics.

Scanniello *et al.* [7] analyzed the text of software entities and computed the dissimilarity between all pairs of the software entities using LSI in both C/C++ and Java projects. During preprocessing, they listed a set of stopwords (a set of keywords of the C/C++ and Java languages) and then filtered the list from the extracted text. Furthermore, the authors applied the Porter stemmer [25]—a method similar to natural language processing—to reduce the number of semantically duplicated terms.

Corazza *et al.* [26] proposed a zone-based textual analysis technique. They extracted terms from four zones: natural language comments, Javadoc comments, method/class signatures, and variable identifiers in Java classes. The authors refined the semantic information using term frequency–inverse document frequency (tf–idf) and then weighted them with a probabilistic model.

Furthermore, LDA-based semantic analysis has attracted attention in the field of software analysis. LDA identifies latent topics and the probability distribution of the topics within a given corpus. A latent topic is denoted by a set of terms, which can be derived from statistical word co-occurrences. Each topic corresponds to a concept in the document. In LDA, to compute the topic distribution, two critical Dirichlet prior parameters are required: α (per-document topic distribution) and β (per-topic word distribution). With LDA, Maskeri *et al.* [27] extracted domain topics from source code, Lukins *et al.* [28] automated bug localization, Tian *et al.* [29] categorized software systems regardless of the programming language, and Garcia *et al.* [8] improved architecture recovery performance.

Concerned with LDA-based software analysis, Gethers and Poshyvanyk [9] proposed a coupling measurement between classes based on LDA and its extension. They studied each source code document's topic distribution using LDA and then measured the link probability between the documents using the relational topic model [30]. The authors demonstrated with a case study that their coupling measure captures new dimensions not covered by the existing coupling metrics.

Furthermore, some studies focused on a semantic measurement to capture the software cohesion. Marcus *et al.* [31] proposed Conceptual Cohesion of Classes (C3), which measures how class methods are coherent. Their semantic coherence is based on the cosine similarity between method vectors derived from LSI. The authors adopted the average similarity among methods as conceptual cohesion. Liu *et al.* [32] proposed an LDA-based cohesion measure, Maximal Weighted Entropy (MWE), which captures topic cohesion using information entropy. They demonstrated that MWE is available in bug prediction.

These previous studies focused on extracting and refining the semantic information in a software system and using it helpfully. However, they did not consider semantic quality. Because they assumed that all text data are valuable in analysis, they used the data as-is, irrespective of text data quality. In contrast, we concentrate on the semantic quality of the source code in this study. We propose a novel measure to judge whether the semantic information is useful.

III. IDENTIFYING SEMANTIC OUTLIERS OF SOURCE CODE ARTIFACTS

A. MOTIVATION AND APPROACH

In this section, we explain the semantic outliers of source code artifacts. As mentioned in the introduction section, the accuracy of semantic-information-based software analysis depends on the text quality's trustworthiness. Thus, identifying text quality is essential.

Manual code review by experts (e.g., a senior programmer, component designer, or project manager) is the optimal solution for judging the quality of a text document. However, it is neither practical nor possible because the scale of a software system is continuously increasing. Moreover, it is not easy to quantify whether the document's terms are appropriate for the development concept. Therefore, we adopt an automated, objective method.

A software component or subsystem of software has a design concept for satisfying a particular requirement and is

designed at an early stage of the software lifecycle. Because the domain expert determines each component's design concept, such design decisions are reflected in the source code as textual information during the implementation stage. However, developers occasionally make minor decisions to name variables for various reasons (e.g., a lack of concern about semantics, time-to-market pressure, and the absence of naming criterion). Such decisions lower the software's textual quality and hinder semantics-based analysis. We intend to identify source code that inadequately follows the design concept. We try to understand the semantics of a component and identify the most heterogeneous entity among its members. Suppose the semantics of a particular entity differ from the design concept of the component. In that case, we assume that the entity is not useful for analysis, and we define it as a semantic outlier. In this paper, a software entity represents a class (or a file), and a component or subsystem indicates a software package.

B. SEMANTIC QUALITY MEASURE : CONCEPTUAL CONFORMITY (CC)

We judge whether the semantic information of a source code artifact complies with its design concept by comparing the latent topic distribution in source code to that of a package—latent topic distribution can explain the meaning of a document concept. Accordingly, we propose a novel measure, CC, which computes the similarity between the two topic distributions obtained from both the source code and the package. The CC's similarity is computed using the Jensen-Shannon divergence (JSD) [33], a suitable metric for measuring the similarity between two probability distributions.

We compute the CC of a source code document by applying a latent topic distribution. The latent topic is composed of a set of words and is stochastically determined by LDA. In LDA, if a document focuses on a specific subject, particular words associated with the subject may occur more frequently. For example, assume a document focuses on the subject "network communications." The document may use words such as "server," "socket," "connect," and "listen," which are related to "network communications." With LDA, we take a set of terms as a topic based on the likelihood of co-occurrence to obtain the topic distribution composed of several topics. The topic distribution of a document reveals how many different topics are distributed, so we can use it to determine the document's subject matter.

TABLE 1. An example of the topic distribution.

	Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
Doc 1	0.5	0.1	0.1	0.2	0.1
Doc 2	0.1	0.6	0.1	0.1	0.1
Doc 3	0.3	0.2	0.1	0.2	0.2

Table 1 presents an abstract model of the LDA topic distribution. The example assumes that there are five topics in a corpus with three documents. This structure makes it easy to identify which documents address which topics. Several notations in our study are defined as follows.

P_i : topic distribution of Doc i

p_i^j : probability of topic j in Doc i

$P_i = (p_i^1, p_i^2, p_i^3, \ldots, p_i^k)$, where k is the total number of topics

P_i is a semantic concept of document i, and the sum of all values in P_i becomes 1 because it is a probability distribution. Applying this to the software, we can obtain the topic distribution P from each source code file. The following is an example of the topic distribution in Table 1.

$$P_1 = (0.5, 0.1, 0.1, 0.2, 0.1)$$
$$P_2 = (0.1, 0.6, 0.1, 0.1, 0.1)$$
$$P_3 = (0.3, 0.2, 0.1, 0.2, 0.2)$$

After obtaining P vectors for each file, semantic information for each package is required. We obtain the mean value of each topic in a package to capture package-level semantic information. Related notations are defined as follows.

Pkg_i: a package containing Doc i

Q_i : topic distribution of Pkg_i

q_i^j : mean probability of topic j in Pkg_i

$Q_i = (q_i^1, q_i^2, q_i^3, \ldots q_i^k)$, where k is the total number of topics

$$q_i^j = \frac{\sum_{m \in Pkg_i} p_m^j}{|Pkg_i|} \tag{1}$$

$|Pkg_i|$ is the number of documents in a package containing document i. Therefore, q_i^j is the mean value of a specific topic in a particular package. For example, Table 2 presents the topic distribution of a software system with a package structure—there are two packages and six documents with five topics. We compute the mean value of each topic in a package and then generate a package-level topic distribution. The following presents the calculation process of the case in Table 2.

$$q_1^1 = \frac{p_1^1 + p_2^1 + p_3^1}{|Package\ A|} = \frac{0.5 + 0.1 + 0.3}{3} = 0.3$$
$$q_1^2 = 0.3,\ \ q_1^3 = 0.1,\ \ q_1^4 = 0.17,\ \ q_1^5 = 0.13,$$
$$\therefore \mathrm{Q}_1 = (0.3, 0.3,\ \ 0.1, 0.17, 0.13),\ \ \mathrm{Q}_1 = \mathrm{Q}_2 = \mathrm{Q}_3$$

TABLE 2. A software representation model with package structure.

Decomposition	Source code	Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
Package A	Doc 1	0.5	0.1	0.1	0.2	0.1
	Doc 2	0.1	0.6	0.1	0.1	0.1
	Doc 3	0.3	0.2	0.1	0.2	0.2
Package B	Doc 4	0.1	0.2	0.1	0.1	0.5
	Doc 5	0.1	0.1	0.2	0.4	0.2
	Doc 6	0.1	0.1	0.1	0.5	0.2

Subsequently, we measure whether the subject of each document relates to the subject of the package. In the software semantics study, the cosine similarity is used for word vector base, but this study introduced information-theory metric because it is based on probability distribution of topics. We measure the similarity using the JSD, an information-theoretic metric for computing the information divergence of two probability distributions P and Q. JSD is a symmetrized and smoothed equation of Kullback-Leibler divergence [34] $D(P||Q)$ and is denoted by

$$JSD(P||Q) = \frac{1}{2}D(P||M) + \frac{1}{2}D(Q||M) \tag{2}$$

where $M = 1/2(P + Q)$. We measure the similarity between the document concept and package concept using the JSD. The original JSD only measures the distance of the probability distributions; it has a value between 0 and 1 and converges at 0 when the distance is closest. Therefore, we modify the metric to the concept of similarity. The proposed measure CC is defined as follows.

$$CC\ (i) = 1 - JSD(P_i||Q_i) \tag{3}$$

where P_i is the topic distribution of document i and Q_i is the mean topic distribution of the package containing document i. Because CC is a document level measure, we can compute the CC score for all source code files in a software system. Theoretically, if a document perfectly complies with a package design concept, CC is 1; otherwise, it is 0. The CC aims to quantify how much the semantic information of the source code complies the design concept of the architecture.

C. IDENTIFYING SEMANTIC OUTLIERS

We apply the proposed measure CC to the identification of semantic outliers. An outlier is defined as an object with distinctly unique properties in a dataset. We can find semantic outliers in disparate text documents that may harm the consistency of a software design concept.

1) EXPERIMENTAL DESIGN

We conducted an experiment to evaluate the feasibility of CC in detecting semantic outliers. The purpose of the experiment was to examine how accurately CC discriminates semantic outliers from software documents. Accordingly, we intentionally created qualitatively low-quality documents. We hypothesize that if a document describes concerns unrelated to the design concept or contains several meaningless terms, the CC value of the document is lower than that of a normal document. In this context, we injected fake documents into Apache Hadoop-core 0.19 [35], which is an open-source project written in the Java language. We randomly selected one package into which we injected three fake documents. The descriptions of the fake documents are as depicted in Table 3.

We generated a topic distribution using "LDA: Collapsed Gibbs Sampling Methods for Topic Models" [36]. The parameter setting for LDA is not trivial. We referred to Griffiths and Steyvers's study [37]. They tried to obtain a superior LDA result experimentally. According to their report, the output of LDA is sensitive to the number of topics. Thus, we varied the number of topics T from 10 to 100 by increments of 10 and used an alpha of 10/T and a beta of 0.1—alpha is the topic distributions per document and beta is the topic distributions per topic. Because LDA is a statistical approach, the more the algorithm iterates, the more stable the output. Therefore, we set the iteration to 250, a reasonable value for the LDA parameter considering the computational cost of our data.

TABLE 3. Description of outlier files.

Fake file	package	Description
file A	org.apache.hadoop.fs	A is a normal text document obtained from Wikipedia by searching with the "java" keyword. This file is not source code.
file B		B is a source code written in Java, but its naming rules are broken. Most of the method and variable names are modified to meaningless words. Furthermore, comments are messed.
file C		C is normal source code. However, it is moved from another package. (from security to fs)

In our experiment, we adapted an additional outlier detection technique to evaluate CC more objectively. We applied a distance-based outlier detection approach [38]. With this technique, we could identify a distinguishing document from all documents by considering the similarity between documents. We adopted the k-Nearest Neighbors (k-NN) algorithm [39] for outlier scoring. Although determining a proper k value is not a trivial problem in k-NN, we used some typical k values to facilitate our experiment. The k values in the experiment were set to 10, 2/n, and n-1, but we report only the 10 case because it produced the optimal output—a small k outperforms a large one. Another small k value should be considered, but we omitted a discussion about k because k optimization is not a significant issue in this study.

2) EXPERIMENTAL RESULT AND RQ1

Table 4 presents the outlier rank of the injected files in Hadoop-core 0.19. The rank is based on the measured values of both CC and k-NN. Because we obtained multiple LDA outputs to vary the number of topics, both the highest and mean ranks are presented. In 608 Java source files, injected file A ranks highest, which indicates that file A is the most disparate document. Similarly, files B and C are ranked second and eighth, respectively, for CC's optimal case. The experiment demonstrates that the proposed measure can identify lexically heterogeneous documents. Furthermore, CC outperforms k-NN in identifying those documents.

Moreover, CC can identify not only lexically low-quality documents but also semantically-unrelated documents. The CC value of file C is ranked eighth among all 608 files. Although file C contains ordinary Java source code, it does

TABLE 4. Comparison between CC and k-NN outlier detection in Hadoop-core 0.19.

Fake file	Rank by CC (1~608)		Rank by k-NN (1~608)	
	best	mean	best	mean
file A	**1**	1	1	1
file B	**2**	5.5	6	19.8
file C	**8**	18.6	51	56

not belong to the org.apache.hadoop.fs package. This observation is hardly discernible in the k-NN manner.

Recall the first research question as follows:

RQ1 - Can we identify source code that is semantically far from the software design decision?

In response to RQ1, our outlier detection method tries to identify documents that do not follow the component's concept, and the proposed measure operates effectively.

Figure 1 presents the distribution of the CC score in Hadoop-core 0.19. The horizontal axis denotes the CC value with a 0.1 range; the vertical axis illustrates the document frequency of each CC range. Most of the documents have a high CC value, which indicates significantly high semantic conformity to the design decision of Hadoop-core.

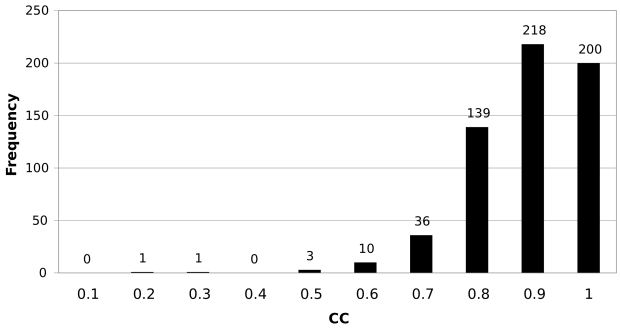

FIGURE 1. CC distribution in Hadoop-core 0.19.

IV. IMPROVING SOFTWARE ARCHITECTURE RECOVERY USING SEMANTIC OUTLIERS REMOVAL

A. SOFTWARE CLUSTERING USING LATENT TOPIC

For software architecture recovery, we use software clustering techniques introduced in the literature. Most software clustering techniques are based on two types of approaches: (1) a graph-based approach using a heuristic method to satisfy its objective function [16], [40], [18] and (2) a vector space-based approach using a hierarchical clustering algorithm and a distance metric [5], [6], [21], [8]. In this study, we use the latter techniques to cluster a topic-vector model. Figure 2 illustrates an overview of the proposed recovery process. The gray highlighted part of Figure 2 is the difference from the existing study, indicating the originality of this study.

1) REPOSITORY DATA ACQUISTION

In the first step, all terms contained in the source code are extracted. Because the term extractor that we implemented targets source code written in the Java language, some reserved keywords—such as "continue," "extends," and "throw"—are eliminated in the extraction process. Then, the use of the stemming algorithm ensures that only the stem words remain. We executed the stemming process using the snowball library [41], which provides a stemmer for the C and Java languages.

2) EXTRACTING SEMANTIC INFORMATION (LDA)

Using the collected terms, LDA determines the topic distribution of each source code document. Then, a document is denoted by an n-dimensional vector, where n denotes the number of topics. We set the LDA parameters to be the same as those used in the previous experiment: number of topics $|T| = 10$–100, alpha $= 10/T$, beta $= 0.1$, and iteration $= 250$. From the LDA, several topic-document matrices are obtained. A column denotes a topic in the matrices, and a row denotes a document (as in Table 1).

3) REFINING SOFTWARE REPRESENTATION MODEL

Before software clustering, we refine the topic-document matrices to improve recovery. Refinement refers to the detection and removal of semantic outliers—refined software models are downsized matrices. The details are described in a later section.

4) SOFTWARE CLUSTERING

This step decomposes the software entities into several subsystems using clustering algorithms. Our recovery is similar to [8] because it uses LDA data. We use agglomerative hierarchical clustering [42] like studies of [19], [20], a bottom-up clustering algorithm that groups the target entities based on their connectivity. Typically, this algorithm employs Jaccard similarity, but we use JSD because our software entity is based on probability distribution using LDA. We implemented a clustering tool using scikit-learn, a package library for machine learning in python.

5) EVALUATION OF THE ARCHITECTURE RECOVERY

The difference between the clustering output and ground-truth architecture [43] is considered for evaluating the recovery result. The ground-truth is an architecture module-view manually recovered by experts. Therefore, we can assess how accurately our output complies with the ground-truth. However, because available ground-truth architectures are limited, we use only three projects—Apache Hadoop-core [35], Apache-oodt [44], and ArchStudio [45]—that have publicly available ground-truth architectures associated with the Java programming language.

MoJoFM [46] quantifies the discrepancy between two architectural decompositions. Because MoJoFM has been widely used in studies on architecture recovery, we also use the measure. Equation (4) defines MoJoFM, where *mno* indicates the minimum number of *Move* or *Join* operations

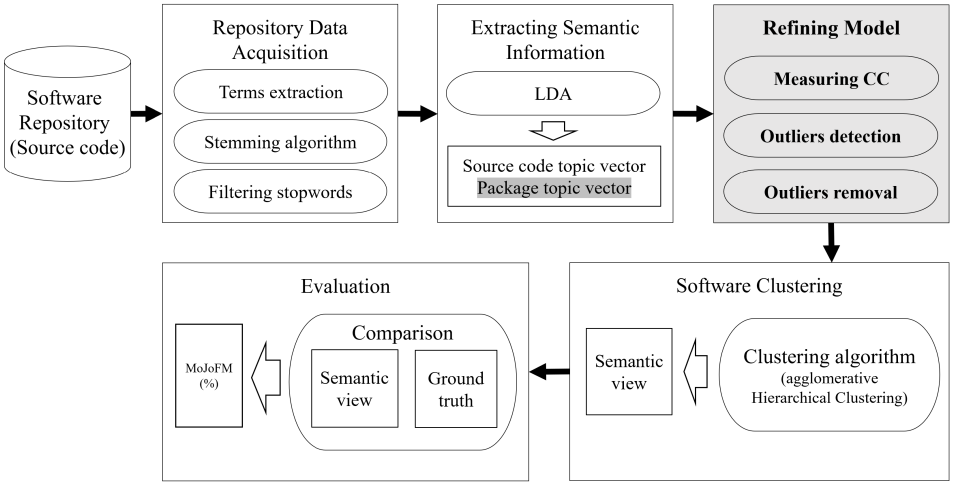

FIGURE 2. Overview of the architecture recovery process.

required to transform one structure into the other.

$$MoJoFM\ (M) = \left(1 - \frac{mno\ (A, B)}{\max\ (mno\ (\forall A, B))}\right) \times 100\% \quad (4)$$

B. SEMANTIC OUTLIERS REMOVAL

If any source code is of low lexical quality, the recovered architecture is not reliable. Therefore, we propose a technique to exclude semantic outliers to improve the accuracy of architecture recovery.

CC measures how well a source code document conforms to the design concept of a component to which the file belongs. As depicted in the previous experiment on fake file injection, CC can be used to identify the degree of semantic quality. Thus, we can discriminate noisy source code from a software system. Based on this idea, we introduce an outlier removal technique into the software clustering process. We refine the software representation model to exclude the outlier entities before clustering.

First, we measure CC for all source code artifacts in a software system. After the files are sorted according to low CC value, high-ranked files are candidates for semantic outliers. Next, we set a cut-off threshold to restrict the size of the outliers. Because the outliers are removed during the refinement, a large threshold causes significant loss of architectural information. Therefore, a valid threshold is needed based on the scale of the software system. Consequently, final outlier candidates are determined. The corresponding outlier tuples are removed from a topic-document matrix, and only the remaining tuples are used for clustering.

V. CASE STUDY

A. ARCHITECTURE RECOVERY FOR OPEN-SOURCE PROJECTS

A case study was conducted on open-source projects to evaluate the effectiveness of the proposed recovery method. Table 5 presents three open-source Java projects that have a ground-truth architecture.

TABLE 5. Open-source projects used in this study.

Project	Description	Version	#Files	#Packages
Hadoop-core	Distributed computing framework	0.19	605	53
Apache-oodt	Distributed data management	0.2	1,017	226
ArchStudio	Architecture development tool	4	610	110

From the software projects, we generated the software representation models (topic-document matrices). Two types of representation models are then used to identify the impact of outlier removal: a base model and an outlier removal model. The former indicates original LDA data, and the latter indicates a model some outliers removed. For the outlier removal, the cut-off threshold is assumed to be from 0.1 to 0.6. For example, for a threshold of 0.1, 10% of documents that have low CC scores are excluded as outliers. A large threshold is not needed because an excessive loss of entities is meaningless in the recovery. We found that an overly large cut-off causes an abnormal or invalid recovery output—the number of remaining entities becomes less than the number of clusters. Therefore, we do not consider values over 0.6.

Next, in the clustering phase, to cover a broad spectrum of parameter configuration, we varied the number of clusters k from 10 to 100 by increments of 10. Furthermore, we increased the number of topics |T| from 10 to 100 to obtain the highest recovery output. The value of |T| for the highest recovery performance varies across projects, so it is desirable to find it experimentally.

For the final step, we evaluated the recovery results by measuring MoJoFM between the model and the ground-truth.

B. EXPERIMENTAL RESULTS AND ANALYSIS (RQ2 AND RQ3)

Table 6 presents the MoJoFM values of Hadoop-core recovery. Because the experiment produced a vast number of

TABLE 6. MoJoFM of recovery results in Hadoop-core (|T| = 80 is optimal).

#Clusters k	Base recovery	Outlier removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
10	13.6	18.8	17.9	19.0	21.2	21.0	**38.7**
20	19.4	17.7	22.7	27.3	26.8	29.0	**31.3**
30	16.3	22.3	22.0	25.5	19.7	29.0	**51.2**
40	20.9	20.6	18.2	23.3	26.7	33.1	**56.6**
50	26.2	21.9	26.2	27.4	34.8	40.5	**61.5**
60	13.2	18.4	24.9	27.0	38.5	18.6	**63.4**
70	15.9	28.2	24.1	22.2	37.2	62.2	**69.1**
80	23.0	26.9	31.5	38.8	48.5	52.5	**60.7**
90	44.4	29.3	24.2	**69.1**	53.9	66.3	61.1
100	30.4	27.0	**57.1**	28.0	34.7	37.3	44.1
Mean	22.3	23.1	26.9	30.8	34.2	39.0	**53.8**

outputs, only the optimal results are reported. The highest recovery performance was observed at |T| = 80 in Hadoop-core.

The optimal MoJoFM of the base recovery is 44.4% when |T| = 80 and k = 90. The bold number indicates the highest score for each removal case; the underlined number is the mean value. For each k, the outlier removal technique illustrates a gradual improvement over the base recovery. Although unstable cases are observed, such as MoJoFM falling below the previous level, the mean value increases linearly. We think these unstable changes in MojoFM are due to the variability of the clustering algorithm. This experiment case confirms that the recovery accuracy of Hadoop-core can be improved gradually through the removal of outliers. This trend was similarly observed in Apache-oodt. Table 7 is the result of Apache-oodt.

TABLE 7. MoJoFM in Apache-oodt (|T| = 100 is optimal).

#Clusters k	Base recovery	Outlier removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
10	6.3	6.1	6.9	7.0	6.5	8.3	**8.7**
20	6.9	7.4	8.0	9.3	10.0	8.4	**10.2**
30	5.8	7.5	9.3	9.8	10.0	11.5	**11.9**
40	5.8	13.6	9.8	10.4	11.8	**14.7**	13.2
50	8.3	7.1	10.8	11.7	15.6	12.9	**22.8**
60	10.2	7.28	10.3	11.0	13.8	15.1	**22.7**
70	8.9	22.1	12.0	18.7	**27.8**	19.0	26.7
80	7.6	8.1	9.2	18.7	20.9	**23.7**	22.0
90	6.5	10.3	24.1	17.0	19.8	**34.3**	17.5
100	27.0	11.6	17.7	19.2	23.6	23.3	**38.6**
Mean	9.3	10.1	11.9	13.3	16.0	17.1	**19.5**

In the Apache-oodt project, the mean value of MoJoFM increases slightly along with the application scale of the outlier removal. LDA-based recovery in Apache-oodt exhibits relatively low accuracy. The optimal MoJoFM is 27%, and the mean MoJoFM is 9.3% at the base recovery. However, regardless of poor performance, our outlier removal method increases MoJoFM linearly.

RQ2 is again presented to associate the results with the research questions, as follows:

RQ2 - Does removing outliers affect software architecture recovery?

In response to RQ2, the outlier removal method outperforms the base recovery. Furthermore, it has the effect of gradually increasing accuracy according to the removal ratio.

However, it is also necessary to verify whether these results are simply due to removing entities or removing real outliers. Therefore, we conducted additional experiments to confirm the relevance of the removal target. We randomly selected some source code and performed clustering without the selected files. Tables 8 and 9 illustrate the comparison between random removal and semantic outlier removal in Hadoop-core and Apache-oodt, respectively. Because the random removal method produces a different output each time, the mean MoJoFM was measured for more than five clustering results.

TABLE 8. Comparison of MoJoFM between semantic outliers removal and random removal in Hadoop-core.

Base recovery (mean)	removal target	removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
22.3	semantic outliers	23.1	26.9	30.8	34.2	39.0	**53.8**
	random choice	**18.8**	18.1	15.2	13.6	15.6	12.3

TABLE 9. Comparison of MoJoFM between semantic outliers removal and random removal in Apache-oodt.

Base recovery (mean)	removal target	removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
9.3	semantic outliers	10.1	11.9	13.3	16.0	17.1	**19.5**
	random choice	6.5	**7.1**	6.6	5.1	4.9	4.8

In both tables, the random choice does not have a particular trend; furthermore, accuracy is worse than in the base recovery. Accordingly, the removal of software entities does not have a positive effect on boosting recovery performance. Nevertheless, the semantic outlier case presents a linearly increasing pattern. Based on this observation, we can infer that exact selection is important. RQ3 is again presented, as follows:

RQ3 - How does the choice of the outlier affects the recovery accuracy?

In response to RQ3, the naïve removal of software entities cannot increase recovery accuracy but functions correctly if the correct outliers are selected and removed.

In contrast, the last open-source project exhibits considerably different results. Tables 10 and 11 present the recovery data from Archstudio. The recovery accuracy is high, such that the optimal MoJoFM at the base model reaches 80%, and the mean value is 52.6%. However, outlier removal does not exhibit a linear increase like those of Hadoop-core and Apache-oodt. At the 0.1 and 0.2 removal ratios, mean MoJoFM values increase temporarily, whereas the rest display an irregular flow.

TABLE 10. MoJoFM in Archstudio (|T| = 60 is optimal).

#Clusters k	Base recovery	Outlier removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
10	45.4	**45.7**	44.2	42.0	38.7	34.4	33.0
20	45.4	**55.6**	54.6	47.2	48.4	37.9	42.3
30	51.1	40.8	50.0	53.0	**55.6**	38.4	45.8
40	47.7	57.9	61.2	**71.3**	60.4	38.5	62.1
50	69.5	53.8	55.6	48.4	**72.3**	44.1	44.6
60	61.8	78.5	77.3	**84.5**	64.2	38.6	41.7
70	74.0	**81.1**	72.6	64.3	45.8	47.0	26.7
80	65.6	**71.2**	62.5	34.9	47.0	20.0	47.2
90	**80.2**	43.3	53.7	67.9	66.7	65.2	34.6
100	38.6	64.7	**67.3**	34.4	31.2	47.7	42.9
Mean	52.6	53.9	**54.5**	49.8	48.2	37.5	38.3.

The irregular flow is also similar to that of random choice, as depicted in Table 11. The MoJoFM values of both the semantic outliers and random case are very close. Unfortunately, in Archstudio, outlier removal techniques do not outperform base recovery as in the previous experiments.

TABLE 11. Comparison of MoJoFM between semantic outliers removal and random removal in Archstudio.

Base recovery (mean)	removal target	removal ratio					
		0.1	0.2	0.3	0.4	0.5	0.6
52.6	semantic outliers	53.9	**54.5**	49.8	48.2	37.5	38.3
	random choice	52.8	**55.5**	49.9	50.7	47.5	38.0

We observe the impact of outlier removal by presenting a mean MoJoFM graph of the three open-source projects in Figure 3. The results illustrate a linear increase and irregular flow observed in the experiments—a noticeable improvement in Hadoop-core accuracy, a slight increase in Apache-oodt, and an irregular decrease in Archstudio.

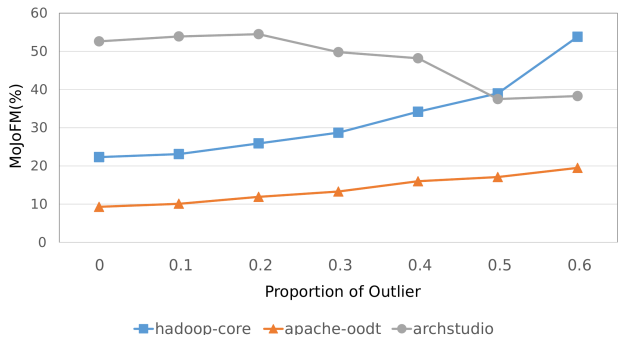

FIGURE 3. Mean MoJoFM variation of three open-sources.

We inferred the reason for the irregularity of Archstudio from the CC distribution. We realized that the CC distribution varies slightly across projects. In some cases, CC values are restricted within a narrow range; in other cases, they are spread across a wide range. Table 12 and Figure 4 illustrate this phenomenon.

In Figure 4, the upper graphs denote the increasing order of CC in each project, and the lower ones are histograms of each case. A similar number of files were used for both Hadoop-core and Archstudio, but their histogram shapes differ significantly. The CC distribution of Hadoop-core extends to a higher value than that of Archstudio. Moreover, in Table 12, the variance σ^2 of Archstudio is approximately three times larger than that of Hadoop-core. The data confirms that the CC of Archstudio is more widely distributed than other projects. In a case similar to Archstudio, outlier removal may be ineffective. Because many files have low CC values, it is difficult to determine which files are outliers precisely.

TABLE 12. Summary of CC value of three open-source projects.

project	# documents	mean	median	σ^2
Hadoop-core	605	0.841	0.848	0.011
Apache-oodt	1,017	0.771	0.785	0.018
Archstudio	610	0.803	0.638	**0.033**

Archstudio has the highest recovery performance among the three projects. The other two projects' maximum recovery accuracy results are 44.4% and 27.0% in base recovery, while Archstudio reaches 80.2%, corresponding to a very high value that is generally difficult to observe. Accordingly, the semantic information in Archstudio follows the ground-truth architecture very closely, indicating it is unnecessary to search for outliers. Consequently, our technique improves accuracy for projects with low semantic-based recovery.

VI. THREATS TO VALIDITY

For internal validity, parameter settings in the experiment do not cover all possible ranges. Although our experiments follow the work of [13], [37] and attempt to change the parameters, we cannot exclude that new observations and analyses can be derived if the parameters have a broader range and more granular steps.

For external validity, because this study is limited to open-source analysis, it may be a threat that our findings are generally applied to all software. We selected three open-source projects in which semantic-based recovery has been performed in previous studies. The number of selected projects is small because there are not enough ground truths and projects in which architecture recovery is performed by semantic analysis. We have selected projects within candidates addressed in related works, rather than using entirely new ones, to align with existing research.

Another threat is that open-source projects have high-quality text information because identifier-naming is relatively well managed. This feature of open projects can undermine our purpose to find semantic outliers. Nevertheless, we have found that outliers affect semantic analysis results in the experimental environment, which can be expected to be more applicable in lower-quality projects.

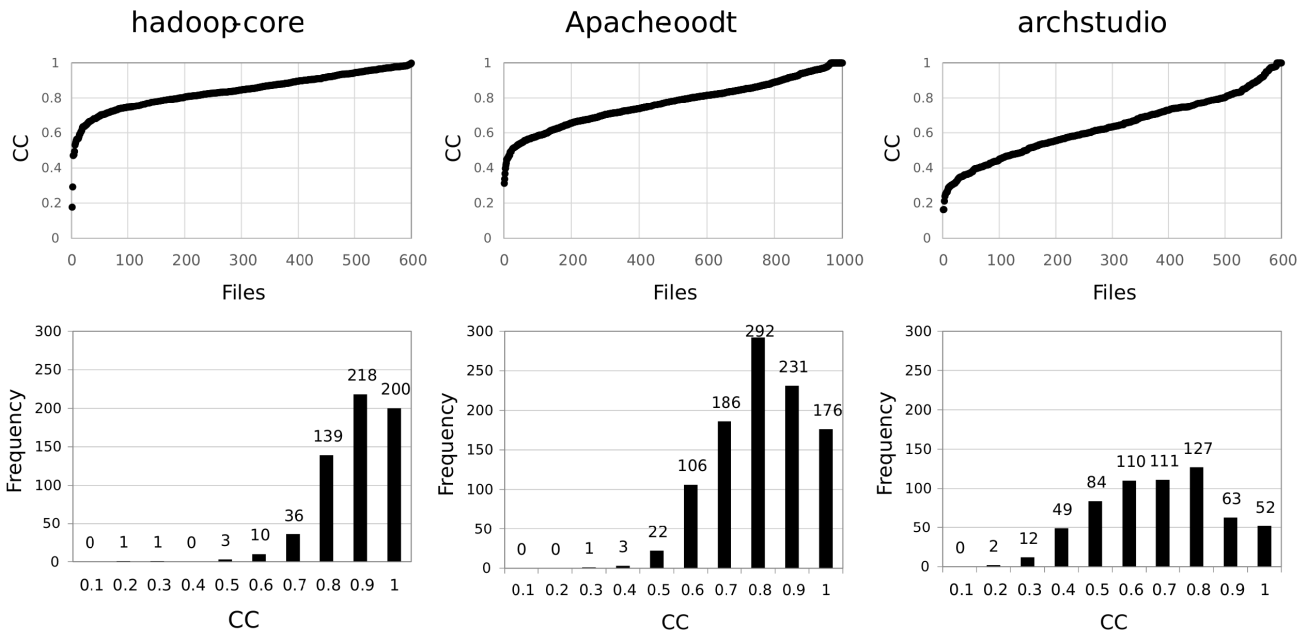

FIGURE 4. CC distribution of three open-source projects.

VII. CONCLUSION

Semantic-information-based architecture recovery provides the conceptual decomposition of a software system. However, the accuracy of the decomposition depends on the semantic quality of the source code. In this paper, we tried to improve recovery accuracy by excluding the semantic outliers. We proposed a software measure for identifying the outliers that quantifies the semantic conformity between the source code semantics and a component design concept. With this measure, the semantic outliers are identified and removed during architecture recovery. The experimental results revealed that the proposed outlier removal technique affects architecture recovery performance in open-source projects.

Our approach contributes to both the semantic quality measure and architecture recovery study. In the future, we plan to focus on architecture assessment using our software measure. We expect to use the proposed measure to assess the quality of software architecture semantically. We also expect it to contribute to the prediction of the defect proneness of software.

REFERENCES

[1] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Trans. Softw. Eng.*, early access, Feb. 25, 2019, doi: 10.1109/TSE.2019.2901468.

[2] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 486–496.

[3] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "A large-scale study of architectural evolution in open-source software systems," *Empirical Softw. Eng.*, vol. 22, no. 3, pp. 1146–1193, Jun. 2017.

[4] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2018, pp. 176–185.

[5] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2011, pp. 35–44.

[6] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[7] M. Risi, G. Scanniello, and G. Tortora, "Architecture recovery using latent semantic indexing and K-means: An empirical evaluation," in *Proc. 8th IEEE Int. Conf. Softw. Eng. Formal Methods*, Sep. 2010, pp. 103–112.

[8] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 552–555.

[9] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[10] D. Link, P. Behnamghader, R. Moazeni, and B. Boehm, "Recover and RELAX: Concern-oriented software architecture recovery for systems development and maintenance," in *Proc. IEEE/ACM Int. Conf. Softw. Syst. Processes (ICSSP)*, May 2019, pp. 64–73.

[11] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering architectural design decisions," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2018, pp. 95–104.

[12] E. Constantinou, G. Kakarontzas, and I. Stamelos, "An automated approach for noise identification to assist software architecture recovery techniques," *J. Syst. Softw.*, vol. 107, pp. 142–157, Sep. 2015.

[13] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 69–78.

[14] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Adv. Softw. Eng.*, vol. 2012, pp. 1–18, May 2012.

[15] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Aug./Sep. 1999, pp. 1–10.

[16] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[17] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock, "Spectral and meta-heuristic algorithms for software clustering," *J. Syst. Softw.*, vol. 77, no. 3, pp. 213–223, Sep. 2005.

[18] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proc. IEEE Work. Conf. Reverse Eng.*, Nov. 2000, pp. 258–267.
[19] O. Maqbool and H. A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Proc. Conf. Softw. Maintenance Re-Eng.*, 2004, pp. 15–24.
[20] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.
[21] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 150–165, Feb. 2005.
[22] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.
[23] V. Tzerpos and R. C. Holt, "MoJo: A distance metric for software clusterings," in *Proc. 6th Work. Conf. Reverse Eng.*, 1999, pp. 187–193.
[24] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.
[25] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, Mar. 1980.
[26] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2010, pp. 88–96.
[27] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent Dirichlet allocation," in *Proc. 1st Conf. India Softw. Eng. Conf. (ISEC)*, 2008, pp. 113–120.
[28] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent Dirichlet allocation," in *Proc. 15th Work. Conf. Reverse Eng.*, Oct. 2008, pp. 155–164.
[29] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent Dirichlet allocation for automatic categorization of software," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 163–166.
[30] J. Chang and D. M. Blei, "Hierarchical relational models for document networks," *Ann. Appl. Statist.*, vol. 4, no. 1, pp. 124–150, Mar. 2010.
[31] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, Mar. 2008.
[32] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 233–242.
[33] J. Lin, "Divergence measures based on the Shannon entropy," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 145–151, Jan. 1991.
[34] S. Kullback, *Information Theory and Statistics*. Mineola, NY, USA: Dover, 1968.
[35] *Apache Hadoop*. Accessed: Oct. 2020. [Online]. Available: http://hadoop.apache.org/
[36] *LDA: Collapsed Gibbs Sampling Methods for Topic Models*. Accessed: Oct. 2020. [Online]. Available: https://cran.r-project.org/web/packages/lda
[37] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. Nat. Acad. Sci. USA*, vol. 101, no. 1, pp. 5228–5235, Apr. 2004.
[38] E. Knorr and R. Ng, "Algorithms for mining distancebased outliers in large datasets," in *Proc. Conf. Very Large DataBases*, 1998, pp. 392–403.
[39] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2000, pp. 427–438.
[40] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar. 2011.
[41] *Snowball*. Accessed: Oct. 2020. [Online]. Available: http://snowball.tartarus.org
[42] O. Maimon and L. Rokach, "Clustering methods," in *Data Mining and Knowledge Discovery Handbook*. Boston, MA, USA: Springer, 2005, pp. 321–352.
[43] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 901–910.
[44] *Apache OODT*. Accessed: Oct. 2020. [Online]. Available: https://oodt.apache.org/
[45] *ArchStudio*. Accessed: Oct. 2020. [Online]. Available: http://isr.uci.edu/projects/archstudio/
[46] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proc. 12th IEEE Int. Workshop Program Comprehension*, Jun. 2004, pp. 194–203.

KI-SEONG LEE was born in Seoul, South Korea. He received the B.S. degree in korean literature in classical chinese from Sungkyunkwan University, Seoul, in 2005, and the M.S. and Ph.D. degrees in computer science and engineering from Chung-Ang University, Seoul, in 2011 and 2015, respectively. From 2006 to 2008, he was a Software Engineer with Internet Service Company, Seoul. Since 2017, he has been an Assistant Professor with the Da Vinci College of General Education, Chung-Ang University. His research interests include software architecture, machine learning, and natural language processing.

CHAN-GUN LEE was born in Seoul, South Korea, in 1972. He received the B.S. degree in computer engineering from Chung-Ang University, Seoul, in 1996, the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 1998, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2005. From 2005 to 2007, he was a Senior Software Engineer with Intel, Hillsboro, OR, USA. Since 2007, he has been a Professor with the Department of Computer Science and Engineering, Chung-Ang University. He is the author of more than 30 articles and conference papers. His research interests include software engineering and real-time systems. He was a recipient of the Korea Foundation of Advanced Studies (KFAS) Fellowship from 1999 to 2005.