A Case Study on Recommending Reusable Software Components using Collaborative Filtering

Frank McCarey, Mel Ó Cinnéide and Nicholas Kushmerick
Department of Computer Science,
University College Dublin,
Belfield, Dublin 4, Ireland.
{frank.mccarey, mel.ocinneide, nick} @ucd.ie

Abstract

The demand for quality, highly functional software reinforces the need for reusable software components. However, as repositories of reusable components increase in size and complexity, the challenge for developers to remain conversant with all components becomes greater. This paper proposes a software recommendation system based on collaborative filtering, which has been shown to be effective in other domains. Based on the usage patterns of existing classes and the class currently being developed, our system proposes a set of reuse candidates to the programmer. We present the results of our analysis of the usage of Swing classes in several open-source applications and find that the collaborative filtering technique is promising in providing recommendations in this context.

1. Introduction

It is of growing importance for enterprises to have effective reuse of software components as they invest in developing and maintaining large software systems [6]. Software reuse is an approach to developing systems where artifacts that already exist are used again. Artifacts vary from software components to analysis models; this paper concentrates exclusively on software components. A component is a well-defined unit of software that has a published interface and can be used in conjunction with other components to form larger units [9]. Using existing components can help develop better, faster and cheaper software systems in an industrial context, e.g. [8].

Developers are not always eager to use reusable components, even if these components may be useful and improve productivity. A Productivity Paradox, as identified in [5], exists. Although reusable components for solving this problem are available, most developers are not motivated to learn these reusable components. The reasons behind the lack of motivation are discussed in [13]. A developer will give preference to a suboptimal solution as they perceive the time and effort to locate and learn components to be too costly. Even if a developer is willing to reuse a component they may not be able to locate it. As the repositories of components increase, there exists a real challenge for developers to remain conversant with all components. To assist software developers in making full use of large component repositories, information access needs to be complemented by information delivery [14].

Software repositories contain a wealth of valuable information. Usage of software components can be automatically extracted from these repositories. This may then be used to infer links between components, examine usage patterns of a component or set of components and examine the relationship between the user task and components used.

This paper describes a recommendation system, based on a collaborative filtering approach, that allows developers discover reusable components for the purpose of supporting learning on demand, improving developer productivity/quality and promoting software reuse. Repositories of open-source Java code, available from *SourceForge* [4], are mined and usage histories of components are automatically collected. Based on the collaborative filtering approach, we use collected histories to recommend to the developer a set of candidate components that may be useful to this individual developer.

Learn on demand is an approach which allows users to learn new information or components as they are needed. Benefits of this style of learning are discussed by [13]. One such benefit is that the user can immediately see the convenience of this reusable component for an authentic problem situation, thereby increasing the motivation for learning. The system will proactively provide users with task-relevant and personalised recommendations by inferring the need for components based on a collaborative filtering technique.

A basic principle of most collaborative systems is that most users can be clustered into groups. Users in a group share preferences and dislikes for particular items and are likely to agree on future items. A recommendation for a user is based on the opinions or ratings of other like-minded users. This principle can be extended to software classes. A Java class can be considered a user and a software component as an item. A Java class may use zero or more components. If two Java classes share similar ratings for a component, or a set of components, then there is evidence that they will share the same ratings in future components. Thus a recommendation for a particular component to a user (Java class) is based on the rating of like-minded users on this component.

The rest of the paper is organized as follows. Related work is introduced in section 2 while section 3 formally describes the collaborative filtering process and details the algorithms used. Results are displayed and evaluated in section 4 followed by a discussion on future research directions in section 5. Finally Section 6 shows conclusions.

2. Related Work

Traditional methodologies for component search and retrieval can be classified into four different categories, namely *Keyword Search*, *Faceted Classification*, *Signature Matching* and *Behavioral Matching* [10]. Each of the retrieval schemes have a number of limitations that result in less than adequate retrievals. A common shortcoming of these schemes is the failure to take into account the user or relevant domain information when querying the component repository.

Semantic-Based Method Retrieval [12] improves on the above; the user specifies component requirements using natural languages. Domain information, user context and component relationships are all considered. This methodology relies on ontology as a knowledge base. Empirical results indicate this technique is superior to traditional schemes. However, in reality developers are not aware of all available components. If they believe a reusable component for a particular task does not exist then they are unlikely to query the component repository. Thus component retrieval must be complemented with component delivery.

CodeBroker [13] infers the needs for components and proactively recommends components, with examples, that match the inferred needs. The need for a component is inferred by developer comments and method signature. This solution greatly improves on previous approaches however the technique is not ideal. Firstly the reusable components in the repository must be sufficiently commented, to allow matching, this may exclude many components. Secondly the developer must actively and correctly comment his/her code.

Ohsugi et al. [11] propose a system, based on collaborative filtering, to recommended useful functions in application software such as MS-Word or MS-Excel. Software usage or function execution histories are automatically collected from many users in order to provide opportunities to commence user clustering. A set of candidate functions is then recommended to the individual, based on the opinions of like-minded users. Our work applies a similar approach to a different problem domain, namely reusable software components, recommending a set of candidate software components to the developer, based on the opinions of like-minded users.

3. Collaborative Filtering

The goal of collaborative filtering algorithms is to suggest new items or to predict the utility of a certain item for a particular user based on the user's previous likings and the opinions of other like minded users [1]. The first recommender systems based on collaborative filtering to automate predictions was *GroupLens* [3]. Collaborative filtering systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items and are likely to agree on future items. In the context of this paper, a user can be considered to be a Java class and an item refers to a software component.

Collaborative filtering algorithms can be divided into two classes, Memory-Based algorithms and Model-Based algorithms. Memory based algorithms operate over the entire user database to make predictions. In contrast, Model-Based algorithms use the user database to learn a model which is then used for recommendations. Memory-based methods are simpler, seem to work reasonably well in practice and new data can be added easily. For these reasons we decided to use a memory-based algorithm. A key aspect of collaborative filtering is the identification of similar users to the active user for whom the recommendation is being sought. This similarity is based on item usage history. Our prediction system has 3 components:

1. Usage History Collector.
2. User Similarity Analyser.
3. Recommender.

3.1. Usage History Collector

The usage history collector records all invocations of a particular method. The collector stores information about both the user (invoking class) and the item (method invoked). A distinction is made between overloaded methods by recording method signatures. The collector is implemented using the Byte Code Engineering Library [2]. This data is then transformed into a user-item matrix.

3.2. User Similarity Analyser

Each user is treated as a vector; the vector holds a count for all components/methods that the user can invoke. Count will hold a value of zero if the user has never used the particular method. The similarity between two users can be computed by determining the cosine of the angle formed by their vectors. This cosine will fall in the range [-1, 1]. A cosine of 1 indicates two users are identical. A cosine of -1 indicates two users share no similarities. This technique needs to be tailored to the problem domain. To illustrate this, table 1 displays the usage history for a small number of Swing method invocations taken from random GUI Java applications in *SourceForge* [4].

Item	User	User	User
	RemoteD	**HostList**	**CompileDlg**
JTextComponent: getText	1	1	8
JTextComponent : setText	2	2	4
JList: getSelectedValue	2	1	0
JList: getSelectedIndex	**1**	**2**	**0**

Table 1: Sample Usage History for Swing Methods

If compared on the *JList* method *getSelectedIndex*, user *RemoteD* and user *HostList* will be considered as similar as user *RemoteD* and user *CompileDlg*, as both counts differ by one. However it is more likely that user *RemoteD* is more similar to user *HostList* than user *CompileDlg*. This is because we know that both *RemoteD* and *Hostlist* have *JList* objects and that both need to get the index of the item in the *JList* which the user has selected, therefore we can see clear similarities between the two classes at both code and task level. Hence a check is needed before vector similarity is computed to ensure both users have used this item, if not then an arbitrary small similarity value of 0.2 is assigned.

3.3. Recommendations

Recommendations were based on the following algorithm. Let *U* and *C* denote the set of similar users and components respectively. The recommendation R_{ac} for user *a* on component *c* is:

$$R_{ac} = \sum_{i \in U} (v_{ic} \times sim_{ai}) \tag{1}$$

where v_{ic} is the count for user *i* on component *c* and *sim*$_{ai}$ is the similarity between user *a* and user *i*, as calculated in subsection 3.2. A recommendation is made if R_{ac} is greater than or equal to the arbitrary threshold value, τ = 1. The results of this equation depend on the component count of other users who are similar to user *a*. A class that is dissimilar to user *a* will have minimal impact on the final recommendation.

The set of users *U* is based on the standard *k*-Nearest Neighbour algorithm. The optimal value for *k* was found to be 4. Hence for the following experiments we used *k* = 4 neighbours.

4. Recommendation Evaluation

4.1. Outline of Experiment

We have conducted experiments to investigate whether the algorithm described in Section 3.3 could accurately predict useful components to the developer. The component repository used in the experiment contained 1123 methods from the standard Java Swing library. Recommendations were made for a total of 343 Java classes, taken from over 40 GUI applications on SourceForge. In each class several sets of recommendations were made. For example, if a fully developed class used 10 Swing components, then we removed 1 random component from the class and a recommendation set was produced for the developer based on the remaining 9 components. Following this recommendation a further component was removed from the class and a new recommendation set was formed for this developer based on the remaining 8 components. This process was continued until all but 1 component remained.

4.2. Evaluation

Precision and *Recall* are the most popular metrics for evaluating information retrieval systems. *Precision* is defined as the ratio of relevant recommended items to the total number of items recommended, as shown in equation 2. This represents the probability that a selected item is relevant.

$$P = N_{rs}/N_s \tag{2}$$

where N_{rs} is the number of relevant items selected and N_s is the number of items selected. An item, or component, is deemed relevant if it is used by the developer. *Recall*, as shown in equation 3, is defined as the relevant items selected to the total number of relevant items. This represents the probability that a relevant item will be selected.

$$R = N_{rs}/N_r \tag{3}$$

where N_{rs} is the number of relevant items selected and N_r is the number of relevant items.

4.3. Experiment 1

Initial experiments were carried out on the most suitable classes for prediction. A suitable class has one or more close neighbours, i.e., similar classes. An unsuitable class

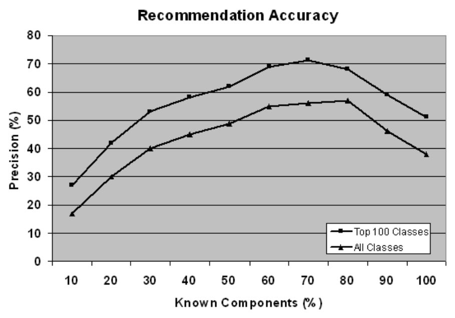

Figure 1. Average Accuracy of Recommendations

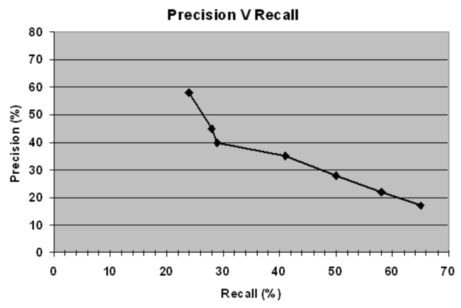

Figure 2. Precision versus Recall

has no close/similar neighbours and thus reduces prediction quality. The upper plotted line in figure 1 displays the accuracy of recommendations for the 100 most suitable classes for predictions:

Focusing on the recommendations for the top 100 classes (the upper plotted line); the first point displays the average accuracy of a recommendation when the developer has utilised between 0% and 10% of the total components they will use. When the developer has utilised only 10% to 20% of the components, recommendations are remarkable good with an average accuracy of over 40%. Recommendation accuracy steadily increased until over 70% of components that the developer will actually employ are used. The decline in accuracy after this point can be explained as follows: Consider a class that uses 10 components and the system is making a recommendation when 50% of the components are known, i.e., 5 components. If the system correctly recommends the remaining 5 components plus 2 incorrect components, then recommendation accuracy will be 71%. However, if the system is making a recommendation when 90% of the components are known and correctly recommends the 1 remaining component plus the 2 incorrect components then recommendation accuracy will be 33%. It is likely that the two incorrect components are used in a very similar class to this active class for whom the recommendation is being sought, thus explaining the recommendation in the first place. One possible way to overcome repeating incorrect recommendations is to allow the developer to explicitly reject a recommended component. As a result the component will not be suggested again. It is the author's belief that a number of the incorrect recommendations may not have been out of context, that is to say the incorrectly recommended component may indeed have been suitable for the active class.

The promising results for the top 100 classes illustrate the system potential as an effective recommender system. The lower plotted line displays the average recommendation accuracy for all 343 classes. When the developer has utilised between 10% and 20% of the total components that they will actually use, recommendations are 25% accurate. Recommendation accuracy peaked to 43% when between 50% and 60% of components were employed. The noticeable fall in recommendation accuracy is due to the fact that recommendations are for all classes, including classes that are unique or at least considerably different from all other classes under test.

To evaluate the performance of the recommendation algorithm, it is important that both *precision* and *recall* are considered together. Figure 2 displays the trade off between precision and recall. *Prediction* and *Recall* figures are an average value based on recommendations for all classes. As expected the lower the recall, the greater the precision. When threshold $\tau = 3$, as discussed in subsection 3.3, *recall* is relatively small however precision is almost 60%. When $\tau = 0.1$, *recall* improves greatly at the expense of *precision*.

4.4. Experiment 2

It has been suggested that 40-60% of code is functionally identical to previously written code [7]. Therefore if we increase the number of usage histories collected it is likely that the probability of finding a class similar to the active class, for whom the recommendation is being sought, should also increase. 100 usage histories were randomly removed from the usage history database leaving a total of 243. The fall in recommendation accuracy, as shown in figure 3 on the following page, confirms the relationship between the number of usage histories and recommendation accuracy.

The average recommendation accuracy for the top 100 classes is 44%. This compares poorly to an average of 56% when all 343 usage histories are collected. This would suggest that the greater the number of usage histories collected, the more reliable the recommendations. Clearly this trend will not continue indefinitely, at some point adding more usage histories will have little or no effect on recommendation accuracy.

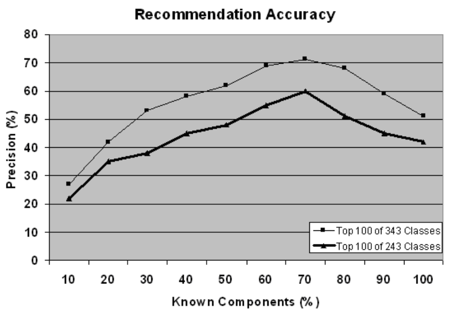

Figure 3. Average Accuracy for top 243 classes

5. Future Work

In our system, classes are considered similar if they use the same or a similar set of components. This similarity measure needs to be extended. Firstly, we will consider the sequence in which components are actually used and secondly, different granularities of similarity will be considered such as fields, methods and components used. Finally a run-time analysis may be useful in highlighting different run-time usage patterns for a particular component between two different classes.

With any recommendation system it is important to explain how the recommendation was derived. Explanations increase a programmer's confidence in recommendations and provide him/her with a mechanism for handling incorrect recommendations. *CodeBroker* [13] provides an interesting feature that allows developers to view example code of a particular component in use. Experiments carried out on *CodeBroker* discovered that developers prefer code examples as opposed to descriptive texts. We will create an intelligent IDE by developing a non-intrusive component recommender as an Eclipse plug-in. The recommender will support explanation by code example and Java documentation retrieval. Finally confidence measures will be added for recommended components, this measure will be based on the closeness of the active class with neighbouring classes.

6. Conclusion

A recommender approach to enable developers discover useful and relevant reusable software components has been presented. Our recommendation scheme addresses various shortcomings of previous solutions to the component retrieval problem. Recommendations consider the developer and problem domain whilst avoiding placing any additional requirements on the developers

Recommender systems are a powerful technology that can extract additional knowledge for a software company from its code databases and then exploit this in future developments. From several experiments, we have demonstrated that this approach offers real promise for allowing developers discover reusable components with minimal effort.

7. Acknowledgments

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology under grant RS/2003/127

References

[1] B.Sarwar et al. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the tenth international conference on World Wide Web*, Hong Kong, 2001.

[2] *Apache Jakarta Project*. Bytecode Engineering Library. http://jakarta.apache.org/bcel/index.html.

[3] P. Resnick et al. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of CSCW '94*, Chapel Hill, NC, 1994.

[4] *VA Corporation*. SourceForge. http://sourceforge.net.

[5] J. Carrol and M. Rosson. *The paradox of the active user. In J.M. Carroll (Ed.), Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. MIT Press, Cambridge, 1987.

[6] K. Daudjee and A. Toptsis. A technique for automatically organising software libraries for software reuse. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, Canada, 1994. IBM Press.

[7] W. Frakes and P. Gandel. Representation methods for software reuse. In *Proceedings of the conference on Tri-Ada '89: Ada technology in context: application, development, and deployment*, pages 302–314, Pennsylvania, USA, 1989.

[8] M. Griss. Software reuse at hewlett-packard. In *Proceedings of the 1st International Workshop on Software Reusability*, Dortmund, Germany, 1991.

[9] J. Hopkins. Component primer. *Communications of the ACM*, Vol. 43:pages 27–30, 2000.

[10] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, Vol. 5:pages 349–414, 1998.

[11] N. Ohsugi, A. Monden, and K. Matsumoto. Recommendation system for software function discovery. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC2002)*, Dec. 2002.

[12] V. Sugurmaran and V. Storey. A semantic-based approach to component retrieval. *ACM SIGMIS Database*, Vol. 34:pages 8–24, 2003.

[13] Y. Yunwen and G. Fischer. Information delivery in support of learning reusable software components on demand. In *Proceedings of the 7th international conference on Intelligent user interfaces*, California, USA, 2002. ACM Press.

[14] Y. Yunwen and G. Fischer. Personalizing delivered information in a software reuse environment. In *Proceedings of 8th International Conference on User Modelling (UM2001)*, Sonthofen, Germany, 2002.