Javawock: A Java Class Recommender System Based on Collaborative Filtering

Masateru Tsunoda Takeshi Kakimoto Naoki Ohsugi
Akito Monden Ken-ichi Matsumoto
Graduate School of Information Science,
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
{masate-t, takesi-k, naoki-o, akito-m, matumoto}@is.naist.jp

ABSTRACT - Many software development platforms provide a large number of library components to make it easy to build high quality software. On the other hand, it became more and more difficult for developers to find useful components in each development context because the amount of components provided became too large today. This paper proposes a recommender system that provides useful Java components (library class files) to a developer based on *collaborative filtering (CF)*. When a developer gives an unfinished Java program to the system, it investigates Java library class files used in the given program and finds Java programs that are similar to the given program from a program repository. Then, the system recommends to the developer Java library class files that were used in the similar programs but were not used in the developer's program. An experimental evaluation showed that the recommendation accuracy of the proposed system was much higher than that of a naïve (non-CF) method in all four evaluation criteria (recall, precision, F1 value, and half-life utility).

KEYWORDS - information retrieval, cosine similarity, software component, recommender system, J2SE

1. INTRODUCTION

Today's software development platforms provide various types of library components to satisfy varying developers' demands and needs. Such platforms enable developers to build software that has rich features in shorter development periods and lower costs [9]. For instance, Java 2 SDK, Standard Edition (J2SE) Version 1.4.1_02 provides 5568 classes and interfaces as library components.

In spite of a large library provided, most developers use only a small fraction of the library. For instance, Fig. 1 shows the number of library classes actually used in four famous open source products. Each product used 224 library classes on average. Since this is only 4% of whole J2SE library classes, there may be unaware library classes that can improve developer's productivity or software quality. However, it is quite difficult for developers to find useful library classes just because the library itself is too large. We need a system that makes it easy to find useful library class files in each development context.

This paper proposes *Javawock*, a Java class recommender system based on collaborative filtering (CF). CF is considered a powerful information filtering method, and has been used in *recommender systems* that estimate end-users' preferable items (books, movies, tunes, etc). Typically such system determines items to be recommended in the following way. First, it collects ratings of items from many users. A set of ratings of each user is called a *preference*. Next, when a target user was specified, the system chooses similar users in terms of their preferences. Finally, it determines recommended items by using preferences of similar users. Usually, items that had high ratings in the similar users' preferences are recommended if the items are not used (rated) by the target user.

Javawock uses the idea of CF by replacing users with Java programs and items with Java library class files. Then, a set of library class files used in each program is regarded as a preference of the program. Javawock makes a recommendation of library class files as follows. First, it collects preferences of programs by investigating used library class files in each program; then, when a target program was given, it chooses similar programs in terms of their preferences; finally, it recommends library class files by preferences of similar programs.

In what follows, Section 2 introduces related works. Section 3 explains the detail of CF. Section 4 explains recommendation procedure and algorithms of Javawock. Section 5 reports an experiment to evaluate the recommendation accuracy of Javawock. In the end, Section 6 concludes the paper with a summary and some future topics.

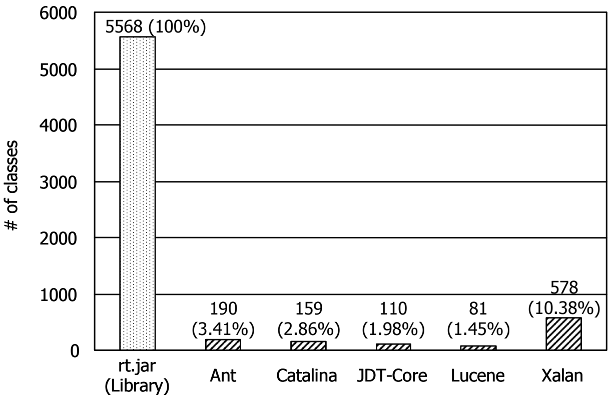


Fig. 1. The number of library classes actually used in each product

2. RELATED WORK

Inoue et al. [4] proposed a Java source code retrieval system SPARS-J. Although SPARS-J is not a recommender system, it can help developers to find useful Java classes based on keyword search. SPARS-J analyzes Java source programs (*.java files) for a keyword indexing, and it stores both program files and indexed keywords to a repository beforehand. SPARS-J provides developer a web-based interface including query edit box to enter keywords. The developer can search program files from the repository with keywords related to program's features, algorithms or authors. It is intended to be used as "Google" to find Java source files instead of HTML texts. One drawback of this approach is that it cannot output relevant programs unless the developer can enter appropriate keywords expressing what the developer wants to search. Generally, it is very difficult to choose appropriate keywords for programs that are unimaginable to the developer. On the other hand, Javawock can recommend Java class files, regardless of whether or not they are imaginable to the developer, without giving any keyword since it requires only an unfinished Java program written by the developer as an input.

Ye and Fischer [11] proposed CodeBroker, a non-CF based recommender system. It recommends a developer useful Java components (library packages and library class files) related to the developer's current task. CodeBroker automatically extracts keywords from comments in program source codes to capture the development context. Then, CodeBroker uses these keywords as a query to retrieve components, and recommends them to the developer. Also, CodeBroker recognizes already used components to avoid recommending components already be known by the developer. The developer does not have to enter any keywords to get recommendation because CodeBroker automatically gathers keywords from source code. One drawback of this approach is that precision (ratio of appropriately recommended items in all recommended items) greatly depends on quality of comments written in source code. It requires fully-commented source code stored in the program repository. In our approach of Javawock, only class files (without source code) are required to be stored in the program repository.

McCarey et al. [5] proposed RASCAL, a CF-based software component recommender system, which employs a similar approach to ours. Their system makes a recommendation of Java methods in the following way. First, the system counts the number of invocations of each method in a Java class file written by the developer, as a preference of the class file. Next, the system finds class files similar to the developer's file from a program repository containing other developers' Java class files. Then, the system recommends methods that were invoked in the similar class files but were not invoked in the developer's class file. There are several differences between Javawock and their system. First, Javawock recommends Java class files while their system recommends Java methods. Their method recommendation is useful if there are too many methods in a class. However, since most of library classes have not so many methods, we put our priority on class recommendation with better accuracy. To gain high accuracy, Javawock counts only *well-known* classes (e.g. classes in J2SDK and Jakarta project) used in each program to capture the preference of the program, while RASCAL counts methods of all the classes including locally-developed classes used in each program. This difference affects the similarity computation of programs. Our approach is based on the idea that a set of well-known classes used in a program is considered a native characteristic (birthmark) of the program [10], while local classes can be easily replaced or changed to other classes. Second, we employ an *item-based* CF algorithm as well as a *user-based* CF algorithm, while RASCAL uses only user-based one. Generally, item-based algorithms are much more useful than user-based ones in practical setting with a large repository. It is because only item-based ones allow recommender systems to *cache* the result of similarity computation (For this reason, Amazon.com uses an item-based algorithm in their book recommender system). Third, we used more criteria to evaluate the recommendation accuracy.

3. COLLABORATIVE FILTERING

CF is one of the key techniques for implementing a recommender system that recommends to a user a set of candidate items which may be preferable or useful to the user. For example, Resnick et al. [6] developed GroupLens system which recommends interesting Usenet articles to users. It draws on a simple idea; people who agreed in their subjective evaluation of past articles are likely to agree again in the future. Resnick et al. proposed a basic CF algorithm known as a user-based method.

User-based method makes recommendations with the following procedure:

1. After using items (Usenet articles, books, movies, etc.), users explicitly assign numeric ratings to the items.
2. A recommender system correlates the ratings in order to determine which user's ratings are the most similar to other ones.
3. The system predicts ratings of new items for the target user, based on the ratings of similar users.
4. If these new items seem to be preferred, the system recommends them to the user.

Sarwar et al. [8] proposed another basic CF algorithm called item-based method. Item-based method can make recommendation with extremely sparse dataset whose ratio of rated items to whole items is only 1%. Item-based method makes recommendations with executing 2' and 3' instead of 2 and 3 in the above procedure respectively:

2'. A recommender system correlates the ratings in order to determine *which item's ratings are most similar to other item's.*

3'. The system predicts ratings of new items for the target user, based on the ratings of *similar items* already

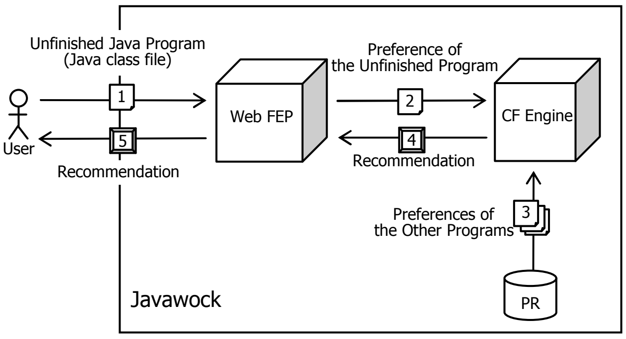


Fig. 2. The architecture of Javawock

	l_1	l_2	$\cdots$	l_j	$\cdots$	l_b	$\cdots$	l_n
p_1	$u_{1,1}$	$u_{1,2}$	$\cdots$	$u_{1,j}$	$\cdots$	$u_{1,b}$	$\cdots$	$u_{1,n}$
p_2	$u_{2,1}$	$u_{2,2}$	$\cdots$	$u_{2,j}$	$\cdots$	$u_{2,b}$	$\cdots$	$u_{2,n}$
$\cdots$	$\cdots$	$\cdots$		$\cdots$		$\cdots$		$\cdots$
p_i	$u_{i,1}$	$u_{i,2}$	$\cdots$	$u_{i,j}$	$\cdots$	$u_{i,b}$	$\cdots$	$u_{i,n}$
$\cdots$	$\cdots$	$\cdots$		$\cdots$		$\cdots$		$\cdots$
p_a	$u_{a,1}$	$u_{a,2}$	$\cdots$	$u_{a,j}$	$\cdots$	$u_{a,b}$	$\cdots$	$u_{a,n}$
$\cdots$	$\cdots$	$\cdots$		$\cdots$		$\cdots$		$\cdots$
p_m	$u_{m,1}$	$u_{m,2}$	$\cdots$	$u_{m,j}$	$\cdots$	$u_{m,b}$	$\cdots$	$u_{m,n}$

Fig. 3. *m*×*n* table used for recommendation

rated by the users.

Intuitively, an idea of the item-based method can be represented as a popular sentence of Amazon.com's recommendations "Customers who bought this book also bought ...".

Javawock assumes each Java program (a class file written by a developer) as a user, and assumes each Java library class file used in the program as an item (Note that a class file can become both the user and the item since the class file itself uses a library class file). Javawock analyzes which Java library class files were used in each developer's class file. Then, Javawock puts high ratings to library classes used in the developer's class file, and low rating to library classes unused in it. We implemented both user- and item-based method as Javawock's CF algorithms. With the user-based method, Javawock predicts which library class files will be used by the developer, based on the similarity of how each class uses library classes. With item-based method, Javawock predicts which library class files will be used by each developer, based on the similarity of how each library classes are used by other classes.

4. RECOMMENDING JAVA CLASS FILES

4.1. Architecture of recommender System

Fig. 2 shows the architecture of Javawock. Javawock consists of a web-based frond-end processing (web FEP), a collaborative filtering engine (CF engine) and a program repository (PR). Javawock recommends Java library class files for a Java program uploaded by user. The uploaded Java program is a class file (not a source file), thus, the users must compile their Java program before uploading it.

Javawock recommends library class files as follows:

1. User uploads an unfinished Java program (target program) to the web FEP.
2. The web FEP analyzes the target program to get a set of library class files used in the given program (preference). To get the preference, the web FEP uses Used-Class analysis engine proposed in jbirth [10].
3. The web FEP sends the preference to the CF engine.
4. The CF engine receives the preference and makes a recommendation by using other programs' preferences stored in the PR.
5. The CF engine returns the recommendation result to the web FEP.
6. The web FEP shows a user the recommendation. The recommendation result consists of library class names, recommendation scores, abstracts of Java API documents and links (URLs) to them.

Fig. 4 shows an input screen that uploads class file, and Fig. 5 shows an output screen of the recommendation result. Javawock has a Google like interface. Each recommended library class is linked to the Java API document.

4.2. Recommendation methods

Javawock provides three types of recommendations. The first one employs a user-based CF method. With this method, Javawock makes recommendation based on the similarity between programs. The similar programs use similar library class files, compared with the target program. The user-based method proceeds as follows:

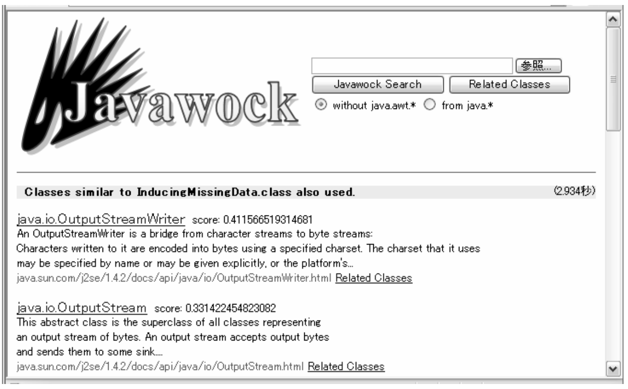
1. Javawock chooses several similar programs from PR.
2. Javawock computes a recommendation score of each library class file whom similar programs use and the target program does not use.
3. Javawock shows recommended library class files ranked by their score.

The second method employs an item-based CF method. Javawock makes recommendation based on the similarlity between library class files (similarity of how they are used). The class file l_0 and l_1 are considered similar each other if l_0 and l_1 are used in the same set of programs. The item-based method proceeds as follows:

1. Javawock chooses several similar library class files for each library class file.
2. Javawock computes a recommendation score of each library class file l_i if the target program does not use it but uses a library class file l_k that is similar to l_i.
3. Javawock shows recommended library class files ranked by their score.

The third method simply outputs Java class files that are similar to the given program based on the user-based

Fig. 4. The input screen of Javawock



Classes similar to InducingMissingData.class also used. (2.934秒)

java.io.OutputStreamWriter score: 0.411566519314681
An OutputStreamWriter is a bridge from character streams to byte streams:
Characters written to it are encoded into bytes using a specified charset. The charset that it uses
may be specified by name or may be given explicitly, or the platform's...
java.sun.com/j2se/1.4.2/docs/api/java/io/OutputStreamWriter.html Related Classes

java.io.OutputStream score: 0.331422454823082
This abstract class is the superclass of all classes representing
an output stream of bytes. An output stream accepts output bytes
and sends them to some sink....
java.sun.com/j2se/1.4.2/docs/api/java/io/OutputStream.html Related Classes

Fig. 5. The output screen of Javawock

similarity computation algorithm. (Note that we haven't experimentally evaluated this method in this paper).

4.3. Collaborative Filtering Algorithm for the Recommendation

The CF engine computes the similarity and the recommendation score to make a recommendation. When making a recommendation, CF engine uses the program repository in form of $m \times n$ matrix as shown in   where $p_i \in \{p_1, p_2, ..., p_m\}$ denotes i-th program, $l_j \in \{l_1, l_2, ..., l_n\}$ denotes j-th library class file, and $u_{i,j} \in \{u_{1,1}, u_{1,2}, ..., u_{m,n}\}$ denotes status of whether library class file l_j is used or not by program p_i. If program p_i uses library class file l_j, the value of $u_{i,j}$ (library class status) is set to 1, and if program p_i does not use library class file l_j, the value of $u_{i,j}$ is set to 0. So there is no missing value in the matrix.

Our CF engine uses the cosine similarity algorithm to compute similarity. Similarity is computed to choose similar programs or similar library class files. Although various similarity computation algorithms are proposed for CF [1][2][6][8], we selected the cosine similarity algorithm because it showed the highest accuracy in our pilot experiment. This algorithm was originally proposed to evaluate the similarity between two documents in the field of information retrieval. The similarity is often evaluated by treating each document as a vector of word frequencies and computing the cosine of the angle formed by the two frequency vectors [7].

On the user-based method, the similarity, $sim(p_a, p_i)$ between the target program p_a and other program p_i is formally defined as the following (1). In this equation, programs, library class files and library class statuses are used instead of documents, words and word frequencies. The value range of $sim(p_a, p_i)$ is [0, 1].

$$sim(p_a, p_i) = \frac{\sum u_{a,j} \times u_{i,j}}{\sqrt{\sum (u_{a,j})^2}\sqrt{\sum (u_{i,j})^2}} \tag{1}$$

In the item-based method, similarity $sim(l_b, l_j)$ between the library class file l_b and other library class file l_j is formally defined as (2).

$$sim(l_b, l_j) = \frac{\sum u_{i,b} \times u_{i,j}}{\sqrt{\sum (u_{i,b})^2}\sqrt{\sum (u_{i,j})^2}} \tag{2}$$

Our CF engine uses the weighed sum algorithm [8] to compute the recommendation score. The recommendation score is a predicted value of $u_{a,j}$ owned by program p_a. The CF engine predicts $u_{a,j}$ of each library class file which is not used by the target program. Although various recommendation score computation algorithms are proposed for CF [1][2][6][8], we selected the weighed sum algorithm because it showed the highest accuracy in our pilot experiment.

When computing a score, the CF engine does not use all similar programs or all similar library class files, but uses k similar programs or k similar library class files. k is called *neighborhood size*.

In the user-based method, the recommendation score $R_{a,b}$, which is a predicted value of $u_{a,b}$ owned by program p_a is formally defined as (3). The recommendation score is computed with the weighed average of $u_{i,b}$ owned by similar programs p_i. Each weight is similarity between the target program p_a and each program p_i. *k-nearestPrograms* means a set of k similar programs.

$$R_{a,b} = \frac{\sum_{K \in k-nearestPrograms} sim(p_a, p_K) \times u_{K,b}}{\sum_{K \in k-nearestPrograms} sim(p_a, p_K)} \tag{3}$$

In the item-based method, the recommendation score $R_{a,b}$, which is a predicted value of $u_{a,b}$ owned by program p_a is formally defined as (4). The recommendation score is computed with the weighed average of $u_{a,j}$ owned by similar library class files l_j. Each weight is similarity between library class file l_b and other library class l_j. *k-nearestLibraryClassFiles* means a set of k similar library class files.

$$R_{a,b} = \frac{\sum_{K \in k-nearestLibraryClassFiles} sim(l_b, l_K) \times u_{a,K}}{\sum_{K \in k-nearestLibraryClassFiles} sim(l_b, l_K)} \tag{4}$$

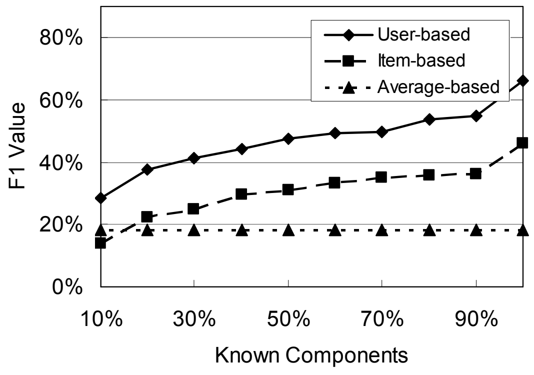


Fig. 6. F1 value of each method

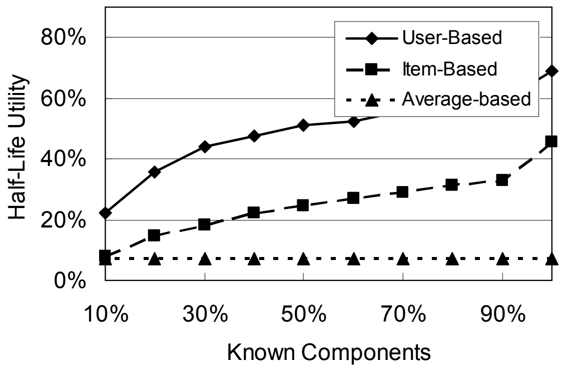


Fig. 7. Half-life utility of each method

5. EXPERIMENTAL EVALUATION

5.1. Dataset

For an experimental dataset, we extracted classes from rt.jar, class library of J2SE (Java 2 Platform Standard Edition) Software Development Kit. We selected 371 commonly used classes as programs $P=\{p_1, p_2, ..., p_m,\}$ and 331 classes as library class files $L=\{l_1, l_2, ..., l_n\}$, which is a subset of P. The number of library class files n is smaller than that of programs m because we excluded 40 library class files (from L) that were not used by any program in P, Consequently, we made 371×331 size dataset from them.

5.2. Experimental Procedure

In the experiment, we evaluated both the user-based method and the item-based method. The experiment proceeds as follows (leave-one-out cross-validation):

1. i-th program p_i is regarded as the target program, and it is removed from the dataset.
2. $u_{i,j}$ is regarded as unknown, and Javawock computes $R_{i,j}$, a recommendation score for library class file l_j (i.e. Javawock predicts the value of $u_{i,j}$).
3. Repeat Step 2 for all j.
4. Repeat Step 1, 2, 3 for all i.

In order to virtually produce an unfinished program of Pi, we also used a criterion q, denoted as *Known Component* in Fig 4,..,8, which is a percentage of library class files regarded to be already used in P_i to the total library class that will be used when P_i is finished. We varied q from 10% to 100% at 10% intervals by step-by-step adding actually used library class files in P_i. Note that at least one library class file is remain unknown even if q=100% (see Step 2).

We conducted pre-examination to define the neighborhood size k used in (3) and (4), and defined k to be 3 for both (3) and (4) since recommendation accuracy became the highest in this case.

We also computed recommendation scores using a naïve (non-CF) method called *average-based method* to compare with proposed methods. In the average-based method, the recommendation score $R_{a,b}$, predicted value of $u_{a,b}$ owned by program p_a is formally defined as (5), where N_c is the number of entire programs, and N_l is the number of programs that uses library class file l_b.

$$R_{a,b} = \frac{N_l}{N_c} \tag{5}$$

5.3. Evaluation Criteria

We used four criteria (recall, precision, F1-value and half-life utility) to evaluate recommendation accuracy of the proposed methods. These are often used to evaluate accuracy of CF based system [3]. The higher these values are, the more accurate evaluated method is.

Precision is a ratio of appropriately recommended library class files to entire recommended library class files, formally defined as (6), where N_r is the number of entire recommended library class files, and N_a is the number of appropriately recommended library class files. That is, N_r

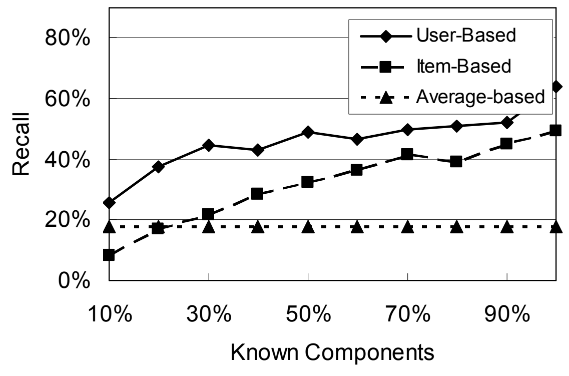


Fig. 8. Precision of each method

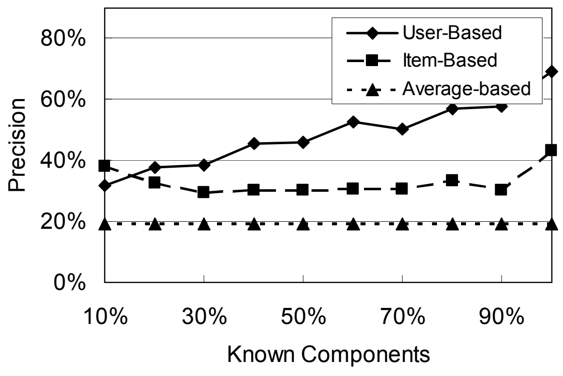


Fig. 9. Recall of each method

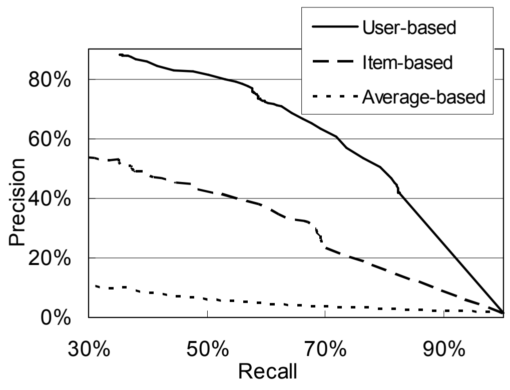


Fig. 10. The relation between recall and precision

is the number of scores that satisfy $R_{i,j}$=1, and N_a is the number of scores that satisfy both $R_{i,j}$=1 and $u_{i,j}$ (predicted values of $R_{i,j}$) =1.

$$Precision = \frac{N_a}{N_r} \tag{6}$$

Recall is a ratio of appropriately recommended library class files to entire library class files actually used in the program p_i, formally defined as (7), where N_u is the number of entire library class files actually used in p_i, and N_a is the number of appropriately recommended library class files. That is, N_u is the number of scores that satisfy $u_{i,j}$=1, and N_a is the number of scores that satisfy both $R_{i,j}$=1 and $u_{i,j}$=1.

$$Recall = \frac{N_a}{N_u} \tag{7}$$

F1 value is a combined criterion of recall and precision, formally defined as (8).

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{8}$$

Half-life utility (H) is a criterion to evaluate the ranking based on recommendation score, formally defined as (9) and (10), where $s_{a,d}$ is the predicted library class status of library class file l_j recommended to the target program p_a at d-th rank (That is, the value of $s_{a,d}$ is same as the predicted value of $u_{a,j}$). α is the rank of a recommended library class file whom users will view with a probability of 50%. We defined α to be 10. H_a^{max} is equal to H_a when the system makes perfect recommendation.

$$H_a = \sum_d \frac{s_{a,d}}{2^{(d-1)/(\alpha-1)}} \tag{9}$$

$$H = 100 \times \frac{\sum_a H_a}{\sum_a H_a^{\max}} \tag{10}$$

5.4. Experimental Result

Experimental result is shown in Fig. 6,…,Fig. 10. These graphs show that user-based method has good recommendation accuracy. Fig. 6 shows the relation between the highest F1 value of each method and the percentage of known components (library class files considered to be already used by the developer). We changed threshold to find the highest F1 value at each percentage of known components. The line of the average-based method in Fig. 6 is parallel to x-axis because average-based method is independent of p_i to make recommendation.

Fig. 6 shows that the user-based method is the most accurate of the three and the average-based method is the most inaccurate. In this figure, when known components ≈100%, i.e. programming is almost finished, F1 value of the user-based is 66%, that of the item-based is 46%, and that of the average-based is 18%. When q<20%, the item-based method is less accurate than the average-based method. This indicates that the developer needs to build at least 20% of the target program to get better recommendation when he/she wants to use the item-based method. Fig. 7,…, Fig. 9 show other criteria when F1 value is the highest. The trends in these graphs are similar to Fig. 6.

Fig. 10 shows the relation between recall and precision of each method (when q≈100%). The curve of the user-based method is always the highest of three methods and that of the average-based is always the lowest. From Fig. 6,…, Fig. 10, we conclude that the user-based method is always more accurate than the average-based method, and the item-based method is more accurate than the averaged-based method if q≥30%.

6. CONCLUSION

In this paper, we proposed Javawock, a Java class recommender system, based on CF. Javawock employs both user-based and item-based method to make recommendations. An experimental evaluation showed that the user-based method is always more accurate than the naïve (non-CF) average-based method, and the item-based method is more accurate than the averaged-based method if the percentage of known components≥30%.

The limitation of our experiment is that we used only one dataset for evaluation. We will conduct further experiment using other datasets to extensively evaluate the proposed method.

ACKNOWLEDGEMENTS

This work is supported by the EASE (Empirical Approach to Software Engineering) project of the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology of Japan, and Grant 15103 of the Open Competition for the Development of Innovative Technology program.

REFERENCES

1. J. Breese, D. Heckerman, and C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering," in *Proc. Conf. on Uncertainty in Artificial Intelligence*, Madison, WI, pp. 43-52, Jul. 1998.

2. D. Goldberg, D. Nichols, B.M. Oki, and D. Terry, "Using collaborative filtering to weave an information tapestry," *Communications of the ACM*, vol..35, no.12 pp. 61-70, Dec. 1992.
3. J. Herlocker, J. Konstan, L. Terveen, and J. Riedl, "Evaluating collaborative filtering recommender systems," *ACM Transactions on Information Systems (TOIS)*, vol.22 , no.1, pp.5-53, 2004.
4. K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, S. Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search," In *Proc. International Conference on Software Engineering*, pp.14-pp.24, Portland, Oregon, 2003.
5. F. McCarey, M. Ó Cinnéide, and N. Kushmerick, "RASCAL: A Recommender Agent for Software Components in an Agile Environment," In *Proc. Artificial Intelligence and Cognitive Science Conference*, Castlebar, Ireland, pp.107-pp.116, Sep. 2004.
6. P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "GroupLens: An Open Architecture for Collaborative Filtering of Netnews," In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, pp.175-pp.186, Oct. 1994.
7. G. Salton, and M. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, 1983.
8. B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-Based Collaborative Filtering Recommendation Algorithms," In *Proc. International World Wide Web Conference*, Hong Kong, China, pp. 285-295, May 2001.
9. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. New York: Addison-Wesley, 1998.
10. H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Design and evaluation of birthmarks for detecting theft of Java programs," In *Proc. IASTED International Conference on Software Engineering*, Innsbruck, Austria, pp.569-575, Feb. 2004.
11. Y. Ye, and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," In *Proc. International Conference on Software Engineering*, Orlando, FL, pp.513-523, May 2002.