A Hybrid Weighted Probabilistic Based Source Code Graph Clustering Algorithm For Class Diagram And Sequence Diagram Visualization

Nakul Sharma, Prasanth Yalla

Abstract: The Unified Modeling Language (UML) model is an essential component in the process of software system development and complex compiled class files. Natural language processing and clustering measures are used to find the relational keywords and methods in the large source code documents. Most of the traditional source code visualization tools are based on limited source code files and fixed similarity threshold. Also, these source code visualization models are independent of source code dependency graph and relational methods for class and sequence diagram representation. In order to overcome these issues, a weighted probabilistic contextual source code pattern extraction model is designed and implemented on the large open source software projects. In this work, a complex class diagram is constructed on the large source code documents using the weighted probabilistic graph clustering algorithm. Similarly, a novel sequence diagram is constructed using the compiled class files metrics. Experimental results are performed on the different open source java software systems with the traditional models in terms of runtime and computational similarity.

Keywords: Natural language process, Source code, class diagram, sequence diagram.

1. INTRODUCTION

With the increasing volume of computer systems, computational algorithms and information constructions are no longer significant architecture issues. The organisation of the entire system—the architecture of software—presents a fresh range of design issues when devices are built from many parts. This layout was discussed in a variety of respects, including informative diagrams and descriptive conditions, words of module linkages, templates and systems frameworks for particular domain requirements and formal designs of the inclusion processes for component models. A program or a computing system software architecture is the design or framework of the system, which includes software components, externally noticeable characteristics and links of those parts. The word references also to software architecture paperwork of a system. Software architecture documentation promotes stakeholder communication and records early decision-making regarding a high level of design and permits re-use of design parts and design patterns among project group.

Many businesses have enormous source code repositories, usually in various languages. A number of information with problems, metrics, and dependencies in source code also come from automatic source code analysis tools. In order to provide IT performance professionals, designers, customers and executives with insight, this data must be handled and visualized. There are several visualization techniques and instruments for the source code that deal with this issue at various achievement rates. We intend to apply the methods of Artificial Intelligence in this project to the issue of the visualization of source code.

The cooperation, word association and protein interaction applications of AI (Cluster Analysis), the social network and WWW assessment where a lot of information also needs to be handled are well established and generate fruitful outcomes. In this venture, we hope that we will find comparable possibilities in the field of software viewing. We also understand that our task is best defined as reverse architecture, a term which is written in the literature: reverse architecture is a flavor of reverse engineering that concerns the extraction from system implementation of softwares architecture models. Natural language processing (NLP) investigates how a machine works, understands and processes the language of human humans, understands human understanding, and communicates naturally with human humans. The NLP’s Apps include: IR recovery, knowledge extraction, QA, categorization of text, machine translation, reading help, voice identification, structure, and so forth. The growth of the Internet and the wide range of digital papers have led to an immediate need for smart text processing, which makes the concept and the ability of NLP more essential. The use of N-gram and WordNet –An algorithm has been created to recognize actual phrases with shortened identifiers and to divide identifiers that do not follow specific separators. We used N-gram method in the algorithm for the above-mentioned issues. N-gram is an NLP method, depending on the value N and dividing one phrase in pieces. For instance, "file" and N=2 are the chunks "fi," "il" and "le." The data and identifiers are not even combined sometimes even with the same significance. We have used WordNet to get a phrase and to make the corresponding method again to avoid synonyms, to discourage synonyms. WordNet is an English term and meaning relationship lexical database. A meaning is a specific significance of a phrase. WordNet offers a synset, a synonym list for this meaning for each meaning of a specific term. Many NLP apps need the significance resemblance between two brief documents to be calculated. For instance, the search engines must, over and above the overlap between words, model the relevance of a document to a query. Likewise, Q&A locations like Quora need to
determine if a query has previously already been requested. This text resemblance is often calculated when the two brief extracts are first embedded and the cosine similarity between them is calculated. Although phrases such as word2vec and GloVe have become normal methods in order to detect the semanticized resemblance between two phrases, there is little consensus about how to calculate phrase embeddings. Below are some of the most popular techniques to be reviewed and their results compared to two benchmarks. Traditionally, resemblance detection methods among documents focused on creating model documents. Several types of document models, such as the Boolene model, the vector model and the statistical probability model, have been developed in recent years. The boolean model provides a crossover and unification of sets to cover keywords. The boolean algorithm is susceptible to misuse, so that an extraction technique approximating a natural vocabulary is an advantage. The UML is an essential component of the software system development method and a significant knowledge base is included in class model de-indications. A UML class model creates an intricate hierarchy of specifications and it takes on significant human knowledge when compared with various classes to determine the respective components. An exponentially complicated assignment must be tackled in order to imitate such human ability. However, study involving UML class matching mainly revolves on detecting design patterns and surveys that address the issue of matching any class model are rather rare. Objective oriented strategy has been dominating the method of software development for more than two centuries. During this time the unified modeling language (UML) has become an essential structural foundation for software modeling and class modeling. The primary goal of an reverse engineering activity is to identify system components and interrelationships to create the system’s representation. The removal of factual bases begins with a normal inverse engineering activity. Source code is the most common, useful and trusted data source for factual analysis, because the real application may be lacking, obsolete or unsynchronized with other objects (records, release notes, version management information gathered, bug tracking system etc.).

2.RELATED WORKS

The renowned algorithms of Artificial Intelligence, such as graph clustering, either optimize particular statistical criteria or use a underlying structure or other recognized information characteristic. In this case, the data is taken from the software source code. On evaluation, the vertices of the chart are computer technology artefacts where the objects can have distinct granularities, starting with instructions / operators, methods / fields and finally courses, applications, applications and libraries. The edges of our chart are dependence between artefacts, with different granularities in turn: from the edges of the control flow graph to the edges of the calling method and the field graph and then to the edges of the coupling graph, the use of the packages or the use of library graphs. The next stage involves visualizing the obtained information and presenting it to the customer interactively, zooming, drilling, or expanding / collapsing. A prevalent issue with such instruments is that too many SE artefacts exist to look at all at once. In order to resolve this issue, the DA4Java instrument allows the User to add or extract the objects the User wants to visualise, while also drilling from the artefacts to the artefacts (for example, packages into courses). By combining AI methods such as clustering, teaching and classifying, we want to resolve the issue of overwhelming numbers of devices so that the customer receives an appropriate amount of organizations. The graph clustering in the software viewing domain is recognized from the available AI methods. Many software artifacts clusterers are apparently using non-MinCut-based methods for a wide analysis of current clusterizers and for a new clusterizer. It appears to be the case. This is based on some ground, since a MinCut-based clustering algorithm tends to generate skewed reductions. In other words, every slice contains a very tiny subgraph. But for graphs from the source code analysis field, this may not be a issue. The reality that the software usually contains clearly defined entrance points (solutions, in MaxFlow conditions, in terms of the maxflow-like algorithms) and lower defined departure points leads to another cause for MinCut-based clustering algorithms to be applied to our domain. A nice selection of sink points is also a question of study while present applicants are concerned with library function, dead-end function (which does not call anyone else) and termination points for program runtime. Fact extraction is an automatic procedure during which software entities and their shared interactions can be identified by analyzing the source code. This method leads to an abstract (model) depiction of the data extracted. Exports include syntactic removal of variable references and categories, procedure calls, package use, connection and class relationships. Class and interface collaborations in the OO scheme form the Class Cooperation Network (CCN). The word collaboration network of modules in procedural programming languages is also used as a module collaboration network. The concepts of grouping function-level entity in various paradigms constitute classes and modules. Similarly, we do not distinguish explicitly between interfaces and definition modules. The commentary that provides insight into code functionality is commonly used to help designers understand and maintain the system. Comments are being used to improve readability of code and communicate programmer plans as the most used documentary artifact for code comprehension, in addition to code, as a normal exercise in computer development[1]. The scope of a comment means a region in which the comment is included in the program. It includes some claims that are consistent with the description or the features listed in the remark. Three kinds of remarks exist: commentary on doc, comment blocks and comment lines. Comments from docs in Java, recognized as Javadocs, are immediately connected to courses, techniques or features while remarks from blocks and row are nestled within code. The range of a remark is therefore clearly the entire technique and the range of a commentary on a set or a row is not evident. Often software packages lack information about the layout and architecture of their source code or software[2]. Though full documents can only be produced by human developers, a high level description of source code can also be provided through groupings, generalization, and abstraction with a
instrument for automatic inferences. Such a instrument could also show areas to be taken care of by human specialists. Semi-automatic inference can be used when partly accessible information is provided. They imply the identification of the common purpose that software products serve in a groups by generalization. One method of determining the objective is to use the language information discovered in the source code, such as identification designations and remarks. This was performed, but before linguistic data was obtained, software engineering products were not partitioned into structurally linked organizations. We think that the first step is to consider official relationships (for instance feature calls or variable accesses), then to analyze the linguistic relationships in the recognized communities (e.g. by means of call graph clustering) instead of to analyze the vocabulary over the source code[3]. A huge quantity of source codes are accessible and free. The use of open source codes in projects can reduce the duration and cost of the project. While several code search engines (CSE) are accessible, it may be difficult to find the most appropriate code. We offer a structure for overcoming the above-mentioned challenge in this document. It begins with XML Class Diagram (Software Architecture) and gathers data from the XML folder, then selects appropriate initiatives using three kinds of GitHub, Source Forge, and Google Code crawlers. Then it discovers among the large number of downloaded applications the most appropriate initiatives. The Abstract Syntax Tree (AST), which extracts identifiers (class names, process names and feature name) and remarks, is presented in all java documents in every project. Words of action (verbs) are obtained from observations based on a POS. For matching, those identifiers and XML file data must be evaluated. With software source codes growing and developing further, it becomes more hard to identify appropriate code for a specific assignment within millions of code lines. When software reuse functions are performed, designers must first recognize the corresponding code pieces for reuse[4]. Temporarily, several code search engines such as Google Code Search, Krugle, Koders, Sourcerer and Codase are used to define code pertaining to the job designers. In this way a request is placed in one of CSEs by the designer. The customer will rewrite the request and search again depending on the significance of the outcomes. This procedure goes on until the outcomes are met by the customer. The customer has two significant tasks: (1) the question structure and (2) the determination of the relevance of the test outcomes. The most effective way to search for data is the keywords of several reference methods and the source code that is most suitable and relevant. The search centered on keywords can however produce appropriate outcomes together with meaningless outcomes. Identify the most significant outcome is the biggest task. Analyzing source code identifier with relevant class diagram data is the best way to tackle the task. Program Identifiers are a key source of software systems understanding data. Because programmers select program names to convey their software domain ideas. (Classes, Methods, Areas). Several NLP methods and methods for machine learning were used for the analysis of identifiers. Salton and Lesk suggested the first vector space model (VSM) recovery scheme, not just a binary technique of contrast. In proposing the ideas partial comparison and resemblance, the main input of the procedure was to allow the systems to calculate the resemblance between a text and a request depending on the distinct weights of index terms and to produce the results of the recovery classification. As regards updating a vector model, consumers first need to transform queries and records into vectors of the same scale in a database. The most common evaluation of semantic resemblance in the high dimensional space is to calculate the similitude between two vectors using cosin, whose value should be between 0 and 1 while the papers and queries are presented in the same vector spatial dimension. Overall, the following include the benefits of a space model. (1) With weights provided, VSM can choose features more efficiently than with the Boolean model and the efficiency of recovery is greatly enhanced. (2) VSM offers a partial comparative system, enabling the recovery of the most comparable allocation of papers. Wu et al. has a FAQ recovery scheme based on the VSM. The components of a vector are made up by the section issue and keyword[4]. Chim and Deng are proposing a sentence-based paper resemblance measure[5]. In[5], the Suffix Tree[5] stages weighted by TF-IDF are mapped to a high-dimensional VSM word area. Very lately,[6] provided a new phrase of computing resemblance. Their measurement is basically a VSM based model, based on semantical data and word order, which has achieved excellent efficiency in testing. The algorithms for machine learning work in a number vector. An picture can be considered a serial number — three digits for color definition — but it turns out that it is more helpful to take these figures and transform them. This phase is called feature extraction to take an picture and to create a vector of helpful figures. We have extracted features with various algorithms. This stage is especially used for pre-processing the source code file to enhance semantic code identification data collection when irrelevant and unwanted data such as pointless phrases, signs and phrases etc. are deleted. In order to minimize the size of the information, the semantic image of each source code file needs to be taken further. In order to identify the plagiarized source code appropriately, the objective of this module is to collect a big amount of sources. Source code pre-processor can be two types such as a pre-processing restriction that explicitly refers to source code and non-source code parameters. In[7] an adaptive network strategy to establish a plagiarism-recognition classifier that is capable of creating the weights and settings of a neural network was put forward. The neural network is tested empirically and is shown to work better in databases and documents. In[8] the genetic algorithm is used in two sequences to detect identity by correcting an mistake in the correction of problems of sub-gram isomorphism in graphical dependence. A methodology that leverages lexical data and fuzzy clustering is described in[9]. The results of a case study conducted on four software systems running in java are suggested in order to assess the effectiveness of the procedure. The information analyzes indicate that the use of lexical data and the furious clustering enhances the precision of the results achieved by prior design methods for the recovery of information according to structural understanding but retains the appropriate instances of design patterns. The methodology
for intrinsic plagiarism identification is included according to Latent Semantic Indexing (LSI) and the systematic approach[10]. LSI is used for the word matrix for information samples, but for estimating the intrinsic inscriptions styles, stylometry is used. This included an experiments series which examined the skills of the limit on dimensionality as the basic principle for LSI in order to gain insight into its impacts using certain small repositories. In[11], a relative assessment was provided to exploit the gourmets ‘ methods for the string tiling. This method mainly selects from the data sample a seed-source source document and, using a source code metric operation (for example, McCabe’s Cyclomatic Complication, the count logical, physical, commentary, and empty lines as well as lists of characteristics and processes) acknowledges the top K identical documents. Chava can process Java source files or class files, enabling distant applet with unusable source code to be analyzed. Analysis using class files is mainly feasible because of Java language characteristics. Failure to have a Java preprocessor implies that we do not have to cope with constructions like macros, databases and templates that do not contain any data in an object file. Java is also an architecture-neutral language, so you can inspect object data to find relations with your program in a machine-independent way. The performance figures show that well operating instrument scales show that Chava is working quicker than javac (the Java compiler) in order of size. Indeed, Chava’s Java program dumping the contents of a class file is also quicker than javap. The data base quantity produced is in the order of the class file type which can be managed very well. When compression is used on the database, size could be considerably decreased. There are tiny enough organizations and interactions for efficient queries. Users conduct tailored queries or relationships that investigate multiple organizational elements of the analysis of the C, C++ or Java program. An entity request enables a customer to use the attribute value only indicated in the left part of the query panel for selecting database documents. A connection request looks at interactions between organizations with the attribute numbers (left column) of the origin unit and the target unit (correct column) of a connection. The outcome can be shown in database mode, in graphic (these connections are laid out as a chart) or in text mode (the source lines can be shown). Software devices are typically altered to expand or change their features, increase their efficiency, take them to distinct platforms etc. The layout is carried out via piping of the intermediate database to the respective submission tool. It is important to designers to know the system’s high-level design before it is modified. The high level design may, however, not be visible to new developers because of the lack or, worse, inconsistency of the design documentation. This issue could be somewhat alleviated if designers could generate low-level source code layout descriptions. The first stage in the method of model regeneration is to remove module-level dependence from the source code and store the results in a database. These models are used in this phase by the software system. After a database has recorded all module-level dependence, the software system executes a data-database request text, filters the request outcomes and generates the Module Dependency Graph (MDG) visual description. For example: Different characteristics can be used in DOT-files for charts, nodes and corners. These characteristics can regulate color, form and row styles. For nodes and edges, after an announcement and before a semicolon, one or more attribute value pairs are placed in square brackets. Chart attribute pairs under graph component are defined as immediate value pairs. This paper is focusing on Java source code, presenting a graphic visualization strategy called DA4Java (Dependency Analyzer for Java), to understand dynamic dependencies between sources. DA4Java mainly has the aim of reducing cognitive effort to comprehend graphs of dependence. Long dependency graphs that overlap with many nodes and edges are generally not esthetic and need more understanding. With this strategy, we can only show the data that will address a certain understanding assignment by producing condensed, esthetic charts. DA4Java utilizes embedded charts and a number of functions to add and filter nodes and corners, for this purpose. The characteristics added enable the customer to create the dependency graph incrementally. For example, the function ‘Add callers’ provides techniques to call a chosen entity and calls only the appropriate procedure. Rigi is a tool focusing on mastering the structural complexity of large structures with graphical visualization. This is done by removing the data that is meaningless to the program and focuses on understanding the task of programming. It primarily follows a top-down assessment method and utilizes simple multi-perspective opinions of hierarchy (SHriMPs). They diminish confusion while maintaining the large image. Rigi offers a series of filters via border and node types or incoming and exiting interactions of dependence. It presents the notion of interchangeable nesting opinions to allow a customer to discover various data perspectives at various rates of abstraction. Creole is a Shrimp-based Eclipse plugin. The primary distinction with DA4Java is that these instruments lack characteristics for the graph structure.

3.PROPOSED MODEL
As the size of the open source projects is increasing exponentially, it is practically difficult to visualize the large volumes of source code due to noise and independent relationships. In the static and dynamic source code analysis, it is difficult to find and extract the essential keywords and methods due to similarity metric. In this proposed model, source code and compiled class files are used to analyse the contextual key patterns and relational patterns using the key phrase extraction and source code dependency graphs. Proposed framework is summarized in figure 1

---

3145

www.ijstr.org
In the figure, initially source code documents 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. Source code dependency graph is used to find the relational source code and class file metrics for pattern extraction. In the proposed model, a novel weighted contextual measure is used to improve the similarity context of the source codes for UML representation. Here, a novel clustering method is used to find the contextual relations among the different source code files. These relational source code metrics are used to construct the complex class diagram. Similarly, a novel class clustering measure is used to find the relational metrics in the compiled class files for sequence diagram construction.

Algorithm 1: Data Filtering
Input : Source code files SC, Class files CF.
Step 1: Read input source codes files SC.
Step 2: Read input class files CF.
Step 3: for each source code SC, in SC[]
          Do
            Project source codes
            Source code parsing
            Code dependency Graph
            Probabilistic weighted based Contextual dependency graph similarity
            Highest probabilistic relational methods and keywords
            Relational Class Diagram for source codes
            Relational Sequence Diagram for class files
            Extract Code metrics
            Project compiled class files

Figure 1: Proposed Model
Parse source code $SC_i$ with methods $M$ and Fields $F$.

\[ M_i = \text{ExtractMethods}(SC_i) \]
\[ F_i = \text{ExtractFields}(SC_i) \]

Mapping $(M_i, F_i)$ to $CS_i$

| $SC_1$ | $(M_1, F_1)$ |
| $SC_2$ | $(M_2, F_2)$ |
| $\ldots$ | $\ldots$ |
| $SC_n$ | $(M_n, F_n)$ |

done

Step 4: for each class file $CF_i$ in $CF[]$

Do

Parse class files $CF_i$ with methods $M$ and Fields $F$.

\[ M_i = \text{ExtractMethods}(CF_i) \]
\[ F_i = \text{ExtractFields}(CF_i) \]

Mapping $(M_i, F_i)$ to $CF_i$

| $CF_1$ | $(M_1, F_1)$ |
| $CF_2$ | $(M_2, F_2)$ |
| $\ldots$ | $\ldots$ |
| $CF_n$ | $(M_n, F_n)$ |

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

\[ M_i \in \text{Prob}(M_i \cap M_j / C); i = j \]
\[ F_i \in \text{Prob}(F_i \cap F_j / C); i = j \]

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 5: // Pre-processing source code comments using Stanford NLP parser.

For each document $d_i$ in $D$

Do

$T[]=$Tokenize($d_i$)

For each token $t$ in $T[]$


Do
Apply stemming, stopword removal using Stanford NLP library.

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 stopword removal as source code entities.

Probabilistic Weighted based contextual similarity measure for Source code and class files dependency graph
Input : Project source codes SC, Project class files CF, Project source metrics (SM\textsubscript{i},SF\textsubscript{i}) and Project class metrics (CM\textsubscript{i},CF\textsubscript{i}).

Procedure:
Step 1: Read source code metrics , sc\rightarrow(SM\textsubscript{i},SF\textsubscript{i}) and Project class metrics cf\rightarrow(CM\textsubscript{i},CF\textsubscript{i})
Step 2: Constructing a source code dependency graph SDG\rightarrow(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 probabilistic weights of the edges are computed using the vertex terms \(t_i\) and \(t_j\) where \(t_i \in V_i\) and \(t_j \in V_j\).

\[
\text{Edgeweight : } w(i, j) = \sum_{t_i, t_j} \frac{\text{Prob}(t_i, t_j)}{2 \cdot \max\{\text{Prob}(t_i), \text{Prob}(t_j)\} - \text{Prob}(t_i, t_j)}
\]

\(\text{Prob}(t_i, t_j)\) is the number of times both terms \((t_i, t_j)\) occurred together.
\(\text{Prob}(t_i)\) is the number of occurrence of \(t_i\) in vertex \(V_i\)
\(\text{Prob}(t_j)\) 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 \(U(SM_i) \leftarrow (m_1,m_2,...,m_n)\) denotes the source codes metrics vector at vertex \(i\).
\(V(SM_j) \leftarrow (m_1,m_2,...,m_n)\) denotes the source code metrics vector at vertex \(j\).

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

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

\[
|U(SM_i).V(SM_j)| = U(m_1).V(m_1) + U(m_2).V(m_2) + ... + U(m_p).V(m_q)
\]

Proposed Contextual source code dependency graph dissimilarity index is computed as

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

Contextual source code dependency graph similarity index
\(\text{CSDGSI} = 1 - \text{CSDGDI}\);
Contextual source code 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-IDF[i] = Compute term frequency tf-id
  Done
Step 5: Repeat until c clusters
  Find nearest cluster distance metrics using the following equation
  Let Document vector one V1, document vector 2 V2
  \[ \text{Dist}(V1, V2) = \sqrt{\text{Cosine}(V1[i], V2[i])} \]
  \[ \text{Correlation}(V1, V2) \times \sqrt{\sum V1[i]^2 + \sum V2[i]^2} \]
  Done
Step 6: Merge the graph nodes using the nearest distance measure.
Step 7: Update cluster centroid using mean distance.
Step 8: Construct the class diagram using the plant UML library to the filtered top k-clusters C[k].
Step 9: For each source code file SC[i] do
  Check the source code file has distance metric >0
  If(dist(SC[i], C[k]) > 0)
    Then
      Display class diagram in source code file SC[i].
  End if
Step 10: done

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

\[ |U(CM_i)| = \sqrt{U(m_1)^2 + U(m_2)^2 + \ldots + U(m_p)^2} \]

\[ |V(CM_j)| = \sqrt{V(m_1)^2 + V(m_2)^2 + \ldots + V(m_q)^2} \]

\[ |U(CM_i), V(CM_j)| = U(m_1) V(m_1) + U(m_2) V(m_2) + \ldots + U(m_p) V(m_q) \]

Proposed Contextual class code dependency graph dissimilarity index is computed as

\[ \text{CCDGDI} = \frac{\sqrt{|U(CM_i), V(CM_j)| \times \cos(|U(CM_i)| + |V(CM_j)|)}}{2 \times (|U(CM_i)| + |V(CM_j)|)} \text{; where } i \neq j \]

Contextual class code dependency graph similarity index

\[ \text{CCDGSI} = 1 - \text{CCDGDI}; \]

For each class file in CF[i]

Do

  Add to Sequence diagram designer S.

Done

Visualize sequence diagram to all the class files in the given relational packages.
4. EXPERIMENTAL RESULTS

Experimental results are performed on different open source object-oriented software projects such as Weka, Apache, JAMA etc. A total of five java open source projects 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. 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 \( N_C \) is source code that predict correctly and \( N_{IC} \) 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. Context similarity defines the number of source codes that are predicted correctly over the project source code documents with high contextual similarity.

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

<table>
<thead>
<tr>
<th>Software project</th>
<th>Number of lines</th>
<th>Number of Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weka</td>
<td>32544</td>
<td>425</td>
</tr>
<tr>
<td>Apache Commons Collections</td>
<td>26371</td>
<td>441</td>
</tr>
<tr>
<td>Jeuclid</td>
<td>12666</td>
<td>230</td>
</tr>
<tr>
<td>JfreeChart</td>
<td>95763</td>
<td>1013</td>
</tr>
<tr>
<td>JAMA</td>
<td>5644</td>
<td>253</td>
</tr>
</tbody>
</table>

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.

Figure 2: Sample directories of Weka class files

Figure 3, 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.
Figure 3, 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.

Table 2: Computational results of proposed model on the user selected package directory.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>associations</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>attributeSelection</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>classifiers</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>clusterers</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>core</td>
<td>04-09-2018 10:45</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>datagenerators</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>estimators</td>
<td>25-04-2016 21:33</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>experiment</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>filters</td>
<td>27-08-2018 16:44</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>gui</td>
<td>31-08-2018 15:14</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>knowledgeflow</td>
<td>11-12-2017 15:08</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>AbstractAssociate</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>9 KB</td>
</tr>
<tr>
<td>Apriori</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>61 KB</td>
</tr>
<tr>
<td>AprioriItemSet</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>22 KB</td>
</tr>
<tr>
<td>AssociationRule</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>5 KB</td>
</tr>
<tr>
<td>AssociationRules</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>4 KB</td>
</tr>
<tr>
<td>AssociationRulesProducer</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>3 KB</td>
</tr>
<tr>
<td>Associate</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>2 KB</td>
</tr>
<tr>
<td>AssociatorEvaluation</td>
<td>11-04-2016 13:06</td>
<td>JAVA File</td>
<td>9 KB</td>
</tr>
</tbody>
</table>

Figure 3: Sample directories of Weka class files

This program is free software you can redistribute it and/or modify under the terms of GNU General Public License as
null
obj1: name (#272) getDebuggingOutput signature (#240) ()Z=obj2. name (#273) numExecutionSlotsTipText signature (#233) ()Ljava/lang/String; type java/lang/String

obj1: name (#274) getNumExecutionSlots signature (#252) ()I=obj2. name (#275) setNumExecutionSlots signature (#249) ()V

obj1: name (#277) listOptions signature (#278) ()Ljava/util/Enumeration; type java/util/Enumeration=obj2. name (#285) setOptions signature (#286) ()Ljava/lang/String;V type java/lang/String

obj1: name (#293) getOptions signature (#294) ()Ljava/lang/String; type java/lang/String=obj2. name (#297) startSetToString signature (#233) ()Ljava/lang/String; type java/lang/String

obj1: name (#304) toString signature (#233) ()Ljava/lang/String; type java/lang/String=obj2. name (#306) search signature (#307) (Lweka/attributeSelection/ASEvaluation;Lweka/core/Instances;)[I type weka/attributeSelection/ASEvaluation type weka/core/Instances

obj1: name (#342) rankedAttributes signature (#343) ()[[D=obj2. name (#345) determineNumToSelectFromThreshold signature (#346) ()[[DV

obj1: name (#351) attributeList signature (#352) ()Ljava/util/BitSet;[I type java/util/BitSet=obj2. name (#355) resetOptions signature (#226) ()V

45. Node : 1 value: m_DoNotCheckCapabilities
46. Node : 1 value: Z
47. Node : 1 value: <init>
48. Node : 1 value: ()V
49. Node : 1 value: Code
50. Node : 1 value: LineNumberTable
51. Node : 1 value: LocalVariableTable
52. Node : 1 value: this
53. Node : 1 value: Lweka/attributeSelection/ASEvaluation;
54. Node : 1 value: doNotCheckCapabilitiesTipText
55. Node : 1 value: ()Ljava/lang/String;
56. Node : 1 value: setDoNotCheckCapabilities
57. Node : 1 value: ()Z
58. Node : 1 value: doNotCheckCapabilities
59. Node : 1 value: getDoNotCheckCapabilities
60. Node : 1 value: ()Z
61. Node : 1 value: buildEvaluator
62. Node : 1 value: (Lweka/core/Instances;)[V
63. Node : 1 value: Exceptions
64. Node : 1 value: postProcess
65. Node : 1 value: ()[I
66. Node : 1 value: attributeSet
67. Node : 1 value: []
68. Node : 1 value: forName
69. Node : 1 value: (Ljava/lang/String;Ljava/lang/String;)Lweka/attributeSelection/ASEvaluation;
70. Node : 1 value: evaluatorName
71. Node : 1 value: Ljava/lang/String;
Cluser-1
ASSearch.java
BestFirst.java
CheckAttributeSelection.java
ClassifierAttributeEval.java
ClassifierSubsetEval.java
GreedyStepwise.java
RankedOutputSearch.java
Ranker.java
StartSetHandler.java
WrapperSubsetEval.java

[D@281e2377 = [2, 12, 23]
Cluser-2
AttributeEvaluator.java
ErrorBasedMeritEvaluator.java
SubsetEvaluator.java

[D@46a2eca6 = [0, 7, 11, 13, 16, 17, 21, 24]
Cluser-3
ASEvaluation.java
CfsSubsetEval.java
CorrelationAttributeEval.java
GainRatioAttributeEval.java
InfoGainAttributeEval.java
OneRAttributeEval.java
ReliefFAttributeEval.java
SymmetricalUncertAttributeEval.java

[D@4ecbc9f0 = [4, 15, 25, 26]
Cluser-4
AttributeSetEvaluator.java
HoldOutSubsetEvaluator.java
UnsupervisedAttributeEvaluator.java
UnsupervisedSubsetEvaluator.java

[D@3033bb96 = [3, 5, 18]
Cluser-5
AttributeSelection.java
AttributeTransformer.java
PrincipalComponents.java
Figure 4: Complex class diagram

Figure 5: Sample Sequence Diagram
Table 3: Runtime comparison of proposed model on different open source java projects

<table>
<thead>
<tr>
<th>Tested Packages</th>
<th>Weka</th>
<th>JFreeChart</th>
<th>Jeuclid</th>
<th>JAMA</th>
<th>Apache lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4</td>
<td>2351</td>
<td>4193</td>
<td>3422</td>
<td>2749</td>
<td>2945</td>
</tr>
<tr>
<td>#5</td>
<td>2446</td>
<td>3846</td>
<td>3924</td>
<td>2742</td>
<td>2485</td>
</tr>
<tr>
<td>#2</td>
<td>1645</td>
<td>2943</td>
<td>2715</td>
<td>2364</td>
<td>1849</td>
</tr>
<tr>
<td>#3</td>
<td>1749</td>
<td>2013</td>
<td>2946</td>
<td>2553</td>
<td>2845</td>
</tr>
<tr>
<td>#6</td>
<td>2763</td>
<td>3975</td>
<td>3946</td>
<td>3953</td>
<td>3756</td>
</tr>
</tbody>
</table>

Figure 6: Runtime Comparison of computational runtime of class diagram on different open source projects

Figure 6, describes the comparison of open source projects and its average runtime computation for data processing. In this figure,

Table 4: Comparison of proposed conceptual similarity weights to the traditional measures.

<table>
<thead>
<tr>
<th>Models</th>
<th>Weka</th>
<th>JFreeChart</th>
<th>Jeuclid</th>
<th>JAMA</th>
<th>Apache lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling between objects</td>
<td>0.45</td>
<td>0.64</td>
<td>0.683</td>
<td>0.759</td>
<td>0.824</td>
</tr>
<tr>
<td>Message passing coupling</td>
<td>0.65</td>
<td>0.74</td>
<td>0.675</td>
<td>0.75</td>
<td>0.814</td>
</tr>
<tr>
<td>Data abstraction coupling</td>
<td>0.69</td>
<td>0.735</td>
<td>0.693</td>
<td>0.867</td>
<td>0.857</td>
</tr>
<tr>
<td>Conceptual Coupling Be- tween two Classes</td>
<td>0.864</td>
<td>0.824</td>
<td>0.874</td>
<td>0.903</td>
<td>0.873</td>
</tr>
<tr>
<td>Conceptual Coupling of a Class</td>
<td>0.894</td>
<td>0.913</td>
<td>0.893</td>
<td>0.874</td>
<td>0.921</td>
</tr>
<tr>
<td>ProposedWeightedConceptualSimilarity</td>
<td>0.953</td>
<td>0.946</td>
<td>0.93</td>
<td>0.964</td>
<td>0.972</td>
</tr>
</tbody>
</table>
CONCLUSION
In this paper, a novel complex class diagram and sequence diagram are generated using the contextual meaning of the source code and compiled files. In this work, a complex class diagram is constructed on the large source code documents using the weighted probabilistic graph clustering algorithm. Similarly, a novel sequence diagram is constructed using the compiled class files metrics. Experimental results are performed on the different open source java software systems with the traditional models in terms of runtime and computational similarity.

REFERENCES