Lightweight Sample Code Recommendation System to Support Programming Education

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 to learn programming techniques is to refer to sample programs. As the number of sample programs increases, however, it becomes difficult and time-consuming to find appropriate sample code visually. To overcome this shortcoming, research and development of program recommendation systems have been actively conducted. This paper discusses a recommendation system for Java sample programs using an unsupervised machine learning technique. The proposed system includes three major steps: (1) extracting invoked methods used in each sample program, (2) clustering the sample programs by applying a data mining technique to the extracted methods, and (3) ranking the programs by calculating a weighted average of the extracted methods. Experiments using file input and output sample programs indicate that the proposed system has sufficient potential to support programming education.**

Keywords—Recommendation System for Software Engineering; Mining Software Repository; Maximal Frequent Itemset; Tf-idf; Unsupervised Machine Learning; Programming Education.

I. INTRODUCTION

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

Recommendation systems are generally 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 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) [1].

Gasparic and Janes [2] survey 46 research and development articles on RSSE published between 2003 and 2013, and categorize them with respect to the covered data and the methods for recommendation. The most common type of covered data is source code with 21 papers, followed by help information to perform changes with 6 papers. As for ranking method, list format is the most common with 33 papers, followed by document format with three papers, and table format with two papers.

Hsu and Lin [3] 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* is applied to recommend API usage patterns.

Katirtzis, Diamantopoulos, and Sutton [4] 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 algorithm. Then, code slice techniques are applied to create a source code snippet.

Diamantopoulos and Symeonidis [5] 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.

Hora [6] 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 in data analysis, while readability and reusability are calculated using indicators developed in software engineering studies.

Nguyen, Rocco, Sipio, Ruscio, and Penta [7] 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 Term Frequency-Inverse Document Frequency *(tf-idf)* [8] and ranks API usage patterns using a collaborative filtering technique [9].

This paper discusses a lightweight recommendation system for analyzing Java sample programs that are collected from the Internet. The system clusters Java sample programs based on the methods that are invoked by the programs so that each cluster represents a programming subject. The system ranks the sample programs using a *tf-idf* weighted vector space model for each clustering. Since the higher ranked samples contain more invoked methods than those ranked lower, this system assists a student in selecting sample code suitable for learning.

The contributions of this study are as follows:

I. In general, method call patterns differ from one programming subject to another. This system can automatically cluster sample programs by programming subjects and represent them to students.

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. Therefore, a sample program can belong to multiple clusters, and a cluster only contains related programs.

III. By modifying *tf-idf* to give greater weights to the methods that frequently appear in a cluster, sample programs that fit the subject of a cluster and invoke many rare methods are ranked higher.

IV. The proposed system employs unsupervised machine learning, making it lightweight to use, operate and maintain the system.

The remainder of the paper is organized as follows. Section II gives the architecture of the proposed system. Section III describes the implementation of the main functions of the proposed system. Section IV shows the experimental results using file I/O sample programs. Section V discusses other implementation options. Section VI concludes the paper with our plans for future work.

II. OVERVIEW OF PROPOSED SYSTEM

This section describes the architecture of the proposed system from the functional point of view, and outlines typical usage.

A. Code Analyzer

Figure 1 depicts the architecture of the proposed system. The input for this system is sample programs available on the Internet. Currently, sample programs are collected manually and stored in a specific project typically in *Eclipse*, an Integrated Development Environment (IDE) for Java [10]. In this study, we assume that all sample programs are correct and work properly.

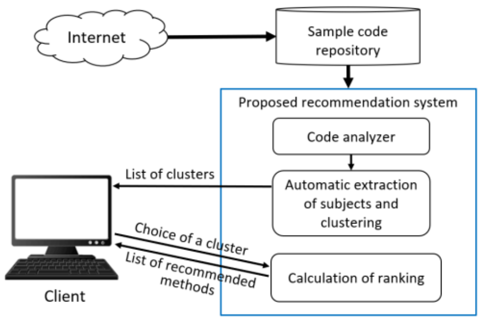
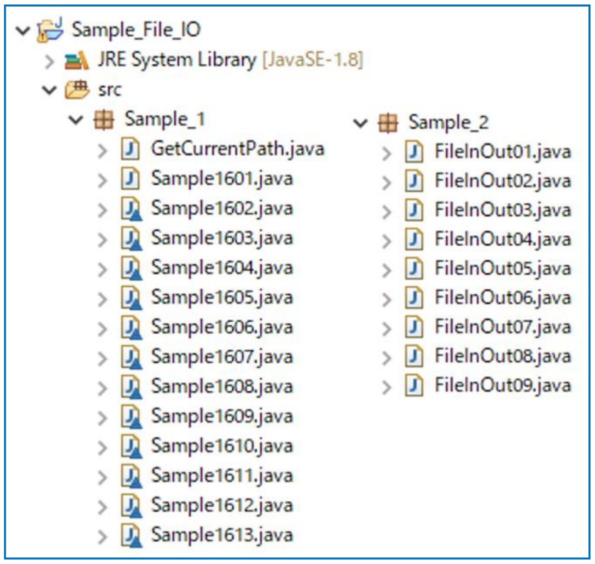


Figure 1. Overview of the proposed system.

Figure 2 shows a set of sample programs used in this study, which is stored in a project named *Sample_File_IO* in *Eclipse*. The sample program can be stored in packages. There is no limitation to the depth of the package hierarchy. As discussed later sections, these programs are concerned with binary and string file I/O. Programs can be stored in any directory other than an *Eclipse* project.



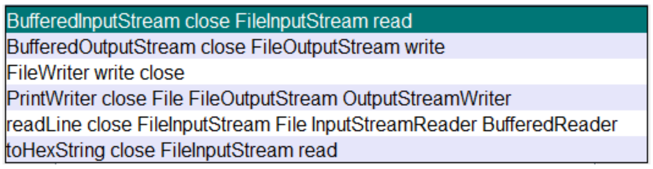
```
Sample_File_IO
  JRE System Library [JavaSE-1.8]
  src
    Sample_1                      Sample_2
      GetCurrentPath.java           FileInOut01.java
      Sample1601.java               FileInOut02.java
      Sample1602.java               FileInOut03.java
      Sample1603.java               FileInOut04.java
      Sample1604.java               FileInOut05.java
      Sample1605.java               FileInOut06.java
      Sample1606.java               FileInOut07.java
      Sample1607.java               FileInOut08.java
      Sample1608.java               FileInOut09.java
      Sample1609.java
      Sample1610.java
      Sample1611.java
      Sample1612.java
      Sample1613.java
```

Figure 2. Sample programs stored in *Eclipse* project.

The *code analyzer* in Figure 1 extracts method declarations and invoked method names from all Java files under the specified directory or project. A list of method names being invoked is used for clustering the declared methods and ranking them.

B. Automatic Identification of Subjects and Clusters

Following code analysis, the *Apriori* algorithm [11] is started to identify the set of invoked methods that occur frequently. Based on the frequent method set, programming subjects are automatically identified. Each subject corresponds to a cluster. Figure 3 shows an example of identified clusters.



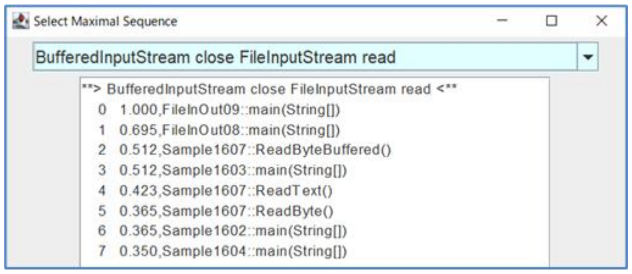
BufferedInputStream close FileInputStream read
BufferedOutputStream close FileOutputStream write
FileWriter write close
PrintWriter close File FileOutputStream OutputStreamWriter
readLine close FileInputStream File InputStreamReader BufferedReader
toHexString close FileInputStream read

Figure 3. Identified programming subjects and clusters.

This study uses a soft-clustering technique based on a maximal frequent itemset [12], i.e., a compact itemset that represents a frequent itemset. Strictly, the method names displayed in each cell of the combo box in Figure 3 are elements of a maximal frequent itemset. For example, “*BufferedInputStream close FileInputStream read*” suggests from the method names that the cluster is related to the subject of reading binary data.

C. *Calculation of Recommended Ranking*

Selecting a cell in the combo box in Figure 3 causes to specify a cluster of methods, which starts calculations of recommendation values for each of the declared methods in the cluster. Figure 4 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.



```
Select Maximal Sequence
BufferedInputStream close FileInputStream read
**> BufferedInputStream close FileInputStream read <**
 0  1.000,FileInOut09::main(String[])
 1  0.695,FileInOut08::main(String[])
 2  0.512,Sample1607::ReadByteBuffered()
 3  0.512,Sample1603::main(String[])
 4  0.423,Sample1607::ReadText()
 5  0.365,Sample1607::ReadByte()
 6  0.365,Sample1602::main(String[])
 7  0.350,Sample1604::main(String[])
```

Figure 4. Sample of program recommendation.

Method names are prefixed with class names, so that a student can easily check method source code using an IDE, such as *Eclipse, NetBeans and IntelliJ IDEA*.

III. IMPLEMENTATION

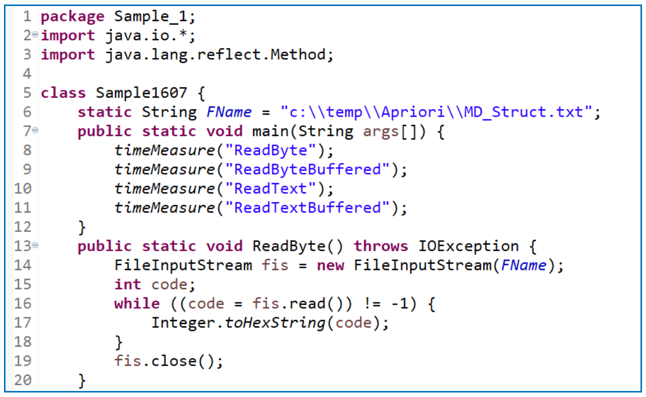
This section describes the implementation of three major steps of the proposed system. Those steps are code analysis, clustering, and ranking.

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 program. In this study, we make the assumption 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 methods. We implemented the code analyzer using the *Scanner* class [13], a tokenizer in *Eclipse* Java Development Tools (JDT) core. This class provides the ability to classify the tokens in a Java program into more than 100 types, and excludes comments allowing efficient analysis of executable statements. The class is widely used in *Eclipse* for navigating Java programs, including a class-method hierarchy.

Figure 5 shows a sample of a Java program. Figure 6 shows a list of declared methods and invoked ones that are generated from the Java program. A method with the same name is usually invoked multiple times in a declared method. Therefore, the analyzer extracts the method name and the number of times invoked, which are used for calculating cosine similarity. For example, the *main()* method in the *Sample1607* class in Figure 5 invokes the *timeMeasure()* method four times.



```java
package Sample_1;
import java.io.*;
import java.lang.reflect.Method;

class Sample1607 {
    static String FName = "c:\\temp\\Apriori\\MD_Struct.txt";
    public static void main(String args[]) {
        timeMeasure("ReadByte");
        timeMeasure("ReadByteBuffered");
        timeMeasure("ReadText");
        timeMeasure("ReadTextBuffered");
    }
    public static void ReadByte() throws IOException {
        FileInputStream fis = new FileInputStream(FName);
        int code;
        while ((code = fis.read()) != -1) {
            Integer.toHexString(code);
        }
        fis.close();
    }
```

Figure 5. Sample Java programs.

```
Sample1607::main(String[])
   timeMeasure,4
Sample1607::ReadByte()
   FileInputStream,1
   close,1
   read,1
   toHexString,1
```

Figure 6. Extracted method names and the number of times invoked.

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 and fail to characterize the function of a declared method.

B. *Apriori Algorithm for Identifying Subjects and Clusters*

Apriori algorithm proposed by Agrawal and Srikant [11] starts by identifying the frequent individual items and extending them to larger itemset as long as those itemset frequently appear in the database under consideration.

Let a database *D* be a set of transactions *t*, i.e., $D= \{t_1, t_2, \ldots, t_n\}$. Let 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.

A support value of an itemset refers to the number of transactions that contain the itemset. In terms of *D* and t_i, the support value of an itemset *X* is defined by the following formula:

$$Support(X) = | \{ t_i \in D : X \subseteq t_i \;\&\; 1 \le i \le n \} | \tag{1}$$

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 *k*-frequent item sets are utilized to identify *k+1* frequent item sets.

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 for a particular database. One such approach is to use a maximal frequent itemset [12].

Definition:

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

By definition, all frequent itemset can be derived from the set of maximal itemset. Table I shows an example of a database consisting of five transactions of itemset. Figure 8 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 [12]. *MinSup* is set to 1 or 20% (= 1/5*100).

TABLE I. EXAMPLE OF DATABASE

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

In Figure 7, the nodes surrounded by solid lines indicate the frequent itemset, while the nodes with yellow backgrounds indicate the maximal frequent itemset.

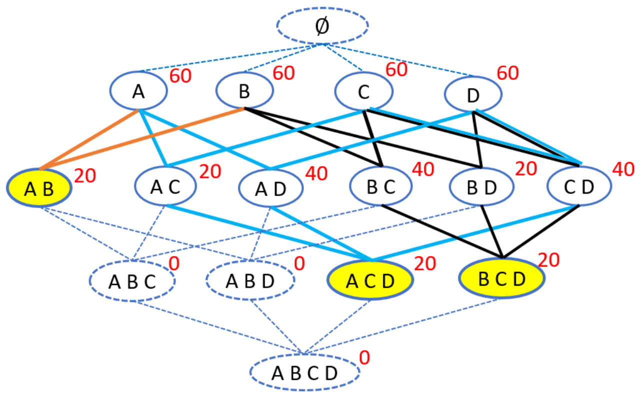


Figure 7. Maximal frequent itemset in lattice structure.

The frequent itemset can be soft-clustered by the maximal frequent itemset. For example, the subsets of {A, C, D} are {A, C}, {A, D}, {C, D}, {A}, {C}, {D}. The subsets of {A, B} are {A}, {B}. Analogously, the subsets generated from {B, C, D} are {B, C}, {B, D}, {C, D}, {B}, {C}, {D}. These subsets formulate three soft-clusters sharing the subsets, such as {A}, {B}, {C}, {D}.

Table II shows the number of the frequent itemset, the maximal frequent itemset, and the compression ratio of when the minimum support *minSup* varies. The experiment is performed using 33 methods declared in 23 Java files. The number of unique invoked methods is 36.

TABLE II. NUMBER OF ELEMENTS IN ITEMSET

MinSup	No. of frequent itemset	No. of maximal frequent itemset	Ratio (%)
6%	236	11	4.7
8%	236	11	4.7
10%	131	7	5.3
12%	127	6	4.7
14%	127	6	4.7
16%	29	7	24.1
18%	25	6	24.0
20%	25	6	24.0
22%	15	3	20.0
24%	13	4	30.8
26%	13	4	30.8
28%	11	5	45.5
30%	11	5	45.5
32%	11	5	45.5
34%	10	5	50.0
36%	3	1	33.3

For example, when *minSup* is 12%, the number of frequent itemset is 127 and the number of maximal frequent itemset is 6. The number of elements compresses to 4.7%. The maximal frequent itemset headlines the frequent itemset and defines the cluster.

C. Clustering Methods Using Maximal Frequent Itemset

More than ten binary programs that implement the *Apriori* algorithm are available on the web page maintained by Borgelt [14]. 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 in a separate process. The input data for this program is the set of invoked methods for each declared method shown in Figure 6, ignoring the number of invoked methods citations.

Figure 8 shows the maximal frequent itemset obtained from the sample program shown in Figure 2, with a *minSup* of 11%. The maximal frequent itemset corresponds to the programming subjects and is shown in Figure 3 as well.

```
0 :: BufferedInputStream close FileInputStream read
1 :: BufferedOutputStream close FileOutputStream write
2 :: FileWriter write close
3 :: PrintWriter close File FileOutputStream OutputStreamWriter
4 :: readLine close File FileInputStream InputStreamReader BufferedReader
5 :: toHexString close FileInputStream read
```

Figure 8. Example of generated maximal frequent itemset.

Figure 9 shows a list of declared methods that contain at least two invoked method names that are elements of a maximal frequent itemset. These clusters are broadly classified into two categories, i.e., those related to reading files and those related to writing files.

```
0  BufferedInputStream close FileInputStream read
   FileInOut08::main(String[])
   FileInOut09::main(String[])
   Sample1602::main(String[])
   Sample1603::main(String[])
   Sample1604::main(String[])
   Sample1607::ReadByte()
   Sample1607::ReadByteBuffered()
   Sample1607::ReadText()
1  BufferedOutputStream close FileOutputStream write
   FileInOut07::main(String[])
   FileInOut09::main(String[])
   Sample1608::main(String[])
   Sample1609::main(String[])
   Sample1610::main(String[])
   Sample1613::byteBufferedWrite()
   Sample1613::byteWrite()
2  FileWriter write close
   Sample1611::main(String[])
   Sample1612::main(String[])
   Sample1613::textBufferedWrite()
   Sample1613::textWrite()
3  PrintWriter close File FileOutputStream OutputStreamWriter
   FileInOut01::main(String[])
   FileInOut02::main(String[])
   FileInOut03::main(String[])
   FileInOut06::main(String[])
   FileInOut07::main(String[])
   FileInOut09::main(String[])
   Sample1610::main(String[])
4  readLine close File FileInputStream InputStreamReader BufferedReader
   CountUniqueMathod::main(String[])
   FileInOut04::main(String[])
   FileInOut05::main(String[])
   FileInOut06::main(String[])
   FileInOut08::main(String[])
   FileInOut09::main(String[])
   GetCurrentPath::main(String[])
   Sample1604::main(String[])
   Sample1607::ReadText()
5  toHexString close FileInputStream read
   FileInOut08::main(String[])
   FileInOut09::main(String[])
   Sample1602::main(String[])
   Sample1603::main(String[])
   Sample1604::main(String[])
   Sample1607::ReadByte()
   Sample1607::ReadByteBuffered()
   Sample1607::ReadText()
```

Figure 9. Methods belonging to each cluster.

Due to soft-clustering, one method belongs to multiple clusters. For example, *Sample1607::ReadByte()* is included in clusters 0 and 5, and *Sample1607::ReadText()* is included in clusters 0, 4 and 5.

D. Calculation of Recommendation Ranking

1) Definition of tf-idf

The Term Frequency-Inverse Document Frequency *(tf-idf)* weight [8] is one that commonly used in information retrieval. 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 that reflects how rare or important 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*.

Idf_i is defined as $log(N/DF_i)$, where *N* is the total number of declared methods that occur in a set 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. The maximal frequent itemset is displayed on the combo box in the GUI as shown in Figure 3. The proposed system identifies a set of declared methods when a user selects a cell on the combo box, and then starts to compute *tf* and *idf* for the set of declared methods. Table III lists the *tf* and *idf* values of the invoked method names corresponding to the maximal frequent itemset *{BufferedInputStream, close, FileInputStream, read}* that is shown at the top of Figure 9. There are 19 invoked methods in the sample programs related to the maximal frequent itemset.

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

No.	Invoked method name	Tf	Idf
0	BufferedInputStream	4	0.574
1	BufferedOutputStream	1	1.176
2	BufferedReader	6	0.398
3	File	6	0.398
4	FileInputStream	12	0.097
5	FileOutputStream	2	0.875
6	FileReader	3	0.699
7	InputStreamReader	6	0.398
8	OutputStreamWriter	1	1.176
9	PrintWriter	1	1.176
10	close	15	0
11	flush	1	1.176
12	format	1	1.176
13	length	1	1.176
14	read	11	0.135
15	readLine	4	0.574
16	toChars	2	0.875
17	toHexString	4	0.574
18	write	1	1.176

Since the proposed system uses clustering based on a maximal frequent itemset, the method names that are included in the Maximal Frequent Itemset (MFI) 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 *MFI* be the maximal frequent itemset specified by a user and idf_{max} be the maximum *idf* values.

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

Table IV shows the adjusted *idf* values for the maximal frequent itemset *{BufferedInputStream, close, FileInputStream, read}*.

TABLE IV. ADJUSTED *IDF* VALUES

Invoked method name	Tf	Idf	Adjusted idf
BufferedInputStream	4	0.574	1.75
FileInputStream	12	0.097	1.273
close	15	0	1.176
read	11	0.135	1.311

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

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

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 related to the maximal frequent itemset *{BufferedInputStream, close, FileInputStream, read}*.

TABLE V. DEGREES OF RECOMMENDATION FOR SAMPLE PROGRAMS

No.	DegR	Name of sample program
0	8.26	FileInOut08::main(String[])
1	11.885	FileInOut09::main(String[])
2	4.334	Sample1602::main(String[])
3	6.084	Sample1603::main(String[])
4	4.158	Sample1604::main(String[])
5	4.334	Sample1607::ReadByte()
6	6.084	Sample1607::ReadByteBuffered()
7	5.033	Sample1607::ReadText()

The maximal degree of recommendation is normalized to be 1 and displayed in the GUI. For the lists in Table 3, the normalized degrees of recommendation are obtained by dividing all the degrees by 11.885. This calculation generates the final list of recommendations shown in Figure 4.

Performance is measured 10 times for the following two processes that comprise this system. Both include the time displayed in the GUI.

(1) From the start of parsing to the end of clustering: average 360.8ms, standard deviation 10.5ms

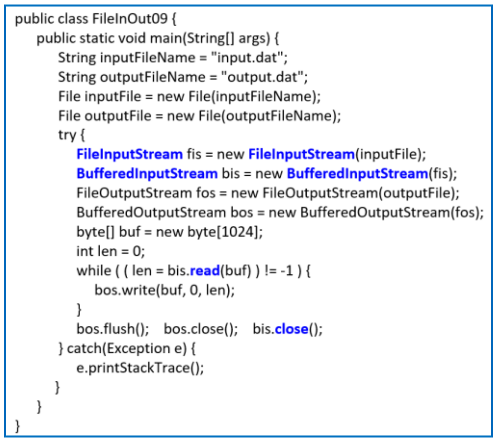
(2) After specifying a cluster to generating a list of recommendations: average 128.6ms, standard deviation 5.95ms

The specifications of a PC used are as follows:

CPU: AMD Ryzen 7 5700U (Laptop PC)
RAM: 16.0 GB
OS: Windows 10 Home 64 bit.

IV. EXPERIMENTAL RESULTS

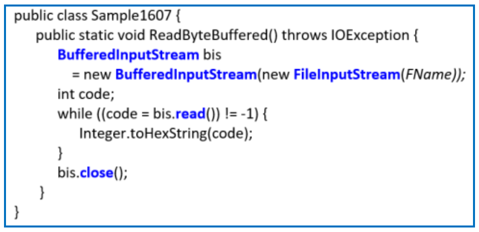
This section describes experimental results. Figure 10 shows the sample program or the declared method that corresponds to the top of the recommended list in Figure 4 with a normalized recommendation value of 1.000. The sample program includes all of the invoked methods that constitute the maximal frequent itemset *{BufferedInputStream, close, FileInputStream, read}*. In addition, it contains essential methods for binary file outputs, e.g., *FileOutputStream(), write()*.



```
public class FileInOut09 {
    public static void main(String[] args) {
        String inputFileName = "input.dat";
        String outputFileName = "output.dat";
        File inputFile = new File(inputFileName);
        File outputFile = new File(outputFileName);
        try {
            FileInputStream fis = new FileInputStream(inputFile);
            BufferedInputStream bis = new BufferedInputStream(fis);
            FileOutputStream fos = new FileOutputStream(outputFile);
            BufferedOutputStream bos = new BufferedOutputStream(fos);
            byte[] buf = new byte[1024];
            int len = 0;
            while ( ( len = bis.read(buf) ) != -1 ) {
                bos.write(buf, 0, len);
            }
            bos.flush();    bos.close();    bis.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 10. Sample program with recommendation value 1.000.

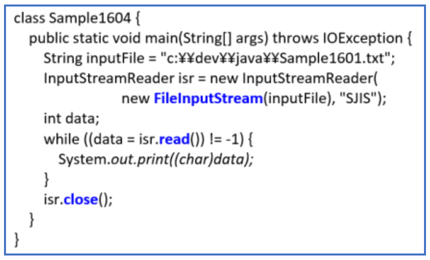
Figure 11 shows a sample program with a normalized recommendation value of 0.512. It contains all four method names that constitute the maximal frequency itemset and another method, i.e., *toHexString()*. Compared to the sample program in Figure 10, this provides a concise one.



```
public class Sample1607 {
    public static void ReadByteBuffered() throws IOException {
        BufferedInputStream bis
            = new BufferedInputStream(new FileInputStream(FName));
        int code;
        while ((code = bis.read()) != -1) {
            Integer.toHexString(code);
        }
        bis.close();
    }
}
```

Figure 11. Sample program with recommendation value 0.512.

Figure 12 shows a sample program with a normalized recommendation value of 0.350. It includes three method names of the maximal frequency itemset and another method, i.e., *InputStreamReader()*.



```
class Sample1604 {
    public static void main(String[] args) throws IOException {
        String inputFile = "c:¥¥dev¥¥java¥¥Sample1601.txt";
        InputStreamReader isr = new InputStreamReader(
                    new FileInputStream(inputFile), "SJIS");
        int data;
        while ((data = isr.read()) != -1) {
            System.out.print((char)data);
        }
        isr.close();
    }
}
```

Figure 12. Sample program with recommendation value 0.350.

Compared to the sample programs in Figures 10 and 11, Figure 12 shows the most concise program regarding a method usage for file read operations. These results demonstrate that the proposed system works as expected.

V. Discussion

A. Syntax Analysis

In this study, the *Scanner* [13] class is used for parsing sample programs mainly because it reduces development effort. There are several options of parsing tools, including *JavaParser* [15] and *ANTLR* [16], both of which generate an Abstract Syntax Tree (AST). AST is an intermediate representation of a program's source code in a tree structure. “Traversing” an AST that would require a few hundred lines of programming allows applications to perform more complex operations than a mere method name extraction. ANTLR can parse formal languages other than Java. All parsing tools work independently of IDEs and can parse sample code stored in arbitrary directories.

B. ChatGPT

ChatGPT is a chat-based tool released by OpenAI in Nov 2022 [17]. The latest *ChatGPT Feb 13* version allows users to chat about Java sample code for File I/O successfully. However, the sample code is limited to what *ChatGPT* has already learned. Since a learning process is exclusively conducted by an OpenAI team, it is difficult for a lecturer to configure sample programs to fit her/his classes. The method proposed in this study allows the lecturer to compose sample programs tailored for a class, even if those programs are specific or even unusual.

VI. Conclusion and Future Work

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

It is confirmed through experiments using file I/O sample programs that declared methods including useful information on the programming subjects, such as read and write string/binary data, are ranked in higher position. This result indicates that the proposed recommendation system has sufficient potential to support programming education.

The *Apriori* algorithm employed in this study requires the minimum number of supports, i.e., *minSup*, to be specified in advance. The ability to automatically determine the optimal *minSup* is left as a topic for future research. Manual collection of sample programs is a drawback of this study. Sample code downloader is an issue for future development. Additional experiments on larger sample programs are planned to verify the effectiveness of the proposed recommendation system for programming education.

References

[1] 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

[2] 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

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

[4] N. Katirtzis, T. Diamantopoulos, and C. Sutton, “Summarizing Software API Usage Examples using Clustering Techniques,” Proc. of 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

[5] 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

[6] A. Hora, “APISonar: Mining API usage examples,” Wiley Online Library, Software: Practice and Experience Vol. 51, Issue 2, pp. 319-352, Oct. 2020, DOI: 10.1002/spe.2906

[7] 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

[8] G. Sidorov. “Vector Space Model for Texts and the tf-idf Measure,” In Syntactic n-grams in Computational Linguistics, pp.11-15, Apr. 2019, Springer, Cham, ISBN: 978-3-030-14770-9.

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

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

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

[12] J. Rousu, “Finding frequent itemsets - concepts and algorithms,” University of Helsinki, Available from: https://www.cs.helsinki.fi/group/bioinfo/teaching/dami_s10/dami_lecture4.pdf [retrieved: Apr. 2010]

[13] IBM Rational Software Architect, “Interface IScanner,” in org.eclipse.jdt.core.compiler, Available from: https://www.ibm.com/docs/ja/developer-for-zos/9.5.1?topic=SSQ2R2_9.5.1/org.eclipse.wst.jsdt.doc/reference/api/org/eclipse/wst/jsdt/core/compiler/IScanner.htm [retrieved: Mar. 2021]

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

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

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

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