Lightweight Approach to Java Sample Code Recommendation System Using Apriori-Based Soft Clustering

Yoshihisa Udagawa
Faculty of Informatics
Tokyo University of Information Sciences
Chiba-city, Chiba, Japan
e-mail: yu207233@rsch.tuis.ac.jp

Abstract— **One effective way of learning programming techniques is to refer to sample programs. However, it becomes difficult and time-consuming to find a suitable sample program visually for a complex programming subject. To overcome this shortcoming, research and development of recommendation systems for software engineering have been actively conducted. This paper discusses a recommendation system for Java sample programs using unsupervised machine learning techniques. The proposed system includes three major steps: (1) extracting invoked methods used in each sample program, (2) soft clustering the sample program by applying a data mining technique to the extracted methods, and (3) ranking programs in a cluster by calculating a weighted average concerning the extracted methods. Experiments using sample programs related to a graphical user interface and string handling have confirmed the effectiveness of the proposed recommendation system. A generative artificial intelligence model can successfully generate a description of the cluster using invoked method names that specify each soft cluster. The proposed recommendation system and a generative artificial intelligence model can collaborate for improving a programming education environment.**

Keywords-component; Recommendation System for Software Engineering; Maximal Frequent Itemset; Unsupervised Machine Learning; Soft clustering.

I. INTRODUCTION

This research paper is an extension of the previously reported contribution to the Java sample program recommendation system [1]. The study includes improvement of algorithms for analyzing program structure, and performing soft clustering. Additional experiments on programming subjects covering standard Java classes including a Graphical User Interface (GUI), string handling, file Input/Output (I/O), socket, multithreading, collection, etc. are performed.

It is widely recognized that sample programs provide an effective means for learning new programming techniques. In particular, sample programs for using Application Programming Interfaces (API) related to open-source programs are widely available on the Internet. Since the amount of publicly concerning sample programs becomes enormous, it might become time-consuming and error-prone to find an appropriate sample program visually. Over the past few decades, there has been a great deal of research and development on the software recommendation systems that provide useful programming information for students and developers.

Recommendation systems are originally employed in online stores and video/music websites, where rankings of items are calculated based on users' reactions and similarities among products and/or works. The recommendation system for software development is intended to assist programmer's effort. It is designed to deals with artifacts, such as sample programs, specifications, test cases and bug reports. Several techniques have been developed to collect, rank, and visualize similar artifacts based on various indicators reflecting their nature. These techniques are often specific to software engineering and cause a recommendation system to be called a Recommendation System for Software Engineering (RSSE) [2].

This paper discusses a sample program recommendation system using a soft clustering technique. The proposed system deals with Java sample programs that are collected from the Internet. Assuming that a characteristic of a Java program is determined by the API calls, the name of a declared method and the names of invoked methods are extracted from these sample programs. The system automatically clusters Java sample programs based on the invoked methods applying a data mining technique named *Apriori* algorithm [3]. Because the *Apriori* algorithm is based on a set theoretic relation, the algorithm implements soft clustering, where one sample program belongs to multiple clusters. The system ranks the sample programs in each cluster using a Term Frequency-Inverse Document Frequency (*tf-idf*) [4] or weighted vector space model. Experiments confirm that higher ranked samples tend to contain more types of invoked methods than those ranked lower, which means this system assists a student in selecting sample programs suitable for learning.

The contributions of this study are as follows:

I. In general, API call patterns differ from one programming subject to another. This system can soft cluster sample programs for each programming subjects based on the API call patterns. This process is automatic, as the system automatically determines parameters for optimal soft clustering, which is newly implemented in this study.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

II. The RSSEs proposed so far employ hard clustering, if any. In hard clustering, the results depend on the initial values and have the restriction that one sample belongs to only one cluster. This study employes soft clustering supported by a set theoretic relation. Therefore, a sample program can belong to multiple clusters, and a cluster only contains programs related by set theory. Soft clustering provides the optimal access paths that reflect characteristics of the sample program.

III. By modifying *tf-idf* to give greater weights to the methods that are used to define a cluster, sample programs that fit the subject of a cluster and include rare APIs are ranked higher.

IV. The proposed system employs unsupervised machine learning, making it lightweight to use, operate and maintain the system. In fact, simply by collecting sample programs and running the proposed system, a student can get suitable sample programs to support his/her learning.

The rest of this paper is organized as follows. Section II describes the state-of-the-art research on the RSSEs. Section III overviews the proposed system. Section IV describes the implementation of the main functions of the proposed system. Section V shows the experimental results using typical Java programming techniques. Section VI discusses other implementation options and collaboration with a generative AI model. Section VII concludes the paper with our plans for future work.

II. State-of-the-Art Research

This section outlines recent studies concerning recommendation systems for software engineering. Technically, they can be broadly classified into clustering, pattern mining, and similarity. Many studies use multiple techniques.

A. *Survey*

Gasparic and Janes [5] survey 46 research and development articles on RSSE published between 2003 and 2013, and categorize them with respect to covered data and methods for recommendation. The most common type of covered data is source code with 21 papers, followed by help information to perform source code changes with 6 papers. As for the recommendation methods, list format is the most common with 33 papers, followed by document format with three papers, and table format with two papers.

Ko, Lee, Park, and Choi [6] discuss the recommendation system research trends from a macro perspective using top-ranking articles and conference papers electrically published between 2010 and 2021. The study analyzes how the recommendation models and technologies are utilized in seven main service fields including education service and academic information service. Smart education that accesses vast digital resources has stimulated a rapid increase of educational recommendation systems. The goal of the systems is to provide learners with personalized educational materials.

B. *Clustering*

Katirtzis, Diamantopoulos, and Sutton [7] discuss an algorithm that extracts API call sequences and then clusters them to create an API usage summary known as a source code snippet. Hierarchical clustering is performed by calculating the distance of extracted API call sequences using the longest common subsequence (LCS) algorithm [8]. Then, code slice techniques are applied to create a source code snippet.

Chen, Peng, Chen, Sun, Xing, Wang, and Zhao [9] propose an approach for API sequence recommendation with three strategies, i.e., heuristic search using a modified longest common subsequence algorithm, clustering API sequence using a hierarchical clustering algorithm, and summarizing API sequence recommendations. They use the clustering to make it easier for programmers to find similar API recommendations and to facilitate the API selection. Since they use a modified hierarchical clustering algorithm, each API is always hard clustered belonging to just one cluster.

C. *Pattern Mining*

Hsu and Lin [10] propose a recommendation system based on frequent patterns in source code. They originally define 17 syntax patterns and extract them from the source code under study. A sequence pattern extraction algorithm based on frequency known as *Prefix-Span* [11] is applied to generate recommended API usage patterns.

Chen, Gao, Ren, Peng, Xia, and Lyu [12] discuss a method to mine the usage patterns of low frequency APIs. Their method is based on three views, i.e., method-API relationship for local view, API-API co-occurrence for global view, and project structure for external view. With experiments of several hundreds of Java projects, their method is confirmed to achieve an increased rate for retrieving the low-frequency APIs.

D. *Similarity*

Diamantopoulos and Symeonidis [13] develop a system to recommend sample code stored in software repositories on the Internet, such as GitHub, GitLab and Bitbucket. The input to the system is a code fragment presented by a user, and the output is a set of sample codes similar to the code fragment. Similarities among source codes are calculated based on the vector space model and the Levenshtein distance [14].

Hora [15] discusses a source code recommendation system that analyzes source code contained in a particular project and creates ranked API usage examples on a web site. The system ranks the source code based on three quality measures, i.e., similarity, readability, and reusability. The similarity is calculated using the cosine similarity [4][16] in data analysis, while readability and reusability are calculated using indicators developed in software engineering studies.

Nguyen, Rocco, Sipio, Ruscio, and Penta [17] implement a system to present API usage in a timely manner during a coding process and discuss the evaluation of experimental results. The system calculates the similarity among similar projects by *tf-idf* and ranks API usage patterns using a collaborative filtering technique [18].

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

E. Approach of this Study

This study concerns a recommendation system for sample programs based on API call patterns, which is similar to many of the studies described in this section. The system first soft clusters sample programs based on a set of frequently occurring APIs. Next, the *tf-idf* model is used to calculate the recommendation of the programs belonging to each cluster. The significant difference from previous studies is the implementation of soft clustering that allows a single program to belong to multiple clusters. The study also automatically adjusts a clustering parameter to optimize the number of clusters. This implementation allows us to efficiently handle the hundreds of sample programs required in programming education.

III. OVERVIEW OF PROPOSED SYSTEM

This section describes the architecture of the proposed system from the functional point of view and outlines typical usage with an example from experiments performed in this study.

A. Architecture

Figure 1 depicts the architecture of the proposed system. The input for this system is a collection of sample programs stored in the *Sample code repository*. Currently, these sample programs are manually collected from the Internet, and stored in a specific project typically in *Eclipse*, an Integrated Development Environment (IDE) for Java [19]. In this study, we assume that all sample programs are correct and work properly.

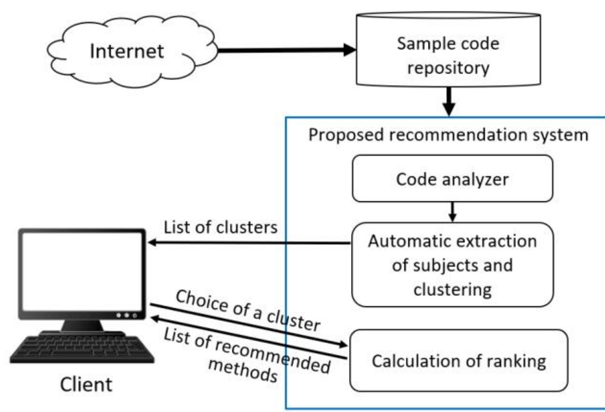

Figure 1. Overview of the proposed system.

Java programming techniques typically classified into several subjects, such as File I/O, collection, GUI, socket, and multithreading. This classification is widely accepted in programming education. The proposed system is designed to store Java sample programs in multiple packages or categories. Figure 2 shows the package structure used in this study, which is stored in a project of *Eclipse* named *Sample_Code*.

Programming education typically requires several to thirty Java sample programs in a package, though there is no limit to the number of Java files to include in each package. The *File_IO.Sample_1* and *File_IO.Sample_2* packages contain a set of sample programs for file IO, which is used for the experiments described in the previous paper [1].

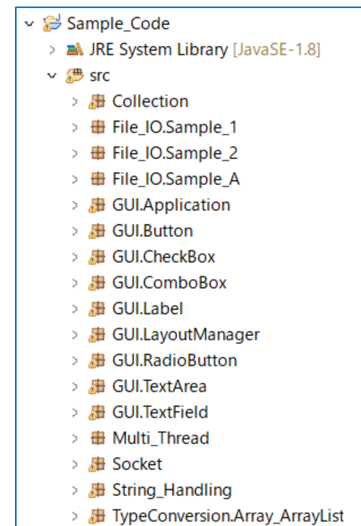

Figure 2. Package structure of *Sample_code*.

More than ten packages covering typical Java programming subjects are newly added. The *GUI.ComboBox* and *String_Handling* packages are used for the experiments described in the rest of this paper.

B. Starting Code Analyzer

The initial GUI screen of the proposed system contains only one *JComboBox* with the top directory of sample programs as an argument. The user of this system can view the package structure of the sample programs, and select one of the packages by pulling down the *JComboBox*. Figure 3 shows the screen dump that selects the *GUI.ComboBox* package.

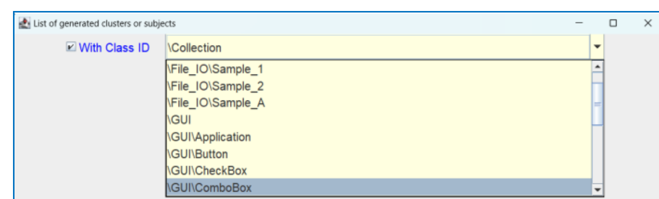

Figure 3. Screen dump for selecting the *GUI.ComboBox* package.

Then, the *code analyzer* in Figure 1 starts to extract declared method names and invoked method names from all Java files under the selected package or directory. A list of invoked method names is used for clustering the declared methods and ranking them.

Since Java allows a class to define its own methods, the same method name can be defined across multiple classes. In a Java program, a non-static method needs a reference variable to identify the class to which the method belongs. In this study, a new function to convert variable names to

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

class names is implemented in the *code analyzer* in order to uniquely distinguish non-static methods.

For example, Java has the *HashMap* and *TreeMap* classes. The both classes have the non-static *put* methods to insert an element to the map classes. The *code analyzer* generates *HashMap.put* and/or *TreeMap.put* by converting a reference variable to a class name. This conversion process is newly implemented in this study, and allows us to identify the difference between the *put* method in the *HashMap* class method and that in the *TreeMap* class.

C. Automatic Identification of Subjects and Clusters

Following code analysis, the *Apriori* algorithm [3] runs to identify the set of invoked methods that occur frequently. Programming subjects are automatically identified based on the frequent method set. Each subject corresponds to a cluster featured by the frequent method name set. Figure 4 shows an example of clustering with 17 identified clusters for the *GUI.ComboBox* package that includes 18 Java files and 45 declared methods.

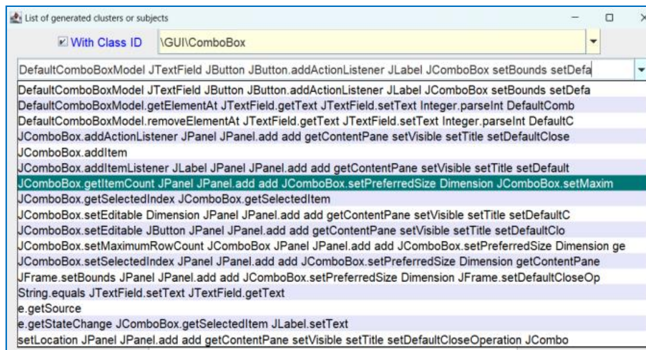

Figure 4. Identified programming subjects or clusters with class name.

Every non-static method name is preceded by a class name in Figure 4. Sometimes class names are so long that it is better to omit them for the purpose of a concise display. Unchecking the *With Class ID* checkbox at the top left corner of the initial GUI, the method names without class names are displayed as shown in Figure 5.

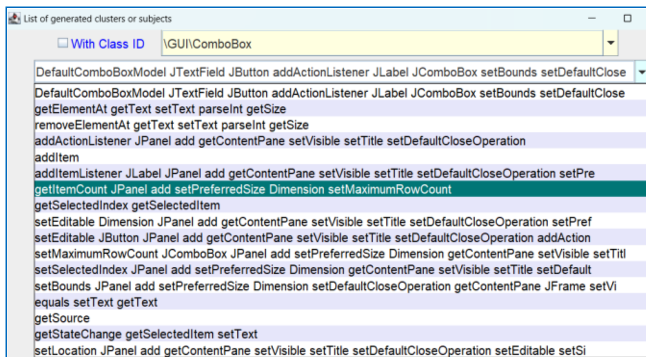

Figure 5. Identified programming subjects or clusters without class name.

Strictly, this study uses a soft clustering technique based on a maximal frequent itemset [20], i.e., a compact itemset that represents a frequent itemset. The method names displayed in Figures 4 and 5 are elements of a maximal frequent itemset. For example, "*getItemCount JPanel add setPreferredSize Dimension setMaximumRowCount*" suggests from the method names that the cluster is related to the programming techniques that specify the number of elements in a *JComboBox*, the size of a *JComboBox*, and the maximum number of rows that can be displayed.

D. Calculation of Recommended Ranking

Selecting an element in the *JComboBox* shown in Figures 4 and 5 causes to specify a cluster of methods, which starts calculations of recommendation values for each of the declared methods in the cluster. Figure 6 shows an example of a method recommendation. The values of recommendation for each declared method are normalized so that the maximum value is equal to one.

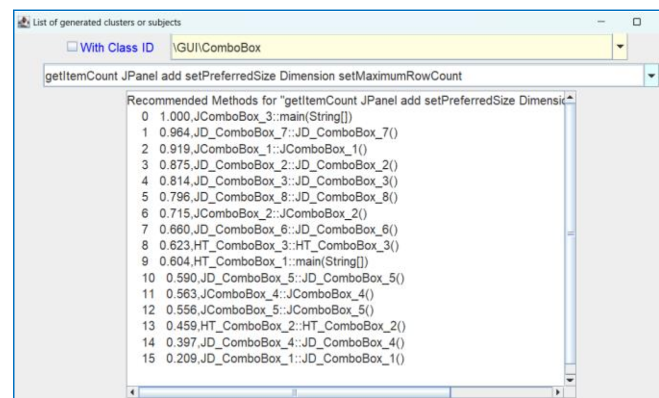

Figure 6. Sample of program recommendation.

Each method name is prefixed with a class name or a Java file name, so that a student can easily find out source code using an IDE, such as *Eclipse, NetBeans and IntelliJ IDEA*.

IV. IMPLEMENTATION

This section describes the implementation of three major steps for generating a recommendation. Those steps are code analysis, soft clustering, and ranking.

A. Code Analysis for Extracting Invoked Method Set

Functions necessary for system development are typically provided as runtime methods in Java. After learning the control structure of programs and object-oriented techniques, students and developers enhance their programming skills by learning how to use the runtime methods provided by Java communities. Therefore, the methods being invoked are closely related to the functionality of the programs under development. In this study, we assume that program similarity can be computed by the similarity of the method sets being invoked.

The *code analyzer* in Figure 1 extracts a declared method signature and a set of invoked method names. We implemented the *code analyzer* using the *Scanner* class [21], a tokenizer in Eclipse Java Development Tools (JDT) core.

This class provides the functionality to classify the tokens in a Java program into more than 100 types, and excludes comments for facilitating efficient analysis of executable statements. The *Scanner* class is also used in *Eclipse* [19] for navigating Java programs, including a class-method hierarchy and a list of field variables.

Figure 7 shows a sample of a Java program. Figure 8 shows the declared method signature and a list of invoked method names that are extracted from the Java program. A method or API with the same name is usually invoked multiple times in a declared method. Therefore, the *code analyzer* extracts the invoked method name and the number of times invoked, which are used for calculating cosine similarity [4][16]. For example, the *main* method in the *JComboBox_3* class in Figure 7 invokes the *Dimension* method twice, and *JComboBox.setPreferredSize* method twice, etc.

```
 1 package GUI.ComboBox;
 2 //  A sample code using "setMaximumRowCount()"
 3 import javax.swing.*;
 4 import java.awt.Dimension;
 5
 6 public class JComboBox_3 {
 7     public static void main(String[] args){
 8         String[] color= {"Light red", "Red", "Dark red",
 9                 "Light blue", "Blue", "Dark blue",
10                 "Light green", "Green", "Dark green"};
11         JPanel p = new JPanel();
12         //  Show 8 elements by default
13         JComboBox<String> comboA = new JComboBox<>(color);
14         comboA.setPreferredSize(new Dimension(120,25));
15         p.add(comboA);
16         // Pull-down to display all elements
17         JComboBox<String> comboB = new JComboBox<>(color);
18         comboB.setMaximumRowCount(comboB.getItemCount());
19         comboB.setPreferredSize(new Dimension(120,25));
20         p.add(comboB);
21
22         JFrame fm= new JFrame();
23         fm.getContentPane().add(p);
24         fm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25         fm.setBounds(10, 10, 400, 200);
26         fm.setTitle("JComboBox_3");
27         fm.setVisible(true);
28         System.out.println("--- Finished ---");
29     }
```

Figure 7. Sample Java program named *JComboBox_3.java.*

```
JComboBox_3::main(String[])
        Dimension, 2
        JComboBox.getItemCount, 1
        JComboBox.setMaximumRowCount, 1
        JComboBox.setPreferredSize, 2
        JFrame, 1
        JFrame.getContentPane, 1
        JFrame.setBounds, 1
        JFrame.setDefaultCloseOperation, 1
        JFrame.setTitle, 1
        JFrame.setVisible, 1
        JPanel, 1
        JPanel.add, 2
        add, 1
```

Figure 8. Invoked method names and the number of invoked times.

It should be noted that the methods, such as *println()* and *printStackTrace()*, are intentionally excluded from the extraction process because they are often used to print data values for debugging purpose. They are considered to fail to characterize the function of a declared method.

B. Soft Clustering Based on Apriori Algorithm

1) Apriori algorithm and maximal frequent itemset

Apriori algorithm proposed by Agrawal and Srikant [3] starts by identifying the frequent individual items of length one, and extending them to larger itemset as long as those itemset frequently appear in the database under consideration.

Let us a database D be a set of transactions t, i.e., $D= \{t_1, t_2, \ldots, t_n\}$. Let us each transaction t_i be a nonempty set of itemset, i.e., $t_i = \{i_{i1}, i_{i2}, \ldots, i_{im}\}$. The itemset is a nonempty set of items observed together.

The support value of an itemset is defined as the number of transactions in the database D. Using terms of the database D and transaction t_i, the support value of an itemset X is defined by the following formula:

$$Support(X) = |\{ t_i \in D : X \subseteq t_i \ \& \ 1 \leq i \leq n \}| \quad (1)$$

A set of items is called frequent if its support value is greater than a user-specified minimum support value, i.e., *minSup*.

Here, we cite the *Apriori* principle:

> *If an itemset is frequent, then all of its subsets are also frequent.*

This means that if a set is infrequent, then all of its supersets are infrequent. The *Apriori* algorithm works based on this principle, in which the frequent item sets of length k are utilized to identify frequent item sets of length $k+1$.

Since the frequent itemset generated by the *Apriori* algorithm tends to be very large, it is beneficial to identify a compact representation of all the frequent itemset. One such approach is to use a maximal frequent itemset [20].

Definition:

A maximal frequent itemset is a frequent itemset for which none of its immediate supersets are frequent.

Table I shows an example of a database consisting of five transactions of itemset.

TABLE I. EXAMPLE OF DATABASE

Transaction ID	Item set
1	A B C
2	A C D
3	A D
4	B C
5	B C D

Figure 9 illustrates an example of the maximal frequent itemset in a lattice structure where a node corresponds to an itemset and arcs correspond to the subset relation [20]. *MinSup* is set to 20% (= 1/5*100). Since the number of transactions in the database is 5, *minSup* 20% means if an

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

itemset appears once or more than once, it is frequent. In Figure 9, the nodes surrounded by solid lines indicate the frequent itemset, while the nodes with yellow backgrounds indicate the maximal frequent itemset. By definition, the maximal frequent itemset forms the boundary between frequent and infrequent itemset.

All frequent itemset can be derived from the set of maximal itemset. In Figure 9, the following three sets of itemset are generated form the maximal frequent itemset:

{{A, B, C} {A, B}, {A, C}, {B, C}, {A}, {B}, {C}}
{{A, C, D} {A, C}, {A, D}, {C, D}, {A}, {C}, {D}}
{{B, C, D} {B, C}, {B, D}, {C, D}, {B}, {C}, {D}}.

Each maximal frequent itemset defines a soft cluster of itemset where one element belongs to multiple clusters. For example, the itemset {A, C} belongs to the two clusters {A, B, C} and {A, C, D}.

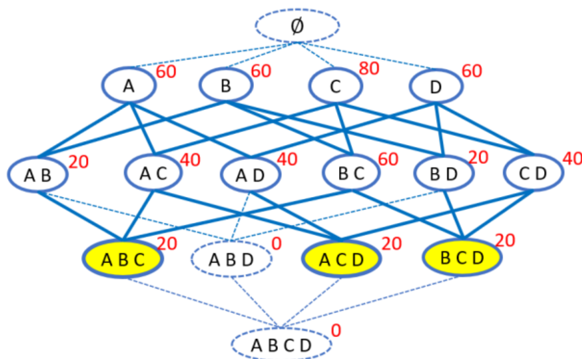

Figure 9. Maximal frequent itemset in lattice structure with 20% *minSup*.

Figure 10 illustrates an example of the maximal frequent itemset with *minSup* of 40%. In the case of Figure 10, the following four sets of itemset are derived:

{{A, C}, {A}, {C}}
{{A, D}, {A}, {D}}
{{B, C}, {B}, {C}}
{{C, D}, {C}, {D}}.

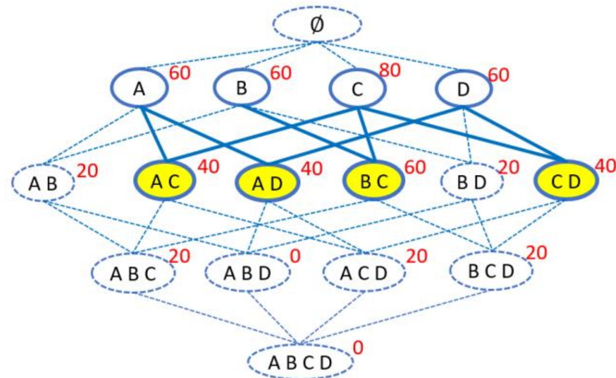

Figure 10. Maximal frequent itemset in lattice structure with 40% *minSup*.

It should be noted that the maximal frequent itemset, and thus the number of elements in the itemset, changes according to the value of *minSup*. In the proposed system, the value of *minSup* is varied by 1% to find the *minSup* that produces the maximal frequent itemset with the largest number of itemsets.

Table II shows some of the package names containing Java programs used in the experiment, the number of Java files, and the number of declared methods. Because *String* and *Collection* APIs are rather simple to use, only one declared method, i.e., *main*, is used in each Java file. Therefore, the number of Java files is equal to the number of declared methods. In contrast, the API related to *GUI* is complex to use, and multiple declared methods are used in a Java file. Therefore, the number of declared methods is larger than the number of Java files.

TABLE II. NUMBER OF JAVA FILES AND METHODS

Package	No. of Java files	No. of methods
Collection	11	11
String_Handring	31	31
File_IO	30	40
GUI.Label	6	12
GUI.ComboBox	18	45
GUI	72	152

Figure 11 shows the value of *minSup* and the number of clusters or itemsets in the maximal frequent itemset for each package. The maximum number of clusters is reached when the *munSup* is between 4% and 6%.

In general, there is a certain trend between the number of declared methods and the number of clusters. The *Collection* and *GUI.Label* packages have the eleven declared methods and the twelve declared methods, respectively. The number of clusters is maximized when *minSup* is between 4% and about 9%. The *File_IO* and *GUI.Combobox* packages have the 40 methods and the 45 methods, respectively. The maximum number of clusters is observed when *minSup* is between 4% and 6%. The *GUI* package consists of nine sub-packages and contains 152 declared methods. Maximum number of clusters occurs when *minSup* is 4%.

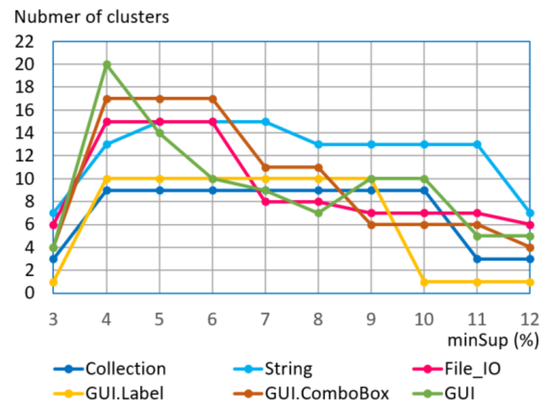

Figure 11. Values of *minSup* and the number of clusters.

In the current implementation, *minSup* is varied from 3% to 12% to count the number of generated clusters. Then, the

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

minSup that maximizes the number of clusters is determined. The lists of invoked methods shown in Figures 4 and 5 present clusters of a *GUI.ComboBox* with a *minSup* of 4%.

2) Soft clustering sample programs

More than ten binary programs that implement the *Apriori* algorithm are available on the web page maintained by Borgelt [22]. For the sake of openness and efficiency of implementation, this study uses *fpgrowth.exe* listed on the web page. Specifically, we implement a maximal-frequent-itemset generating function by calling *fpgrowth.exe* using *java.lang.Runtime.exec()* that executes the specified command and arguments as a separated process. The input data for this program is the set of invoked methods for each declared method, which is generated by the *code analyzer* ignoring the number of invoked methods citations. The result of running *fpgrowth.exe* is written to a file. Next, this file is read by the proposed recommendation system, which implements the linkage with the *Apriori* algorithm.

Figure 12 shows the maximal frequent itemset generated from the sample programs in *GUI.ComboBox* package shown in Figure 5, with a *minSup* of 4%. The maximal frequent itemset corresponds to the programming subjects. Figure 4 shows the complete set of method names preceded by the class name, which is actually used to calculate the recommended values.

```
0) DefaultComboBoxModel JTextField JButton addActionListener JLabel
   JComboBox setBounds setDefaultCloseOperation setTitle setVisible
   getContentPane Dimension setPreferredSize add JPanel
1) getElementAt getText setText parseInt getSize
2) removeElementAt getText setText parseInt getSize
3) addActionListener JPanel add getContentPane setVisible setTitle
   setDefaultCloseOperation
4) addItem
5) addItemListener JLabel JPanel add getContentPane setVisible setTitle
   setDefaultCloseOperation setPreferredSize Dimension setBounds
   JComboBox
6) getItemCount JPanel add setPreferredSize Dimension
   setMaximumRowCount
7) getSelectedIndex getSelectedItem
8) setEditable Dimension JPanel add getContentPane setVisible setTitle
   setDefaultCloseOperation setPreferredSize
9) setEditable JButton JPanel add getContentPane setVisible setTitle
   setDefaultCloseOperation addActionListener
10) setMaximumRowCount JComboBox JPanel add setPreferredSize
   Dimension getContentPane setVisible setTitle setDefaultCloseOperation
   setBounds
11) setSelectedIndex JPanel add setPreferredSize Dimension
   getContentPane setVisible setTitle setDefaultCloseOperation setBounds
   JComboBox
10) setMaximumRowCount JComboBox JPanel add setPreferredSize
   Dimension getContentPane setVisible setTitle setDefaultCloseOperation
   setBounds
11) setSelectedIndex JPanel add setPreferredSize Dimension
   getContentPane setVisible setTitle setDefaultCloseOperation setBounds
   JComboBox
12) setBounds JPanel add setPreferredSize Dimension
   setDefaultCloseOperation getContentPane JFrame setVisible
13) equals setText getText
14) getSource
15) getStateChange getSelectedItem setText
16) setLocation JPanel add getContentPane setVisible setTitle
   setDefaultCloseOperation setEditable setSize
```

Figure 12. Example of generated maximal frequent itemset.

Figure 13 shows a list of declared methods that contain at least one invoked method name that is included in a maximal frequent itemset. For example, clusters 1, 2, 4, and 7 are about *actionPerformed,* and *itemStateChanged*.

```
0)  DefaultComboBoxModel JTextField JButton addActionListener
    JLabel JComboBox setBounds setDefaultCloseOperation setTitle
    setVisible getContentPane Dimension setPreferredSize add JPanel
    <Intentionally omitted>
1)  getElementAt getText setText parseInt getSize
  JComboBox_2::actionPerformed(ActionEvent)
  JD_ComboBox_6::actionPerformed(ActionEvent)
  JD_ComboBox_7::actionPerformed(ActionEvent)
  JD_ComboBox_8::actionPerformed(ActionEvent)
2)  removeElementAt getText setText parseInt getSize
  JComboBox_2::actionPerformed(ActionEvent)
  JD_ComboBox_6::actionPerformed(ActionEvent)
  JD_ComboBox_7::actionPerformed(ActionEvent)
  JD_ComboBox_8::actionPerformed(ActionEvent)
3)  addActionListener add JPanel getContentPane setVisible setTitle
    setDefaultCloseOperation
  HT_ComboBox_1::main(String[])
  HT_ComboBox_2::HT_ComboBox_2()
  HT_ComboBox_3::HT_ComboBox_3()
  JComboBox_1::JComboBox_1()
  JComboBox_2::JComboBox_2()
  JComboBox_3::main(String[])
  JComboBox_4::JComboBox_4()
  JComboBox_5::JComboBox_5()
  JD_ComboBox_1::JD_ComboBox_1()
  JD_ComboBox_2::JD_ComboBox_2()
  JD_ComboBox_3::JD_ComboBox_3()
  JD_ComboBox_4::JD_ComboBox_4()
  JD_ComboBox_5::JD_ComboBox_5()
  JD_ComboBox_6::JD_ComboBox_6()
  JD_ComboBox_7::JD_ComboBox_7()
  JD_ComboBox_8::JD_ComboBox_8()
4)  addItem
  HT_ComboBox_3::actionPerformed(ActionEvent)
  JComboBox_1::JComboBox_1()
  JComboBox_1::actionPerformed(ActionEvent)
5)  addItemListener JLabel JPanel add getContentPane setVisible
    setTitle setDefaultCloseOperation setPreferredSize Dimension
    setBounds JComboBox
    <Intentionally omitted>
6)  getItemCount JPanel add setPreferredSize Dimension
    setMaximumRowCount
  HT_ComboBox_1::main(String[])
  HT_ComboBox_2::HT_ComboBox_2()
  HT_ComboBox_3::HT_ComboBox_3()
  JComboBox_1::JComboBox_1()
  JComboBox_2::JComboBox_2()
  JComboBox_3::main(String[])
  JComboBox_4::JComboBox_4()
  JComboBox_5::JComboBox_5()
  JD_ComboBox_1::JD_ComboBox_1()
  JD_ComboBox_2::JD_ComboBox_2()
  JD_ComboBox_3::JD_ComboBox_3()
  JD_ComboBox_4::JD_ComboBox_4()
  JD_ComboBox_5::JD_ComboBox_5()
  JD_ComboBox_6::JD_ComboBox_6()
  JD_ComboBox_7::JD_ComboBox_7()
  JD_ComboBox_8::JD_ComboBox_8()
7)  getSelectedIndex getSelectedItem
  HT_ComboBox_3::actionPerformed(ActionEvent)
  HT_ComboBox_3::itemStateChanged(ItemEvent)
  JComboBox_4::itemStateChanged(ItemEvent)
  JD_ComboBox_3::actionPerformed(ActionEvent)
  JD_ComboBox_5::itemStateChanged(ItemEvent)
8 to 16  <Intentionally omitted>
```

Figure 13. Declared methods belonging to each cluster.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Cluster 3 is about how to create a GUI containing a *ComboBox*. Cluster 6 is related to the *JComboBox* property settings. Since the *JComboBox* needs to be placed in a screen frame, the API for *JFrame*, e.g., lines 22-27 of Figure 7, are commonly included. The declared methods of sample programs of clusters 3 and 6 are overlapping as soft clustering is employed in this study. Since this stage is before the recommended ranks are calculated, only the declared method names belonging to each cluster are listed.

Due to space constraints, clusters 0, 5, 8 through 16 are intentionally omitted. Many of the clusters consist of the same 16 declared methods as listed in clusters 3 and 6. In this implementation, if a declared method includes one or more invoked methods that comprise a maximal frequent itemset, then it is treated as an element of the cluster corresponding to that maximal frequent itemset. Therefore, the same set of methods appears in many clusters. The implemented condition seems to be most appropriate. However, if a user wants to reduce the number of elements belonging to each cluster, it can be easily implemented by setting the number of methods included in a maximal frequent itemset to two or more.

C. Calculation of Recommendation Ranking

1) Definition of tf-idf

The Term Frequency-Inverse Document Frequency *(tf-idf)* weight is a statistical measure that is commonly used in information retrieval [4]. In the context of our study, the *tf-idf* can be rephrased as follows:

Tf (term frequency) means the frequency of an invoked method name in a sample program,

Idf (inverse document frequency) indicates a numerical value used for measuring the importance of an invoked method name in a set of sample programs.

Among several options to calculate the *tf* and *idf*, we adopt the following definitions.

Tf_i is defined as the number of occurrences of an invoked method i in declared method.

Idf_i is defined as $log(N/DF_i)$, where N is the total number of declared methods that occur in a package of sample programs, and DF_i is the number of declared methods where an invoked method i appears at least once. It should be noted that idf_i of an invoked method i that appears in all declared methods is equal to $log(N/N)$, which is equal to 0.

2) Calculating Tf-idf for Sample Program Recommendation

As mentioned earlier, the maximal frequent itemset consists of a set of method names that suggest programming subjects. Examples of the maximal frequent itemset is displayed on the *JCombobox* in the GUI as shown in Figures 4 and 5. The proposed system identifies a set of declared methods related to the maximal frequent itemset when a user selects a cell on the *JCombobox*. Then, the proposed system starts to compute *tf* and *idf* for each of invoked methods that are defined in the set of declared methods.

Table III lists the *tf* and *idf* values of the invoked method names relating to the maximal frequent itemset *{getItemCount, JPanel, add, setPreferredSize, Dimension, setMaximumRowCount}* that is shown on the seventh line from the top in Figure 5. There are 35 invoked methods in the 16 declared methods in cluster 6 that concerns the maximal frequent itemset.

TABLE III. *TF* AND *IDF* VALUES FOR INVOKED METHOD NAMES

No.	Invoked method name	Tf	Idf
0	DefaultComboBoxModel	2	0.903
1	Dimension	14	0.058
2	Font	1	1.204
3	JButton	6	0.426
4	JButton.addActionListener	6	0.426
5	JButton.setFont	1	1.204
6	JComboBox	9	0.25
7	JComboBox.addActionListener	2	0.903
8	JComboBox.addItemListener	3	0.727
9	JComboBox.getItemCount	2	0.903
10	JComboBox.setEditable	3	0.727
11	JComboBox.setEnabled	1	1.204
12	JComboBox.setMaximumRowCount	3	0.727
13	JComboBox.setPreferredSize	14	0.058
14	JComboBox.setSelectedIndex	2	0.903
15	JComboBox.setSelectedItem	1	1.204
16	JFrame	2	0.903
17	JFrame.getContentPane	2	0.903
18	JFrame.pack	1	1.204
19	JFrame.setBounds	2	0.903
20	JFrame.setDefaultCloseOperation	2	0.903
21	JFrame.setTitle	1	1.204
22	JFrame.setVisible	2	0.903
23	JLabel	8	0.301
24	JPanel	16	0
25	JPanel.add	16	0
26	JTextField	4	0.602
27	add	16	0
28	getContentPane	14	0.058
29	setBounds	11	0.163
30	setDefaultCloseOperation	13	0.09
31	setLocation	2	0.903
32	setSize	2	0.903
33	setTitle	13	0.09
34	setVisible	13	0.09

Since the proposed system uses soft clustering based on a maximal frequent itemset, the method names that are included in the maximal frequent itemset should be considered to characterize the sample programs more strongly than the others. In this study, the weights of the invoked method names are adjusted using the following formula.

Let us *MFI* be the Maximal Frequent Itemset specified by a user and idf_{max} be the maximum of *idf* values.

$$\begin{aligned} Adjusted\ idf_j &= idf_j + idf_{max} \quad \text{if } j \in MFI \\ &= idf_j \quad \text{if } j \notin MFI \end{aligned} \tag{2}$$

Table IV shows the adjusted *idf* values for the maximal frequent itemset *{getItemCount, JPanel, add, setPreferredSize, Dimension, setMaximumRowCount}*. The

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

add methods are defined in both of the *JPanel* and *JFrame* classes. They are distinguished in the internal processing. Because the *add* method of the *JFrame* class is called via the *getContentPane* method, as shown in line 23 of Figure 7, it is simply denoted by *add*.

TABLE IV. ADJUSTED *IDF* VALUES

Invoked method name	Idf	Adjusted idf
Dimension	0.058	1.262
JComboBox.getItemCount	0.903	2.107
JComboBox.setMaximumRowCount	0.727	1.931
JComboBox.setPreferredSize	0.058	1.262
JPanel	0	1.204
JPanel.add	0	1.204
add	0	1.204

The degree of recommendation $DegR_i$ for a declared method *i* is calculated as:

$$DegR_i = \sum_{k=0}^{k=M-1} tf_{ik} * (Adjusted\ idf_k) \quad (3)$$

where tf_{ik} is the number of occurrences of the invoked method *k* in the declared method *i,* and idf_k is the inverse document frequency of the invoked method *k*.

Table V shows the degrees of recommendation for the declared method regarding the maximal frequent itemset *{getItemCount, JPanel, add, setPreferredSize, Dimension, setMaximumRowCount}*.

TABLE V. DEGREES OF RECOMMENDATION FOR SAMPLE PROGRAM

No.	DegR	Name of sample program
0	19.623	JComboBox_3::main(String[])
1	18.926	JD_ComboBox_7::JD_ComboBox_7()
2	18.038	JComboBox_1::JComboBox_1()
3	17.178	JD_ComboBox_2::JD_ComboBox_2()
4	15.967	JD_ComboBox_3::JD_ComboBox_3()
5	15.614	JD_ComboBox_8::JD_ComboBox_8()
6	14.028	JComboBox_2::JComboBox_2()
7	12.956	JD_ComboBox_6::JD_ComboBox_6()
8	12.216	HT_ComboBox_3::HT_ComboBox_3()
9	11.856	HT_ComboBox_1::main(String[])
10	11.576	JD_ComboBox_5::JD_ComboBox_5()
11	11.041	JComboBox_4::JComboBox_4()
12	10.916	JComboBox_5::JComboBox_5()
13	8.998	HT_ComboBox_2::HT_ComboBox_2()
14	7.781	JD_ComboBox_4::JD_ComboBox_4()
15	4.104	JD_ComboBox_1::JD_ComboBox_1()

The maximal degree of recommendation is normalized to be 1.000 and displayed in the text area of the GUI. For the lists in Table V, the normalized degrees of recommendation are obtained by dividing all the degrees by 19.623. This calculation generates the final list of recommendations shown in Figure 6.

V. EXPERIMENTAL RESULTS

This section describes two sets of experimental results. The first set of experimental results is about declared methods or sample programs relating to the *GUI.Combobox* package. Because GUI components in Java are typically embedded in a screen frame called *JFrame*, the declared methods for the *GUI.Combobox* package inevitably accompany *JFrame* APIs. Consequently, they are often complicated. The other set of experimental results concerns the *String_Handring* package. APIs for the *String* class tend to be called alone. Therefore, the declared methods for the *String_Handring* package are often concise.

A. Experiment on Programs in GUI.Combobox Package

Figure 14 shows the sample Java files included in the *GUI.Combobox* package that is listed on the 10th line from the bottom in Figure 2. The number of Java files is 18, and the number of declared method is 45 as shown in Table II.

Figure 14. Sample Java files included in *GUI.Combobox* package.

Let us the programming subject be "*getItemCount JPanel add setPreferredSize Dimension setMaximumRowCount*" as listed on the seventh line from the top in Figure 5. The generated recommendation list is shown in Figure 6. Figure 7 shows the source program of the declared method named *JComboBox_3.java::main(String[])* with the normalized recommendation value of 1.000. This method has the top recommended rank because it contains all the invoked method names or APIs that make up the programming subject.

Figure 15 shows the declared method of eighth recommended rank with the normalized recommendation value of 0.660 named *JD_ComboBox6()*. This method fails to include two APIs of programming subjects, i.e., *getItemCount* and *setMaximumRowCount*. Instead, it includes APIs, such as *JTextField* and *JButton*.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
22      JD_ComboBox_6(){
23          String[] combodata = {"Swing", "Java2D", "Java3D", "JavaMail"};
24          model = new DefaultComboBoxModel<String>(combodata);
25          JComboBox<String> combo = new JComboBox<String>(model);
26          combo.setPreferredSize(new Dimension(80, 30));
27          JPanel p = new JPanel();
28          p.add(combo);
29
30          JPanel controlPanel = new JPanel();
31          text = new JTextField(10);
32          button = new JButton("add");
33          button.addActionListener(this);
34          controlPanel.add(text);
35          controlPanel.add(button);
36
37          getContentPane().add(p, BorderLayout.CENTER);
38          getContentPane().add(controlPanel, BorderLayout.PAGE_END);
39
40          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41          setBounds(10, 10, 300, 200);
42          setTitle("JD_ComboBox_6");
43          setVisible(true);
44      }
```

Figure 15. Sample program with normalized recommendation value of 0.660.

Figure 16 shows the declared method named *JD_ComboBox_1()* that is ranked at the end of recommendation list with the normalized recommendation value of 0.209. The method is a basic program for the usage of *JComboBox* and its integration into *JFrame*.

```
19      JD_ComboBox_1(){
20          String[] combodata =
21              {"Swing", "Java2D", "Java3D", "JavaMail"};
22          JComboBox<String> combo = new JComboBox<String>(combodata);
23          JPanel p = new JPanel();
24          p.add(combo);
25
26          getContentPane().add(p, BorderLayout.CENTER);
27          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28          setBounds(10, 10, 300, 200);
29          setTitle("JD_ComboBox_1");
30          setVisible(true);
31      }
32  }
```

Figure 16. Sample program with normalized recommendation value of 0.209.

Because of the recommended value calculation formula, a declared method that is closely related to a programming subject is ranked high. In general, a lower-ranked declared method is concise and better suited for beginners in learning programming because it contains fewer APIs or invoked methods. A declared method containing many APIs and less related to the programming subject tends to be ranked in the middle of the recommendation list.

B. *Experiment on Programs in String_Handring Package*

In Java programming language, the *String* class provides various APIs that can be used to handle string data. It includes APIs like *length, charAt, equals, indexOf, substring, toUpperCase, toLowerCase*, etc. These APIs facilitate string processing.

Figure 17 shows the sample Java files included in the *String_Handring* package that is located on the second line from the bottom in Figure 2. The number of Java files is 31, which is the same as the number of the declared method named *main* as shown in Table II. Since sample programs on the *String* class are rather simple, only one declared method is defined in each Java file.

Figure 17. Sample Java files included in *String_Handring* package.

Figure 18 shows 15 identified programming subjects or clusters of the *String_Handring* package. Figure 18 reveals some commonly used APIs for string processing, such as *equals*, *indexOf*, and *substring*. Since this system uses soft clustering, there are APIs common to multiple clusters. For example, the *equals* API appears in four clusters, the *compareTo* API in three clusters as shown in Figure 18.

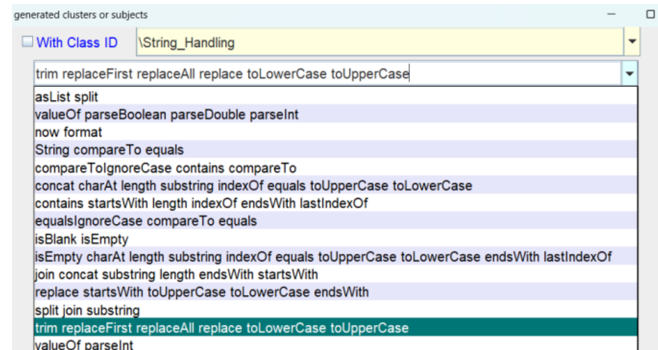

Figure 18. Identified programming subjects of *String_Handring* package.

Figure 19 shows a list of recommended declared methods for the programing subject "*trim replaceFirst replaceAll replace toLowerCase toUpperCase.*" The proposed system lists five declared methods with recommended values from 1.000 to 0.275.

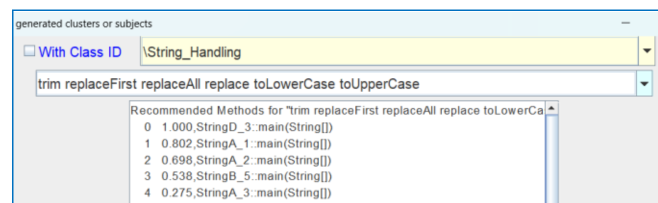

Figure 19. List of recommended declared methods.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Figure 20 shows the top-ranked sample program with the normalized recommendation value of 1.000. This sample program contains all the method names that constitute the programming subject.

```
1 package String_Handling;
2
3 public class StringD_3 {
4     public static void main(String[] args) {
5         String a = "Java 4 all ??";
6         System.out.println(a.replace("?", "!"));          // Java 4 all !!
7         System.out.println(a.replaceAll("[a-z]+", "_"));   // J_ 4 _ ??
8         System.out.println(a.replaceFirst("[a-z]+", "_")); // J_ 4 all ??
9         System.out.println(a.toUpperCase());              // JAVA 4 ALL ??
10        System.out.println(a.toLowerCase());              // java 4 all ??
11        //  to remove whitespace from both ends of a string.
12        String b = "      Java 4 Every Person !!    ";
13        System.out.println(b.trim());             //Java 4 Every Person !!
14    }
15 }
```

Figure 20. Sample program with normalized recommendation value of 1.000.

Figure 21 shows the sample program with a normalized recommendation value of 0.802. This sample program only contains two method names that constitute the programming subject, i.e., *toLowerCase* and *toUpperCase*. However, this program has a high recommended value because it contains many methods related to *String* class, such as *equlas*, *indexOf*, and *substring*.

```
1 package String_Handling;
2 //  http://www.btechsmartclass.com/java/java-string-handling.html
3 //  Java String Handling
4
5 public class StringA_1 {
6
7     public static void main(String[] args) {
8         String title = "Java Tutorials";
9         String siteName = "www.btechsmartclass.com";
10        System.out.println("Length of title: " + title.length());
11        System.out.println("Char at index 3: " + title.charAt(3));
12        System.out.println("Index of 'T': " + title.indexOf('T'));
13        System.out.println("Last index of 'a': " + title.lastIndexOf('a'));
14        System.out.println("Empty: " + title.isEmpty());
15        System.out.println("Ends with '.com': " + siteName.endsWith(".com"));
16        System.out.println("Equals: " + siteName.equals(title));
17        System.out.println("Sub-string: " + siteName.substring(9, 14));
18        System.out.println("Upper case: " + siteName.toUpperCase());
19        System.out.println("Lower case: " + siteName.toLowerCase());
20    }
21 }
```

Figure 21. Sample program with normalized recommendation value of 0.802.

Figure 22 shows the sample program with the normalized recommendation value of 0.275. This program only includes the *replace* method twice, causing to a low recommendation.

```
1 package String_Handling;
2
3 public class StringA_3 {
4     public static void main(String[] args) {
5         String s1="your name is java, isn't it?";
6         String str=s1.replace('a','w'); //replaces all of 'a' to 'w'
7         System.out.println(str);     // your nwme is jwvw, isn't it?
8
9         str=s1.replace("is","was"); // replaces all of "is" to "was"
10        System.out.println(str);     // your name was java, wasn't it?
11    }
12 }
```

Figure 22. Sample program with normalized recommendation value of 0.275.

In the recommendation calculation proposed in this study, sample programs with fewer method types generally rank lower than those with richer in method types. However, the simpler program can be useful for beginners in programming because of its conciseness.

VI. DISCUSSION

A. Syntax Analysis

In this study, the *Scanner* [21] class is used for parsing sample programs mainly because it reduces development effort. There are several options of parsing tools, including *JavaParser* [23] and *ANTLR* [24], both of which generate an Abstract Syntax Tree (AST). An AST is an intermediate representation of a source program represented by a tree structure. A few hundred lines of programming for traversing an AST allow an application to perform more complex operations than a mere method name extraction. ANTLR can parse formal languages including Java. All parsing tools work independent of IDEs and can parse sample programs stored in arbitrary directories.

B. Generative AI

ChatGPT is a chat-based generative AI released by OpenAI in Nov 2022 [25]. The *ChatGPT August 3* version allows users to get Java sample programs for *JComboBox* successfully. Since Java programs are generally characterized by APIs they call, *ChatGPT* precisely generates a report that contains a targeted sample program using a prompt including those APIs.

For example, the following prompt generates a report with a sample program that sets the number of elements to be displayed in a *JComboBox*'s dropped-down list using the *setMaximumRowCount* method:

> Would you show me a sample Java program about *JComboBox* using *setMaximumRowCount* method?

The proposed system and *ChatGPT* can be used to support each other. The proposed recommendation system automatically generates a list of APIs, which is helpful for writing prompts to *ChatGPT*.

For example, the list of APIs identified by the proposed system facilitates writing the following prompt:

> Would you show me a programming subject using the following Java APIs: "*getItemCount JPanel add setPreferredSize Dimension setMaximumRowCount*"?

Figure 23 shows a gist of *ChatGPT*'s response to this prompt. The response briefly states the subject and also suggests areas for further study.

> **Subject:** Creating a *JComboBox* within a *JPanel*
> **Objective:** Build a GUI application that contains a *JComboBox* inside a *JPanel*, allowing users to select options from the dropdown list. < *Intentionally omitted* >
> This subject will allow you to explore GUI customization and layout management in Java Swing while using the mentioned Java APIs to create a visually appealing and interactive user interface.

Figure 23. Gist of response from *ChatGPT*.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The proposed recommended system has a lot of potential to improve the programming learning environment by working together with generative AI models, such as *ChatGPT*, *Bing*, *Bard* and *Claude* [26].

VII. CONCLUSION AND FUTURE WORK

This study deals with a recommendation system of Java sample programs using unsupervised machine learning. The proposed system soft clusters the sample program based on the set of invoked method names that are frequently observed. The clustering that corresponds to a programming subject is performed automatically using the *Apriori* algorithm. The recommended ranking of the sample programs in a cluster is calculated based on an adjusted *tf-idf* model that takes the method name and the number of times it is invoked.

This study is an extension of a previously published study [1]. The system described in this paper has been significantly enhanced in its functionality to perform source program parsing and soft clustering. Enhancements in parsing have made it possible to accurately parse complex sample programs, which allows the proposed system to handle sample programs on a variety of Java programming subjects. The functionality to optimize the value of *minSup*, i.e., a parameter of the *Apriori* algorithm, has been introduced to automatically perform optimal soft clustering.

It is confirmed through experiments using sample programs in the *File_IO, GUI*, and *String_Handring* packages, etc. that the sample programs containing APIs related to a programming subject are ranked high on a produced recommendation list. In addition, the set of APIs automatically identified by the proposed recommendation system is helpful for writing successful prompts for generative AI models including *ChatGPT*. The combination of the proposed system and the generative AIs offers significant potential to provide an unprecedented programming education environment.

Manual sample program acquisition from the Internet is time consuming and is a subject for future research. Additional experiments with larger number of sample programs are planned for a programming class room. Experiments in cooperation with generative AI are also planned.

REFERENCES

[1] Y. Udagawa, “Lightweight Sample Code Recommendation System to Support Programming Education,” The Ninth International Conference on Advances and Trends in Software Engineering (SOFTENG 2023), IARIA, Apr. 2023, pp. 1-7, ISSN: 2519-8394, ISBN: 978-1-68558-042-1.

[2] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” IEEE Software 27, pp. 80-86, Jul. 2010, DOI: 10.1109/MS.2009.161

[3] R. Agrawal and R. Srikant, “Mining sequential patterns,” The 11th IEEE International Conference on Data Engineering (ICDE), pp. 3-14, 1995, DOI: 10.1109/ICDE.1995.380415

[4] G. Sidorov. “Vector Space Model for Texts and the tf-idf Measure,” In Syntactic n-grams in Computational Linguistics, pp. 5-14, Apr. 2019, Springer, Cham, ISBN: 978-3-030-14770-9.

[5] M. Gasparic and A. Janes, “What Recommendation Systems for Software Engineering Recommend: A Systematic Literature Review,” Journal of Systems and Software 113, pp. 101-113, Mar. 2016, DOI: 10.1016/j.jss.2015.11.036

[6] H. Ko, S. Lee, Y. Park, and A. Choi, “A Survey of Recommendation Systems: Recommendation Models, Techniques, and Application Fields,” Electronics vol. 11, pp. 141-188, Jan. 2022, DOI: 10.3390/electronics11010141

[7] N. Katirtzis, T. Diamantopoulos, and C. Sutton, “Summarizing Software API Usage Examples Using Clustering Techniques,” The 21st International Conference on Fundamental Approaches to Software Engineering, vol. 10802, Springer, pp. 189-206, Apr. 2018, DOI: 10.1007/978-3-319-89363-1_11

[8] “Longest Common Subsequence,” Available from: https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_longest_common_subsequence.htm

[9] C. Chen, X. Peng, B. Chen, J. Sun, Z. Xing, X. Wang, and W. Zhao, “More Than Deep Learning: Post-processing for API Sequence Recommendation,” Empirical Software Engineering. vol.27, pp. 1-32, Oct. 2021, Available from: https://ink.library.smu.edu.sg/sis_research/6580

[10] S.-K. Hsu and S.-J. Lin, “Mining Source Codes to Guide Software Development,” Asian Conference on Intelligent Information and Database Systems (ACIIDS 2010), pp. 445-454, Mar. 2010, DOI: 10.1007/978-3-642-12145-6_46

[11] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, “Prefixspan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,” The 17th international conference on data engineering, pp. 215-224, Apr. 2001.

[12] Y. Chen, C. Gao, X. Ren, Y. Peng, X. Xia, and M. R. Lyu, “API Usage Recommendation Via Multi-View Heterogeneous Graph Representation Learning,” IEEE Transactions on Software Engineering, vol. 49, pp. 3289-3304, May 2023, DOI: 10.1109/TSE.2023.3252259

[13] T. Diamantopoulos and A. Symeonidis, “Mining Source Code for Component Reuse,” Mining Software Engineering Data for Software Reuse, Advanced Information and Knowledge Processing, Springer, pp. 133-174, Mar. 2020, DOI: 10.1007/978-3-030-30106-4_6

[14] “Levenshtein Distance,” Wikipedia, Nov. 2023, Available from: https://en.wikipedia.org/wiki/Levenshtein_distance

[15] A. Hora, “APISonar: Mining API usage examples,” Wiley Online Library, Software: Practice and Experience, vol. 51, issue 2, pp. 319-352, Oct. 2[4]020, DOI: 10.1002/spe.2906

[16] “Cosine Similarity,” Wikipedia, Oct. 2023, Available from: https://en.wikipedia.org/wiki/Cosine_similarity

[17] P. T. Nguyen, J. D. Rocco, C. D. Sipio, D. D. Ruscio, and M. D. Penta, “Recommending API Function Calls and Code Snippets to Support Software Development,” IEEE Transactions on Software Engineering, vol. 48, issue 7, pp. 2417-2438, Jul. 2022, DOI: 10.1109/TSE.2021.3059907

[18] A. Roy, “Introduction to Recommender Systems-1: Content-Based Filtering and Collaborative Filtering,” Jul. 29, 2020, Available from: https://towardsdatascience.com/introduction-to-recommender-systems-1-971bd274f421

[19] Eclipse foundation, “Download Eclipse Technology that is right for you,” Nov. 2023, Available from: https://www.eclipse.org/downloads/

[20] J. Rousu, “582364 Data mining, 4 cu Lecture 4: Finding frequent itemsets - concepts and algorithms,” University of Helsinki, Apr. 2010, Available from: https://www.cs.helsinki.fi/group/bioinfo/teaching/dami_s10/dami_lecture4.pdf

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[21] Eclipse documentation “Interface IScanner,” in org.eclipse.jdt.core.compiler, Dec. 2023, Available from: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fcompiler%2Fpackage-summary.html

[22] “Christian Borgelt’s Web Pages,”, Nov. 2022, Available from: https://borgelt.net/fpgrowth.html

[23] JavaParser.org, “Tools for your Java code,” 2019, Available from: https://javaparser.org

[24] T. Parr, “Download ANTLR”, Sept. 2023, Available from: https:// www.antlr.org/download.html

[25] OpenAI, “Introducing ChatGPT,” Nov. 2022, Available from: https://openai.com/blog/chatgpt

[26] J. Horsey “ChatGPT vs Bing vs Bard vs Claude comparison which ones right for you?” Aug. 2023, Available from: https://www.geeky-gadgets.com/chatgpt-vs-bing-vs-bard-vs-claude/

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org