


Reconstructing and evolving software architectures using a coordinated clustering framework

Sheikh Motahar Naim¹  · Kostadin Damevski² · M. Shahriar Hossain¹

Received: 6 November 2015 / Accepted: 30 January 2017 / Published online: 7 February 2017
© Springer Science+Business Media New York 2017

Abstract During a long maintenance period, software projects experience architectural erosion and drift, making maintenance tasks more challenging to perform for software engineers unfamiliar with the code base. This paper presents a framework that assists software engineers in recovering a software project’s architecture from its source code. The architectural recovery process is an iterative one that combines clustering based on contextual and structural information in the code base with incremental developer feedback. This process converges when the developer is satisfied with the proposed decomposition of the software, and, as an additional benefit, the framework becomes tuned to aid future evolution of the project. The paper provides both analytic and empirical evaluations of the obtained results; experimental results show a reasonably superior performance of our framework over alternative conventional methods. The proposed framework utilizes a novel compartmentalization technique Coordinated Clustering of Heterogeneous Datasets (CCHD) that relies on contextual and structural information in the code base, but, unlike most previous approaches, does not require specific weights for each information type, which allows it to adapt to different project types and domains.

✉ Sheikh Motahar Naim
snaim@miners.utep.edu

Kostadin Damevski
kdamevski@vcu.edu

M. Shahriar Hossain
mhossain@utep.edu

¹ Department of Computer Science, University of Texas, El Paso, TX 79968, USA

² Department of Computer Science, Virginia Commonwealth University, Richmond, VA 23284-3019, USA

Keywords Software architecture · Coordinated clustering · Heterogeneous data clustering · Architecture recovery

1 Introduction

Over time, software project architectures diverge from their original design and cease to follow their written documentation due to the dual effect of architectural erosion and drift (Taylor et al. 2009). Architectural recovery techniques can be used to prevent erosion and drift, by recommending restructuring opportunities (Bavota et al. 2013) to developers, or to treat these conditions, by recovering the architectures of software projects for which the architectural decomposition has been lost (Bauer and Trifu 2004).

Due to these and other applications, automatically extracting software architectures, usually by using a clustering algorithm on the code base, has been a long standing pursuit in the software engineering and software maintenance research communities (Shtern and Tzerpos 2012). While a number of approaches and tools have been proposed for architectural reconstruction, more research is needed for such tools to become integral part of the IDE and achieve wide application by developers.

Recently, a number of proposed software clustering techniques have relied on lexical information, found in identifier names and comments in the source code, as a means to perform software clustering, often combining this type of information with structural information available in the source code (e.g., method caller to callee relationships) (Scanniello and Marcus 2011; Misra et al. 2012; Bavota et al. 2013). In software clustering approaches that use both lexical and structural types of information, the distance function must combine both types of information extracted from the code base into a single metric, trading off the importance of one type of information for the other. However, the ratio of structural to lexical information is impossible to know at the outset and is dependent on a *project-by-project basis* analysis. This information weighing problem has also been outlined in a recent survey of software clustering performed by Shtern and Tzerpos (2012).

A recent comparative study of architectural recovery techniques by Garcia et al. (2013a) shows that even the best software architecture extraction algorithms cannot produce better than 50% average accuracy, compared to “ground truth” architectures derived from expert developer feedback. Our experience in a smaller scale experiment also indicates that extracting software architectures is a very subjective task, where the results vary significantly from developer to developer. Therefore, evaluations of architectural recovery techniques based on “ground truth” architectures can be subjective, and penalize logical decompositions that disagree with few developers’ opinion. The approach presented in this paper, follows a few other solutions to the architecture recovery problem that dynamically integrate developer opinion to adapt its clustering results (Christl et al. 2005; Koschke 2002). When used by experienced developers, such semi-automatic approaches can achieve improved reconstruction accuracy based on minimal, as-needed, developer feedback.

In this paper, we describe a framework based on our novel CCHD algorithm, short for *Coordinated Clustering of Heterogeneous Datasets*, which clusters a software project’s source code in order to discover its inherent architecture. Our framework

leverages three datasets extracted from a software project's source code: (1) text retrieved from comments and identifiers in the source code, (2) method (function) caller to callee relationships, commonly known as a call-graph, and (3) method to class relationships. In addition to automatically producing a coherent software architecture, the framework provides a mechanism to maintain the quality of the extracted architecture by placing newly developed code into appropriate architectural components. In summary, the contributions of this paper are as follows.

1. The architectural recovery framework seamlessly combines program structure with the natural language context of the code. The two types of data (lexical and structural) leveraged by our CCHD technique complement each other in describing relationships between program elements in the source code. Unlike prior work that views structural and lexical information as integrated data, we present a clustering technique that allows two heterogeneous datasets (e.g., a call graph and a lexical dataset) to be partitioned simultaneously using relationships between the two without explicitly weighing their importance levels in the distance metric.
2. The architectural recovery framework is flexible. Along with automatic placement of new code, the framework allows the user to refine an existing architecture when necessary.
3. We provide a broad range of evaluations to verify if the framework meets the needs of software engineers. The evaluations include information theoretic measures, conventional compartmentalization assessment techniques, as well as empirical justifications by professional software engineers.

The recent surge in the use of a combination of lexical and structural information (Bavota et al. 2013; Bauer and Trifu 2004; Shtern and Tzerpos 2012; Scanniello and Marcus 2011; Misra et al. 2012) to solve the architectural decomposition problem provides an indication of the growing importance of software architecture analysis tools. Our approach addresses the problem of combining these two types of information in a novel way. It also addresses a larger question present in many software maintenance tools: how to smoothly and seamlessly incorporate contextual information found in the program identifiers with structural information exhibited by the code? This problem is present in a variety of software maintenance tools, including feature location, program comprehension and software testing, all of which are of great relevance to industrial applications with a strong software emphasis.

The rest of the paper is organized as follows. Section 2 reviews the literature related to this work from different perspectives. We provide a formal description of the problem we are trying to solve in Sect. 3. Section 4 describes the process of preparing the raw code base for further analysis. Section 5 describes the proposed CCHD framework in details. Different evaluation metrics used in our experiments are defined in Sect. 6. We describe the experimental set-up and present the results in Sect. 7. Finally, Sect. 8 discusses the implications of the experimental results.

2 Related work

Our solution to the problem of reconstructing software architecture involves extracting lexical and structural information of a software code base. It generates two heteroge-

neous datasets and the relationships between them from a single code base. Then a new architecture for the code base is obtained by clustering the datasets simultaneously using an information theory based algorithm. Initial clustering results are refined using the feedback of the users in an iterative fashion to finally achieve a stable structure. In this section, we review the literature from these different aspects of our solution framework.

2.1 Software clustering

Some previous research that relies on both lexical and call graph information to perform software clustering are (Garcia et al. 2011; Bavota et al. 2013; Misra et al. 2012; Scanniello and Marcus 2011). Bavota et al. (2013) propose a weighted combination of two metrics that expresses the strength of the lexical and structural relationship between two classes to modularize (or package) a software. In a similar approach, Misra et al. (2012) uses a few more parameters to reflect both lexical similarity (e.g. between class names, method names, or just plain program text), as well as structural relationships (e.g. caller-callee, inheritance). Scanniello and Marcus (2011) leverage graph partitioning techniques where the call graph edges are weighed based on shared semantic information between two methods. We claim that the weakness of each of these approaches are the arbitrary sets of weights applied to each class of information (or metric) to compute similarity between two program elements. Instead of combining the call graph and lexical information in one dataset, we keep the information separate and consider them relational data. Our approach co-clusters lexical and structural information simultaneously without resorting to choosing a set of constants to weigh the importance levels of the structural and the lexical information.

Recent work by Corazza et al. (2016) consider lexical information within the source code but ignore the connectivity between them. Moreover, they use a probabilistic method to weigh different code elements before clustering them whereas our approach does not require any such pre-weighting. Though Zhu et al. (2013) consider both similarity and connectivity between program elements, they have to adjust the contributions of each of these factors using a random walk model. Praditwong et al. (2011) use a multi-objective optimization based approach to find software modules of high cohesion and low coupling. One of their objectives is to find nearly equal-sized clusters, which we do not agree with due to our assumption that, naturally, there should be modules of greatly different sizes in many software code base. This assumption is based on the ground truth information we have for multiple code bases, labeled by expert developers. For example, in the Hadoop data set, the five largest clusters include more than 54% of the total classes, while the total number of clusters is 67. Similar characteristics were found in the Apache OODT code base.

Design rules clustering, proposed by Cai et al. (2013), augments architectural recovery techniques with the notion that certain design decisions (e.g. exhibited by inheritance hierarchies) are fundamental to the software's architecture and should be given precedence over other, typical measures of cohesion and coupling. While we have not experimented with the types of design rules described in their paper, we believe that integrating them into a mixed-influence framework such as ours may be one way of effectively applying them to a wide variety of code bases.

2.2 Software clustering improvement via software engineers' feedback

Semi-automatic software clustering, which integrates human feedback into an automated technique, has previously been studied by a few researchers (e.g. [Bavota et al. 2012](#); [Koschke 2002](#); [Christl et al. 2005](#)). The strength of these types of approaches in finding solutions that are close to developer opinions is also their weakness: poor feedback can lead such algorithms to poor architectural reconstruction. We believe that this weakness can be addressed by better project planning, i.e. by selecting the most experienced developer or group of developers to guide the architecture recovery process in order to produce a good result.

[Bavota et al. \(2012\)](#) considered a software clustering approach where the initial result is improved via feedback from software engineers, an aspect which is similar to the one described in this paper. However, our software clustering technique differs in several ways, including the clustering algorithm, the objective function, and the way in which developer feedback is incorporated.

[Christl et al. \(2005\)](#) used semi-automatic software clustering that integrates developer feedback for reflexion analysis, which maps software architectures extracted from code into a hypothesized conceptual architecture. On one example application, the necessary feedback was very small and the algorithm produced very high mapping correctness. The authors do not consider evolving the code after the mapping is complete, as our approach does, while the clustering algorithm relies only on structural dependencies in the code.

2.3 Heterogeneous data clustering

In the literature, research on heterogeneous clustering appears in different forms. Some researchers focus on clustering based on heterogeneous features ([Yang and Zhou 2006](#); [Yoon et al. 2006](#)) to partition the data into multiple views and then combine the results in a systematic way. There is also a surge in the use of transfer of knowledge from one domain to another ([Dai et al. 2007](#); [Hossain et al. 2014](#)). Some of the techniques to cluster heterogeneous datasets depend on the concurrent influence of the clusterings on each other based on implicit or explicit relationships between the datasets ([Hossain et al. 2010](#); [Momtazpour et al. 2012](#)). [Gao et al. \(2005\)](#) propose an algorithm to cluster heterogeneous objects of two types—a central type and objects connected to the central type of objects. The solution leverages a combination of pairwise co-clustering applied on subdivided problems. In this paper, we propose a technique that clusters relational heterogeneous data (e.g., a graph and a collection of text documents) simultaneously without leveraging feature level heterogeneity.

2.4 Information theoretic approaches to co-clustering

Several clustering algorithms use information theoretic formulations to cluster objects and features simultaneously ([Gokcay and Principe 2002](#); [Dhillon and Guan 2003](#); [Böhm et al. 2006](#)). [Dhillon et al.](#) presents an algorithm ([Dhillon et al. 2003](#)) for simultaneously clustering the rows and the columns of a contingency table of data.

The method views the contingency table as an empirical joint probability distribution of two discrete random variables and finds a clustering result that maximizes the mutual information between the cluster random variables. Banerjee et al. (2004) propose a solution to the co-clustering problem using the minimum Bregman information (MBI) principle that simultaneously generalizes the maximum entropy. Our proposed CCHD algorithm has resemblance with the mathematical machinery of the co-clustering of Dhillon (2001) but it has broader functionality in terms of heterogeneity.

3 Problem description

3.1 Formalism

In modern object-oriented programming languages code is organized in classes,¹ where commonly one class is stored in one file. The set of classes, $C = \{c_1, c_2, \dots, c_{n_c}\}$, form the code base of a software project. Developers may provide comments in the code for readability and to aid future modifications as a part of their standard software development practice. Comments and thoughtfully declared identifiers (e.g., variable names and method/function names) can create a rich natural language vocabulary, $T = \{t_1, t_2, \dots, t_{n_t}\}$, where t_i is referred as the i th term of vocabulary T . Classes consist of a set of methods/functions, $M = \{m_1, m_2, \dots, m_{n_m}\}$, where M is the set of all methods in all classes of a specific software project. That is, each method, m_k of M , is associated with a unique identifier that combines a class name with a method name.

C , T , and M have complex relationships between them. For example, there is an inherent bipartite relationship between the terms in T and the classes in C since each term can be associated with multiple classes. Classes commonly contain significant numbers of terms, which makes this relationship more interesting than that of T and M . These term-class relationships can be expressed as: $\mathcal{S} = \{(t_i, c_j) : t_i \in T, c_j \in C\}$.

A second set of relations can be formed using the caller-callee relationships of the methods. Each method of a class can call itself or other methods in the same class or in another class. A call graph $\mathcal{G}(M', E)$ represents these relationships where each method $m \in M' \subseteq M$. M' is the set of all methods that are categorized either as callers or callees. The graph \mathcal{G} is an undirected graph based on the assumption that if one method calls another method, both the methods are expected to have strong relationship and should be in the same cluster, irrespective of the direction of the edge. The set disjunction, $M - M'$ is the set of isolated nodes. An isolated method that is neither a caller nor a callee does not impact the software architecture since it will never execute, and therefore is external to the functionality of the software. As a result, it is redundant to keep these isolated methods in the call-graph. In the rest of the paper, for simplicity, when we use M we actually refer to M' . Each edge $(m_i, m_j) \in E$ represents a method call either from m_i to m_j or from m_j to m_i .

¹ In this paper, we use *class* to refer to the programming language context of the word, rather than to a collection or category.

The third set of relationships appears from the existence of the methods within specific classes. Every method in the code base belongs to a particular class and most classes contains one or more methods. We represent the method-class relationships as a bipartite graph $\mathcal{R} = (C, M, R)$ where C and M are two sets of vertices representing the classes and the methods respectively, and $R = \{\{c_i, m_j\} : c_i \in C, m_j \in M\}$ is the set of relationships.

Now the problem of extracting a good architecture of a software project can be expressed as developing the function $\Omega : \{C, T, M\} \rightarrow \mathcal{P}$ which maps three parameters C , T and M into clusters of class-files \mathcal{P} using the relationships \mathcal{S} , \mathcal{G} and \mathcal{R} . Ω 's objective is to extract the inherent software architecture with limited or no feedback from the software engineers. In addition, such a function has the capability to determine the 'proper' location for any newly written (or modified) methods or classes so that architectural drift can be prohibited in the future.

3.2 The proposed framework

Our solution to the described problem comprises of three stages: (1) capturing the caller-callee relationships of the methods of a software code base to construct the call-graph \mathcal{G} , extracting relational dependence \mathcal{R} between the methods and the classes, and constructing term-class relation \mathcal{S} for the class files; (2) incorporating call-graph \mathcal{G} , method-class relation \mathcal{R} , and term-class relation \mathcal{S} to discover the architecture of a code base; and (3) automatic placement of newly written or modified code into the extracted architecture. Figure 1 shows our software architectural recovery framework. The framework applies a series of data extraction techniques and preprocessing to construct \mathcal{S} , \mathcal{G} and \mathcal{R} . Section 4 provides a detailed description of the techniques utilized in this stage. The call-graph \mathcal{G} is represented by an adjacency matrix, the term-class relationships \mathcal{S} are converted to a vector space model, and method class relationships are stored in a binary matrix. The next stage combines these three heterogeneous data sources and applies our coordinated clustering mechanism (CCHD) to produce a compartmentalization of the codes that reflects the relationships among the heterogeneous data sets. We explain this process in Sect. 5. After the clusters are obtained, we characterize each of them by selecting representative terms in a systematic and automated way, perform cluster enrichment to distinguish our results from a vanilla clustering algorithm that does not take the relationship between these heterogeneous datasets into account.

Based on the characterization and enrichment results discovered in the second stage, the software engineer can apply her/his knowledge and provide feedback to update some of the results. After adjusting the results the framework trains a classifier for automatic placement of new code in this software architecture. CCHD framework has the ability to categorize both new classes and new methods since it produces clusters for classes, methods and terms. However, in this paper, we build a classifier for categorizing only new classes since the number of methods is too large. Huge number of methods would make the task of evaluation cumbersome for the developers. A software engineer can iteratively provide feedback and modify the architecture as required. In practice, this cycle can iterate a few times until the software engineers

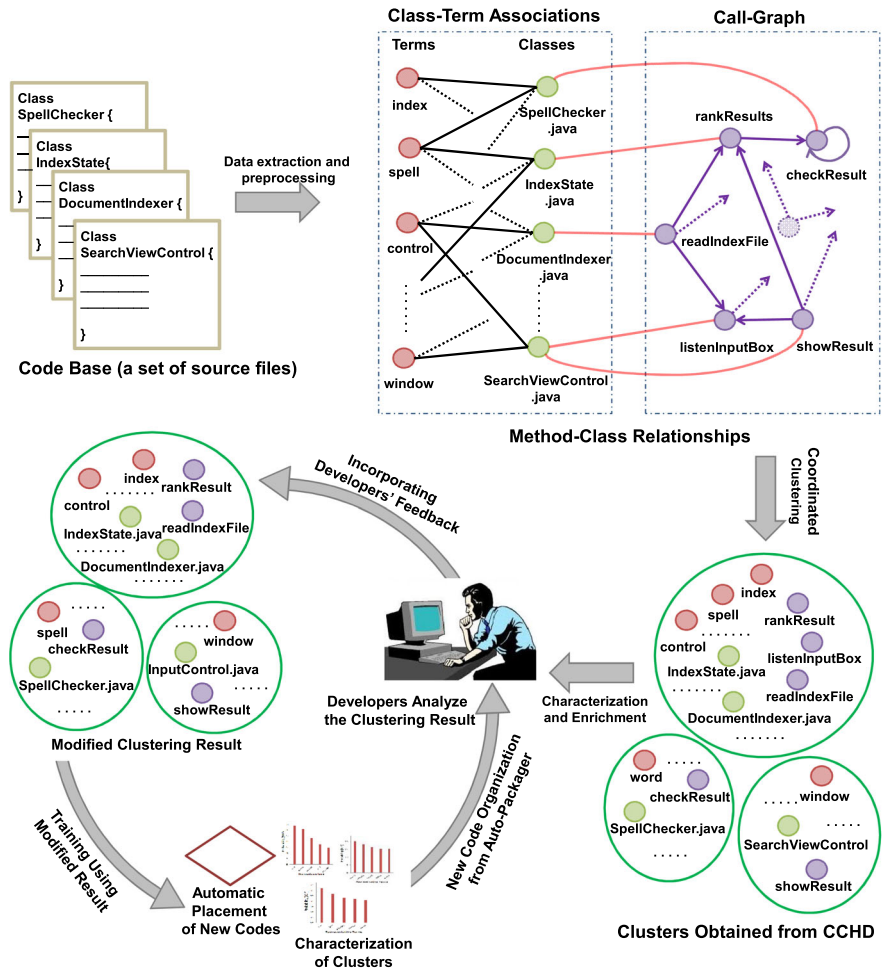


Fig. 1 Framework of the complete system

are completely satisfied with the architecture and there are no more suggestions for change.

3.3 Illustrative example

In the following we show a brief and very simplified example of how CCHD operates. For this purpose, we consider a subsystem of three classes—SpellChecker, IndexState and DocumentIndexer—extracted from the code base of the Sando code search tool, shown in Fig. 1. Each class contains exactly one method, e.g. class SpellChecker contains the method checkResult. One component of CCHD’s input is exactly this method to class membership. Another input to CCHD is the relationships between the

system's classes and natural language terms. Strength of a relationship between a class and a term depends on how many times that term appears in that class file compared to other class files. Exact formula to compute this strength is discussed in the next section.

In Fig. 1, we observe that some of the methods are connected in the call graph, i.e. a method calls another method. We can use this relationship as a bidirectional, binary indication of a relationship between each two methods. This example, so far, encompasses the three relationships that CCHD requires: (1) methods to classes, where each method has a containing class; (2) classes to terms, where frequency of terms can be used; and (3) methods to methods, where the call graph is used. CCHD optimizes a cost function that combines all three of these relationships into a decomposition of classes, e.g. classes `DocumentIndexer` and `IndexState`, were placed into one group, while class `SpellChecker` was placed in another group.

Once CCHD has completed the initial assignment of classes to groups (or clusters), the developer is asked to consider the choices the algorithm made. If, for instance, the developer indicates that a better grouping is between classes `IndexState` and `SpellChecker`, and places these two classes in one group, leaving class `DocumentIndexer` in the other, CCHD learns from this feedback, creating an automated classifier for each of the groups. So, when a new class d is introduced in the system (or if the existing classes were modified in a way that their relationship strength is affected), CCHD will act to automatically reorganize the system's architecture. As the software system evolves, CCHD incorporates developers' feedback at each step, using it to tune the way it evolves the decomposition of this particular software system.

4 Data preparation

4.1 Data extraction

Our architectural recovery framework directly relies on three type of relationships—class-lexical terms, method-class, and callee method-caller method relationships. The effect of other types of connections between these types of data, such as method-lexical terms, is expressed indirectly in the system via the above three, i.e. by the combination of class-terms and method-class relationship. In the following section, we describe the way we extract these three types of relationships from a raw software codebase. All of the six real world codebases we used in our experimental evaluations (Sect. 7) were synthesized following this process.

To extract the list of methods and classes as well as a call graph containing the invocation relationship between methods we rely on the SrcML.NET code analysis framework.² SrcML.NET constructs the list of method-class memberships and a method-method call graph using lightweight program analysis techniques, while also building an XML representation of each method's inner syntactic structure. This XML is used to construct the other type of relationships i.e. class-term relationships required by the CCHD framework. The XML representation provides a convenient means to

² <https://github.com/abb-iss/SrcML.NET>.

extract the identifiers (variable names, method and class names) and comments from each method. Javadoc-style comments, located directly above a method in some software projects, are also parsed and inserted into each method’s term list. The term list of a class comprises of all the terms of its methods as well as any other identifiers or comments within its definition. Since SrcML.NET supports several programming languages, including C,C++,C#, and Java, the data extraction routines are able to rapidly obtain lexical and structural information from a variety of software projects. In the paper’s companion website,³ we have provided the raw codebase of an open source software proeject (Sando v1.7) and the corresponding relationships extracted by applying this synthesis process.

4.2 Preprocessing

Several preprocessing steps are commonly applied to lexical data in order to remove spurious matches based on common words in the language (e.g. “the”, “is”, “at”) or common words in a particular software project, while also providing the ability to match words which are semantically but not syntactically similar (e.g. “parse” and “parsing”). Our goal in choosing from a variety of such preprocessing mechanisms was to achieve the above mentioned goals in a straightforward way, without imposing processing pipeline bulk that is unlikely to have widespread benefits.

We use vector space modeling (Manning et al. 2008) to represent the natural language terms in each class in the source code, after removing the stop words and numerals and applying Porter stemming. Each term t of class c is weighted as

$$w_{t,c} = \frac{(1 + \log(tf_{t,c}))(\log \frac{N}{df_t})}{\sqrt{\sum_{j=1}^{n_c} ((1 + \log(tf_{j,c}))(\log \frac{N}{df_j}))^2}} \quad (1)$$

where $tf_{t,c}$ is the frequency of term t in class file c , df_t is the number of class-files containing term t , n_c is the number of terms in class c , and N is the total number of class files. The above equation is a variant of tf-idf modeling with cosine normalization. The comments and the identifiers of the class files of a code base differ in size. In general, longer classes have higher term frequencies because many terms are repeated. The cosine normalization helps lessen the impact of size of the class files in the vector space modeling.

Some methods in the constructed call graph \mathcal{G} may be isolated since they do not have either a caller or a callee relationship. We eliminate these isolated methods from the call graph, as they do not carry any useful relational information.

5 Coordinated clustering of heterogeneous datasets

In the core of the CCHD framework we use a graph clustering algorithm named spectral clustering (see Sect. 5.2) that tries to bring closely connected nodes of a graph in the

³ <http://vcu-swim-lab.github.io/cchd>.

same cluster so as to minimize the weights of the edges across clusters. Motivated by the nature of spectral clustering algorithm, we extract different entities, namely classes, methods and terms, and their relationships from a code base (described in Sect. 4), and represent them in a heterogeneous graph so that relevant entities are grouped in the same cluster. Notice that spectral clustering is usually used on homogeneous graph where all the nodes are of same type. Though our dataset contains variety of entities and relations, we are still able to represent them in a single graph maintaining all the graph properties. This enables us to simultaneously cluster terms, classes, and methods. Many clustering frameworks (Bae and Bailey 2006; Basu et al. 2008) suffer from the necessity of expensive post-processing steps to relate different types of entities after discovering homogeneous clusters. CCHD overcomes the necessity of those post-processing steps through the use of an Eigenvalue method.

5.1 Unification of heterogeneous information

After the data extraction and the preprocessing steps on the original code base, we have three datasets: call graph \mathcal{G} , method-class associations \mathcal{R} , and class-term bipartite relationships \mathcal{S} . We situate these three datasets on a common footing to be able to perform simultaneous clustering of all of them.

We use an $n_t \times n_c$ matrix S to store the vector space representation of the class-term associations \mathcal{S} . $S(t, c) = w_{t,c}$ records the weight of the t th term of the c th class file (computed using Eq. 1). We build an $(n_t + n_c) \times (n_t + n_c)$ adjacency matrix \mathcal{W}_1 for the weighted bipartite relationships of S :

$$\mathcal{W}_1 = \begin{bmatrix} 0 & S \\ S^T & 0 \end{bmatrix} \quad (2)$$

Similarly, we build an adjacency matrix \mathcal{W}_2 for the call graph \mathcal{G} :

$$\mathcal{W}_2(i, j) = \begin{cases} 1, & \text{if method } m_i \text{ calls method } m_j \text{ or vice versa} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

Now, we merge \mathcal{W}_1 and \mathcal{W}_2 to obtain a combined adjacency matrix \mathcal{W} .

$$\mathcal{W} = \begin{bmatrix} \mathcal{W}_1 & 0 \\ 0 & \mathcal{W}_2 \end{bmatrix} = \begin{bmatrix} 0 & S & 0 \\ S^T & 0 & 0 \\ 0 & 0 & \mathcal{W}_2 \end{bmatrix} \quad (4)$$

\mathcal{W} does not capture the relationships between \mathcal{W}_1 and \mathcal{W}_2 . We bridge \mathcal{W}_1 and \mathcal{W}_2 by putting the associations from method-class relationships \mathcal{R} in the following way:

$$\mathcal{W}(n_t + i, n_t + n_c + j) = \begin{cases} 1, & \text{if class } c_i \text{ includes method } m_j \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

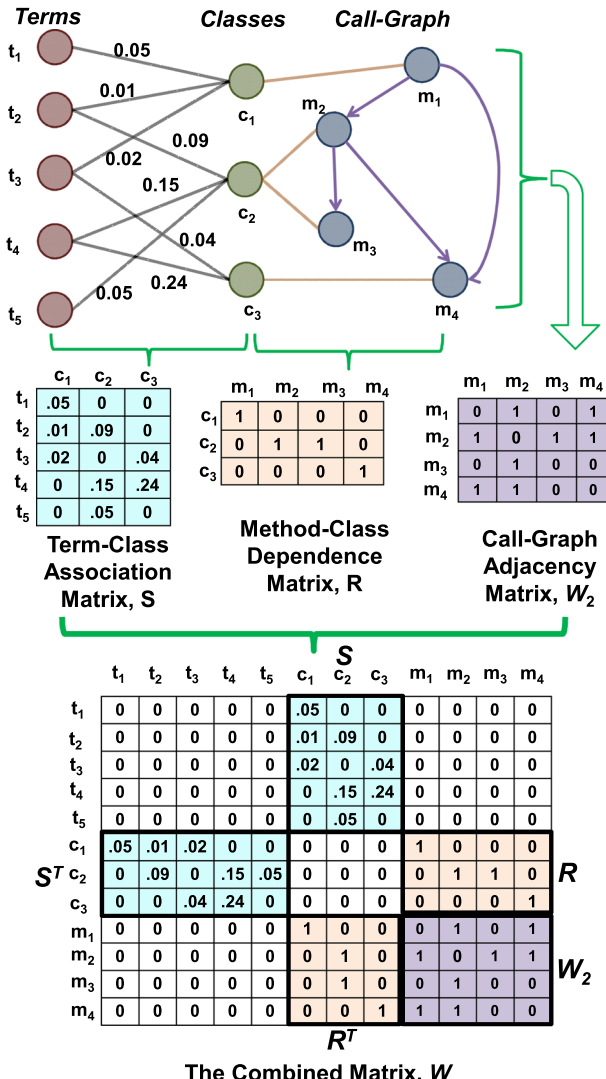


Fig. 2 Unification of heterogeneous datasets and their relationships

As a result, we have all three data sets within matrix W .

$$W = \begin{bmatrix} 0 & S & 0 \\ S^T & 0 & R \\ 0 & R^T & W_2 \end{bmatrix} \tag{6}$$

Figure 2 demonstrates the steps of unifying the three heterogeneous datasets with an example with three classes, five terms and four methods. The combined matrix, W is somewhat similar to the Design Structure Matrix (DSM) proposed by Cai et al. (2011).

However, while the basic design structure matrix focuses on dependencies only, our CCHD architecture is able to leverage vectors for nodes (e.g., classes), weights for edges (e.g., term-class weights), and binary relations (e.g., call-graph relationships). Such representation of the matrix enables us to accommodate all the entities extracted from the code base and at the same time retain their mutual relationships. That is why it produces meaningful results after being fed to spectral clustering algorithm (ALGORITHM 1).

5.2 Partitioning the code base data

We partition the combined data \mathcal{W} in such a way that connected methods, class files, and relevant terms appears together in one group. We utilize graph Laplacian theory in this space. In our algorithm, we utilized the unnormalized graph Laplacian matrix (Luxburg 2007) for partitioning our datasets. The unnormalized graph Laplacian matrix is given as:

$$L = D - \mathcal{W} \quad (7)$$

where D is the diagonal matrix defined as

$$d_i = \sum_{j=1}^n \mathcal{W}(i, j) \quad (8)$$

Our algorithm focuses on the following two properties of the unnormalized graph Laplacian matrix.

- The smallest eigenvalue of L is 0, the corresponding eigenvector is the constant one vector 1.
- L has n non-negative, real-valued eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$.

An overview of many of its properties can be found in Mohar and Alavi (1991), Mohar (1997). We use Shi and Malik (2000) normalized spectral clustering algorithm to partition classes C , terms T , and methods M all packaged together in \mathcal{W} . Algorithm 1 outlines the spectral clustering we use for our software architectural recovery framework. The exact number of clusters in a codebase is very subjective—the developers themselves usually have different opinions—and hence difficult to know beforehand. The general spectral clustering algorithm takes a fixed number of clusters as an input but in our framework we run spectral clustering multiple times with different number of clusters. We report the experimental results in Sect. 7 for different values of k (Figs. 4, 5, 6), and from those results we attempt to find a suitable value of k for a particular code base. Although there are some quantitative measures like Average Silhouette Coefficient (Struyf et al. 1997) and Dunn Index (Dunn 1973), which use inter-cluster distances and intra-cluster distances to determine the best value for k , they are not guaranteed to give exact results that will satisfy a subject matter expert. Though the clustering algorithm described above is a general purpose one, it suits very well with our purpose of clustering classes, methods and terms simultaneously. We

represent class-term relationships, call graph and method-class relationships of a code base in a heterogeneous graph and apply the spectral clustering algorithm to obtain heterogeneous clusters. Inside ALGORITHM 1, the column-wise Eigen vectors compose the nodes relevant to classes, methods and terms. That is, each row represents a vector for either a class, or a method, or a term.

Algorithm 1: Pseudocode for spectral clustering.

Data: The combined matrix obtained from Eq. 6 containing all the code elements and their relationships, \mathcal{W} and the number of clusters k

Result: Cluster assignment for each row of \mathcal{W} .

Steps:

1. Compute the diagonal matrix D , using Eq. 8.
 2. Compute the unnormalized Laplacian $L = D - \mathcal{W}$.
 3. Compute the first k generalized eigenvectors u_1, \dots, u_k of the generalized eigenvalue problem $Lu = \lambda Du$. Package the eigenvectors u_1, \dots, u_k as columns into a $n \times k$ matrix \mathcal{X} where n is the total number of rows in \mathcal{W} . Notice that each row of \mathcal{X} is still representing a class, or term, or method.
 4. Apply k -means clustering on \mathcal{X} to find k clusters.
 5. Return cluster assignment for each row of \mathcal{X} .
-

The result obtained from our coordinated clustering algorithm is unique in nature because it combines methods, terms, and classes through an amalgamation of the natural language context and the existing structural usage pattern of the methods. This provides a primary architecture of the code base to the software developer, which can be modified through subsequent passes as illustrated by the loop in Fig. 1.

5.3 Architectural refinement via developer feedback

We expect that a developer (or software architect) feedback is required in order to tailor the initial clustering result into the most appropriate architectural decomposition. In soliciting feedback, our framework adopts two goals: (1) provide context by characterizing each of the clusters using representative natural language terms; and (2) minimize the number of program elements that require input from the developer.

To achieve the first goal, we show an list of terms for each cluster, ordered in ascending order of their distances from the corresponding cluster prototypes. To achieve the second goal, apart from showing a similarly ordered list of program elements for each cluster, we highlight the difference between the previous and current architectural decomposition (i.e. what program elements were added or removed). This provides enough information to the developer on the progress of the algorithm as it iteratively incorporates their feedback.

At the outset of the iterative architectural decomposition process, we use simple k -means clustering as the prior decomposition on which CCHD performs its initial clustering. To map a CCHD cluster to a k -means cluster, we compare a CCHD cluster with all the k -means clusters and select the most similar one for the mapping. We used three similarity measures for this mapping—hyper geometric distribution (Berkopec

2007), number of common pair of classes between two clusters, and number of common classes between two clusters.

To provide feedback the developers mark the program elements first answer whether the specific set of methods, classes, and terms comprise an individual cluster, and if yes, then they mark the elements that do not belong. This is forms the input of a Naive Bayes classifier which becomes trained on the features of each cluster. Naive Bayes is a relatively simple classification technique, which is computationally efficient, and amenable to this specific problem.

After the training process is complete, an added benefit is that as new code is written in the software project, the system will automatically organize it in the existing architecture. This can ensure that the system, from that point on, does not stray from the selected architectural decomposition.

6 Evaluation plan

In this section, we provide descriptions of the evaluation techniques we used to assess the outcome of our framework. We used a wide spectrum of evaluation techniques—from cluster quality measurement to computing information theoretic mutual dependence between heterogeneous components to empirical assessments. We provide the results of these evaluations next, in Sect. 7.

6.1 Evaluation datasets

We use seven different open source software projects, Sando (v. 1.7), Apache httpd (v. 2.0), JEdit (v. 5.1.0), Apache OODT (v. 0.2), Hadoop (v. 0.19.0), ArchStudio (v. 4) and ITK (v. 4.5.2) to evaluate our approach. Table 1 summarizes the characteristics of these datasets. We use only the Sando dataset for the empirical (human subject) evaluation, as we were able to recruit some of the primary developers as participants. We use only the Hadoop, OODT, ArchStudio and ITK datasets for comparative evaluation, as those as where both “ground truth” results and previous benchmarks for other architectural recovery techniques are also available (Garcia et al. 2013a). In some of the other evaluation categories where we evaluate a parameter or detailed result of our

Table 1 Summary of the evaluation datasets

Dataset	Lines of code	Classes	Terms	Methods	Method-class relations	Call graph edges
Sando	20K	142	2687	819	841	414
httpd	60K	366	12,994	3488	3495	4365
JEdit	100K	830	14,514	6878	7014	18,598
OODT	180K	940	11,023	11,271	5812	15,310
Hadoop	200K	2895	25,382	31,300	17,199	49,812
ArchStudio	280K	1854	19,691	7285	6015	10,010
ITK	1M	989	11,608	5205	4143	4799

technique, for clarity, we select and focus our discussion only on one or a few of the datasets.

CCHD's initial clustering output for the above systems, for a fixed number of clusters, is provided in a companion website⁴ to this paper.

6.2 Evaluation for relations clustering

CCHD utilizes the relationships in \mathcal{R} to cluster the methods in \mathcal{G} and the class files in \mathcal{S} . One endpoint of a relationship in \mathcal{R} is associated with a method while the other endpoint is associated with a class. In an ideal architecture, the endpoints of each relationship of \mathcal{R} should be in the same cluster. In the worst case, with an arbitrarily constructed architecture, all endpoints of the relationships tend to scatter in multiple clusters forming a uniform distribution of the relationships over the clusters. One measure to evaluate the relational nature of the methods and the class files across clusters is the divergence of a clustering from its possible worst case scenario. We build a k -by- k contingency matrix B_r to capture the overlap of clustering agreements between two endpoints of the relationships. We record the percentage of relationships having one end in a class file cluster i and the other end in a method cluster j in $B_r(i, j)$. B_r captures the distribution of the class file clusters in the method clustering and each row sums up to 1. Similarly, we build another k -by- k contingency matrix B_c which represents the distribution of the method clusters in the class file clustering. The columns of B_c sums up to 1. In an ideal case, both B_r and B_c will constitute diagonal matrices since the endpoints of each relationship will be assigned to the same cluster.

$$B_r(i, j) = \frac{N_R(C_i, M_j)}{\sum_{j'=1}^k N_R(C_i, M_{j'})} \quad (9)$$

$$B_c(i, j) = \frac{N_R(C_i, M_j)}{\sum_{i'=1}^k N_R(C_{i'}, M_j)} \quad (10)$$

The function $N_R(C_i, M_j)$ returns the number of relationships between a class file cluster C_i and a method cluster M_j .

Since each row of B_r sums up to 1 it is possible to compute Kullback–Leibler (KL) divergence, $\delta_r(i)$, between the i th row and a corresponding uniform distribution $U(\frac{1}{k})$. The KL divergence is an information theoretic metric for the difference between two probability distributions. The uniform vector $U(\frac{1}{k})$ represents lack of any architecture where relationships are scattered arbitrarily in all clusters. Therefore, $\delta_r(i)$ is a measure of how well the i th cluster of the class file clustering is distributed across the method clusters.

$$\delta_r(i) = KL-Div \left(B_r(i, :), U \left(\frac{1}{k} \right) \right) \quad (11)$$

⁴ <http://vcu-swim-lab.github.io/cchd>.

Similarly, we can compute the divergence of each column of B_c with the corresponding uniform distribution.

$$\delta_c(j) = KL-Div \left(B_c(:, j), U \left(\frac{1}{k} \right) \right) \quad (12)$$

We use median of all the $2k$ KL-Divergence values (for k rows of B_r and k columns of B_c) as our overall measure of relational architecture evaluation. Higher median KL-Divergence indicates better relationship between class file clustering and method clustering, i.e., methods are well placed in the class files. Section 7.2 presents detailed experimental results for this evaluation.

6.3 Mutual information

Another means of evaluating our software clustering algorithm is by considering how well the resulting clusters capture the relationship between methods and classes within a single cluster; a poor clustering would contain many methods that are clustered separately from their containing classes. We measure the information shared by the methods in one side of the relationships in R and the class files in the other side using mutual information (MI) (Chaitin 1982).

$$MI = \sum_{i=1}^k \sum_{j=1}^k \frac{N_{ij}}{N} \log_2 \left(\frac{\frac{N_{ij}}{N}}{\frac{a_i b_j}{N^2}} \right) \quad (13)$$

where N_{ij} is the number of relationships between class file cluster i and method cluster j , a_i is the total number of relationships associated with class file cluster i , b_j is the total number of relationships associated with method cluster j , and N is the total number of relationships in R .

6.4 Local evaluation of the class file clustering

Finally, we evaluate the quality of the resulting class clustering using several metrics intended to reveal how closely the classes within each of the resulting clusters are related. After running our architectural recovery algorithm, we evaluate the resultant class file clustering locally by using Sum of Squared Distances (SSD) (Cressie 1993), Dunn Index (DI) (Dunn 1973), and mutual information between terms and class files. SSD provides a measure of cohesion of a clustering and smaller values are desired. Dunn index takes both cohesion and clusters' separation into account. It is defined as the ratio between the minimal inter-cluster distance and the maximal intra-cluster distance.

$$DI = \min_{1 \leq i \leq k} \left\{ \min_{1 \leq j \leq k} \left\{ \frac{d(i, j)}{\max_{1 \leq l \leq k} d'(l)} \right\} \right\} \quad (14)$$

where $d(i, j)$ is the distance between clusters i and j , and $d'(l)$ measures the intra-cluster distance of cluster l . Higher Dunn index values are better.

We use Eq. 13 to compute the mutual information between terms and the class files. The only difference is that the relationships considered in Eq. 13 are replaced by the bipartite relationships between the terms and the class files. Section 7.3 describes the experimental findings for local evaluation of the class file clustering.

6.5 Local evaluation of the call graph partitioning

The call graph is composed of a complex set of relationships between the methods. Our CCHD-based approach clusters both the call graph and the class files simultaneously to obtain a meaningful partition for both. The call graph itself can be partitioned independently using Shi and Malik (2000) normalized cut algorithm. Then we can compare the results obtained by any other approach to this independent graph clustering to verify if the other approach provides a different result set than the standalone graph partitioning. We compute the Jaccard index (JI) (Yue and Clayton 2005) between two clustering outcomes to find their similarity. For every pair of methods in the call graph, JI investigates whether two methods are clustered together in both clusterings or separate in both clusterings, or together in one but separate in the other. JI is defined as follows.

$$J = \frac{a + b}{\binom{n}{2}} \quad (15)$$

where a is the number of agreements across all pairs, b is the number of disagreements, and n is the number of vectors in the dataset.

We use a second method to evaluate the quality of the call graph partitions. One of the objectives of graph clustering is to minimize the number of edges that have their endpoints in two different partitions. In an ideal case, each cluster will be a graph component with no edge between the clusters. We compute the percentage of edges of the call graph that have their two endpoints in the same cluster as a quality measure of the call graph partitions.

Finally, we use conventional sum of squared distance (SSD) as a measure of cohesiveness of the clustering result. During any comparison, we use the same vector representation of the call graph using (Shi and Malik 2000) but compute SSD with different cluster assignments of the vectors for a fair assessment. We describe the experiments with local evaluation of the call graph partitioning in Sect. 7.4.

6.6 Evaluation of the automatic organization

We evaluate the automatic categorization step for newly written codes using a k -fold cross validation technique. k -fold strategy guarantees that every record is used as both training and test over subsequent runs. In addition, we varied k of the k -fold technique to experiment a wide range of data splits. Experimental results are shown in Sect. 7.6.

6.7 Evaluation of clustering accuracy

To evaluate the overall accuracy of the clustering result, we calculate the MoJoFM (Wen and Tzerpos 2004) value between the estimated clustering with the ground truth clustering. MoJoFM provides a measure of how close one clustering result (A) is to another clustering (B), and is calculated as:

$$MoJoFM(A, B) = \left[1 - \frac{mno(A, B)}{\max(mno(\forall A, B))} \right] \times 100\% \quad (16)$$

where $mno(A, B)$ is the minimum number of Move or Join operations one needs to perform in order to transform either A to B or vice versa.

6.8 Empirical evaluation

To evaluate the partitioning we conducted a user study, consisting of four professional developers at ABB Inc. and the Sando open source code search tool,⁵ which was developed in large part by those four developers. Sando was chosen for the study since it is an open source software project of considerable size, consisting of hundreds of classes and many thousands of lines of code, where the developers can still be aware of the purpose of all or most of the classes in the project. The developers were contacted via e-mail, and presented with a listing of seven components, which were initially discovered by the CCHD-based framework. For each of the components, the developers were provided a list of the classes in the component and an ordered list of the most relevant terms from the code base that define that component. The developers were also given, for each component, a list of classes that were added, removed, and retained, compared to a plain k -means clustering of the dataset. The developers were asked to mark each component (as a whole) as either good or bad, based on the class files that are part of it. In addition, the developers were asked to mark each of the class files that were added, removed or retained in the component as either (1) good decision by the algorithm; (2) bad decision by the algorithm; or (3) cannot decide. The developers worked individually and were given an unrestricted time span to perform this task, although most of them reported having completed the task within one hour. The evaluation consisted of only a single iteration of CCHD, and therefore the developers did not experience or evaluate the training and tuning performed by CCHD over multiple iterations, which is one of the strengths of this technique.

7 Results

The research questions we seek to answer in this section are as follows.

⁵ <http://sando.codeplex.com>.

1. How does the runtime of the CCHD approach scale with increases in number of classes and terms? (Sect. 7.1)
2. Are the relationships between the methods in the call graph and the class file data preserved at cluster level using CCHD? (Sect. 7.2)
3. Does the CCHD algorithm improve the quality of the class file clustering? (Sect. 7.3)
4. Is the quality of the call graph partitioning with CCHD better than a direct mapping of an independent class file clustering to related methods? (Sect. 7.4)
5. Are the components of the architecture discovered by the CCHD approach empirically justifiable? (Sect. 7.5)
6. How does the proposed framework perform in categorizing new code into an extracted architecture? Can the framework provide a characterization of each component of a software architecture discovered by the CCHD approach for better understanding of the code base? (Sect. 7.6)
7. How does the initial clustering performed by CCHD compare to other state of the art software clustering techniques? (Sect. 7.7)

7.1 Runtime characteristics

To examine the runtime characteristics of the CCHD approach, we prepared *synthetic* datasets with smoothly varying the number of classes, methods, and strengths of relationships between them. Using synthetic data enabled us to create artificial conditions for evaluating the runtime of CCHD that we would have a hard time to create using data from the wild. The synthetic datasets were generated by randomly creating entries in the matrix of natural language terms, classes and methods, where we constrained the number of methods to be 5 times the number of classes. When varying the number of classes, we kept the number of terms fixed at 500, while we kept the number of classes fixed at 50 when varying the number of terms. Figure 3 shows the runtime behavior of the CCHD approach. It shows that the runtime monotonically increases with number of classes and terms. We experimented the runtime with different numbers of clusters.

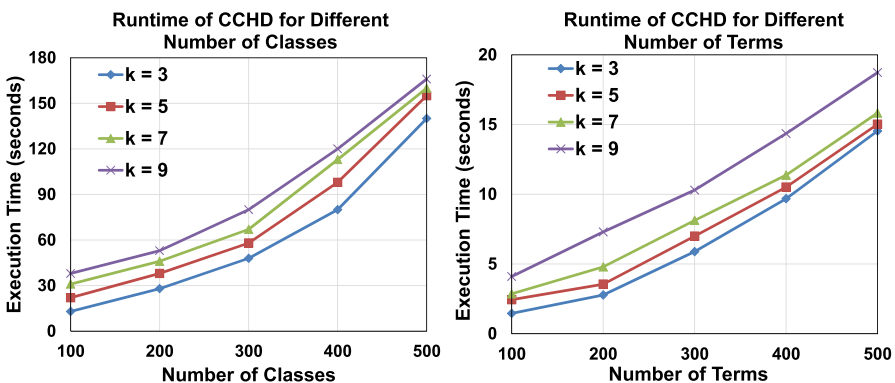


Fig. 3 Runtime characteristics, (left) with varying number of classes (right) with varying number of terms

With any number of clusters in the experiment, we observe that the runtime increases are modest. MATLAB codes for generating these synthetic datasets are provided in the paper's companion website.

7.1.1 Complexity analysis

CCHD framework works in several stages—pre-processing, building the combined matrix, running spectral clustering, and building the classifier. In this section we analyze the computational complexity of the CCHD framework and its different components. The symbols used in this section refer to the same symbols described in Sect. 3.1. The pre-processing step is mostly about removing stop words, performing stemming and computing the tf-idf values of all the terms in each document, which has a time complexity of $O(n_c \times n_t)$ (Hossain et al. 2012). Once we have the tf-idf values for each word-document pair, we just have to go over all the pairs, which is equal to the number of edges in the class-term association graph, and put each tf-idf value in corresponding cell in constant time. Similarly, we have to go over all the edges of call graph and method-class relationship graph to complete building the combined matrix, W . Therefore, total cost for building the combined matrix is $O(|\mathcal{S}| + |\mathcal{R}| + |E|)$. The most expensive parts of spectral clustering are solving the generalized eigenvalue problem that takes $O(n^3)$ time (Mises and Pollaczek-Geiringer 1929; Pohlhausen 1921) and applying k -means clustering that takes $O(knt)$ time (Na et al. 2010), where k is the number of clusters and t is the number of iterations for the algorithm to converge. Notice that in this solution, the number of nodes for spectral clustering is the sum of all the entities in the code base, that is, $n = n_c + n_t + n_m$. Based on the clustering results and developers' feedback we run Naive Bayes classifier to categorize the classes using their term distribution. The training process takes $O(n_c \times n_t)$ time and the categorization step takes $O(k \times n_t)$ time for a single class (Zheng and Webb 2005). Though the framework goes through multiple stages, and some of them have relatively higher time complexities (e.g. spectral clustering), all these things will be done offline except the categorization step of Naive Bayes classification that has very low time complexity. Therefore, once the system goes live, there should not be any issues with time. Section 7.1 characterizes the runtime of CCHD using a synthetic dataset.

7.2 Relationship preservation by CCHD

The CCHD approach provides a balance between the clusters' quality in multiple datasets and preservation of cluster level relationships between those datasets. To measure how well the relationships are preserved at the cluster level, we use median KL-divergence as explained in Sect. 6.2 and mutual information as described in Sect. 6.3. We compare our results with independent executions of spectral clustering on the call graph and the inter-class lexical similarity datasets. Note that our framework, CCHD, is also based on spectral clustering (Ng et al. 2002), but using a combination of the call graph and class lexical similarity. Therefore, in this evaluation we aim to show that this combination outperforms isolated executions of spectral clustering on each of the constituent relationships. We vary the number of clusters from two to ten.

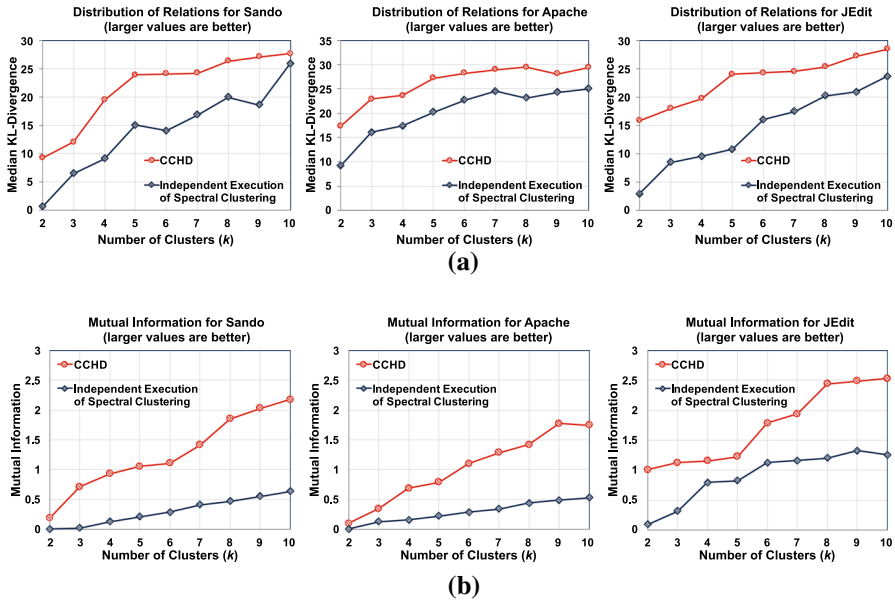


Fig. 4 Information theoretic evaluation to measure the quality of the relationship preservation by CCHD approach and independent execution of k -means on the call graph and the class file datasets. **a** Median KL-Divergence to compute the distribution of the relationships across two clusterings: (*left*) Sando, (*middle*) Apache httpd, and (*right*) jEdit. **b** Shared information between methods and class files in terms of mutual information: (*left*) Sando, (*middle*) Apache httpd, and (*right*) jEdit.

Figure 4a shows the median KL-Divergence with different number of clusters using three different datasets and two approaches—CCHD and independent execution of spectral clustering. All the three plots depict that the CCHD approach provides higher median KL-Divergence from the uniform distribution than the independent spectral clustering executions. This indicates that our CCHD approach provides a mechanism to bring two endpoints of a relationship to the same cluster. With each of the datasets, the results become better with larger number of clusters. Figure 4b shows the corresponding mutual information for each code base. We observe that, with any number of clusters, the mutual information between the method clusters and the class file clusters is higher with the CCHD approach than the independent execution. Although class files and the call graph are generated from the same code base, conventional clustering algorithms cannot take context and modular usage of the code into account, and as a result the relationships are not well preserved. With CCHD, the relationships are preserved along with the clusters' locality. Section 7.3 portrays the experimental results for clusters' locality.

7.3 Quality of the class file clusters

In this section, we assess the quality of the class file clusters discovered by our CCHD approach. We used all the three datasets—Sando, Apache httpd, and jEdit—for the

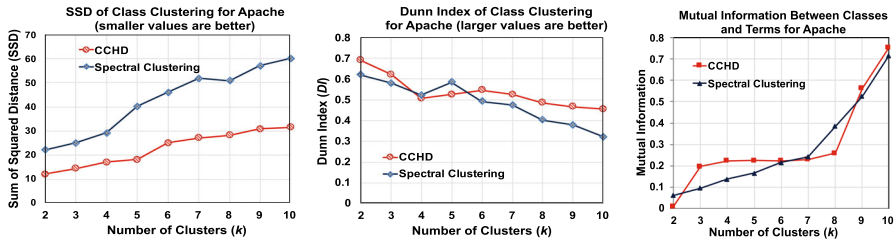


Fig. 5 Class cluster quality comparison between CCHD and a similarity matrix based spectral k -means clustering algorithm using the Apache httpd dataset: (left) SSD, (middle) Dunn index, and (right) term-class mutual information

experiments but are reporting only the results of httpd in this space because the other datasets have similar findings. We compare the results of the CCHD approach to a similarity matrix based spectral k -means clustering algorithm, which is a baseline that is most similar to the clustering algorithm described in this paper. For both the algorithms, the CCHD approach and the similarity matrix based spectral k -means, we map the resultant class file cluster labels to the original vector of terms and compute the SSD. This evaluation ensures the fairness of comparison since SSD is based on the original vector space rather than any transformed one (as in CCHD and spectral clustering). Figure 5(left) shows that the CCHD approach provides lower SSD than the spectral k -means clustering. This indicates that, for the httpd dataset, CCHD provides better locality of a class file clustering than the k -means algorithm. Moreover, the increase in SSD with larger number of clusters using CCHD is smaller than the increase with spectral k -means algorithm.

Figure 5(middle) shows the evaluation with the same setting but using Dunn Index (DI) instead of SSD. Larger DI values are better. Though it is practically very hard for a system to show superior performance from every angle for all the datasets, CCHD shows reasonably better results in most of the cases. We observe that CCHD has competitive DI compared to the spectral k -means algorithm. DI of CCHD is better than the spectral k -means algorithm for almost any k . The exceptions for Apache httpd are $k = 4$ and $k = 5$. For jEdit, CCHD has smaller DI only with $k = 2$. This indicates that although the CCHD approach takes many items into account—e.g., call graph locality, relationship preservation, and class file clustering quality—it does not result in deterioration of quality of the class file clustering. In Sect. 7.4, we report that CCHD does not deteriorate the quality of the call graph partitions either.

Since we aim to capture the holistic features of a code base, we utilize the call graph, class files, and the relationships between methods and the class files while the conventional k -means algorithm uses only the document vectors for clustering a code collection. In a sense, our approach has a possibility of losing mutual information between terms and class files since we use relational information for clustering the call graph and the class files simultaneously. Despite the possibility, Fig. 5(right) shows that the CCHD approach provides better mutual information trends, barring the exceptions for $k = 7$ and $k = 8$, even when compared to the standalone execution of spectral clustering algorithm on the class files. Moreover, for the exceptional cases when CCHD cannot give better results, they are fairly close to the spectral clustering results.

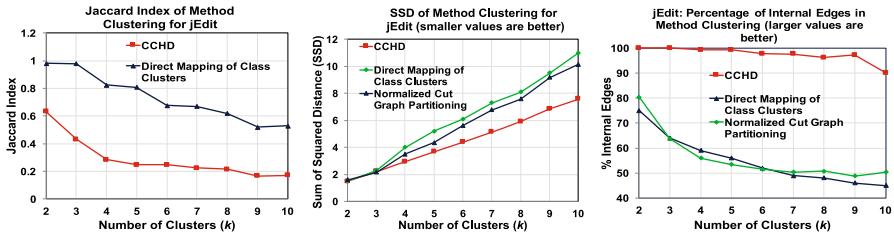


Fig. 6 Evaluation of the call graph partitioning for the jEdit code base: (left) Jaccard similarity index between a clustering mechanism and Shi and Malik (2000) normalized cut algorithm, (middle) SSD of the graph partitioning using different approaches, (right) percentage of call graph edges within each cluster using different approaches

7.4 Quality assessment for the call graph partitions

Figure 6(left) shows that both the CCHD approach and the direct mapping of k -means class clustering to the methods provide different clustering results than Shi and Malik’s (2000) normalized cut algorithm applied on the call graph. The CCHD approach tends to produce lesser similarity to the normalized cut algorithm than the direct class file to method mapping. This illustrates that our approach neither follows direct mapping of class file clusters to label the methods nor it relies on the normalized cut based algorithm to partition the call graph. In addition to providing different clustering results, the CCHD approach tends to provide high quality clusters in terms of locality when compared to the normalized cut algorithm and direct class file cluster mapping as shown in Fig. 6(middle). Figure 6(right) illustrates that our CCHD approach has the best percentage of call graph edges inside the partitions. The results clearly show that the call graph partitioning with the CCHD approach provides unique and high quality graph partitions. We used all the three code bases for this experiment and obtained similar trends.

7.5 Empirical evaluation

All of the four participants of our developer case study reported that examining the Sando classes for each of the 7 components, categorized by CCHD, was not difficult and consumed less than one hour of their time. Most of the participants, 3 out of 4, found that 6 components (out of 7) were logically constructed, while one developer found only 5 components as satisfactory. There was some disagreement among the developers on which components were inappropriate: two developers found *Cluster 1* as poorly constructed, while two others found *Cluster 7* as poor. Another developer thought that *Cluster 6* was incorrect. *Cluster 1* was the largest, consisting of 82 Sando classes, which bothered the developers that marked it as inappropriate, while the others regarded *Cluster 1* as defining the core functionality of Sando. On the other hand *Cluster 7* was one of the smallest, consisting of only 4 Sando classes. The two developers that found *Cluster 7* as inappropriate felt that while it included a coherent set of Sando classes, it did not include several other related Sando classes that should

have been included. Not having included necessary functionality was also the reason *Cluster 6* was marked as incorrect by the single developer.

Software architecture recovery is highly subjective, as shown by previous studies as well as by the general lack of agreement among the developers in our study on the accuracy of each of the extracted clusters. Overall, CCHD was successful at providing the developers a good initial architecture, as the vast majority of the clusters were marked as appropriate by the developers, while the framework’s ability to fine tune the clustering by training a classifier is aimed at addressing developer subjectivity in software architectural recovery.

7.6 Automatic categorization and characterization of clusters

To evaluate the accuracy of our automatic categorization technique after applying the CCHD approach, we use cross fold validation over the CCHD outcome. We report the accuracies with different number of clusters and varying test and training splits in Fig. 7. We only report the results with the Sando code base because we designed the empirical evaluation using Sando. Our observation with the CCHD outcome for the Sando code base is that the class are well distributed across the clusters when there are seven groups. Figure 7(left) shows that with different training and test splits, we obtain an accuracy of around 97% with seven clusters. With $k = 3$ and 5 the accuracies are more than 90%. With large number of clusters (e.g., $k = 9$) the accuracies reduce which can be expected with any automatic categorization.

The software engineers provided feedback on CCHD results with $k = 7$ (Sect. 7.5). Since three out of software engineers found that six clusters (out of seven) were logically constructed, we removed one suggested cluster (Cluster 7) from the list and applied cross fold validation over six clusters. Another important aspect of verify is a simple spectral clustering over the similarity matrix for the code classes provides high quality classification. The categorization accuracy comparison in Fig. 7(right) illustrates that both CCHD outcomes and modified CCHD outcomes provide more accurate

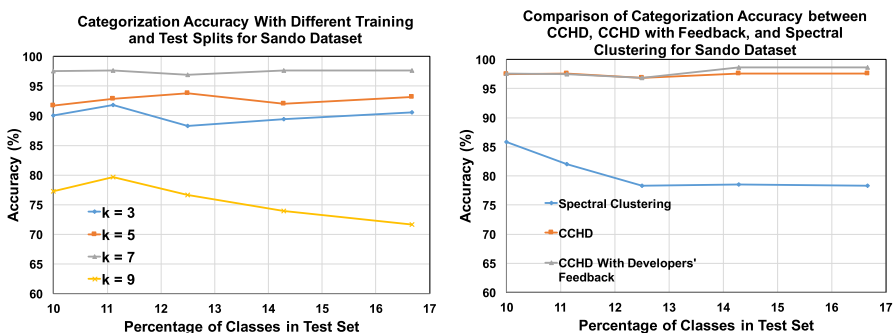


Fig. 7 Results for automatic placement of codes into existing architecture: (left) after applying CCHD with different number of clusters and (right) a comparison of categorization accuracy between three approaches using different clustering outcomes—a similarity matrix based spectral clustering with $k = 7$, CCHD approach with $k = 7$, and modification of the CCHD outcome using software engineer’s feedback

Table 2 Characterizations of three example clusters

Cluster label	Representative terms (class probability of terms $p(t c)$)
Graphical user interface	box (0.0006), text (0.0006), window (0.0005), control (0.0005), event (0.0005)
Search	search (0.0002), criteria (0.00018), result (0.00016), index (0.00015), query (0.00015)
Spell checking	Spell (0.0005), engine (0.0005), suggest (0.0005), word (0.0004), language (0.0004)

Terms with highest class probability $p(t|c)$ are listed

categorization than the spectral clustering without considering any relationships and the call graph.

To aid the process of analytical evaluation of the clustering result, we provide the software developers with a characterization of each cluster. When we train our system for automatic placement of new codes, we obtain the probability of each term being associated with each cluster. For each cluster, we sort these probabilities in descending order and present the corresponding terms as a characterization of each cluster. Table 2 shows the characterizations of three clusters (out of seven) discovered from the Sando dataset.

7.7 Comparison with other techniques

To evaluate the quality of software architecture recovered by CCHD framework, we measure its accuracy against the ground truth information of four open source code bases: Apache OODT (version 0.2), Hadoop (version 0.19.0), ArchStudio (version 4) and Insight Segmentation and Registration Toolkit (ITK, version 4.5.2) (Garcia et al. 2013b; Lutellier et al. 2015). The accuracy of an architecture is calculated as MoJoFm using Equation 6.7. We then compare the accuracies of CCHD on OODT, Hadoop, ArchStudio and ITK data sets with six other state-of-the-art software architecture recovery techniques, namely, Algorithm for Comprehension-Driven Clustering (ACDC) (Tzerpos and Holt 2000), Weighted Combined Algorithm (WCA) using UE measure (Maqbool et al. 2004), scaLable InforMation BOttleneck (LIMBO) (Andritsos and Tzerpos 2005), Bunch (Mancoridis et al. 1999), uniform version of Zone-Based Recovery (ZBR) (Corazza et al. 2010) and Architecture Recovery using Concerns (ARC) (Garcia et al. 2011). The MoJoFM values of these techniques on the four data sets were collected from the survey of Garcia et al. (2013a) and their later work (Lutellier et al. 2015). Table 3 summarizes the performance of different techniques. The MoJoFm value for LIMBO was not available because this technique produces an architecture of OODT dataset for which MoJoFM calculation does not terminate. We found that the CCHD framework achieved the best accuracy on OODT and ITK data sets, was close to best one on ArchStudio, and came out as second-best on Hadoop data set. This is to note that the accuracies achieved by the CCHD framework is based on only the initial clustering (without developers' feedback).

Table 3 Comparison of architecture recovery accuracies (in MoJoFm) of different methods

Method	OODT	Hadoop	ArchStudio	ITK
ARC	48.48	54.28	62	59
ACDC	46.01	62.92	77	59
WCA-UE	43.67	42.15	33	32
LIMBO	–	19.23	26	31
Bunch	31.56	51.24	–	–
ZBR-UNI	30.89	36.00	48	–
CCHD	52.83	60.36	76.8	65.63

Bold values of the last row are to highlight the values for our method. Remaining bold values show the best result among other methods for each data set

8 Implications of the results

Our CCHD software clustering approach presents a novel model and algorithm for software clustering, coupled with an iterative process of integrating developer feedback to improve the clustering that was initially obtained. The approach builds on previous successes of integrating lexical and structural information for improving the quality of software clustering, but presents an approach that is free of pre-weighting of information and other assumptions, which may limit techniques that use them in applying across a wide set of software projects.

CCHD uses relationships between methods, relationships between classes and the correspondence of methods and classes to perform the clustering. As method relationships, in this paper, we used the call graph, and as class relationships, we used lexical similarity between natural language terms in the classes. Other relationships between classes or methods can easily be integrated, as well as composite metrics that combine several relationship types, e.g. class inheritance and lexical similarity.

While the initial CCHD clustering is competitive with the best approaches in the field, we find, based on a small-scale study of developers in the field, that improving the initial clustering to fit a particular project or specific developer is necessary in order to achieve industrially-usable architecture recovery. Also, CCHD will not effectively decompose extremely degraded legacy systems that have strong inter-dependencies and contain similar, repetitive natural language semantics. Better metrics, which may be integrated into the CCHD framework, that can tease out the original architecture are required to improve the system's effectiveness for such cases.

An additional benefit of the proposed technique is the ability to automatically characterize the clusters by providing the probabilistically strongest terms. This characterization was very important in conducting our developer study, in order to further clarify to the participants the nature of each cluster. The idea of characterizing software clusters has been discussed by others, but is not yet a required part of each technique or tool. We believe that automatic cluster characterization is absolutely integral to developers' use of architecture recovery systems, and should be performed by all such approaches that utilize natural language information.

9 Conclusions

We have presented a data analytic framework leveraging a palette of data mining techniques to recover software architecture from a code base. Experimental results and empirical evaluations show that the framework discovers software project architectures systematically to help software engineers maintain complex code bases.

Our directions for future work are two fold. Currently, our framework allows software engineers to improve the clustering outcomes at the instance level. A future direction is to allow the users to provide abstract level feedback, for example, how clusters can be merged together, how some clusters can be subdivided, or a combination of both, where clusters can be regrouped with a *scatter and gather* approach. This would help software engineers capture more expressive relationships between the call graph and the lexical dataset. Secondly, we aim to enrich the CCHD approach by providing temporal information about the development history of the code base using version control. This will incorporate the code base knowledge propagated over time to obtain a better architecture.

References

- Andritsos, P., Tzerpos, V.: Information-theoretic software clustering. *IEEE Trans. Softw. Eng.* **31**(2), 150–165 (2005)
- Bae, E., Bailey, J.: Coala: a novel approach for the extraction of an alternate clustering of high quality and high dissimilarity. In: *Proceedings of the Sixth International Conference on Data Mining (ICDM'06)*, IEEE, pp 53–62 (2006)
- Banerjee, A., Dhillon, I., Ghosh, J., Merugu, S., Modha, D.: A generalized maximum entropy approach to Bregman co-clustering and matrix approximation. In: *Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining (KDD'04)*, pp. 509–514 (2004)
- Basu, S., Davidson, I., Wagstaff, K.: *Constrained Clustering: Advances in Algorithms, Theory, and Applications*. CRC Press, Boca Raton (2008)
- Bauer, M., Trifu, M.: Architecture-aware Adaptive Clustering of OO Systems. In: *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pp. 3–14 (2004)
- Bavota, G., Carnevale, F., Lucia, A., Penta, M., Oliveto, R.: Putting the developer in-the-loop: an interactive GA for software re-modularization. In: *Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE'12)*, pp. 75–89 (2012)
- Bavota, G., Lucia, A., Marcus, A., Oliveto, R.: Using structural and semantic measures to improve software modularization. *Empir. Softw. Eng.* **18**(5), 901–932 (2013)
- Berkopec, A.: HyperQuick algorithm for discrete hypergeometric distribution. *J. Discrete Algorithms* **5**(2), 341–347 (2007)
- Böhm, C., Faloutsos, C., Pan, J., Plant, C.: Robust information-theoretic clustering. In: *Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pp. 65–75 (2006)
- Cai, Y., Iannuzzi, D., Wong, S.: Leveraging design structure matrices in software design education. In: *Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET'11)*. IEEE, pp. 179–188 (2011)
- Cai, Y., Wang, H., Wong, S., Wang, L.: Leveraging design rules to improve software architecture recovery. In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, ACM, New York, NY, USA, QoSA'13, pp. 133–142. doi:[10.1145/2465478.2465480](https://doi.org/10.1145/2465478.2465480) (2013)
- Chaitin, G.: *Algorithmic Information Theory*. Wiley Online Library, New York (1982)
- Christl, A., Koschke, R., Storey, M.: Equipping the reflexion method with automated clustering. In: *12th Working Conference on Reverse Engineering*. IEEE, pp. 10–20 (2005)

- Corazza, A., Di Martino, S., Scanniello, G.: A probabilistic based approach towards software system clustering. In: 2010 14th European Conference on Software Maintenance and Reengineering (CSMR). IEEE, pp. 88–96 (2010)
- Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: Weighing lexical information for software clustering in the context of architecture recovery. *Empir. Softw. Eng.* **21**(1), 72–103 (2016)
- Cressie, N.: *Statistics for Spatial Data*, vol. 900. Wiley, New York (1993)
- Dai, W., Xue, G., Yang, Q., Yu, Y.: Co-clustering based classification for out-of-domain documents. In: Proceedings of the 13th International Conference on Knowledge Discovery and Data Mining (KDD'07), pp. 210–219 (2007)
- Dhillon, I.: Co-clustering documents and words using bipartite spectral graph partitioning. In: Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining (KDD'01), pp. 269–274 (2001)
- Dhillon, I., Guan, Y.: Information theoretic clustering of sparse cooccurrence data. In: Proceedings of the 3rd International Conference on Data Mining (ICDM'03), pp. 517–520 (2003)
- Dhillon, I., Mallela, S., Modha, D.: Information-theoretic co-clustering. In: Proceedings of the 9th International Conference on Knowledge Discovery and Data Mining (KDD'03), pp. 89–98 (2003)
- Dunn, J.: A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. *J. Cybern.* (1973)
- Gao, B., Liu, T., Zheng, X., Cheng, Q., Ma, W.: Consistent bipartite graph co-partitioning for star-structured high-order heterogeneous data co-clustering. In: Proceedings of the 11th International Conference on Knowledge Discovery in Data Mining (KDD'05), pp. 41–50 (2005)
- Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., Cai, Y.: Enhancing architectural recovery using concerns. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 552–555 (2011)
- Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th International Conference on Automated Software Engineering (ICASE'13), pp. 486–496 (2013a)
- Garcia, J., Krka, I., Mattmann, C., Medvidovic, N.: Obtaining ground-truth software architectures. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 901–910 (2013b)
- Gokcay, E., Principe, J.: Information theoretic clustering. *Pattern Anal. Mach. Intell.* **24**(2), 158–171 (2002)
- Hossain, M.S., Tadepalli, S., Watson, L., Davidson, I., Helm, R., Ramakrishnan, N.: Unifying dependent clustering and disparate clustering for non-homogeneous data. In: Proceedings of the 16th International Conference on Knowledge Discovery and Data Mining (KDD'10), pp. 593–602 (2010)
- Hossain, M.S., Gresock, J., Edmonds, Y., Helm, R., Potts, M., Ramakrishnan, N.: Connecting the dots between pubmed abstracts. *PLoS ONE* **7**(1), e29,509 (2012)
- Hossain, M.S., Marwah, M., Shah, A., Watson, L., Ramakrishnan, N.: AutoLCA: a framework for sustainable redesign and assessment of products. *ACM Trans. Intell. Syst. Technol.* **5**(2) (2014)
- Koschke, R.: Atomic architectural component recovery for program understanding and evolution. In: IEEE International Conference on Software Maintenance. IEEE Computer Society, pp. 478–488 (2002)
- Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., Kroeger, R.: Comparing software architecture recovery techniques using accurate dependencies. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). IEEE, vol. 2, pp. 69–78 (2015)
- Luxburg, U.: A tutorial on spectral clustering. *Stat. Comput.* **17**(4), 395–416 (2007)
- Mancoridis, S., Mitchell, B.S., Chen, Y., Gansner, E.R.: Bunch: a clustering tool for the recovery and maintenance of software system structures. In: IEEE International Conference on Software Maintenance, 1999 (ICSM'99). Proceedings. IEEE, pp. 50–59 (1999)
- Manning, C., Raghavan, P., Schütze, H.: *Introduction to Information Retrieval*, vol. 1. Cambridge University Press, Cambridge (2008)
- Maqbool, O., Babri, H.A.: The weighted combined algorithm: a linkage algorithm for software clustering. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. IEEE, pp. 15–24 (2004)
- Mises, R., Pollaczek-Geiringer, H.: Praktische verfahren der gleichungsauflösung. *ZAMM* **9**(1), 58–77 (1929)
- Misra, J., Annervaz, K., Kaulgud, V., Sengupta, S., Titus, G.: Software Clustering: Unifying Syntactic and Semantic Features. Working Conference on Reverse Engineering, pp. 113–122 (2012)
- Mohar, B.: *Some Applications of Laplace Eigenvalues of Graphs*. Springer, Berlin (1997)

- Mohar, B., Alavi, Y.: The Laplacian Spectrum of Graphs. *Graph Theory Comb. Appl.* **2**, 871–898 (1991)
- Momtazpour, M., Butler, P., Hossain, M.S., Bozchalui, M., Ramakrishnan, N., Sharma, R.: Coordinated clustering algorithms to support charging infrastructure design for electric vehicles. In: *Proceedings of the 18th International Conference on Knowledge Discovery and Data Mining (KDD UrbComp'12)*, pp. 126–133 (2012)
- Na, S., Xumin, L., Yong, G.: Research on k-means clustering algorithm: an improved k-means clustering algorithm. In: *Proceedings of the 3rd International Symposium on Intelligent Information Technology and Security Informatics (IITSI'10)*. IEEE, pp. 63–67 (2010)
- Ng, A., Jordan, M., Weiss, Y.: On spectral clustering: analysis and an algorithm. *Adv. Neural Inf. Process. Syst.* **2**, 849–856 (2002)
- Pohlhausen, E.: Berechnung der eigenschwingungen statisch-bestimmter fachwerke. *ZAMM* **1**(1), 28–42 (1921)
- Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. *IEEE Trans. Softw. Eng.* **37**(2), 264–282 (2011)
- Scanniello, G., Marcus, A.: Clustering support for static concept location in source code. In: *Proceedings of the 19th International Conference on Program Comprehension (ICPC'11)*, pp. 1–10 (2011)
- Shi, J., Malik, J.: Normalized cuts and image segmentation. *Pattern Anal. Mach. Intell.* **22**(8), 888–905 (2000)
- Shtern, M., Tzerpos, V.: Clustering methodologies for software engineering. *Adv. Softw. Eng.* (2012). doi:[10.1155/2012/792024](https://doi.org/10.1155/2012/792024)
- Struyf, A., Hubert, M., Rousseeuw, P.: Clustering in an object-oriented environment. *J. Stat. Softw.* **1**(4), 1–30 (1997)
- Taylor, R.N., Medvidovic, N., Dashofy, E.M.: *Software Architecture: Foundations, Theory, and Practice*. Wiley, New York (2009)
- Tzerpos, V., Holt, R.C.: Acdc: an algorithm for comprehension-driven clustering. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, pp. 258–258 (2000)
- Wen, Z., Tzerpos, V.: An effectiveness measure for software clustering algorithms. In: *12th IEEE International Workshop on Program Comprehension, 2004. Proceedings*. IEEE, pp. 194–203 (2004)
- Yang, C., Zhou, J.: Hclustream: a novel approach for clustering evolving heterogeneous data stream. In: *Proceedings of the 6th International Conference on Data Mining (ICDM'03)*, pp. 682–688 (2006)
- Yoon, H., Ahn, S., Lee, S., Cho, S., Kim, J.: Heterogeneous clustering ensemble method for combining different cluster results. *Data Min. Biomed. Appl.* **3916**, 82–92 (2006)
- Yue, J., Clayton, M.: A similarity measure based on species proportions. *Commun. Stat. Theory Methods* **34**(11), 2123–2131 (2005)
- Zheng, F., Webb, G.I.: A comparative study of semi-naive Bayes methods in classification learning. In: *Proceedings of the Fourth Australasian Data Mining Conference (AusDM05)*, Citeseer, pp. 141–156 (2005)
- Zhu, J., Huang, J., Zhou, D., Yin, Z., Zhang, G., He, Q.: Software architecture recovery through similarity-based graph clustering. *Int. J. Softw. Eng. Knowl. Eng.* **23**(04), 559–586 (2013)