Mapping Program Elements to Layers using Centrality Measures and Machine Learning Techniques

Sanjay B Thakre, Arvind W Kiwelekar

Department of Computer Engineering, Dr Babasaheb Ambedkar Technological University, Lonere-Raigad, 402103, India

Abstract

The necessity of explicit architecture descriptions for communicating system functionalities and system maintenance activities has been continuously emphasized. This paper presents an approach to extract layering information, a form of architecture descriptions, using the *centrality measures* from Social Network Analysis theory and supervised machine learning algorithms. The layer recovery approach presented in this paper works in two phases. The first phase calculates centrality measures for each program element in an application. The second phase assumes that the application has been implemented around the layered architecture style and maps program elements to appropriate layers. Two techniques for mapping program elements to layers are presented. The first technique maps program elements to layer using pre-defined rules, while the second technique learns the mapping rules from a pre-labelled data set. The paper presents the evaluation of both approaches.

Keywords

Layered Architecture Style, Architecture Descriptions, Architecture Recovery, Centrality Measures, Module Dependency View, Supervised Classification.

1. Introduction

The value of explicit software architecture has been increasingly recognized for software maintenance and evolution activities [1]. Especially architecture descriptions in terms of high-level abstractions such as patterns, styles and views have been found as a valuable tool to communicate system functionalities effectively [2, 3]. Despite the numerous benefits, a legacy or open-source software system often lacks such kind of architecture descriptions. Moreover, when such architecture descriptions are available, they are not aligned with the latest version [3].

A lightweight architecture recovery approach that approximately represents a system decomposition may be more convenient than sophisticated architecture recovery techniques in such situations. Such a light approach shall quickly extract relevant information necessary to build a system decomposition so that it can provide much-needed assistance to software architects dealing with re-engineering or modernization of existing systems, thus increasing their productivity.

Intending to design a lightweight approach, this paper presents an architecture recovery to extract layered decomposition of an implemented system. The method uses *centrality measures* from the theory of Social Network Analysis [4] to analyze software structure formed by *dependency relationship*. Three observations drove the rationale behind using centrality measures for architecture extraction: (i) Most of these measures provide a highly intuitive and computationally simple way to analyze interactions when a graph represents the structure of a system. (ii) These measures quantify the structure of a system at multiple levels, i.e., at a particular node level, concerning other nodes in the graph, and at a group of nodes or communities. (iii) These measures support the development of data-driven approaches to architecture recovery. Hence such approaches can learn the rules of architecture recovery from given data. The approach recovers layering information in two phases. In the first phase, a centrality score is assigned to each program element. We assume that system functionalities are decomposed among multiple layers, and so in the second phase, a layer is assigned to each program element.

The paper contributes to the existing knowledge base of the architecture recovery domain in the following ways. (1) It demonstrates *the use of centrality measures* to recover layering information. (2) It describes a *data-driven* approach to mapping program elements to layers using supervised classification algorithms. (3) It presents an evaluation of supervised classification algorithms to extract layering information. The rest of the paper is organized as follows: Section II defines various centrality measures. Section III describes the central element of the approach. The algorithmic and data-driven approaches to the problem of layer assignment are explained in Section IV. The evaluation of the approach is presented in Section V. Section VI puts our approach in the context of existing approaches. Finally, the paper concludes in Section VII.

ECSA2021 Companion Volume, Robert Heinrich, Raffaela Mirandola and Danny Weyns, Växjö, Sweden, 13–17 September, 2021

mail2sbt@gmail.com (S. B. Thakre); awk@dbatu.ac.in (A. W. Kiwelekar)

https://awk-net.group/ (A. W. Kiwelekar)

0000-0002-9647-6403 (S. B. Thakre); 0000-0002-3407-0221 (A. W. Kiwelekar)

2. Social Network Analysis Measures

The theory of Social Network Analysis (SNA) provides a generic framework to analyze the structure of complex systems. It includes a rich set of measures, models, and methods to extract the patterns of interactions among systems' elements. A complex system is expressed as a network of nodes and edges to support systems from diverse application domains.

A few examples of complex systems that have been analyzed with the help of SNA include communities on social media platforms [5], and neural systems[6]. The techniques from SNA have been applied to control the influence of diseases [7] to understand biological systems [8], and to investigate protein interactions [9]. In these applications of SNA, a complex system is represented as a graph, and then they are analyzed through measures such as centrality, scale-free, small world, and community structure [10, 11, 12, 13]. Some of these commonly used SNA measures relevant to our study are described below.

The theory of Social Network Analysis (SNA) provides a range of measures with varying levels. Some are applied at the *node-level* while others are applied at the *network-level*. The node-level measures are calculated from the nodes which are directly connected to a given node. The *centrality measures* are the node-level measures quantifying the importance of an individual node in the network. A central node is an influential node having significant potential to communicate and access information. There exists different *centrality measures*, and they are derived from the connections to a node, position of a node in the network, and relative importance of nodes.

2.1. Degree centrality

This measure determines a central node based on the connections to individual nodes. A node with a higher degree in a network is considered the most influential one. In a directed graph, two different centrality measures exist *in-degree* and *out-degree* based on the number of incoming and outgoing edges, respectively. The degree centrality, $C_D(v)$, of a node, v, is equal to the number of its connections, $deg(v)$, normalized NC_D to the maximum possible degree of the node.

$$C_D(v) = deg(v) \tag{1}$$

$$NC_D = \frac{C_D(v)}{n-1} = \frac{deg(v)}{n-1} \tag{2}$$

2.2. Closeness centrality

This measure identifies an influential node in terms of a faster and broader spread of information in a network. The influential nodes are characterized by a smaller internode distance which signifies the faster transfer of information. The closeness centrality is derived from the average distance from a node to all the connected nodes at different depths. However, the distance between the disconnected components of the network is infinite, and hence it is excluded. For the central node, the average distance would be small and is calculated as the inverse of the sum of the distances to all other nodes (d_{vw}). The normalized closeness (NC_C) is in the range from 0 to 1, where 0 represents an isolated node, and 1 indicates a strongly connected node.

$$C_C(v) = \sum(\frac{1}{d_{vw}}) \tag{3}$$

$$NC_C = \sum(\frac{n-1}{d_{vw}}) \tag{4}$$

2.3. Betweenness centrality

This measure identifies those central nodes which are responsible for connecting two or more components of the network. Removal of such a central node would mean a disconnection of the complete network. Hence, these nodes act as a bridge to pass the information [12, 14]. Betweenness centrality ($C_B(v)$ is defined as the number of shortest paths passing through a node v.

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5}$$

where, σ_{st} is the total number of shortest paths from a node s to t and $\sigma_{st}(v)$ is the number of paths that pass through v. The relative betweenness centrality, $C_B^{'}(v)$, of any node in a graph with respect to the maximum centrality of the node is calculated from $C_B(v)$.

$$C_B^{'}(v) = \frac{2C_B(v)}{n^2 - 3n + 2} \tag{6}$$

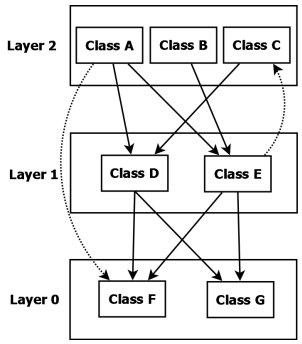
2.4. Eigenvector centrality

The Eigenvector centrality is a relative centrality measure, unlike the last three measures that are absolute. The Eigenvector centrality calculation depends on the largest real Eigenvalue present in the symmetric adjacency matrix. The centrality of a node v is proportional to the sum of the centralities of the nodes connected to it [15, 12].

$$\lambda v_i = \sum_{j=1}^{n} a_{ij} v_j \tag{7}$$

In general, it requires the solution of the equation $Av = \lambda v$ where A is an adjacency matrix.

Figure 1: An Example of Class Dependencies with and their centrality scores.



PID	in-degree	out-degree	$C_D(v)$	$C_B(v)$	$C_C(v)$	λv_i
Class A	0	3	3	0	0.71	0
Class B	0	1	1	0	0.5	0
Class C	1	1	2	2	0.6	0.055
Class D	2	2	4	2.5	1	0.27
Class E	2	3	5	5.5	0.8	0.0055
Class F	3	0	3	0	0	1
Class G	2	0	2	0	0	0.99
Layer 2	1	5	6	0	1	0.5
Layer 1	4	5	9	1	1	0.5
Layer 0	5	0	5	0	0	1

Figure 1 shows the centrality scores of various programming elements calculated by considering the dependencies as shown in the figure. Here, it is to be noted that centrality scores can be calculated at different granularity levels, i.e., at the object, method, class, package, or logical layer. In the figure, centrality scores are computed at class and layer level. Here, we have considered a layer as a logical encapsulation unit, loosely holding multiple classes.

3. Approach

The broad objective of the approach is to extract high-level *layering information*, a form of architecture descriptions, from the implementation artefacts so that software architects can perform the analysis specific to layer architecture style. In this paper, we demonstrate our approach with the help of implementation artefacts available in a Java-based system. The method assumes that a system under study is implemented around the layered architecture style. Software maintenance engineer can verify such assumptions from the earlier documentation of the system when available. As shown in Figure 2, the approach consists of following two phases.

1. **Dependency Network Builder and Analysis:** This phase retrieves the dependencies present among implementation artefacts. For a Java-based system, this phase takes Java or Jar files as an input and generates a dependency network. The programming elements such as *Classes, Interfaces*, and *Packages* are the nodes of the network and the Java relationships such as *extends*, *implements*, and *imports* are the edges in the dependency network. The output of this stage is represented as a graph in Graph Modeling Language (GML). In the second stage, a centrality score to each node is assigned. The centrality score includes the different measures described in Section 2, and they are calculated at the Class and Interface levels. The output of this stage is a data file in the CSV format describing the centrality score assigned to each program element.
2. **Layer Assignment:** The layer assignment activity aims to assign the most appropriate layer to a program element. Additional style-specific analyses such as analysis of layer violations, and performance modeling can be supported once the programming elements are assigned to appropriate layers.

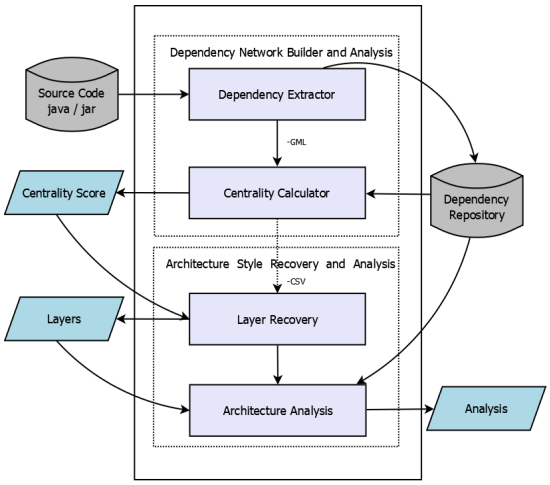


Figure 2: Block diagram of a tool implement in Java to discover layered architecture using centrality.

Building a dependency network and calculating centrality scores are straightforward activities to realize when the tools such as *JDependency* [1] are available. However, assigning program elements to layers is a trickier issue considering the large number of program elements and

[1] https://sourceforge.net/projects/javadepend/

Centrality	upper	middle	lower
in-degree	low	-	high
out-degree	high	-	low
between-ness	low	high	low
close-ness	high	high	low
eigenvector	low	low	high

Table 1
Relative Significance of centrality measures with respect to layers

δ_{il}, δ_{iu}	Lower and upper bound for in-degree centrality values.
δ_{ol}, δ_{ou}	Lower and upper bound for out-degree centrality values.
δ_b	Critical Value for between-ness centrality
δ_c	Critical Value for close-ness centrality
δ_e	Critical Value for eigen-value centrality

Table 2
Configuration Parameters

relationships among them are present. We describe two techniques for this purpose in the following section.

4. Layer Assignment

The objective of the layer assignment stage is to identify the most appropriate layer based on the centrality measures. Here, we use the term *layer* in the loose sense that a *layer* is a logical coarse-grained unit of decomposing system functionalities and encapsulating program elements according to common system functionalities. It is not the term *layer* in the strict sense as that used in *Layer architecture style* [16].

We assume a *three-layers* based decomposition. The decision to decompose all the system responsibilities into three layers is based on the observation that most architectural styles' functionalities can be cleanly decomposed into three coarse-grained units of decomposition. For example, architectural style such as Model-View-Controller (MVC), Presentation-Abstraction-Control (PAC),[16], and 3-tier architectural patterns have three units of system decomposition.

Two different techniques based on centrality measures are developed to assign program elements to layers. The first techniques uses a set of pre-defined rules. The second technique automatically learns the assignment rules from the pre-labelled layer-assignments using a supervisory classification algorithm.

4.1. Rule-Driven Layer Assignment

This technique uses centrality measures, described in Section 2, to map program elements to the most appropriate layer. The measure of *degree centrality* from Section 2.1 is further divided as *in-degree* and *out-degree* which count the number of incoming and outgoing edges of a node. Total five measures of centrality are used. Five accessor functions namely *in*, *out*, *between*, *closeness* and *eigen* are defined to get the values of in-degree centrality, out-degree centrality, betweenness centrality, closeness centrality and eigenvector centrality associated to a specific node respectively.

Three layers are labelled as, i.e. *upper, middle and lower*. Table 1 describes the relative significance of various centrality measures with references to *upper, middle and lower* layers. A set of configuration parameters, as shown in Table 2, are defined. These parameters provide flexibility while mapping program elements to a specific layer.

The **Algorithm** 1 is operated on a dependency-network in which nodes represent program elements while edges represent dependencies. The objective of this algorithm is to partition the node space representing into three segments corresponding to lower, middle and upper layers. This objective is achieved in two stages.

Algorithm 1 : primaryLabel(inDegree, outDegree, n)
Input: inDegree[1:n],outDegree[1:n]: Vector, n:Integer
Output: inPartition[1:n], outPartition[1:n] Vector

```
1:  Initialize δ_iu, δ_il, δ_ou and δ_ol
2:  for node in 1 to n do
3:      if in(node) = 0 and out(node) = 0 then
4:          inPartition[node] ← lower
5:          outPartition[node] ← lower 5
6:      else
7:          if in(node) > δ_il then
8:              inPartition[node] ← lower
9:          else
10:             if in(node) < δ_iu then
11:                 inPartition[node] ← upper
12:             else
13:                 inPartition[node] ← middle
14:             end if
15:         end if
16:         if out(node) > δ_ou then
17:             outPartition[node] ← upper
18:         else
19:             if out(node) < δ_ol then
20:                 outPartition[node] ← lower
21:             else
22:                 outPartition[node] ← middle
23:             end if
24:         end if
25:     end if
26: end for
```

First, the algorithm calculates two different partitions. The first partition i.e. *inPartition* is calculated using the *in-degree* centrality measure while the second partition is calculated using the *out-degree* centrality measure.

Algorithm 2 refineLabel(inParticion, outPartition, n)
Input: inPartition[1:n], outPartition[1:n]: Vector
n: Integer
Output: nodeLabels[1:n]: Vector

1: Initialize δ_b, δ_c and δ_e
2: **for** *node* in 1 to n **do**
3: **if** $inPartition[node] = outPartition[node]$ **then**
4: $nodeLabels[node] \leftarrow outPartition[node]$
5: **else**
6: $nodeLabels[node] \leftarrow upDown(inPartition[node], outPatition[node])$
7: **end if**
8: **end for**

After the execution of **Algorithm** 1 each node is labeled with two labels corresponding to layers. The various combination of labels include *(lower, lower), (middle, middle), (top, top), (middle, top),* and *(middle, lower).* Out of these six labelling, the labels *(middle, top),* and *(middle, lower)* are conflicting because two different labels are assigned to a node. This conflict needs to be resolved.

The **Algorithm** 2 resolves the conflicting labels and assigns the unique label to each node. The conflicting labels are resolved by using the rules described in the decision Table 3. The function *upDown* called in the **Algorithm** 2 uses these rules. The rules in Table 3 resolve the conflicting assignments using the centrality measures of *close-ness, between-ness*, and *Eigen vector*, while the primary layer assignment by **Algorithm** 1 is done with *in-degree* and *out-degree* centrality measures. When **Algorithm** 2 is executed, some of the nodes from the middle layer bubble up to the upper layer, and some nodes fall to the lower level. Some nodes remain at the middle layer. The vector *nodeLabels* holds the unique labelling of each node in the dependency network after resolving all conflicts.

4.2. Supervised Classification based Layered Assignment

The *configuration parameters* need to be suitably initialized for the correct functioning of the *rule-driven* approach discussed in the previous section. The system architect responsible for architecture recovery needs to fine tune the parameters to get layering at the desired level of abstraction. To overcome this drawback a *data-driven* approach is developed to assign labels to the programming elements.

Table 3
Decision Table used to Refine Layering

Layer	Measure	Significance	Rationale
upper	in	0	Classes with in-degree value equal to 0 are placed in the upper layer.
	out	high	Classes with high out-degree are placed in the top layer because they use services from layers beneath them.
	closeness	high	Classes with high closeness value are placed in the upper layer because of large average distance from top layer to bottom layer.
middle	between	high	Classes with high betweenness value are placed in the middle layer as they fall on the path from top layer to bottom layer.
lower	in	high	Classes with high in-degree value are placed in the bottom layer because they are highly used.
	out	0	Classes with out-degree value equal to zero are placed to bottom layer because they only provide services.
	eigen	1	Classes with eigen value equal to 1 are placed to bottom layer because they are highly reused.
	in	-	Classes with in-degree and out-degree values are equal to 0 are placed to bottom layer, because they are isolated classes.

In the data-driven approach, the problem of layered assignment is modeled as a multi-class classification problem with three labels i.e. *lower (1), middle (2) and upper (3)* with numerically encoded as 1,2, and 3 respectively. The classification model is trained on the labeled data-

Table 4
Sample observations from the Datasets used for Supervised Learning

Id	Label	In-Degree	Out-Degree	Close-ness	Betweenness	Eigenvec-tor	Layer
HealthWatcher							
1	ComplaintRecord	1	10	1.714	19	0.0056	2
2	ObjectAlreadyInsertedException	37	0	0	0	0.347	1
3	ObjectNotFoundException	53	0	0	0	0.943	1
4	ObjectNotValidException	41	0	0	0	0.883	1
5	RepositoryException	60	0	0	0	1	1
ConStore							
1	Cache	2	1	1	0	0.0162	2
2	CacheObject	4	0	0	0	0.053	2
3	LRUCache	0	2	1	0	0	2
4	MRUCache	1	2	1	7	0.0246	2
5	ItemQuery	1	20	0.412	47.166	0.0388	2

set. The data set, as shown in Table 4, includes program element identifiers, values of all the centrality measures and layering labels as specified by the system architect responsible for architecture recovery. The layering labels can be used from the previous version of the system under study or the labels guessed by system architect to explore different alternatives for system decomposition.

We implement three supervised classification algorithm namely K-Nearest Neighbour, Support Vector Machine, and Decision Tree. These are the machine learning algorithms particularly used for multi-class classification problems. A detailed comparison of these various algorithms can be found in [17]. Python's Scikit-Learn [18] library is used to develop classification model based on these algorithms. Table 4 shows the format of the sample dataset used to train the classification models. The developed models are evaluated against the classification metrics such as accuracy, precision, recall, and F1-Score.
.

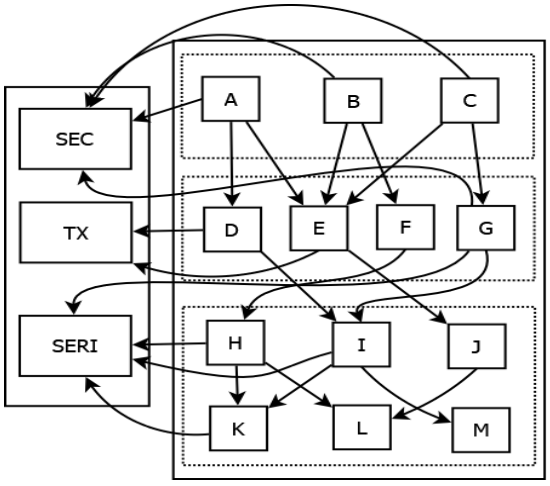


Figure 3: An Architecture of a System Designed to Train the approach

5. Model Development

The machine learning model development includes phases like training, testing and evaluation. This section describes how these phases are carried out.

5.1. Model Training

The following software systems are used to train and build the machine learning models.

1. **Training Architecture system**: A small-scale sample architecture system, as shown in Figure 3, has been designed specially to train the approach. It includes 16 classes without the implementation of any functionalities; it contains only dependencies among classes, as shown in Figure 3. The classes named as *SEC, TX, SERI* in the figure represent crosscutting concerns.
2. **HealthWatcher:** The HealthWatcher is a web-based application providing healthcare-related services [19]. This application is selected to train the model because existing literature in the public domain confirms that the application follows a client-server and layered architecture style. Initially, the first author has manually assigned labels to all the programming elements. Later the second author checked the labelling of individual elements. We used the following rules to label programming elements. These are: (i) Programming elements that access low-level device functions and data access functions are labelled as *lower*. (ii) Programming elements accessing pre-

Table 5

Accuracy and Confusion Matrix for Data-Driven and Algorithmic Approach

SVM				Decision Tree			KNN classifier			Rule based		
HealthWatcher(Size: 135 classes or interfaces) Confusion Matrix												
Layer	lower	middle	upper	lower	middle	upper	lower	middle	upper	lower	middle	upper
lower	47	1	9	49	4	4	41	8	8	28	16	13
middle	20	5	12	15	20	2	7	28	2	6	30	1
upper	5	1	35	7	0	34	6	6	29	3	9	29
Accuracy = 0.64				Accuracy = 0.76			Accuracy = 0.72			Accuracy = 0.63		
Recall (R), Precision (P), F1-Score (F1) Evaluation												
	R	P	F-1	R	P	F-1	R	P	F-1	R	P	F-1
lower	0.65	0.82	0.73	0.69	0.86	0.77	0.76	0.72	0.74	0.76	0.49	0.60
middle	0.71	0.14	0.23	0.83	0.54	0.66	0.67	0.76	0.71	0.55	0.81	0.65
upper	0.62	0.85	0.72	0.85	0.83	0.84	0.74	0.71	0.72	0.66	0.66	0.66
Training Architecture System (Size = 16 Classes) Confusion Matrix												
lower	5	2	0	4	3	0	5	2	0	5	2	0
middle	1	4	0	0	5	0	1	4	0	1	4	0
upper	1	0	3	0	1	3	4	0	0	1	0	3
Accuracy = 0.75				Accuracy = 0.75			Accuracy = 0.56			Accuracy =0.75		
Recall (R), Precision (P), F1-Score (F1) Evaluation												
	R	P	F-1	R	P	F-1	R	P	F-1	R	P	F-1
lower	0.71	0.71	0.71	1.00	0.57	0.73	0.50	0.71	0.59	0.71	0.71	0.71
middle	0.67	0.80	0.73	0.56	1.00	0.71	0.67	0.80	0.73	0.67	0.80	0.73
upper	1.00	0.75	0.86	1.00	0.75	0.86	0.00	0.00	0.00	1.00	0.75	0.86
Constore (Size: 66 classes or interfaces) Confusion Matrix												
lower	43	0	0	43	0	0	40	3	0	27	16	0]
middle	14	0	1	13	2	0	12	3	0	9	5	1
upper	6	0	2	6	0	2	7	1	0	4	2	2
Accuracy = 0.68				Accuracy = 0.71			Accuracy = 0.65			Accuracy =0.52		
Recall (R), Precision (P), F1-Score (F1) Evaluation												
	R	P	F-1	R	P	F-1	R	P	F-1	R	P	F-1
lower	0.68	1.00	0.81	0.69	1.00	0.82	0.68	0.93	0.78	0.68	0.63	0.65
middle	0.00	0.00	0.00	1.00	0.13	0.24	0.43	0.20	0.27	0.22	0.33	0.26
top	0.67	0.25	0.36	1.00	0.25	0.40	0.00	0.00	0.00	0.67	0.25	0.36

sentation functions are labelled as *upper*. (iii) Programming elements that provide business logic or depend on programming elements defined within the application are labelled as *middle*.

Total one hundred fifty-one classes, i.e. data instances, are used to train the model—sixteen classes from the specially designed system and 135 classes from the *HealthWatcher* system.

5.2. Model Testing

We used *ConStore*[20], a small scale Java-based library designed to manage concept networks, to test the model. This application is selected to test the model because the second author has involved in recovering architecture previously and knows the details of the system. The ConStore is a framework for detailing out the concepts and creating a domain model for a given application.

5.3. Model Evaluation

The performance of classification models is typically evaluated against measures such as accuracy, precision, recall, and F1-Score [21]. These metrics are derived from a confusion matrix which compares the count of actual class labels for the observations in a given data set and the class labels as predicted by a classification model. Four different metrics are derived by comparing true labels with the predicted labels. These are accuracy, recall, precision and F1-score. Table 5 shows the performance analysis against these metrics. The table compares the performance of algorithmic-centric approach and data-driven approach.

5.3.1. Accuracy Analysis

The accuracy is the rate of correction for classification models. Higher the value of accuracy, better is the model. From the accuracy point of view, one can observe from the Table 5 that the data-driven approach performs better as compared to the algorithmic-centric approach. The decision-based classifier preforms better on all the test cases with an average accuracy of 74%. This is because the performance of algorithmic approach depends on the proper tuning of various configuration parameters. The results shown in Table 5 are obtained with following values in Table 6 of configuration parameters. The Table 6 shows combination of configuration values for the best accuracy obtained during twenty-five iterations. During each iterations, values of configuration parameters were incremented by 1 or 0.1(for δ_c, δ_e).

	ConStore	Healthwatcher	Test Arch.
δ_{il}	4	10	2
δ_{iu}	1	1	1
δ_{ol}	4	2	2
δ_{ou}	1	5	2
δ_b	6	9	6
δ_c	0.8	0.8	0.6
δ_e	0.6	0.5	0.6

Table 6
Configuration parameters and their values

The machine learning models automatically learn and adjust the model parameters for the better results of accuracy. In case of algorithmic approach, the configuration parameter tuning is an iterative process and need to try different combinations.

5.3.2. Recall, Precision, F1-Score Analysis

Recall indicates the proportion of correctly identified true positives while precision is the proportion of correct positive identification. High values of both recall and precision are desired, but it isn't easy to get high values simultaneously for recall and precision. Hence, F1- score combines recall and precision into one metrics. From the recall, precision, and F1-score point of view, one can observe from Table 5 that decision tree-based classifier performs better with the highest F1-score of 0.86 for the upper layer classification of test architecture system. Recalling class labels with higher precision for *middle layer* is a challenging task for all the models described in this paper. This is because of the presence of many not so cleanly encapsulated functionalities in a module at the middle layer and mapping crosscutting concerns to one of the three layers.

5.3.3. Threats to Validation

The performance of the models has been evaluated in an academic setting with internal validation only. By internal validation, we mean the performance of algorithmic and data-driven techniques that have been compared and analyzed. This is because our prime aim is to demonstrate the significance of social network analysis measures in recovering architecture descriptions. The model's performance needs to be further compared against similar approaches developed earlier[22]. Also, the usefulness of recovered layering information needs to be assessed in the software industry setting.

6. Related Work

Recovering architecture descriptions from the code has been one of the widely and continuously explored problem by Software Architecture researchers. This has resulted in a large number of techniques[23, 24, 25], survey papers [22] and books [26] devoted to the topic. In the context of these earlier approaches, this section provides the rationale behind the implementation decisions taken while developing our approach.

(i) Include Dependencies vs Symbolic Dependencies: The recent study reported in [27] has recognized that the quality of recovered architecture depends on the type of dependencies analyzed to recover architecture. The study analyzes the impact of *symbolic dependencies* i.e. dependencies at the program identifier level versus *include dependencies* i.e. at the level of importing files or including packages. Further, it emphasizes that symbolic dependencies are more accurate way to recover structural information. The use of include dependencies is error prone owing to the fact that a programmer may include a package without using it.

We used *include dependencies* in our approach because extracting and managing include dependencies are simple as compared to symbolic dependencies. Further, we mitigated the risk of unused packages by excluding these relationship from further analysis. Many programming environments facilitate the removal of unused packages. One of our objectives was to develop a data-driven approach and cleaning data in this way is an established practice in the field of data engineering.

(ii) Unsupervised Clustering vs Supervised classification: The techniques of unsupervised clustering have been adopted widely to extract high-level architectures through the analysis of dependencies between implementation artefacts [23]. These approaches use hierarchical and search-based methods for clustering. These approaches usually take substantial search time to find not so good architectures [28]. One of the advantages of clustering methods is that unlabelled data sets drive these methods. But, the identified clusters of

program elements need to be labelled with appropriate labels.

Our choice of *supervised classification method* is driven by the fact that *centrality measures* quantify the structural properties with reference to a node, and relation of the nodes with respect to others. Processing such quantified values in efficient way is one of the advantages of many of supervised classification methods. Further, assigning program elements with layering labels is not an issue if such information is available from the previous version of software, which may be the case for re-engineering or modernization projects. In the absence of such labelled data set, the approach presented in the paper can still be adopted in two stages. In the first stage, a tentative layer labelling can be done through algorithmic approach followed by the labelling through supervised classification method.

(ii) Applications of Social Network Analysis (SNA) Measures: The interest in applying theory of SNA has started growing in recent times. Some of the recent applications of SNA include predicting architectural smell [29], predicting vulnerable software components [30] to measure structural similarity of program elements. Our approach characterizes each program elements through its centrality measures which can be termed as *feature representation*, using machine learning vocabulary, necessary to build data-driven models.

7. Conclusion

The main highlights of the approach presented in the paper include: (i)The dependency graph formed by programming elements (i.e. classes in Java) is treated as a network, and centrality measures are applied to extract structural properties. (ii) It represents each program element as a set of values corresponding to different *centrality measures*. Thus, each program element is represented as a feature vector in machine learning terminology. (iii) The paper treats a *layer* as a coarsely granular abstraction encapsulating common system functionalities. Then it maps a group of programming elements sharing common structural properties to a layer. (iv) The paper describes two mapping methods for this purpose called algorithmic centric and data-driven. (v) Overall, the data-driven method illustrated in the paper perform better compared to the algorithmic centric method.

The paper makes several assumptions, such as (i) availability of Java-based system implementation, (ii) a system is decomposed into three layers, and (iii) availability of pre-labelled data set for supervised classification. These are the assumption made to simplify the realization and demonstration of the approach. Hence, these assumptions do not restrict the approach. However, these assumptions can be relaxed, and the approach is flexible enough to extend.

The layering information extracted by the approach can be viewed as one way of decomposing a system. It is not the single ground truth architecture that is often difficult to agree upon and laborious to discover [22]. Further, the quality of extracted architecture descriptions, i.e. clustering of program elements to layers, need to be assessed for the properties such as a minimal layer of violation [31, 32] or satisfaction of a particular quality attribute[26] or any other project-specific criteria.

We described the working of the approach by assuming a three-layer decomposition. The work presented in this paper can be extended to more than three layers. The algorithm-centric technique needs to be adapted by redesigning rules for additional layers. Also, the supervised classification method can be adjusted by relabelling program elements with the number of layers considered.

Exploring the impact of fusing structural properties and some semantic features such as a dominant concern addressed by a programming element would be an exciting exercise for future exploration.

References

[1] D. Link, P. Behnamghader, R. Moazeni, B. Boehm, The value of software architecture recovery for maintenance, in: Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference), 2019, pp. 1–10.

[2] A. Pacheco, G. Marín-Raventós, G. López, Designing a technical debt visualization tool to improve stakeholder communication in the decision-making process: a case study, in: International Conference on Research and Practical Issues of Enterprise Information Systems, Springer, 2018, pp. 15–26.

[3] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, N. Medvidovic, Recovering architectural design decisions, in: 2018 IEEE International Conference on Software Architecture (ICSA), IEEE, 2018, pp. 95–9509.

[4] P. J. Carrington, J. Scott, S. Wasserman, Models and methods in social network analysis, volume 28, Cambridge university press, 2005.

[5] S. B. Thakare, A. W. Kiwelekar, Skiplpa: An efficient label propagation algorithm for community detection in sparse network, in: Proceedings of the 9th Annual ACM India Conference, 2016, pp. 97–106.

[6] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Reviews of modern physics 74 (2002) 47.

[7] D. J. Watts, Networks, dynamics, and the small-world phenomenon 1, American Journal of sociology 105 (1999) 493–527.

[8] J. S. Silva, A. M. Saraiva, A methodology for applying social network analysis metrics to biological interaction networks, in: Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM International Conference on, IEEE, 2015, pp. 1300–1307.

[9] G. Amitai, A. Shemesh, E. Sitbon, M. Shklar, D. Netanely, I. Venger, S. Pietrokovski, Network analysis of protein structures identifies functional residues, Journal of Molecular Biology 344 (2004) 1135–1146. doi:http://dx.doi.org/10.1016/j.jmb.2004.10.055.

[10] M. E. J. Newman, Random graphs as models of networks, arXiv preprint cond-mat/0202208 (2002).

[11] M. E. Newman, The structure and function of complex networks, SIAM review 45 (2003) 167–256.

[12] S. P. Borgatti, Centrality and network flow, Social networks 27 (2005) 55–71.

[13] L. C. Freeman, Centrality in social networks conceptual clarification, Social networks 1 (1979) 215–239.

[14] D. R. White, S. P. Borgatti, Betweenness centrality measures for directed graphs, Social Networks 16 (1994) 335–346.

[15] P. Bonacich, Some unique properties of eigenvector centrality, Social networks 29 (2007) 555–564.

[16] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-oriented software architecture, on patterns and pattern languages, volume 5, John wiley & sons, 2007.

[17] C. A. U. Hassan, M. S. Khan, M. A. Shah, Comparison of machine learning algorithms in data classification, in: 2018 24th International Conference on Automation and Computing (ICAC), IEEE, 2018, pp. 1–6.

[18] J. Hao, T. K. Ho, Machine learning made easy: A review of scikit-learn package in python programming language, Journal of Educational and Behavioral Statistics 44 (2019) 348–361.

[19] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, et al., On the impact of aspectual decompositions on design stability: An empirical study, in: European Conference on Object-Oriented Programming, Springer, 2007, pp. 176–200.

[20] http://www.cse.iitb.ac.in/constore, http://www.cse.iitb.ac.in/constore, 2009.

[21] C. Goutte, E. Gaussier, A probabilistic interpretation of precision, recall and f-score, with implication for evaluation, in: European conference on information retrieval, Springer, 2005, pp. 345–359.

[22] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 486–496.

[23] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Transactions on Software Engineering 33 (2007) 759–780.

[24] A. W. Kiwelekar, R. K. Joshi, An ontological framework for architecture model integration, in: Proceedings of the 4th International Workshop on Twin Peaks of Requirements and Architecture, 2014, pp. 24–27.

[25] A. W. Kiwelekar, R. K. Joshi, Ontological analysis for generating baseline architectural descriptions, in: European Conference on Software Architecture, Springer, 2010, pp. 417–424.

[26] A. Isazadeh, H. Izadkhah, I. Elgedawy, Source code modularization: theory and techniques, Springer, 2017.

[27] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, R. Kroeger, Measuring the impact of code dependencies on software architecture recovery techniques, IEEE Transactions on Software Engineering 44 (2017) 159–181.

[28] S. Mohammadi, H. Izadkhah, A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code, Information and Software Technology 105 (2019) 252–256.

[29] A. Tommasel, Applying social network analysis techniques to architectural smell prediction, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2019, pp. 254–261.

[30] V. H. Nguyen, L. M. S. Tran, Predicting vulnerable software components with dependency graphs, in: Proceedings of the 6th International Workshop on Security Measurements and Metrics, 2010, pp. 1–8.

[31] S. Sarkar, G. Maskeri, S. Ramachandran, Discovery of architectural layers and measurement of layering violations in source code, Journal of Systems and Software 82 (2009) 1891–1905.

[32] S. Sarkar, V. Kaulgud, Architecture reconstruction from code for business applications-a practical approach, in: 1st India Workshop on Reverse Engineering,(IWRE), 2010.