Probabilistic Component Identification

Hitesh S. Sajnani and Cristina V. Lopes
University of California, Irvine
{hsajnani, lopes}@uci.edu

ABSTRACT

Automatic component identification is an important step for many software engineering tasks and tools, from reverse engineering to program comprehension. Although being a topic of research for a long time, generally applicable solutions remain elusive with very low recall. Most component identification techniques impose structure on the software, therefore requiring substantial knowledge about the software, instead of inferring such structure.

We propose PCI, a non-parametric, probabilistic algorithm to the problem of automatic component identification. The proposed approach uses structural, behavioural and lexical features of the software to infer the underlying structure and identify the components. In order to evaluate this approach, multiple experiments are performed on a software system for which components have previously been identified [by others]. The proposed algorithm is compared to two existing algorithms. The experiments measure precision and recall, and attempt to unveil the impact of each source of information on the accuracy of the approach. We find PCI outperforms the others most notably on recall, with improvements that go from 4x to 10x better recall.

1. INTRODUCTION

The widespread practice of open source development is changing the IT industry in significant ways. Involvement in the open source projects, these days, is a strategy that companies consider as part of their product's marketability. Researchers in the fields of Science and Engineering rely on open source for their research, and the National Science Foundation is further promoting it by making it a requirement in all funded projects. Having the source code available to everyone these days is as important as having the raw data supporting scientific claims available, since Science and Engineering disciplines rely more and more on software for substantiating claims. Unfortunately, undocumented source code is as difficult to understand as raw, undocumented data; having it available without being able to understand it is not of much benefit. Open source projects, in particular, are notorious for their lack of documentation, since the developers often do not have the resources to produce artifacts beyond the code, so "the code is the documentation."

While this minimalist approach to documentation may save some time and effort in the short term, it has serious consequences in the long term. First, it is very time consuming to manually reverse-engineer the principal design decisions that underlie a piece of code. Second, without insight into a system's architecture, new developers who join the project and users who extend the code may make many mistakes and introduce costly bugs. Finally, without the insights on the design principles of the system, the code may deteriorate to the point where its organization is so chaotic that the project needs to be overhauled at a great cost, or simply abandoned and rewritten from scratch [33].

Motivating Scenario: To illustrate the problem, as well as the goal of this paper, consider Apache Lucene [5], a popular high-performance, full featured text search engine library written in Java. Lucene is a medium-sized library with about 2,000 classes and interfaces, some of which are to be extended by the users of the library. The available documentation on the project's web site is scarce, consisting primarily of a couple of usage examples and the automatically-generated API description - enough to get users started with Lucene, but far from fulfilling the needs of advanced usage. Like many popular open source projects, Lucene relies on the community to contribute additional documentation on the Wiki, but the coverage and quality of that documentation is questionable. Something as simple as an overall structural decomposition of the library, and the data it uses, is not available. Unlike many other useful open source libraries, Lucene is extremely popular, so there are some books published about it that contain the in-depth explanations needed to understand how it works. For example, in the book, *Lucene in Action* [25], one can finally find Figure 1, an informal schematic of the functional decomposition. Such a figure increases the user's understanding of the system, helps in understanding the API and gives valuable insights into the entire domain of search.

Goal: Our goal is to develop techniques that will allow us to recover components, as shown in the schematics, in Figure 1, automatically from source code artifacts. Ultimately, we want to be able to take any open source project, feed it to the tools proposed here, and obtain schematics that reasonably represent the project's main components and the relations among them. This goal is usually known in the software engineering literature as *component identification* or *software architectural component recovery* [20], and it has a long history as a topic of inquiry [26].

2. BACKGROUND

Terminology: The terms that are used throughout this paper are defined as follows: A *software entity*, or short an *entity*, is a programming unit with a name. Some examples of entities are source-code statements, methods, and Java classes or packages. An *atomic architectural component*, or

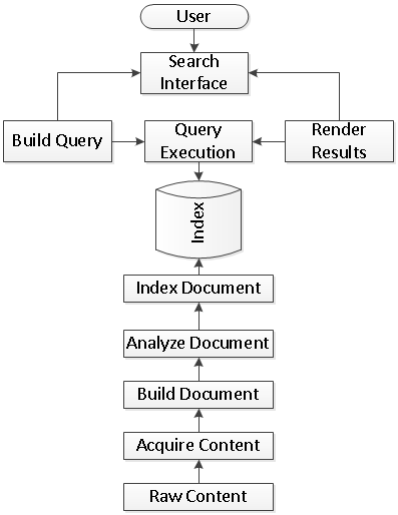

Figure 1: Functional decomposition of the Lucene library. Block represents a component and arrow describes the relationship [2]

short *component* is a set of functionally related entities [20]. A component need not be continuos block of code and it can span disparate parts of the program. *Software architecture* is the set of principal design decisions underlying a software system [31]. These decisions can be of several types; some easier to conceptualize than others. For the remainder of the paper we use the term "software architecture" to refer to design decisions that are structural in nature and/or related to data flows, i.e., in terms of components. Other types of design decisions that do not get explicitly represented in the source code are much harder, and even impossible, to recover. The term *feature* is used in the machine learning context, i.e., an individual measurable heuristic property of a phenomenon being observed [8]. For example, when representing images, the feature values might correspond to the pixels of an image, and when representing texts perhaps to term occurrence frequencies. A *feature vector* is an n-dimensional vector of features that represents some object. The reason for doing this is that the vector can be treated mathematically by algorithms.

In ther remainder of this section, we review some of the exisiting research in this area, discuss important limitations and make a case for proababilistic component identification.

Parameter Tuning and Structure Imposition. Several researchers have used clustering techniques to identify components [20, 4, 36, 3, 2]. Clustering helps to group similar entities and hence identifies smaller and more manageable components to assist program comprehension [40]. The results are promising, however, there are many challenges associated with it. Most of the clustering techniques that work well require heavy human intervention in terms of detailed knowledge of the components which are to be identified [33, 34]. The ones which are automated are parametric, i.e., they require many parameters to be tuned for each subject system. Moreover, the higher the number of parameters, the higher the chances of a clustering solution over-fitting a specific software and hence, it cannot be generalized to work for other systems. However, the fundamental problem with parametric algorithms is that they try to impose the structure on the data instead of inferring it from the data. This property makes them inflexible and increases their dependence on the initial value of the parameters. Thus, if the initial parameters are not set correctly, the clustering algorithms fail to correct themselves in the later stages, producing components with poor precision [24]. Another challenge is that many clustering approaches are transformed into optimization problems, which in general are computationally expensive or an NP-Hard problem. However, generally this is not a big issue because architecture retrieval is not an activity to be conducted *online*.

Using Limited Information. Many clustering techniques utilize information from one aspect of the software, such as, only structural information. This may lead to recovered components that are biased towards one aspect of the software. Moreover, recent research advancements show that it is useful to combine information from various aspects of software (e.g., static, dynamic, lexical, evolutionary, etc.) for various software engineering tasks. Beck and Diehl [6] found correlation between software evolution and concern clustering. They showed that using evolutionary information along with other information improves the accuracy of concern clustering. Similarly, Eaddy et. al increased the accuracy of feature identification by using dynamic and lexical information [13]. DiGiuseppe and Jones [11] proposed a concept-based clustering technique that leverages control flow and the natural language text used in the source code to cluster software failures. They showed that their technique worked better than other approaches which used only control flow information. Hence, utilizing information from multiple aspects may have merits even in the field of component identification. However, this leads to challenges like handling information overload and also designing algorithms to efficiently interpret the combined information from various aspects of software.

Exclusive Assignment of Entities to Components. Many researchers have proposed techniques which perform exclusive assignment of software entities to components [4, 36, 3, 2]. However, software entities, by their very nature, are likely to be associated with multiple components due to cross-cutting concerns and overlap in functionalities. For example, in a banking software, a getAccountReport method is likely to be associated with credit, debit, remittance, and other components. Similarly a calculateFee method that calculates the fee for an account, is likely to be associated with transfer, account, remittance, and other components. Sometimes, the association with one component is so dominant that the association with other components may not be important. However, this may not always be the case, and an entity may have strong associations with multiple components. Nonetheless, it is at least desirable to consider all the associations during the intermediate operations of the algorithm, and eventually filter out weak associations.

A recent empirical evaluation of five existing clustering techniques for component identification show that K-means produces best results in terms of cluster quality [9]. Hence we chose K-Means as one of the clustering algorithms to evaluate against. We chose its variant, Fuzzy K-Means as well, since it overcomes some of the limitations of K-Means. We briefly describe K-Means and argue about the limitations for the task of component identification using an example. It is important to note that the limitations are also pertinent to the class of parametric algorithms which are mostly used so far for the purpose of component identification.

K-Means: In K-Means, K cluster centroids are randomly initialized in the data [8]. The algorithm proceeds in an iterative manner and in each iteration, data points are assigned to the closest cluster centroid in the pool. At the end of each iteration, centroids are re-calculated based on the point assignment and this process is repeated until convergence. The convergence criterion is constant cluster assignment or error threshold.

Limitations: Although popular because of its simplicity, K-Means has several limitations in the context of component identification. To begin with, K-Means is a parametric algorithm, it needs the number of output components k to

[2]Diagram adapted from the book *Lucene in action*

be fixed beforehand. This may not be desirable because: (i) apriori knowledge about the software may not be available to us with sufficient confidence level; and (ii) it gets in the way of automation. Moreover, K-Means (or any such parametric algorithm) is very sensitive to the initial parameter values, hence the final cluster assignments largely depend on the initial choice of the parameters. The algorithm is susceptible to *local minimum* phenomenon as decisions are made locally without any prior information. Another limitation is that it performs exclusive assignment of software entities to components. However, as discussed, software entities, by nature, are likely to be associated with multiple components. Moreover, it makes arbitrary decisions in assigning entities to only one component when it may be close to several. For example, if an entity is equidistant to two components, which cluster should it belong to? In such cases, the assignment is random. Fuzzy K-Means differs slightly from K-Means because it assigns every entity to every component with a membership weight that is between 0 and 1. In other words components are treated as fuzzy sets instead of binary discrete sets as in K-Means. It is important to note that the basic philosophy of the algorithm is still discrete and not probabilistic. The difference between the two algorithms is only in the way they produce the final assignment, not the way they compute.

Our approach to the old goal of component identification is to use machine learning (ML) based non-parametric probabilistic models. Non-parametric models are not very sensitive to parameters and infer the optimal parameter values as they advance in the computation process. Also, being proabilistic, allows them to make soft assignments of entitites to components. Moreover, since ML has an elegant way of combining features using feature vectors, it will allow us to experiment with a large number of features of the software artifacts without having to establish *apriori* our own insights about what is important and what is not. We believe this is a promising approach that is worth exploring.

More precisely, the contributions of this paper are as follows:

- Identification of the limitations of existing clustering techniques when used in the context of component identification.
- A machine learning based non-parametric approach (PCI) for component identification using static, lexical, and dynamic information. It incorporates the fact that an entity can be associated with multiple components.
- A case study that shows the applicability of our approach to produce accurate components.
- Experiments to evaluate the effectiveness of various sources of information in the context of component identification.

3. APPROACH

We formulate the process of component identification as a learning problem, as shown in Figure 2. First, we extract various features associated with the software. Then, we use these features to identify the components using component identification algorithms derived from unsupervised machine learning techniques. We use three different feature sets to represent static, lexical, and dynamic aspect of the software. We propose a non-parametric probabilistic algorithm for component identification, called PCI. The algorithm is evaluated against a popular parametric clustering algorithm called K-Means and its variant, Fuzzy K-Means.

In the subsequent sections, we describe the motivation for using each feature set, the method of feature extraction, and each of the component identification algorithm in detail.

3.1 Feature Extraction

Data describing multiple aspects of the software have been proven to be useful for various software engineering tasks. Thus, in our approach, apart from structural information, we also include lexical and dynamic information. The intuition is that it might improve the accuracy of the identified components. Moreover, it will allow us to evaluate the effectiveness of various type of information for the process of component identification.

Source code Features: Source code features consist of static features and lexical features:

Static Features. These features include structural information about the source code. In our case, it includes method invocations by source code entity, and field accesses referred to or by an entity. Other examples are user defined type accesses, inheritance relations among classes, nature of the class (abstract, concrete or an interface). We included field access as they reflect cohesion present between the entities. The reason for using method invocation is that they represent functional bindings. For example, if two entities invoke the same method or access the same field, the possibility of they being functionality related is higher compared to other entities. Features are usually represented in the form of vectors. So, for example, if $\{f_1, ..., f_k\}$ are the field accesses and method invocations in the body of method m_i, then the static feature vector F_S corresponding to the method m_i, can be represented as a vector of terms: $F_{iS} = \{f_1, ..., f_k\}$.

Lexical features. These features capture domain information and include the tokenized text of and related to a method's content. We use all the comments appearing before the method definition and inside the methods. The identifiers we consider include the names of methods (and fields), enclosing types, packages, parameters, local variables, field accesses, method calls, and type references. Qualified names such as *foo.bar.baz()* are split into separate identifiers *foo, bar, and baz.* Compound identifiers are added as is and also split into separate terms based on standard naming conventions (e.g., camelCaseStyle and underscore style). We filter terms using a standard stop list augmented with the list of Java keywords. These lexical features have shown to work well for similar software engineering tasks e.g., concern location [13]. Similar to static features, if $\{t_1, ..., t_j\}$ are terms resulting by applying the lexical operations mentioned above for method m_i, then the lexical feature vector F_L corresponding to the method m_i, can be represented as a vector of terms: $F_{iL} = \{t_1, ..., t_j\}$

Dynamic features: The architecture of software, along with the static structure, also depends on its runtime behavior. Dynamic features exhibit the behavior of source code during runtime. Runtime information like frequency of method call related to a particular use case gives us information of the strength of their relationship which cannot be obtained from the structural properties. This helps us create specific components related to functionality. Also, methods which are executed together for a use case might have functional association.

The general process for retrieving the dynamic features from the source code is as follows: (1) identify the set of features (functionalities) in the software; (2) identify test cases associated with the features; (3) generate the list of executed methods for each feature by executing the test cases corresponding to that feature; and (4) create a relational table (dynamic features table) describing the method and the associated test cases.

It is important to note few concerns associated with the dynamic features. One of the them is the identification of functionalities in the software. In many cases, this may not be complete due to the lack of knowledge about the software. Another issue is the identification of test cases associated with these functionalities. This can be obtained if the test cases are well organized and packaged based on the function-

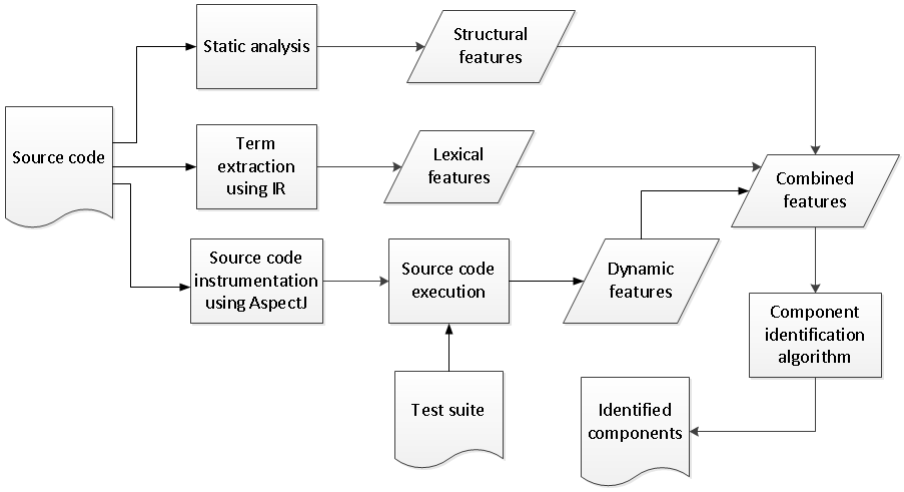

Figure 2: Overall Approach for Architectural Component Recovery

ality they are suppose to test. However, in the absence of such organization, it is difficult to identify this association. Moreover, the test cases associated with the software or certain functionality may not be adequate. This can lead to incomplete dynamic features. All these concerns may have an impact on the completeness of dynamic features, which, in turn, can have an impact on the role of dynamic features in component identification process.

More formally, if method m_i is executed when test cases $\{c_1, ..., c_n\}$ are run, the dynamic feature vector corresponding to m_i, can be represented as: $F_{iD} = \{c_1, ..., c_n\}$.

Combined features: These features represent the union of source code features and dynamic features together. Hence, each feature vector corresponding to method m_i in the dataset is represented as:

$F_i = F_{iS} \bigcup F_{iL} \bigcup F_{iD} = \{f_1, ..., f_k, t_1, ..., t_m, c_1, ..., c_n\}$

We expect it to be rich in information content because it combines three different aspects of software. Ideally, we would like to combine as many different sources of useful information as possible and have the ability expand our feature set as necessary. A good algorithm should be able to effectively exploit this kind of growing feature set.

3.2 Probabilistic Component Identification

The PCI algorithm is based on a probabilistic non-parametric model that does not make any strong assumptions about the parameters controlling the component identification process. Here, non-parametric does not mean a parameter-less model, rather a model in which representations grow as more data is observed. Thus instead of fixing the number of components to be discovered, the algorithm allows it to grow as more entities are seen. In this model, the observed data points are not viewed as belonging to a fixed set of clusters but rather as representatives of a latent structure in which data points belong to one of a potentially infinite number of clusters. As more information about feature differences is revealed, the number of inferred clusters is allowed to grow; allowing algorithm to choose and correct itself at a later stage as it sees more data. This is unlike parametric based approaches, in which, the algorithm keeps advancing in the wrong direction once it makes a mistake in the earlier stage. In case of PCI, we still need to specify parameters, but those are just the seed parameters with minimal impact on the output because the algorithm is self adjusting and infers ideall parameter values automatically from the data. This nature of the algorithm is desirable for the task of component identification as the distinct number of components is unknown beforehand.

Formal description. The basic philosophy of the algorithm is to view components in the software as represented by many distributions. The number of distributions k can be chosen to be any value that is larger than the expected number of components in the software. The number of distributions does not have any impact on the accuracy, however, may increase the convergence time for the algorithm. A software entity can then be modeled on a $(k-1)$ dimensional simplex, such that the probability distribution of an entity across all the components sums up to 1. i.e.,
Entity $e = [Pr(C_1), Pr(C_2), ..., Pr(C_k)]$ such that $Pr(C_i)$ represents the probability of an entity e belonging to component C_i, where $i \in \{1, 2, ..., k\}$, and $\sum_{i=1}^{k} Pr(C_i) = 1$.

The probability distribution G of an entity is actually a distribution over distributions. In probability theory, a random process that is a probability distribution whose domain is itself a set of probability distributions is represented by a Dirichlet process [8]. Hence, G can be represented as a Dirichlet process DP, with a distribution H, and a concentration parameter α.

$$G \sim DP(H, \alpha) \tag{1}$$

H represents a distribution i.e., a component. In our case, it is a gaussian distribution and it can be represented by the parameters mean μ and standard deviation σ of the entities that belong to the component. Parameter α determines the concentration of the component. The larger the value of the concentration parameter, the more evenly distributed is the association of the given entity with many components i.e., cross cutting concerns and overlapping components. The smaller the value of the concentration parameter, the more sparsely distributed is the association of the given entity with components i.e., the entity is associated with only few or one component. As there are many entities in the software, the probability distribution of all the entities is the integration of distributions generated by each entity.

$$P(e_1, e_2, ..., e_n) = \int P(G) \prod_{n=1}^{N} P(e_n|G) dG \tag{2}$$

As these entities are processed sequentially, the probability of assignment of a newly seen entity e_n, to a component, can be computed based on the assignments of the previous $n-1$ entities using Bayes' theorem[3] [8]. In summary,

$$Pr(e_n|e_1,...,e_{n-1}) = \begin{cases} C_i \text{ with probability } \frac{1}{n-1+\alpha} \\ OR \\ \text{new component with } \frac{\alpha}{n-1+\alpha} \end{cases} \quad (3)$$

Here, C_i represents the existing component where entity e_n can be assigned. Let there be K unique components where the previous $(n-1)$ entities are already assigned. i.e., C_k^* where $k \in \{1,...,K\}$ Hence, the above equation can be rewritten as follows:

$$Pr(e_n|e_1,...,e_{n-1}) = \begin{cases} C_k^* \text{ with probability } \frac{NUM_{n-1}(C_k^*)}{n-1+\alpha} \\ OR \\ \text{new component with } \frac{\alpha}{n-1+\alpha} \end{cases} \quad (4)$$

where $NUM_{n-1}(C_k^*)$ is the number of components discovered with $n-1$ entities. To summarize, the entities are assigned to the component based on the prior distribution of the assignments (entity to components) made. Algorithm 1 describes the complete process of PCI with this background.

ALGORITHM 1: Probabilistic Component Identification

[line:1] Data: F is the feature set representing all the entities in the software. i.e. F = $\{f_1,...,f_n\}$, such that, f_i is a feature vector for entity e_i where $i \in \{1,...,n\}$ and n is the total number of entities in the software
[line:2] Result: Components consisting of entities
[line:3] Set *maxIter* as the maximum number of iterations for the algorithm;
[line:4] Randomly initialize k such that the expected number of components is <= k;
[line:5] Sample k components, $C = \{C_1,...,C_k\}$, from a prior distribution;
[line:6] Each component C_i where $i \in \{1,...,k\}$ is represented as a distribution;
[line:7] repeat
1. Calculate the degree of fit between each entity e_i and component C_j where $i \in \{1,...,n\}$, $j \in \{1,...,k\}$. Here n is total number of entities and k is the total number of components. The degree of fit corresponds to a probabilistic value between 0 and 1 using equation (4);
2. Determine the new assignments of entities to components using degree of fit and threshold value;
3. Update the components in C using the new assignments from step 2. Now C contains an updated set of components;
until *C does not change OR maxIter iterations have been completed*;

Algorithm description. To recall, components are distributions (equation 1)). Hence Sampling a component in [line5] means means choosing parameters that represent the distribution. In case of a gaussian distribution the parameters are mean μ and standard deviation σ, along with α which determines the concentration of the distribution. To begin with the values can be random because the algorithm iteratively updates the component configurations so that there is a better degree of fit between the entities and their associated components (step 1 in line[7]). The degree of fit is calculated as a probabilistic value between 0 and 1 using the equation (4). So the higher the probability, the higher the chances of that entity being associated with the component. These new associations can lead to new components, and hence new parameters values for distributions (step2 & 3 in line[7]). After the last iteration or convergence, there are k distributions equivalent to k components in the software system. Some of these k distributions may not have any entities associated with them as initially this number was chosen to be greater than the expected number of components. So effectively, the number of components identified can vary between 1 to k.

[3]A law of probability that describes the proper way to incorporate new evidence into prior probabilities to form an updated probability estimate.

Analogy with Chinese Restaurant Process. The way PCI assign methods to the components can be understood using a Chinese Restaurant Process as follows:

- Imagine a restaurant which is initially empty.
- The first person (method 1) to enter sits down at a table (chooses a component). He then orders food for the table (i.e., selection of parameters for the component); every other person on the table will eat the same food (i.e methods in this component will have the same parameters).
- The second person (method 2) to enter sits down at a table. Which table does he sit at? With probability $\alpha/(1+\alpha)$ he sits down at a new table (i.e, selects a new component), and with probability $1/(1+\alpha)$ he sits at an existing table (chooses component of method 1).
- The N_{th} person (method N) sits down at a new table with probability $\alpha/(N-1+\alpha)$, and at table k with probability $n_k/(N-1+\alpha)$, where n_k is the number of people currently sitting at table k (i.e, the number of methods in the component k).

It is important to note that there is always a small probability that someone joins an entirely new table (i.e., a method forms a new component). As a result, the *calculateFee()* method which was forced to be associated with the credit component in the case of K-Means algorithm, now has the potential to span a new component to reflect its association with the *remittance* component.

4. EVALUATION

This section describes the studies we conducted to assess our approach. We attempt to answer the following research questions:

- How effective is the PCI algorithm in identifying components?
- What features are important in the component identification process?
- How effectively is the PCI algorithm able to combine features compared to the traditional K-Means and its variant?
- What parameters impact the component identification process and why?

4.1 Data

We evaluate our approach on RHINO, an implementation of JavaScript (ECMAScript specification) in Java. Version 1.5R6 of RHINO, which was used, consists of 32,134 source lines of Java code (excluding comments and blank lines), 138 types (classes, interfaces, and enums), 1,870 methods, and 1,339 fields.

Obtaining the Gold Set: Component identification techniques are evaluated by measuring the accuracy of the identified components. This evaluation, however, poses some challenges; unlike other fields that have produced annotated datasets for classification and recommendation problems (e.g. [15, 30] and a variety of them hosted at [16]), software engineering is lacking such datasets.

We chose RHINO because a prior study by Eaddy et al. [14] systematically associated the methods, fields and types in the RHINO source code with the ECMAScript specification. They define specification as description of a pro-

gram which may be executable (e.g., a set of program elements), or nonexecutable (e.g., a requirements specification or architectural design). Since their operational definition of a specification is an item from a program's nonexecutable specification (also known as a concern) [14], they create mapping between requirement specifications and the corresponding program elements implementing it. They manually associated 360 specifications (concerns) with program elements implementing them, thus creating 10,613 concern-element links[4]. It took them 102 man hours to create 10,613 concern-element links. These 360 specifications, according to ECMAscript specification are hierarchical in nature and can be grouped into 9 top-level specifications. We group together methods which were associated to the same top-level concern; this group of methods represents an architectural component. It is important to note that since a method can be associated with more than one concern, it can be part of more than one component. Thus, each method is mapped to one or more such identified components. We found one component in the data that had no association with any method. Hence, the set of 8 components (gold set) produced for the whole system forms the target component configuration against which we evaluate the output produced by three different algorithms under various settings. The 8 components are: Lexical Conventions, Statements, Expressions, Types, Type Conversions, Executable Code and Execution contexts, ECMAScript Objects, and Errors. The average number of methods in the components is around 850.

Creating Dynamic Features: The aforementioned study conducted on RHINO produced a mapping between methods and all the unit tests they activate. The ECMAScript test suite consists of 939 tests, 795 of which directly test 240 of the 360 requirements, i.e., several requirements have multiple tests. These tests activate 1,124 of the 1,870 methods in RHINO. Thus, the feature coverage of the test suite is 67% and the method coverage is 60%. Using this information, we created the dynamic feature set. At the end of the data preparation stage, there were 1,124 unique methods for which the dynamic features were unambiguously produced. The remaining methods were not included as they had no coverage in the test suite.

4.2 Evaluation Metrics

We use precision and recall [17] to evaluate the accuracy of the components identified. Other studies (e.g. [4]) have used these metrics before. To evaluate the identified components, precision and recall are calculated over pairs of methods. For each pair of methods that share at least one component in the identified components, these metrics try to estimate whether the prediction of this pair being in the same component was correct with respect to the underlying true categories in the data (gold set).

Precision is calculated as the fraction of pairs correctly put in the same cluster.

Recall is the fraction of actual pairs that were identified. Intuitively, we look at all pairs of methods, say (x,y), and ask whether one of the components that contains method x also contains method y. A true positive (tp) is when this is both the case in gold set and for the identified components. A false positive (fp) would be when this is not the case in the gold set but it is the case for the identified components. A false negative (fn) would be when this is the case in the gold set but not for the identified components. Then precision = tp / (tp + fp) and recall = tp / (tp + fn).

Other similarity metrics have been proposed and used before (e.g., Koschke-Eisenbarth (KE) metric, and MoJo) [21, 38]. We use precision and recall because they are robust against overlapping elements. Moreover, it is important to evaluate recall as lot of techniques do fairly well on precision, however, they perform poorly on recall. In his thesis, Koschke evaluated 23 techniques on 3 system and found that even the best technique has a recall as low as 34% in the worst case [20].

[4]http://www.cs.columbia.edu/∼ eaddy/concerntagger/

4.3 Experiments

We answer the research questions posed at the beginning of this section by performing several experiments and analyzing the results. The first experiment was performed to find out how effectively software components can be identified given only the structural features. In experiment 2, we study the impact of dynamic features on the recovered component. Experiment 3 attempts to find out which one of the algorithms most effectively combines structural and dynamic information. In all these experiments, we do not make hard assignment of entities with only one component. For example, if at convergence, the probability value of an entity e to be associated with two components A and B is 0.40 and 0.42 respectively, then we do not associate it with only B. However, we take the highest probability value (in this case, highest probability value is 0.42) and allow all the associations under 20% of the highest value. Thus, any component that has a probability value of association between 80% of 0.42 to 0.42 has an association with the entity e. We used Euclidean distance as a similarity measure for the above experiments. We perform sensitivity analysis by observing how different parameters including number of iterations (10, 20, and 40), and number of components (10 and 20) affect the process of component identification. In the end, we also look at the impact of three different distance metrics on the PCI algorithm. All results reported are an average of 3 runs.

Experiment 1: Using only source code features

In this experiment, we perform component identification based on source code features only. All three algorithms are run on the feature set consisting of lexical and structural features. As mentioned above, we use configurations having different number of iterations and components. Table 1 and Table 2 show the precision and recall respectively for all the configurations. As shown, all three algorithms produce very precise output. However, recall values for K-Means and Fuzzy K-Means are very low, whereas the recall of PCI is high.

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	98.8	98.7	98.76	98.77	98.77	98.77
F-KMeans	98.78	98.78	98.78	98.77	98.77	98.76
PCI	98.77	98.74	98.77	98.78	98.75	98.76

Table 1: Precision with only source code features

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	6.03	7.35	7.38	2.80	2.80	2.80
F-KMeans	8.77	8.77	8.77	8.70	8.70	15.43
PCI	77.40	84.55	92.47	68.79	59.25	66.96

Table 2: Recall with only source code features

Experiment 2: Using only dynamic features

In this experiment, we perform component identification based on dynamic features only. Precision and recall are shown in Table 3 and Table 4 respectively. As earlier, all three algorithms produce very precise output, however, recall for K-Means and Fuzzy K-Means is very low again. PCI has relatively good recall, but not as high as in the previous case.

This could be because the number of dynamic features is much smaller than the number of source code features due to its dependency on the completeness of the test suite. At this point, it is important to note that there are many issues

associated with the identification of dynamic features. One of them is the identification of functionalities in the software. In many cases, this may not be complete due to the lack of knowledge about the software. Another issue is the identification of test cases associated with these functionalities. This can be obtained if the test cases are well organized and packaged based on the feature they are suppose to test. However, in the absence of such organization, it is not possible to identify this association. The ECMAScript test suite consists of 939 tests, 795 of which directly test 240 of the 360 requirements, i.e., several requirements have multiple tests. These tests activate 1,124 of the 1,870 methods in RHINO. Thus, the feature coverage of the test suite is 67% and the method coverage is 60%. Moreover, structural features and lexical features are combined to form static features. Under such circumstances it is difficult to conclude that the source code features are actually better indicators of relationships in the software, although the algorithms produced better results when using them.

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	97.46	97.46	97.46	97.99	97.99	97.99
F-KMeans	97.87	97.92	98.05	99.00	99.02	98.97
PCI	99.73	96.09	99.54	98.78	99.56	100

Table 3: Precision with only dynamic features

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	5.93	5.93	5.93	5.54	5.54	5.54
F-KMeans	5.68	5.64	5.59	5.33	5.32	5.33
PCI	67.91	18.98	84.18	13.95	39.49	32.24

Table 4: Recall with only dynamic features

Experiment 3: Using combined features

In this experiment, we combined all the features (source code and dynamic) to identify the components. Table 5 and Table 6 show the precision and recall respectively for all the configurations. As with source code, and dynamic features, all three algorithms produce precise output. K-Means and Fuzzy K-Means again have low recall. The overall output when using combined feature set is not as good as it was when using source code features. This is counter-intuitive as we thought combining features might produce better results overall. In fact, this result motivated us to experiment with various distance metrics which led to different observations. We discuss the details in experiment 4.

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	97.61	97.61	97.61	97.25	97.22	97.22
F-KMeans	96.68	97.12	97.12	96.72	96.77	96.57
PCI	99.60	99.22	99.38	99.66	99.25	99.36

Table 5: Precision with combined features

Experiment 4: Sensitivity analysis

We use three different distance metrics namely Euclidean, Manhattan, and Cosine [8] to study their impact on the component quality.

We ran PCI with fixed number of components (10), all three feature types, and multiple iterations (10, 20 and 40). The results are shown in Tables 7 and 8. It turns out that Cosine metric when used in conjunction with combined features produces the best results. On the other hand, Manhattan and Euclidean perform better when used in conjunction with source code features. We speculate that this is because Euclidean and Manhattan metrics depend on magnitude (length) of the two vectors whereas Cosine does not. This can be reasoned as follows: Given $\vec{x} = (x_1, x_2)$ and

Algorithm/	Components = 10			Components = 20		
Iterations	10	20	40	10	20	40
K-Means	7.00	7.00	7.00	5.92	5.85	5.85
F-KMeans	4.39	5.11	5.11	4.38	4.78	4.48
PCI	48.02	39.58	61.59	56.13	43.45	56.96

Table 6: Recall with combined features

$\vec{y} = (y_1, y_2, y_3, y_4)$ as the two vectors in R^2 and R^4 respectively. Now the Euclidean distance and Manhattan distance between them is given by:

$$E_d(\vec{x}, \vec{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + y_3^2 + y_4^2} \quad (5)$$

$$M_d(\vec{x}, \vec{y}) = |(x_1 - y_1) + (x_2 - y_2) + y_3 + y_4| \quad (6)$$

Similarly, Cosine distance is given by:

$$\cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{\| \vec{x} \| \| \vec{y} \|} \quad (7)$$

As seen in equation 5, since $\vec{y}$ has higher dimension than $\vec{x}$, the terms y_3^2 and y_4^2 are introduced without any subtraction terms. As a result, the overall distance between the vectors increases just because of the difference in their length. This is also the case with Manhattan distance metric. However, Cosine metric has no effect with the change in length as it treats both vectors as unit vectors by normalizing them (see magnitude terms in the denominator of equation 7), giving a metric of the angle between the two vectors. It does provide an accurate measure of similarity but with no regard to magnitude.

The feature vectors representing software entities are asymmetric and sparse because not all entities are related to all other entities in the software. Hence, feature vectors representing software entities have varied length. Now this variation in length is even larger when we combine source code and dynamic features because the length of both vectors increases, but with different proportions. As a result, Manhattan and Euclidean produced good results with only source code features but their performance degraded when the features were combined. On the contrary, Cosine metric gave better results when used with combined features because it is robust against change in the magnitude. This property allows it to compare asymmetric vectors elegantly, at the same time exploiting the information gain due to added features.

4.4 Study Summary

We summarize the main findings of our study which correspond to the four research questions posed in the beginning of this section.

- All three algorithms had very high precision, but only PCI had high recall. So non-parametric probabilistic model based techniques when tailored for software context proved to be very effective in this case.
- Source code features outperformed dynamic features. The algorithms produced better results when using them. But it is unfair to say that they are better indicators of relationships in the software because of the difference in number of features. Also, it is unrealistic to encounter a software with a test suite which has complete feature and method coverage.
- Euclidean distance measure may not be effective in combining features. However, PCI with the Cosine distance measure gave the best results with combined features.
- Parameters like number of iterations, number of components, and distance metric have noticeable impact on the performance of the algorithm. PCI can automatically infer the number of components and hence it is robust against any change in the specified number of components.

Distance Metric	10 Iterations			20 Iterations			40 Iterations		
	Structural	Dynamic	Combined	Structural	Dynamic	Combined	Structural	Dynamic	Combined
Cosine	98.74	98.81	99.54	98.77	99.07	99.59	98.76	99.07	99.51
Manhattan	98.73	98.73	98.94	98.72	98.81	99.21	98.76	99.44	95.24
Euclidean	98.77	99.73	99.60	98.74	96.09	99.22	98.77	99.54	99.38

Table 7: Precision for various distance metrics

Distance Metric	10 Iterations			20 Iterations			40 Iterations		
	Structural	Dynamic	Combined	Structural	Dynamic	Combined	Structural	Dynamic	Combined
Cosine	59.79	83.50	89.87	56.78	88.03	92.39	55.28	87.94	89.95
Manhattan	76.77	14.83	36.69	73.67	9.03	49.55	96.14	34.01	18.94
Euclidean	77.40	67.91	48.02	84.55	18.98	39.58	92.47	84.18	61.59

Table 8: Recall for various distance metrics

5. ANALYSIS AND IMPLICATIONS

In general, the recall for K-Means and Fuzzy K-Means is low irrespective of the type of features, number of components, and iterations. In our understanding, this is because of the exclusive assignment nature of these two algorithms. K-Means creates exclusive assignments, whereas Fuzzy K-Means near exclusive assignments. As a result, they produce small clusters, without much overlap, because of which there are fewer number of intra pairs. This impacts the overall recall because gold components have overlapping components increasing the size of clusters and eventually the number of intra pairs. PCI on the other hand, produces soft entity assignments allowing entity to be associated with multiple components. This increases overall recall because of the increase in total number of intra pairs.

So if the subject software is believed to be modular and having distinct functionalities even K-Means and Fuzzy K-Means should produce realistic components. However, if the software has undergone evolution and degraded from its original modular design, PCI might be useful as it will produce overlapping components allowing the architect to make design decision about separating utility components (set of highly overlapping entities) from domain components (non or minimal overlapping entities), creating APIs or re-architecting the software.

Another observation is the impact of specifying the number of components on the recall. As observed, the recall value decreases for K-Means as the number of specified components increases. We conjecture that this is because K-Means is a parametric algorithm, creating exactly the number of components as specified by the user. So if the user has wrongly estimated number of components to be higher than the actual number, K-Means is forced to allocate the entities to more components. Since K-Means does exclusive assignments, the increase in number of components will ultimately reduce the overall size of each component and hence lower recall. Thus in the result shown above, there is a drop in recall for K-Means when the number of components specified increases from 10 to 20, which is higher than 8 (the actual number of components in the gold set). Fuzzy K-Means is not impacted by this parameter as it does a pre-clustering step called canopy clustering to infer the approximate number of components; whereas PCI, as discussed in section 3.2, has a property to automatically infer the number of components. Thus PCI and Fuzzy K-Means are more or less robust against any change in the specified number of components. We experimented with many configurations (10, 20, 40, and 300) for the number of components for PCI. It did not have any noticeable impact on the accuracy. The average runtime for 10, 20 and 40 components was between 10-12 seconds. However, the average runtime for 300 components was between 45-60 seconds. Thus, increasing the number of components to an unrealistically large value may not have any impact on accuracy but it may increase the convergence time of the algorithm and eventually increases the run time.

The number of iterations also seems to have an impact on the output produced by various algorithms. In case of K-Means and Fuzzy K-Means, the algorithm stops at convergence. Increasing the number of iterations beyond this point does not have any impact on the output. PCI on the other hand is a self adjusting algorithm. The number of components considered at each step can potentially vary at every iteration. For this reason, the number of iterations parameter seems to have a more noticeable impact on PCI's performance. In spite of the minor changes in performance, as observed in the results, PCI remains fairly consistent. This flexible nature of the algorithm is also one of the reasons it is able to infer the number of components automatically.

Although it is observed that component identification is precise with more data, we observed that the choice of distance metric is equally important. Few distance metrics depend on the magnitude of an input vector. So a preliminary analysis of the size of methods in the software can help in choosing the right distance metric. If the distribution of method sizes is uniform, Euclidean family can perform well, however, in case of non-uniform distribution, Cosine or Entropy metrics might yield better results.

Apart from other factors, the choice of the algorithm is driven by the existing knowledge about the software. If one has adequate knowledge about modularity and evolution of the software, then even simple algorithms like K-Means can prove to be useful. For example, if one is already aware about the utility components, removing them from analysis will lead to fewer overlapping components. In such a scenario, K-Means can be a better choice because of its simplicity, provided one specifies near right number of components. However, in case of inadequate knowledge, non-parametric approaches like PCI might have more to offer as they are more elegant in inferring structure without a prior knowledge.

6. THREATS TO VALIDITY

Internal validity: The implementation is based on top of Mahout, which is a reliable machine learning library. K-Means and Fuzzy K-Means have already available implementation in Mahout. We implemented PCI by extending the Dirichlet Process implementation. Dynamic features were generated by instrumenting the code using AspectJ. The results of the experiments are verified for consistency and correctness. Unlike some of the other clustering approaches (based on gradient descent) which have issue of implicit randomness, where initial assignments are by chance, PCI is robust against such changes. However, K-Means and Fuzzy K-Means are susceptible to local minima. The gold set used in our study was created subjectively. Hence, there is a possibility of some bias being introduced in the creation. In the future, we would like to validate our technique on

more systems to address this concern. However, to ensure repeatability of the experiments, we have made the gold set, feature set and identified components data[5] available.

External validity: There is a very limited extent, if any, to which we can generalize our findings to other contexts. Since different architectural views of software reflect the goals and assumptions of users, it is difficult for one technique to produce best results in all cases. Although our technique was demonstrated to be useful in recovering the functional components of the system, it may need to be specialized to perform well for other views and goals. We used RHINO as the subject system as it has been previously used in many studies [12, 13, 14]. A further investigation on different software systems will be helpful. We plan to validate our work by further investigating it on a larger commercial financial software system used in our previous work [33].

7. RELATED WORK

The general practice of recovering the design of a software system takes a knowledge-based approach. This is a bottom up approach in which one reverse engineers units of source code to understand it using existing domain knowledge. Combination of these units together leads to the understanding of system's design. This approach has been shown to work well with small systems [39], and, in the absence of tools, this is the only possible approach to recover software architectures.

However, this approach requires a fairly manual process, and it can be painstaking to do this for large systems. Moreover understanding the architecture of a piece of software in a relatively new domain is often the first step developers take in order to use that software effectively, so requiring domain knowledge upfront is a barrier to effective use. For these reasons, the software research community has been seeking other, more automated, approaches such as structure-based, explained next.

Structure based approaches look at syntactic interactions, such as method invocation or field access, between code entities to arrive at software decomposition. The problem transforms into a graph partition problem where nodes are methods or fields, and the edges are relations between them. The objective function is to achieve a partition such that similar entities are grouped together.

Belady and Evangelisti [7] were the first to extract the information from the system's documentation to automatically cluster a software system in order to reduce its complexity. Later Hutchens and Basili [19] extracted data binding information from the source code instead of documentation. Based on this data binding they construct a hierarchy from which a partition can be derived.

Schwanke's tool called ARCH [36] introduced the concept of clustering software entities based on low coupling and high cohesion heuristics. Although the approach was never tested against a large software system, it demonstrated promise. Also the complexity of algorithm was cubic which makes it difficult to work on large systems. Neighbours [28] investigated compile-time and link time dependencies among components to identify subsytems with the ultimate goal of extracting reusable components. Anquetil and Lethbridge [3] looked at informal information like names of various entities of the system. Their approach showed that the idea of mining identifier names could be useful to gain program understanding. However, the disadvantage is that it relies heavily on the developers' style and consistency for naming identifiers.

Few researchers have also applied techniques from other disciplines to improve software architecture recovery. These techniques include concept analysis [22, 37], association rule mining [29], and clustering [40, 4, 10, 32, 1, 27]. Mancoridis [23] treated clustering as an optimization problem. He used genetic algorithms to increase the speed of clustering process. Andritsos and Tzerpos used entropy metrics from information theory to group entities together based on the minimization of the information loss [2]. Sartipi et al. use rule mining to recover a system's architecture [35].

[5]http://www.ics.uci.edu/∼ hsajnani/files/

8. FUTURE WORK

Automated approaches to component identification are useful. However, there are weaknesses, as decisions (or feedback) provided for some portions of the system influences other parts. This makes it unlikely that a purely automated technique can be used without human intervention to reconstruct the architecture. In our understanding, one of the most challenging issues is to incorporate human knowledge into an architecture recovery algorithm. This is where supervised machine learning approaches could be useful. *How can we extend the above mentioned iterative approach to incorporate human-feedback in the learning process, so as to refine preliminary (automatic) solutions toward a more meaningful result?*

Besides the identification of components, architecture recovery requires recovering knowledge about the types of those components and the relationships between them. For example, it is common for projects to have utility components (e.g. calendar module, error handling module) and application components (e.g. credit module in a banking software). Knowledge about these types is important for understanding a system. *Can we train a classifier by looking at existing utilities and use that to classify components with high accuracy?*(e.g., [18])

Architecture recovery is not complete without human understandable semantics. Knowing that there exist certain components is of little value unless the method also associates meaningful descriptions with those components.*How can advanced information retrieval and statistical techniques improve the labeling of the recovered components?*

9. CONCLUSION

This paper presented the notion of learning based non-parametric probabilistic approach as a valid basis for component identification in software systems. We proposed a new algorithm that is tailored for software and evaluated it along with the two existing algorithms. Our experiments highlighted the impact of various parameters on the component identification process and enumerated which approach is more suitable under what circumstances. The proposed algorithm when used with a proper distance metric has the ability to effectively combine information from various aspects of software. Due to its probabilistic nature, it can produce overlapping components, which is in agreement with the paradigm of software systems. Since the approach is based on a non-parametric model, it enumerates more possibilities as it sees more data and can correct itself at later stages of the component identification process. This makes it an ideal candidate to address our problem, where the distinct number of components is unknown beforehand.

10. REFERENCES

[1] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Trans. Softw. Eng.*, 31(2):105–165, 2005.

[2] Periklis Andritsos and Vassilios Tzerpos. Software clustering based on information loss minimization. In *Proceedings of WCRE*. IEEE Computer Society, 2003.

[3] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON*, 1997.

[4] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of WCRE*, pages 235–255, 1999.

[5] Apache lucene. http://lucene.apache.org.

[6] Fabian Beck and Stephan Diehl. Evaluating the impact of software evolution on software clustering. In *Proceedings of WCRE*. IEEE Computer Society, 2011.

[7] L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 29:2–23, 1981.

[8] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Proceedings of European Conference on Software Maintenance and Reengineering*, 2009.

[10] J. Davey and E. Burd. Evaluating the suitability of data clustering for software remodularization. In *Proceedings of WCRE*, pages 268–277, 2000.

[11] Nicholas DiGiuseppe and James A. Jones. Concept-based failure clustering. In *Proceedings of FSE*, 2012.

[12] Bogdan Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 2012.

[13] M. Eaddy, A. V. Aho, G. Antoniol, and Yann-Gael Gueheneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of ICPC*, 2008.

[14] Marc Eaddy, Thomas Zimmermann, Kaitlin Sherwood, Vibhav Garg, Gail Murphy, Nachiappan Nagappan, and Alfred Aho. Do crosscutting concerns cause defects. *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008.

[15] Hugo Jair Escalantea, Carlos A. HernÃąndeza, Jesus A. Gonzaleza, A. LÃşpez-LÃşpeza, Manuel Montesa, Eduardo F. Moralesa, L. Enrique Sucara, Luis VillaseÃśora, and Michael Grubinger. The segmented and annotated iapr tc-12 benchmark. *Computer Vision and Image Understanding*, 114(4):419–428, 2012.

[16] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[17] Salton G and McGill MJ. *Introduction to modern information retrieval*. McGraw Hill, 1983.

[18] Joshua Garcia, Daniel Posescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing architectural recovery using concerns. In *Proceedings of ASE*, 2011.

[19] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, 11(7):749–757, 1985.

[20] R. Koschke. *Atomic architectural component recovery for understanding and evolution*. PhD thesis, University of Stuttgart, 2000.

[21] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceeding of IWPC*, pages 201–210, 2000.

[22] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of ICSE*, pages 349–359, 1997.

[23] S. Mancoridis and et al. B. Mitchell. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of IWPC*. IEEE Computer Society, 1998.

[24] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33(11):759–780, 2007.

[25] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action*. Manning, 2010.

[26] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution*. PhD thesis, University of California, Irvine, 1999.

[27] Brian Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208, 2006.

[28] J. M. Neighbours. Finding reusable software components in large systems. In *Proceedings of WCRE*. IEEE Computer Society, 1996.

[29] C. Montes De Oca and D.L. Carver. Identification of data cohesive subsystems using data mining techniques. In *Proceedings of ICSM*, pages 16–23. IEEE Computer Society, 1995.

[30] Martin Potthast. Crowdsourcing a wikipedia vandalism corpus. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 789–790, New York, NY, USA, 2010. ACM.

[31] R.N.Taylor, N.Medvidovic, and E.M.Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, 2010.

[32] M. Saeed, O. Maqbool, H.A. Babri, S.M. Sarwar, and S.Z. Hassan. Software clustering techniques and the use of the combined algorithm. In *In Proceedings of European Conference on Software Maintenance and Reengineering*, pages 301–306, 2003.

[33] Hitesh Sajnani, Ravindra Naik, and Cristina Lopes. Application architecture discovery: Towards domain-driven easily extensible code structure. In *Proceedings of WCRE*, 2011.

[34] Hitesh Sajnani, Ravindra Naik, and Cristina Lopes. Easing software evolution: A change-data and domain-driven approach. In *Proceedings of ISEC*, 2012.

[35] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Design recovery using data mining techniques. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 129–140, 2000.

[36] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of International Conference on Software Engineering*, pages 83–92, 1991.

[37] P. Tonella. Concept analysis for module restructuring. *IEEE Trans. Softw. Eng.*, 27(4):351–363, 2001.

[38] V. Tzerpos. *Comprehension-Driven Software Clustering*. PhD thesis, University of Toronto, 2001.

[39] Vassilios Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *In Proceedings of the 9^{th} International Workshop on Database and Expert Systems (DEXA'98)*, pages 811–818, 1998.

[40] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of WCRE*. IEEE Computer Society, 1997.