International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-8 Issue-5, June 2019

Source Code Dependency Graph Based Contextual Probabilistic Clustering Approach on Large Open Source Projects

Nakul Sharma, Prasanth Yalla

Abstract*: Software systems tend to become larger and more complex in terms of different metrics such as fields, methods and classes as the functional requirements of the project increases. Most of the open source or commercial software projects are represented in simple diagrammatic representation in order to understand the analysis of the source code metrics. Also, these projects are difficult to analyze the source code metrics by using traditional similarity measures and graph dependency approaches due to high computational memory and time. Analysis of source code metrics and compiled class metrics are necessary to represent various metrics and its complex relationship for software design representation. In order to overcome these issues, a novel graph dependency based probabilistic similarity clustering approach is implemented on the source codes and class files metrics. Experimental results proved that the present approach is better than the traditional source code analysis methods in terms of contextual similarity and accuracy.

***Index Terms*: source code analysis, class files metrics, graph dependency model.**

I. INTRODUCTION

Understanding the structure and metrics of a large software projects can be quite difficult and complex for software developers and designers. Source code and class files clustering are used to understand the contextual similarity of the code metrics for software design representation. Software architecture is considered as the most essential part of the software design. It involves different complicated design decisions. Most of the vital design decisions are structural design decisions related to system composition. These design decisions can be explained and analyzed during the initial architectural stage of design process. The named entity recognition process has the responsibility to detect particular names such assource code methods, fields, variables etc. The process of text classification can be defined as a specific process that can automatically identify importance of a particular source code document. Apart from all of these, emphasis is also given on recognizing synonyms and term abbreviations. In the process of relationship extraction, emphasis is given on the detection of occurrences of a pre-specified relation such as associative, correlative or semantic relations among various entities.Semantics can be either formal or informal. In case of the informal software modeling languages, there are no precisely defined source code metrics for UML representation. On the contrary, formal language semantics are precisely defined with the help of certain mathematical rules and logic to find the relationships among the source code metrics. Since last two decades, there have been extensive amount of research works carried out in order to perform systematic evaluation of contextual relationship among the source codes in case of object oriented systems. Traditional source code dependency graph (SDG) model is considered as a common standard which is implemented during the process of software design for a single source code metrics. It is mostly used in order to represent the workflow and object flow within a system. Nodes of the activity diagram can be of different type, those are:- control nodes, design nodes, join nodes, and so on. Edges of the source code dependency graph are responsible to detect the code metrics and its relationships. It is also responsible to demonstrate the transition of one state to another using edge weights.The edges have the responsibility to provide the sequence number of each and every operation. The SDG graph has the responsibility to traverse via UML diagram. As we all know, UML is not a completely formal language. Hence, it's semantics are not completely formalized. In most of the cases, natural language is implemented for model specification. It may give rise to a specific condition in which model presentation is quite complicated. Hence, while using any UML diagram during the process of code production, it is necessary to combine it with certain specification languages such as object constraint language using SDG model. Latent topic model is responsibility to help during the understanding of software systems. It analyses the words included in the source code and detects the relationship among them. It is necessary to go for latent factors in order to obtain complete structure of the software application. All of the latent factors are analyzed to find the correlation among attributes;these correlated features are used to find thecomplete structure of the source code analysis. In the natural language processing, the observations can be linked to the word frequency directly.

Manuscript published on 30 June 2019.
***** Correspondence Author (s)
Nakul Sharma, Research Scholar, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation, Guntur(Dist), India
Prasanth Yalla, Professor, Department of Computer Science and Engineering, Koneru Laksmaiah Education Foundation, Guntur(Dist), India.

In this case, word frequency is evaluated as word counts are considered as input for the statistical structure which is known as topic model. In this case, topic has the responsibility to explain the relationship among different portions of data. Graphical modeling languages havevarious applications during the system design and documentation. Unified Modeling
Language is considered as the standard and most commonly used graphical modeling language. At first, UML diagrams were incapable to explain the dynamic aspects such as state transitions of the state machines. Similarly, traditional natural language and existing programming languages are not efficient to explain these dynamic aspects of source code entities. There are certain limitations of implementing these traditional approaches, some of those limitations are mentioned below:-

1. Natural languages are ambiguous in source code prediction due to large number of method, fields and classes.
2. Most of the programming languages are implementation for limited source code analysis. In order to construct realistic characteristics, action semantics are incorporated into UML 2.0. A model can be described as an abstraction of a system. It is very much complicated to obtain modeling consistency due to abstraction gap among modeling and programming languages.
3. Traditional models are used to find the abstraction of the system by removing unwanted information. Code actions, methods, fields and paths can't be explained through UML diagrams. All the constraints for the model construction should be explained with the help of certain programming language.
4. Model aware action languages have the responsibility to decrease the abstraction gap.
5. The Action Language for Foundational UML can be defined through the Object Management Group in order to decrease the source code entities. These models are explained with the help of different graphical representations. In order to explain each and every element of the model and its executable nature, a new UML action language is necessary.

Another problem in the traditional source code analysis for UML representation is code cloning or duplicate entities in the classes. Most of the developers are required to carry out the process of software maintenance at the final stage of code clone detection. A specific part of the source code is copied and changed from the implemented code blocks. The process of code clone identification is considered as the most common use case for source code analysis for UML representation. There are different applications for large scale code entity identification and some of those are mentioned below:- querying library entities , plagiarism identification, detection of product lines in case of reverse engineering process. Object Constraint Language (OCL) is capable enough to overcome limitations of UML notations. Mostly, all of these constraints are written in natural languages which are ambiguous in nature. A formal language is needed to be developed in order to represent the constraints. As OCL is a formal language and all of its constructs are defined, it can be specified ambiguously. Presently, the process of factor analysis is carried out with the help of topic models and it can be implemented in software systems. It uses topic models in order to analyze units of source code such as methods, fields and classes. It is very much vital to know the exact numbers of latent topics needed to be uncovered.The work described in this paper addressed various issues regarding the source code and class files metrics for design diagram representation. In this work, a contextual dependency graph measure between the source code files, class files and its related API comments are analyzed for structural analysis. The main idea behind this approach is that in traditional source code analysis or graph dependency models, all possible candidate sets of the source code metrics and class files are assumed to have equal weightage for similarity computation. In the proposed model, a novel contextual similarity based graph dependency model is used to filter the candidate sets by using probabilistic measure. The rest of this paper is organized as follows. In part II a brief summarization of related works is presented. The proposed model is discussed in part III. In the part IV, experimental analysis of the proposed work and existing works are discussed followed by conclusion in part V.

II. RELATED WORKS

A. *M.* Fernández-Sáez, et.al, discussed the empirical studies related to the maintenance of UML diagrams and its use during the maintenance phase [1]. Since last decade, unified modelling language is used as a common standard during the modelling of object oriented software systems. Hence, it is very much necessary to analyse its advantages and the associated expenses for its implantation. In this piece of research work, they have introduced new and systematic mapping strategy in order to gather the published empirical studies. The prime objective of this research work is to investigate all the previously existing empirical studies related to UML diagrams. As we all know, by using UML diagram, the maintenance process of source code can be enhanced. The quality of the changes is considered to be significant because of the UML diagrams. Most of the previously existing research works emphasised on the maintainability and comprehensibility of UML diagrams. Therefore, more detailed research works are needed to be carried out in industrial contexts. Real systems must be considered for the experiment and the maintenance process must be carried out under real circumstances in order to identifythe actual utility of the UML diagrams. In future, the cost and productivity can be improved significantly.A. M. Fernández-Sáez et.al, discussed the effectively of forward designed and reverse engineered UML diagrams [2]. There exist numbers of different efficient and effective model based techniques. But till date, this model based development strategies are not completely accepted by all of the software organisations. The major limitation of these approaches is the high cost. Therefore, there is a need of some cost effective feasible solutions which could be accepted by all of the software organisations for collecting empirical evidence. The prime objective of this research work is to present perfect empirical evidence so that, the UML diagrams will be more during the software maintenance phase.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

Here, they have concentrated on two types of UML diagrams, those are:- forward designed UML diagrams and reverse engineered UML diagrams.Md. I. Azeem et.al, introduced a new machine learning approach in order to identify code smell [3]. They have performed a through literature survey and meta-analysis. Code smell is the result of suboptimal design or implementation of choices.

Both of the above mentioned factors mostly lead to fault prone condition. Researchers have already developed numbers of different code smell identifiers. It has the major objective to exploit various information sources in order to assist the developers during the diagnosing process of design flaws. Most of the traditional approaches not doubt are giving accurate outcomes, but there are three major limitations of those works:-

1. Subjectiveness of developers with respect to code smells identified by tools.
2. Scarce agreement among various identifiers.
3. Complexities in detecting good thresholds.

In order to resolve the above mentioned issue, machine learning algorithms are the best option. Most of the research works only include the implementation of heuristic based code smell identifiers. But, there is no significant amount of research works have been performed the implementation procedure of machine learning techniques for code smell identification. The aim of the above research paper is to do a thorough survey on various machine learning techniques in the area of code smells. A. S. Nuñez-Varel, et.al, performed a thorough mapping survey on different source code metrics [4]. Source code metrics are considered to be most vital components during the software measurement process. These are collected from the source code and its values provide us the quality attributes evaluated through these metrics. This research paper has an objective to gather source code metrics from the literature, review those and carry out the process of analysis. They also analysed the current state of source code metrics and their current trends. They have performed an efficient mapping study. They considered 226 studies those are published in between 2010-2015 are chosen and analyses. Though object oriented metrics are more important and popular, but presently, the requirement of feature oriented metrics and aspect oriented metrics are also increasing.F. Zhou and X. Luo developed an automatic detection system for source code comments [5]. Comments provide very vital information related to the system functionalities. Most of the software engineering approaches consider comments very seriously because it is the most important source of different processes just like code semantic analysis, code reuse, etc. As we all know, structural comments are the simplest form of comments. But, it is very much difficult to detect the relationship among functional semantics of the code and the related textual descriptions. In this piece of research work, they have presented a standard approach in order to identify the source code comments. According to the concepts of machine learning, this technique included features of code snippets and comments in order to identify the scopes of source code comments automatically. This above presented system is implemented on two software engineering tasks in order to analyse software repositories to identify outdated comments. Apart from this, mining of software repositories for comment production is also considered as an important process. We can say that, this approach can be considered as an effective and efficient solution during the process of comment-code mapping. The above presented technique can enhance the performance of different baseline approaches. In future, further research works are needed to be carried out in order to extend this approach. A. Adamuaet.al, proposed a multiview similarity assessment approach for UML diagrams [6]. This research paper has an objective to match various UML diagrams out of multiple views. It provides an efficient method in order to evaluate the similarity among different UML diagrams. They have considered structural, functional and behavioural perceptive. The Multiview similarity is defined as an aggregation of views. It is scaled with the help of a factor known as inconsistency penalty. The above mentioned factor is responsible for managing the conflicting mapping in between views. Multiview similarity results better performance as compared to the single view. The prime objective of the above proposed technique is to assist software developers in order to extract pre-existing software system those are stored in repositories. Developers can reuse the previously existing software in order to develop new applications. Presently, the above proposed approach depends upon class, sequence and state machine diagrams. Further researches can be performed to enhance the overall matching accuracy.F. A. Fontana and M. Zanoni presented a new code smell severity classification technique with the help of machine learning approaches [7]. There have been numbers of different smells identification tools developed since years. As we all know, smells can actually be interpreted. Therefore, smells can be identified by numbers of different methods. There is wide range of applications of machine learning approaches in the domain of software engineering just like design pattern identification, code smell identification, bug prediction, and so on. In this piece of research work, they have emphasized on classification process of code smell severity with the help of machine learning approaches. The severity of code smells is considered as a vital factor. It permits the prioritization of refactoring efforts. The code smell along with very high severity is actually very large and complex. Additionally, it is the main cause of larger problems during the maintenance phase of a software system. In this research paper, different machine learning algorithms are implemented. The code smell severity is considered as an ordinal variable. The baseline models are gathered from the previously developed research papers. In those above mentioned research papers, they have implemented binary classification techniques in order to carry out the process of code smell identification. In the subsequent time, additional research works can be performed in order to enhance and expand the above proposed model. N. R. Carvalho, et.al, conducted their research works from source code identifiers to natural language terms [8]. Program comprehension approaches use program identifiers in order to infer knowledge related to different programs.

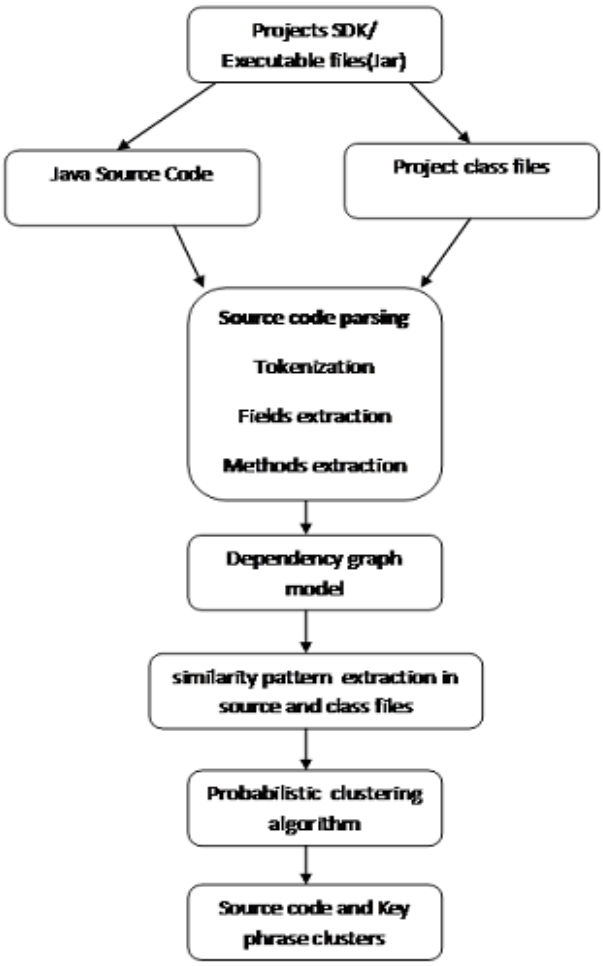
Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

All of the previously existing programming languages include certain rules for identifier strings. They have used combination of several words and abbreviations in order to devise strings which is required to represent a particular or multiple concepts. It has the responsibility to enhance the programming linguistic efficiency significantly. All of these strings never use explicit marks in order to identify the terms. Therefore, hard splitting approaches are not sufficient enough. They also performed comparative study of their presented approach along with other existing approaches.

L. Cerulo, et.al, developed a new hidden Markov approach in order to identify coded information in free text [9]. As we all know, extraction of the content of developers' communication is beneficial in order to support different software engineering processes just like program comprehension, source code analysis, and so on. Automation of the extraction process is very much complicated because of the complex and unstructured nature of the free text. It integrates various coding languages along with natural language parts. Most of the traditional graph based clustering models[10-14] are not appropriate to large size source codes due to high computational time and memory. Also, these models are not appropriate to form the contextual relationships among the source code and class files by using API comments.

III. PROPOSED MODEL

In the static and dynamic source code analysis, it is difficult to find and extract the essential key patterns due to large number of similar patterns. In this proposed model, source code and compiled class files are used to analyze the contextual key patterns and cluster patterns on the large candidate similar code patterns. Proposed framework is summarized in figure 1. In the figure, initially source code documents with API content and compiled class files are used to pre-process the code structure using tokenization, field extraction and methods extraction. Here, code parsers are used to find and extract the essential patterns in source code documents and class documents. Graph dependency model is constructed to each document in the source code and class files for similarity pattern extraction. Proposed source code similarity measure and class similarity measure are used to find the key patterns among the large number of code structural patterns. These graph based similarity patterns are used to cluster the source code documents using the graph based clustering algorithm. As the size of the software source code and compiled data increasing, it is practically difficult to analyze the large volumes of individual source code documents and its essential key structures due to high computational time and memory. Proposed model is summarized in the figure 1. In the figure, initially different types of software projects and its API documentations are taken as input to the proposed model. Project source codes are parsed using the class parsing libraries for code dependency graph. Here, the code dependency graph is constructed to extract the methods and fields in the each project class file. Similarly, project API documents are pre-processed to remove the un-used text information for contextual graph similarity



.Fig. 1. Proposed Framework

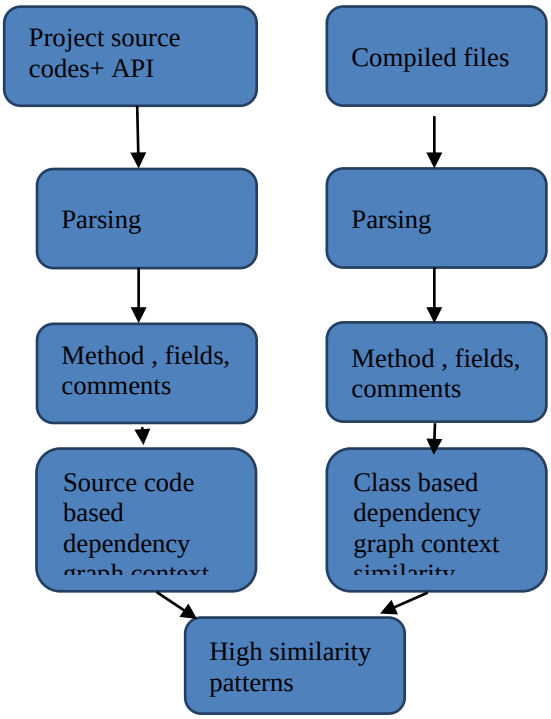


Fig. 2. Source code and class based contextual dependency graph construction.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

Contextual dependency graph is designed and implemented in two phases. In the first phase, source code and class metrics are extracted from the source code and class files for dependency graph modeling. In the second phase, different contextual relationships are extracted in the source code document metrics and the class metrics using the proposed contextual source code and class similarity measures.

Phase 1: Source Code and Class files data filtering

Input : *Source code files SC, Class files CF.*

Step 1: *Read input source codes files SC.*

Step 2: *Read input classfiles CF.*

Step 3:*for each source code SCi in SC[]*

Do

Parse source code SCi with methods M and Fields F.

Mi=ExtractMethods(SCi)

Fi=ExtractFields(SCi)

Mapping (Mi , Fi) to CSi

SC_1	(M_1,F_1)
SC_2	(M_2,F_2)
…	…..
SC_n	(M_n,F_n)

done

Step 4: *for each class file CFi in CF[]*

Do

Parse class files CFi with methods M and Fields F.

Mi=ExtractMethods(CFi)

Fi=ExtractFields(CFi)

Mapping (Mi , Fi) to CFi

CF_1	(M_1,F_1)
CF_2	(M_2,F_2)
…	…..
CF_n	(M_n,F_n)

done

Step 5: *// Remove the duplicate methods and fields in each source code and class files*

For each code C_i *in* $SC_i \cup CF_j$

Do

$$M_i \in \Pr ob(M_i \cap M_j / C); i = j$$

$$F_i \in \Pr ob(F_i \cap F_j / C); i = j$$

If(M_i*!=0 AND* F_i*!=0)*

Then

Remove M_i *in* C_i *or* C_j

Remove F_i *in* C_i *or* C_j

End if

Done

Step 6: *//Pre-processing source code comments using Stanford NLP parser.*

For each document di in D

Do

T[]=Tokenize(di)

For each token t in T[]

Do

Apply stemming, stop word removal using Stanford NLP library.

Done

Done

The source code and class files are used as input to the above algorithm in order to parse the tokens using Stanford NLP and Class parser libraries. Here, each source code and class file is preprocessed using the NLP parsing methods such as tokenization, stemming and stop word removal as source code entities.

Phase 2: Contextual similarity measures for Source code and class files dependency graph

Input: Project source codes SC, Project class files CF, Project source metrics (SMi,SFi)and Project class metrics (CMi,CFi).

Procedure:

Step 1: *Read source code metrics ,sci→(SMi,SFi) and Project class metrics cfi→(CMi,CFi)*

Step 2: *Constructing a source code dependency graph SDG→(V,E) with vertex set V and Edge set E using source code metrics. Here vertex set V is represented with source code methods and fields and edge set E is represented as weighted rank between the vertices.*

Step 3: *The weights of the edges are computed using the vertex terms* t_i *and* t_j *where* $t_i \in V_i$ *and* $t_j \in V_j$.

$$\text{Edgeweight}: w(i,j) = \sum_{\forall i,j} \frac{F(t_i,t_j)}{F(t_i)+F(t_j)-F(t_i,t_j)}$$

$F(t_i,t_j)$ *is the number of times both terms* (t_i,t_j) *occurred together.*

$F(t_i)$ *is the number of occurrence of* t_i *in vertex* V_i

$F(t_j)$ *is the number of occurrence of* t_j *in vertex* V_j

Step 4: *The vertices with positive edge weights are sorted in ascending order in the dependency graph to find the contextual similarity between the source code metrics.*

Step 5: *Source code dependency graph SDG is used to find the contextual similarity between the vertex nodes to the neighbor metrics using the following proposed measure.*

Let U(SMi) ← *(m1,m2,⋯.mn) denotes the source codes metrics vector at vertex i.*

V(SMj) ← *(m1,m2,⋯.mr) denotes the source code metrics vector at vertex j.*

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

$$|U(SM_i)| = \sqrt{U(m_1)^2 + U(m_2)^2 .... U(m_p)^2}$$

$$|V(SM_j)| = \sqrt{V(m_1)^2 + V(m_2)^2 .... V(m_q)^2}$$

$$|U(SM_i).V(SM_j)| = U(m_1).V(m_1) + U(m_2).V(m_2) ... + U(m_p).V(m_q)$$

Proposed Contextual source code depenedency graph dissimilarity index is computed as

$$CSDGDI = \frac{\sqrt[3]{U(SM_i).V(SM_j) * \tan^{-1}\theta(|U(SM_i)| + |V(SM_j)|)}}{2*(|U(SM_i)| * |V(SM_j)|)}; \text{where } i \neq j$$

Contextual source code depenedency graph similarity index

$$CSDGSI = 1 - CSDGDI;$$

Step 6: *Class file dependency graph CDG is used to find the contextual similarity between the vertex nodes to the neighbor metrics using the following proposed measure.*

Let $U(CM_i) \leftarrow (m_1, m_2, ...m_n)$ *denotes the source codes metrics vector at vertex i.*

$V(CM_j) \leftarrow (m_1, m_2, ...m_r)$ *denotes the source code metrics vector at vertex j.*

$$|U(CM_i)| = \sqrt{U(m_1)^2 + U(m_2)^2 .... U(m_p)^2}$$

$$|V(CM_j)| = \sqrt{V(m_1)^2 + V(m_2)^2 .... V(m_q)^2}$$

$$|U(CM_i).V(CM_j)| = U(m_1).V(m_1) + U(m_2).V(m_2) ... + U(m_p).V(m_q)$$

Proposed Contextual class code depenedency graph dissimilarity index is computed as

$$CCDGDI = \frac{\sqrt[3]{U(CM_i).V(CM_j) * \log(|U(CM_i)| + |V(CM_j)|)}}{2*(|U(CM_i)| * |V(CM_j)|)}; \text{where } i \neq j$$

Contextual class code depenedency graph similarity index

$$CCDGSI = 1 - CCDGDI;$$

Contextual graph based Clustering algorithm

Step 1: Read number of clusters c.

Step 2: Read number of iterations I.

Step 3: Initialize k random clusters as centroids.

Step 4: for each document at vertex V in graph

Do

TF-ID[]= Compute term frequency tf-id

Done

Step 5: Repeat until c clusters

Find nearest cluster distance metrics using the following equation

Let Documentvector one V1, document vector two V2

$$Dist(V1, V2) = \frac{\sqrt{V1[i].V2[i]}}{2 * \sqrt[3]{\sum V1[i]^2 + \sum V2[i]^2}}$$

Done

Step 6: Merge the graph nodes using the nearest distance measure.

Step 7: Update cluster centroid using mean distance.

IV. EXPERIMENTAL RESULTS

Experimental results are performed on different object oriented software projects with API comments. A total of five java open source projects with API comments are taken as input to validate the performance of the proposed model to the existing models. The five open source projects are summarized in table 1.The sizes of the open source projects vary from 100 to 1500 classes and 5134 to 142553 lines of code. For the experimental evaluation, various performance metrics such as accuracy, similarity index and computational time are used to compare the proposed model to the existing models.

$$Accuracy := \frac{N_C \cap N_{IC}}{N}$$

In the above formula NC is source code that predict correctly and NIC is source code reports that predict incorrectly. Accuracy defines the number of the files that predict correctly over the number of files that is matched in the API comments. Context similarity defines the number of source codes that are predicted correctly over the project API documents with high contextual similarity.

Table. 1. Summary of selected open source projects(source code files and class files) with API comments

Software project	Number of lines	Number of Classes
Weka	32544	425
Apache Commons Collections	26371	441
Jeuclid	12666	230
JfreeChart	95763	1013
Kryo	24144	347

Table 1, describes the summarization of open source projects and its number of lines and number of classes. From the table, it is clear that the proposed model used different types of complex source code structures and its classes.

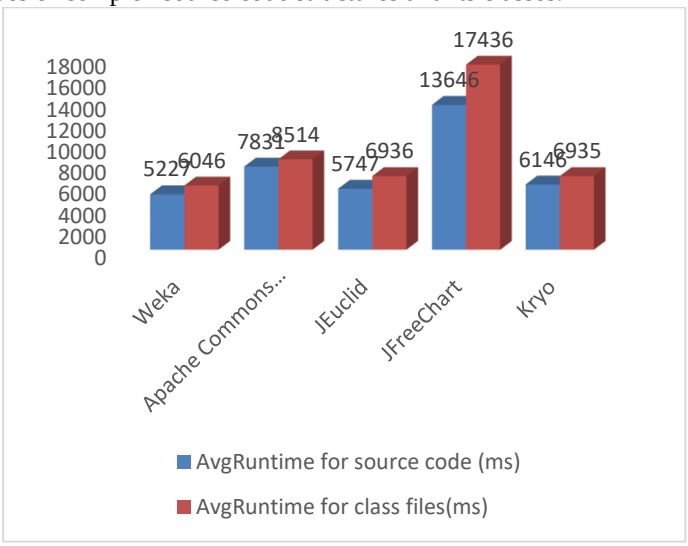


Fig. 3. Runtime Comparison of data pre-processing on different open source projects

Figure 3, describes the comparison of open source projects and its average runtime computation for data processing. In this figure, runtime computation of source code files and class files are visualized in data pre-processing phase.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

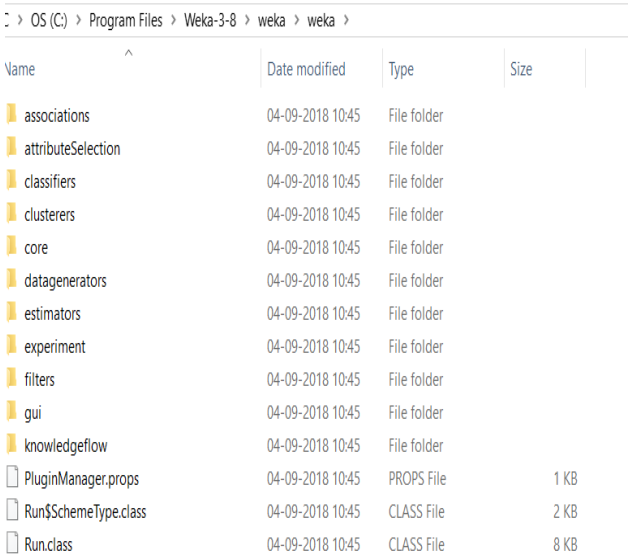


Fig. 4. Sample directories of Weka class files

Figure 4, represents the sample input class files of the open source complex project Weka. As shown in the figure, different packages have different class structures and different methods and fields for structural analysis.

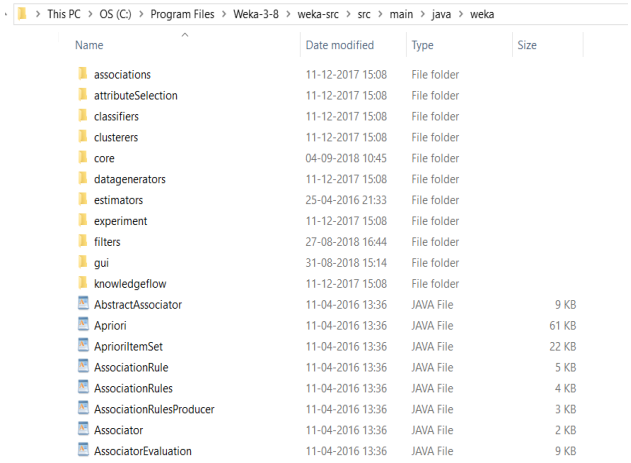


Fig. 5. Sample directories of Weka class files

Figure 5, represents the sample input source code files of the open source complex project weka. As shown in the figure, different packages have different class structures and different methods and fields for structural analysis.

Sample result of the source code and class files dependency graph

Key Phrases in SDG :{m_items.iterator()} {m_items.add(i)} ➔ Score :0.9073701027137411

Key Phrases in SDG :{m_items.iterator()} {Collections.sort(m_items)} ➔ Score :0.9073701027137411

Key Phrases in SDG :{m_items.iterator()} {m_items.get(index)} ➔ Score :0.9073701027137411

Key Phrases in SDG :{m_items.iterator()} {m_items.size()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{m_items.iterator()} {m_items.iterator()} ➔ Score :0.826985987428094

Key Phrases in SDG :{m_items.iterator()} {i.hasNext()} ➔ Score :1.0

Key Phrases in SDG :{m_items.iterator()} {i.next()} ➔ Score :1.0

Key Phrases in SDG :{m_items.iterator()} {i.next().toString()} ➔ Score :1.0

Key Phrases in SDG :{m_items.iterator()} {buff.append(i.next().toString() + "")} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {m_items.add(i)} ➔ Score :0.9073701027137411

Key Phrases in SDG :{i.hasNext()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {m_items.get(index)} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {m_items.size()} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {m_items.iterator()} ➔ Score :1.0

Key Phrases in SDG :{i.hasNext()} {i.hasNext()} ➔ Score :0.826985987428094

Key Phrases in SDG :{i.hasNext()} {i.next()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{i.hasNext()} {i.next().toString()} ➔ Score :0.9073701027137411

Key Phrases in SDG :{i.hasNext()} {buff.append(i.next().toString() + "")} ➔ Score :0.9436832985868555

Key Phrases in SDG :{i.next()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.next()} {m_items.add(i)} ➔ Score :0.9073701027137411

Key Phrases in SDG :{i.next()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.next()} {m_items.get(index)} ➔ Score :1.0

Key Phrases in SDG :{i.next()} {m_items.size()} ➔ Score :1.0

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

Key Phrases in SDG :{i.next()} {m_items.iterator()} ➔ Score :1.0

Key Phrases in SDG :{i.next()} {i.hasNext()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{i.next()} {i.next()} ➔ Score :0.826985987428094

Key Phrases in SDG :{i.next()} {i.next().toString()} ➔ Score :0.8832936425594423

Key Phrases in SDG :{i.next()} {buff.append(i.next().toString() + "")} ➔ Score :0.929045402428935

Key Phrases in SDG :{i.next().toString()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.next().toString()} {m_items.add(i)} ➔ Score :0.9377671555157053

Key Phrases in SDG :{i.next().toString()} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{i.next().toString()} {m_items.get(index)} ➔ Score :1.0

Key Phrases in SDG :{i.next().toString()} {m_items.size()} ➔ Score :1.0

Key Phrases in SDG :{i.next().toString()} {m_items.iterator()} ➔ Score :1.0

Key Phrases in SDG :{i.next().toString()} {i.hasNext()} ➔ Score :0.9073701027137411

Key Phrases in SDG :{i.next().toString()} {i.next()} ➔ Score :0.8832936425594423

Key Phrases in SDG :{i.next().toString()} {i.next().toString()} ➔ Score :0.9102447067835182

Key Phrases in SDG :{i.next().toString()} {buff.append(i.next().toString() + "")} ➔ Score :0.9456308214638183

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {m_items.add(i)} ➔ Score :0.9623025170847559

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {Collections.sort(m_items)} ➔ Score :1.0

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {m_items.get(index)} ➔ Score :1.0

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {m_items.size()} ➔ Score :1.0

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {m_items.iterator()} ➔ Score :1.0

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {i.hasNext()} ➔ Score :0.9436832985868555

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {i.next()} ➔ Score :0.929045402428935

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {i.next().toString()} ➔ Score :0.9456308214638183

Key Phrases in SDG :{buff.append(i.next().toString() + "")} {buff.append(i.next().toString() + "")} ➔ Score :0.9611041333812625

Graph Node ++++++++++++++++++++C:\Program Files\Weka-3-8\weka-src\associations\Item.java

Key Phrases in SDG :{toString(false)} {toString(false)} ➔ Score :0.826985987428094

Key Phrases in SDG :{toString(false)} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {comp.getFrequency()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {comp.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {comp.getAttribute().name()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {m_attribute.name().compareTo(comp.getAttribute().name())} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {comp.getFrequency()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {b.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{toString(false)} {m_attribute.equals(b.getAttribute())} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {toString(false)} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name()} ➔ Score :0.826985987428094

Key Phrases in SDG :{m_attribute.name()} {comp.getFrequency()} ➔

Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name()} ➔ Score :0.826985987428094

Key Phrases in SDG :{m_attribute.name()} {comp.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {comp.getAttribute().name()} ➔ Score :0.9073701027137411

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name().compareTo(comp.getAttribute().name()} ➔ Score :0.9487405555445889

Key Phrases in SDG :{m_attribute.name()} {comp.getFrequency()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {b.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.equals(b.getAttribute())} ➔ Score :0.9299880499551685

Key Phrases in SDG :{comp.getFrequency()} {toString(false)} ➔ Score :1.0

Key Phrases in SDG :{comp.getFrequency()} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{comp.getFrequency()} {comp.getFrequency()} ➔ Score :0.826985987428094

Key Phrases in SDG :{comp.getFrequency()} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{comp.getFrequency()} {comp.getAttribute()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{comp.getFrequency()} {comp.getAttribute().name()} ➔ Score :0.9073701027137411

Key Phrases in SDG :{comp.getFrequency()} {m_attribute.name().compareTo(comp.getAttribute().name()} ➔ Score :0.9644586862698906

Key Phrases in SDG :{comp.getFrequency()} {comp.getFrequency()} ➔ Score :0.826985987428094

Key Phrases in SDG :{comp.getFrequency()} {b.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{comp.getFrequency()} {m_attribute.equals(b.getAttribute())} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {toString(false)} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name()} ➔ Score :0.826985987428094

Key Phrases in SDG :{m_attribute.name()} {comp.getFrequency()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name()} ➔ Score :0.826985987428094

Key Phrases in SDG :{m_attribute.name()} {comp.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {comp.getAttribute().name()} ➔ Score :0.9073701027137411

Key Phrases in SDG :{m_attribute.name()} {m_attribute.name().compareTo(comp.getAttribute().name()} ➔ Score :0.9487405555445889

Key Phrases in SDG :{m_attribute.name()} {comp.getFrequency()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {b.getAttribute()} ➔ Score :1.0

Key Phrases in SDG :{m_attribute.name()} {m_attribute.equals(b.getAttribute())} ➔ Score :0.9299880499551685

Key Phrases in SDG :{comp.getAttribute()} {toString(false)} ➔ Score :1.0

Key Phrases in SDG :{comp.getAttribute()} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{comp.getAttribute()} {comp.getFrequency()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{comp.getAttribute()} {m_attribute.name()} ➔ Score :1.0

Key Phrases in SDG :{comp.getAttribute()} {comp.getAttribute()} ➔ Score :0.826985987428094

Key Phrases in SDG :{comp.getAttribute()} {comp.getAttribute().name()} ➔ Score :0.8832936425594423

Key Phrases in SDG :{comp.getAttribute()} {m_attribute.name().compareTo(comp.getAttribute().name()} ➔ Score :0.9552207506905175

Key Phrases in SDG :{comp.getAttribute()} {comp.getFrequency()} ➔ Score :0.8626786872190586

Key Phrases in SDG :{comp.getAttribute()} {b.getAttribute()} ➔ Score :0.8626786872190586

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

Key Phrases in SDG :{comp.getAttribute()} {m_attribute.equals(b.getAttribute())} ➔ Score :0.9299880499551685
Key Phrases in SDG :{comp.getAttribute().name()} {toString(false)} ➔ Score :1.0
Key Phrases in SDG :{comp.getAttribute().name()} {m_attribute.name()} ➔ Score :0.9073701027137411
Key Phrases in SDG :{comp.getAttribute().name()} {comp.getFrequency()} ➔ Score :0.9073701027137411
Cluster score :1.8915088904591437
Cluster score :1.6548087815435597
Cluster score :10.712497313465224
Cluster score :1.6230128533259884
Cluster score :0.9582862527271474
Cluster score :8.072977858791052
Cluster score :1.006250720645443
Cluster score :2.3205316918872296
Cluster score :1.1039443359178631
Cluster score :0.6642823486428735
Cluster score :7.1738868557022615
Cluster score :1.1523870100959008
Cluster score :2.030141191318216
Cluster score :1.2334755093904384
Cluster score :2.142004956812363
Cluster score :1.0608241208866174
Cluster score :7.718336411341251
Cluster score :4.022114849613585
Cluster score :0.9539618062874109
[D@68183f19 = [6, 9]
Cluser-1{
Associator.java
CARuleMiner.java
}
[D@3fe5c9f1 = [8, 15, 18, 19]
Cluser-2{
BinaryItem.java
Item.java
NominalItem.java
NumericItem.java
}
[D@775429c8 = [0, 7, 20]
Cluser-3{
AbstractAssociator.java
AssociatorEvaluation.java
SingleAssociatorEnhancer.java
}
[D@f7b6b9e = [1, 2, 4, 5, 10, 12, 13, 14, 16, 17]
Cluser-4{
Apriori.java
AprioriItemSet.java
AssociationRules.java
AssociationRulesProducer.java
CheckAssociator.java
FilteredAssociationRules.java
FilteredAssociator.java
FPGrowth.java
ItemSet.java
LabeledItemSet.java
}
[D@7923cb8d = [3, 11]
Cluser-5{
AssociationRule.java
DefaultAssociationRule.java
}

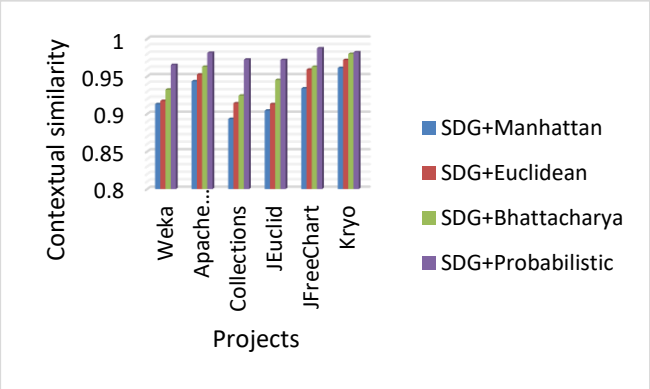


Fig. 6. Contextual similarity of the proposed model to the existing models between the source codes and class files metrics.

V. CONCLUSION

Source code and class files dependency graph analysis is essential for software design diagram representation and project metrics analysis. Most of the traditional projects are difficult to analyze the source code metrics by using traditional similarity measures and graph dependency approaches due to high computational memory and time.

Analysis of source code metrics and compiled class metrics are necessary to represent various metrics and its complex relationship for software design representation. In order to overcome these issues, a novel graph dependency based probabilistic similarity clustering approach is implemented on the source codes and class files metrics. Experimental results proved that the present approach is better than the traditional source code analysis methods in terms of contextual similarity and accuracy.

REFERENCES

1. A. M. Fernández-Sáez, M. Genero and M. R.V. Chaudron, Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study, Information and Software Technology 55 (2013) 1119–1142
2. A. M. Fernández-Sáez, M. Genero, M. R.V. Chaudron, D. Caivano and I. Ramos, Are Forward Designed or Reverse-Engineered UML Diagrams More Helpfulfor Code Maintenance?: A Family of Experiments, Information and Software Technology
3. Md. I. Azeem, F. Palomba, L. Shi and Q. Wang , Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-Analysis, Information and Software Technology
4. A. S. Nuñez-Varel, H. G. Pérez-Gonzalez, F. Martínez-Perez , C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, The Journal of Systems and Software 128 (2017) 164–197.
5. H. Chen, Z. Liu, X. Chen, F. Zhou and X. Luo, "Automatically Detecting the Scopes of Source Code Comments," 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, 2018, pp. 164-173.doi: 10.1109/COMPSAC.2018.00029.
6. A. Adamua and W. M. Zainon, Multiview Similarity Assessment Technique of UML Diagrams, 4th Information Systems International Conference 2017, ISICO 2017, 6-8 November 2017, Bali,Indonesia.
7. F. A. Fontana and M. Zanoni , Code Smell Severity Classification using Machine LearningTechniques, Knowledge-Based Systems, Volume 128, 15 July 2017, Pages 43-58, doi: https://doi.org/10.1016/j.knosys.2017.04.014.
8. N. R. Carvalho, J. J. Almeida, P. R. Henriques and M. J. Varanda, From source code identifiers to natural language terms, The Journal of Systems and Software Volume 100, February 2015, Pages 117-128 doi: https://doi.org/10.1016/j.jss.2014.10.013.
9. L. Cerulo, M. D. Penta, A. Bacchellid, M. Ceccarelli and G. Canfora, Irish: A Hidden Markov Model to detect coded information islands in free text, Volume 105, 1 July 2015, Pages 26-43, https://doi.org/10.1016/j.scico.2014.11.017.
10. Acharya, M., Robinson, B., 2011. Practical change impact analysis based on staticprogram slicing for industrial software systems. In: Proceedings of the 33rdInternational Conference on Software Engineering, ACM Press, pp. 746–755.
11. Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R., 2010. A systematic review ofthe application and empirical investigation of search-based test case generation.IEEE Transactions on Software Engineering 36 (6), 742–762.
12. Hamilton, J., Danicic, S., 2012. Dependence communities in source code. In: 28thIEEE International Conference on Software Maintenance, pp. 579–582.
13. Harman, M., Binkley, D., Gallagher, K., Gold, N., Krinke, J., 2009. Dependence clustersin source code. ACM Transactions on Programming Languages and Systems 32(1), 1:1–1:33.
14. Islam, S.,Krinke, J., Binkley, D., Harman, M., 2010b. Coherent dependence clusters. In:PASTE '10: Proceedings ofthe 9thACM SIGPLAN-SIGSOFTWorkshop on ProgramAnalysis for Software Tools and Engineering. ACM Press, pp. 53–60

AUTHORS PROFILE

Prasanth Yalla received his B.Tech Degree from Acharya Nagarjuna University, Guntur (Dist), India in 2001, M.Tech degree in Computer Science and Engineering from Acharya Nagarjuna University in 2004, and received his Ph.D. degree in CSE titled "A Generic Framework to identify and execute functional test cases for services based on Web Service Description Language" from Acharya Nagarjuna University, Guntur (Dist), India in April 2013. He was an associate professor, with Department of Information Science and Technology in KL University, from 2004 to 2010. Later he worked as Associate professor, with the department of Freshman Engineering from 2011 in KL University. Presently he is working as Professor in the department of Computer Science & Engineering in KL University and also Associate Dean (R&D) looking after the faculty publications. Till now he has published 28 papers in various international journals and 4 papers in conferences. His research interests include Software Engineering, Web services and SOA. He taught several subjects like Multimedia technologies, Distributed Systems, Advanced Software Engineering, Object Oriented Analysis and design, C programming, Object-Oriented programming with C++, Operating Systems , Database management systems, UML etc. He is the Life member of CSI and received "Active Participation- Young Member" Award on 13-12-13 from CSI. He has applied a project to SERB very recently.

.