A Recommender Agent for Software Libraries: An Evaluation of Memory-Based and Model-Based Collaborative Filtering

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

Abstract— **Software Agents can conveniently facilitate knowledge discovery and knowledge sharing across an organisation. We contend that programming tasks are often mimicked, that knowledge concerning reusable libraries can be extracted automatically from source code repositories, and that this knowledge can then be filtered and presented to a developer in a manner that will encourage and support future software reuse. We introduce RASCAL, a recommender agent that continually recommends a set of task relevant library methods to a developer.**

RASCAL learns information regarding how a particular reusable library is used and then employs this insight to make task relevant recommendations to a developer. In this paper we detail our RASCAL agent and describe two recommendation techniques; namely Model-Based and Memory-Based Collaborative Filtering. We are interested in producing a scalable and efficient realtime recommender and thus ideally would favor a Model-Based approach. However, each scheme is evaluated against both runtime performance and recommendation accuracy. We present results and discuss the merits and limitations of each technique.

I. Introduction

Reusable libraries tend to be large; for example, the latest version of the Java API library has over 3000 classes while the Java Swing library contains more than 500 classes. Therefore it is imperative that proper tool support be available and that such support tools allow a developer to easily access components in a library. Further to this, it must be recognised that a developer cannot be fully conversant with an entire library or set of libraries and that there is a need to advise a developer when and how to reuse components.

Traditional reuse support tools have mainly focused on passive search techniques, as summarised by Mili et al. [1]. We shift the focus from retrieval to recommendation akin to work of Ye et al. [2]. Our desire is to share, among developers, knowledge about what reusable components exist in a library and to subsequently inform a developer when it is appropriate to use a particular component. Typically this knowledge is shared between programmers via direct meetings, emails or instant messenger and forces a strong dependence on human memory. As noted in LaToza's et al. study, there is also an expensive overhead associated with this type of task switching for both the requester and responder; for example, simply replying to an email may upset the flow of your primary task. We believe that much of this knowledge could be stored and distributed automatically with the assistance of a software agent. We have developed RASCAL, a recommender agent that encourages and assists a developer to make use of software libraries; in particular we focus on library methods.

A central objective of RASCAL is to create an environment where reuse is straightforward, encouraged and allowed to foster. Proponents of reuse claim improvements in software quality and developer productivity [3] whilst a reduction in defect density [4] and time-to-market [5]. However there are several pragmatic issues that hamper reuse such as developer motivation, time constraints and library accessability, in addition to the inadequacy of existing knowledge sharing mechanisms. We suggest that many of these problems stem from ineffective support tools and propose that RASCAL will help to address many of these issues.

The RASCAL agent constantly monitors the code a developer is writing and, based on this, pro-actively recommends a candidate set of task relevant library methods. Recommendations are produced using a Collaborative Filtering (CF) [6] algorithm. CF algorithms work by clustering users who share preferences and dislikes for particular items. A recommendation for an individual is based on the opinions of other like-minded users within the individuals cluster. We propose that this principle can be applied in the software engineering domain. The code a developer is currently writing can be clustered with other similar source codes and based on these similar source codes, we can produce a recommendation for this developer. Important considerations when implementing RASCAL are scalability, accuracy and response times. With this mind we will investigate, compare and evaluate two CF algorithms in this paper, memory-based and model-based collaborative filtering.

A distinction between our recommender agent and other commercial recommenders is that we are attempting to successfully predict a library method that a developer should invoke. It is possibly that a developer will have invoked this method previously; most recommender systems only make predictions for new or unseen items. Recommendation is

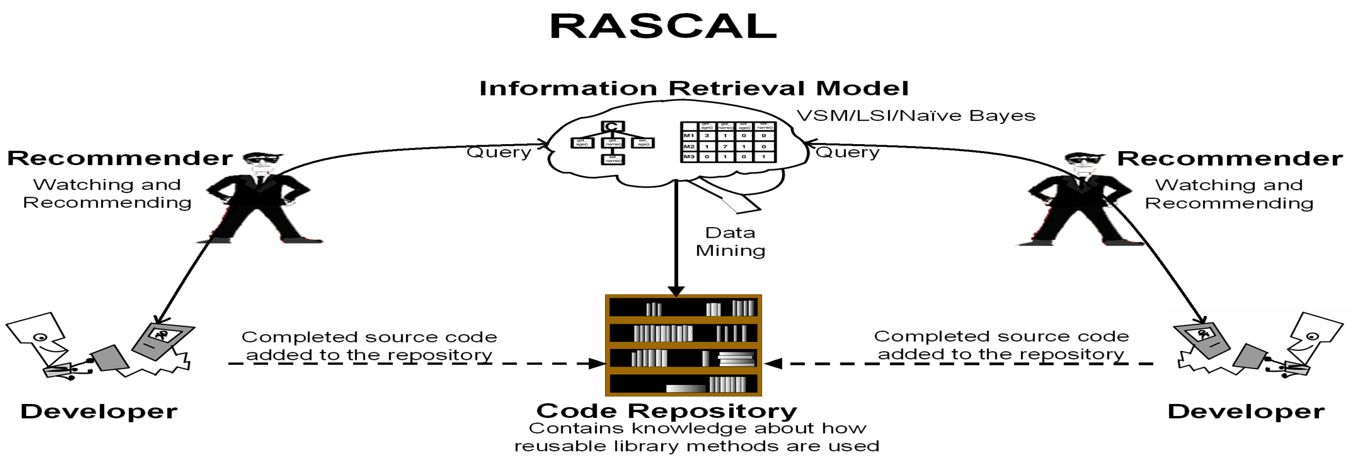

RASCAL
Information Retrieval Model
VSM/LSI/Naïve Bayes
Query
Query
Recommender
Watching and Recommending
Recommender
Watching and Recommending
Data Mining
Completed source code added to the repository
Completed source code added to the repository
Developer
Code Repository
Contains knowledge about how reusable library methods are used
Developer

Fig. 1: The main components of the RASCAL agent.

further complicated by the fact that the order in which methods are invoked is of importance and this needs to be reflected in the recommendation set.

As a prerequisite to producing recommendations, the RASCAL agent needs a repository of source code to allow clustering as required by the recommendation algorithm. In our research, we use the open-source Sourceforge [11] repository. By mining and analysing such a repository, we can establish what reusable library methods exist, obtain knowledge about how they are used by developers and employ an appropriate information retrieval model to efficiently represent these usage patterns. To validate our work we produce recommendations for these mined classes; we compute over 35,000 recommendations for almost 1400 open-source Java classes.

The main contributions of this work are:

- A software agent that supports and encourages software reuse.
- An evaluation of memory-based and model-based collaborative filtering in a software engineering reuse context with regard to recommendation accuracy and runtime performance; we also note that there may be a trade-off between the two.
- A lightweight solution, embedded in RASCAL, to knowledge acquisition and knowledge sharing. This work is specifically geared to supporting reuse but potentially could be used for other tasks such as recommending design practices or identifying code duplication.

The remainder of this paper is organised as follows. In the next section we provide an overview of the main components in the recommender agent. This is followed by a detailed description and comparison of memory-based and model-based collaborative filtering algorithms. Section IV presents experimental results with discussion. Related works are reviewed in section V and finally in section VI we discuss how RASCAL can be extended and draw general conclusions.

II. RASCAL Overview

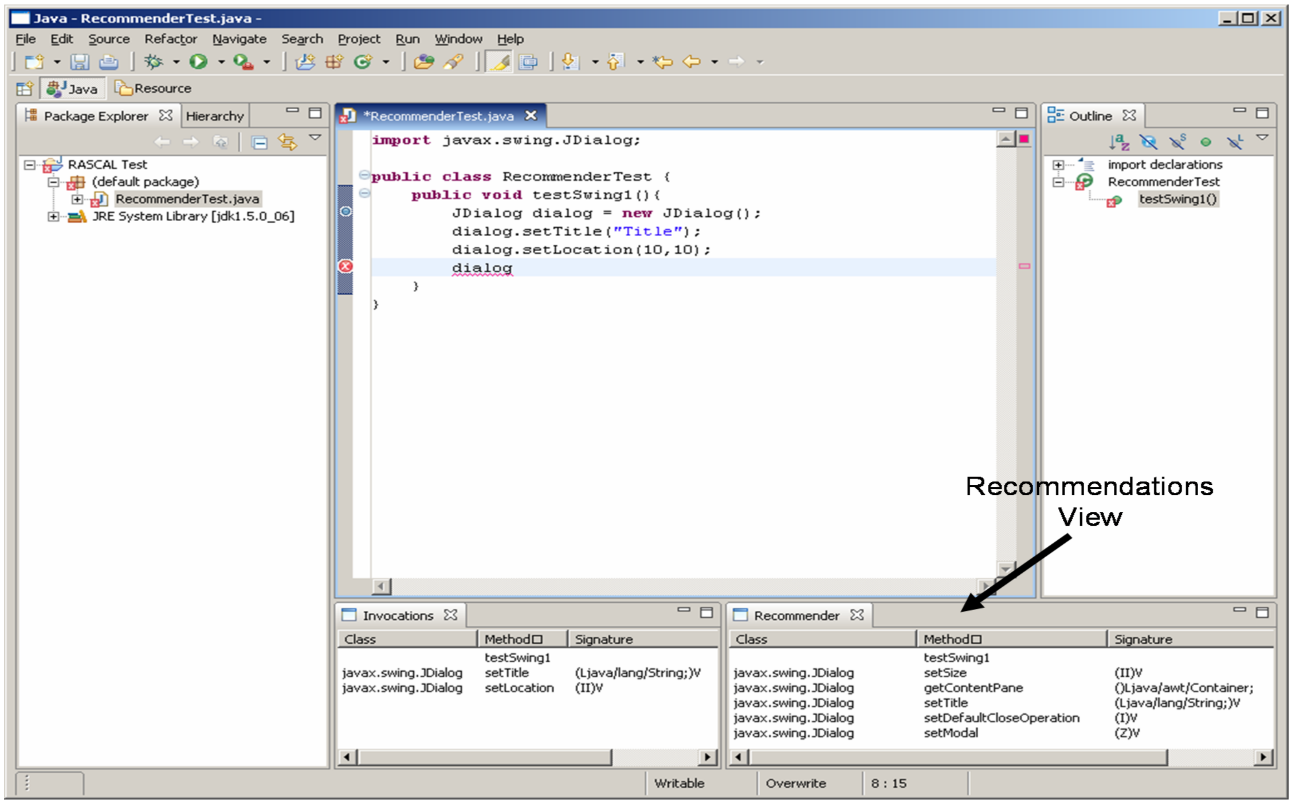
The RASCAL agent presently recommends a set of methods from the Swing and AWT libraries to a developer. Below we describe the four main components of RASCAL, as shown in figure 1.

We produce personalised recommendations for each individual **Developer**. When producing a recommendation, we only consider the content of the current active method which this developer is coding; we do not record any long-term history. In recommender systems, it is common terminology to refer to the user for whom the recommendation is being sought as the active user; likewise here we will refer to the active developer or the active method that a developer is coding. Developer preferences are gathered implicitly.

The **Code Repository** contains code from previous projects, external libraries, open-source projects etc; in our work we used the Sourceforge [7] repository however any repository with sufficient source code is applicable to RASCAL. This source code repository can be considered an unstructured database of user preferences. This repository will be continually updated as new classes/systems are developed. From such a repository, we automatically extract information about what reusable library methods exist and also knowledge about how these are used by developers.

We produce an **Information Retrieval Model** by mining the source code repository. This allows us to make sense of all the user preferences in the code repository and to represent them using a suitable model. We describe both the statistical Vector Space Model (VSM) and the probabilistic Naïve-bayes information retrieval (IR) model in subsection III-A and subsection III-B respectively. This model will need to be created once initially and subsequently updated when a new piece of source code is added to the repository. We extract information from the source code repository using the *Bytecode Engineering Library* [8].

Finally there will be a **Recommender** agent for each individual developer; this agent actively monitors the method that a developer is coding. Based on this, the agent queries



```
import javax.swing.JDialog;

public class RecommenderTest {
    public void testSwing1(){
        JDialog dialog = new JDialog();
        dialog.setTitle("Title");
        dialog.setLocation(10,10);
        dialog
    }
}
```

Class	Method	Signature
	testSwing1	
javax.swing.JDialog	setTitle	(Ljava/lang/String;)V
javax.swing.JDialog	setLocation	(II)V

Class	Method	Signature
	testSwing1	
javax.swing.JDialog	setSize	(II)V
javax.swing.JDialog	getContentPane	()Ljava/awt/Container;
javax.swing.JDialog	setTitle	(Ljava/lang/String;)V
javax.swing.JDialog	setDefaultCloseOperation	(I)V
javax.swing.JDialog	setModal	(Z)V

Fig. 2: RASCAL Eclipse Plugin

the information retrieval model to establish a set of source codes that are most similar to the code currently being written by this developer. Following this, a candidate set of ordered library methods is recommended to the active developer. The recommendation set is produced based on the similar source codes; we explain the collaborative filtering recommendation technique in full in the section III.

RASCAL is currently implemented as a plugin for the Eclipse IDE as shown in figure 2. As a developer is writing code, RASCAL monitors the library methods currently invoked by this developer and uses this information to recommend a candidate set of methods. Recommendations are then presented to the developer in the recommendations view at the bottom right hand corner of the IDE window. Recommendations are unsolicited and thus unobtrusive; a developer need not accept nor reject a recommendation.

It is extremely important that RASCAL integrates seamlessly with the developers IDE. RASCAL should also be able to initialise quickly and produce reliable recommendations in a realtime environment. If even a modest computation time is required then it is highly probable that the plugin would quickly be disabled by the developer. For this reason, we have carried out much research to date in efficient IR models. We have investigated the Vector Space Model (VSM), Latent Semantic Indexing (LSI) [9] and Bayesian Networks (BN) [10]. In the following section we investigate two recommendation algorithms commonly used with such IR models.

III. Recommendations

The goal of a collaborative filtering (CF) algorithm is to suggest new items or predict the utility of a certain item for a particular user based on the users' previous preference and the opinions of other like-minded users [6]. CF systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items and will likely agree on future items. CF algorithms are used in mainstream recommender systems like *Amazon* [11] and *Movielens* [12]. In our work we use a CF algorithm to recommend a set of library methods to a developer. We can infer a developers' previous preferences implicity by examining the set of library methods that they have invoked. Breese et al. [13] identify two classes of CF algorithms, namely memory-based and model-based; we will investigate both.

For clarity we describe three terms, specific to this context, that are common terminology in recommender literature.

Item	This refers to a reusable library method. We wish to predict a developers preference for an item.
User	A user is a Java method in our source code repository. The active user can be considered the method currently being written or indeed the actual developer of that method. We need to establish a database or a repository of users to facilitate recommendation.
Vote	This represents a users' preference for a particular item. In this context, a vote is simply an invocation count for a particular library method.

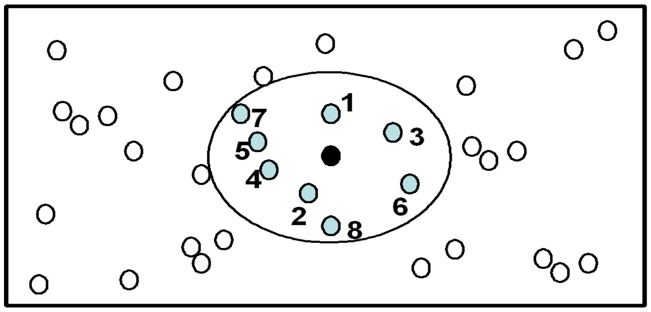
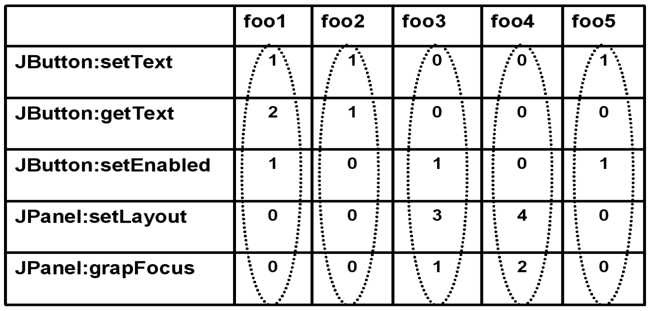


Fig. 3: Illustration of the kNN formation. Here we look for the active methods' k=8 most similar source codes. We need to compare the active user with 34 other users.



	foo1	foo2	foo3	foo4	foo5
JButton:setText	1	1	0	0	1
JButton:getText	2	1	0	0	0
JButton:setEnabled	1	0	1	0	1
JPanel:setLayout	0	0	3	4	0
JPanel:grapFocus	0	0	1	2	0

Fig. 4: Vector Space Model represent User-Preference Matrix

A. Memory-Based Collaborative Filtering

In a memory-based approach, a prediction for the active user is derived by considering all other users in the code repository. Vote v_{ij} corresponds to the vote by user i for item j. The mean vote for user i is:

$$\overline{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j} \tag{1}$$

where I_i is the set of items the user i has voted on. The predicted vote using memory-based CF for the active user a on item j, PV_{aj}, is a weighted sum of the votes of the other similar users:

$$PV_{aj} = \overline{v}_a + N \sum_{i \in kNN} sim\,(a,i)\,(v_{i,j} - \overline{v}_i) \tag{2}$$

where weight $sim(a,i)$ represents the correlation or similarity between the current user a and each user i. kNN is the set of k nearest neighbours to the current user, as illustrated in figure 3. A neighbour is a user who has a high similarity value $sim(a,i)$ with the current user. The set of neighbours is sorted in descending order of weight. For experiments we used a value of $k = 8$. N is the normalising factor such that the absolute values of the weights sum to unity. From equation 2 we can now predict a users' vote for any item. In the context of this work, we can now predict a developers' preference for any library method assuming that there exists at least one user in the code repository that has previously used the particular library method. Library methods are ranked based on their predicted vote and the top n methods are recommended to the developer. In our experiments, we use a value of $n = 7$.

Central to CF is the ability to determine a set of users who are most relevant or similar to the active user for whom the recommendation is being sought, $sim(a,i)$. We want to effectively discover source codes in our repository that are most similar to the code currently being written. As shown in figure 4, we employ the Vector Space Model (VSM) [14] to create a matrix of user votes; this corresponds to the IR component in figure 1. The similarity between any two users, $sim(a,i)$, can now 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 whereas -1 denotes no similarities. When making a recommendation for the active user, we create a query vector and compare this with all other user vectors.

Note, to determine the kNN nearest users, we must compare the active user with each user in the code repository. This property of memory-based CF algorithms has potential implications for scalability and runtime performance. In general however, memory-based CF works reasonably well, are straightforward to implement and easily allow new users to be added. In the RASCAL system, we expect new source code to be added to the code repository frequently and hence this latter characteristic may be important.

B. Model-Based Collaborative Filtering

Model-based CF algorithms seeks to fit a probabilistic model to the data through unsupervised learning techniques and to subsequently use this model for making predictions. We used a similar version of the model-based clustering method described by Breese et al. [13]. In this work we create a set clusters $C = c_1, c_2, ..., c_m$ where m is the number of unique clusters, as shown in figure 5. We estimate the probability that an active user a belongs to a particular cluster c_x, $Pr(c_x, a)$. After establishing the cluster which user a is mostly likely to be a member, we can use a modified version of equation 2 to predict the vote for user a on item j:

$$PV_{aj} = E\,(v_{a,j}) = Pr(c_x, a)(v_{c_x,\overline{j}}) \tag{3}$$

where c_x is the classified cluster and $v_{c_x,\overline{j}}$ is the average vote for item j in cluster c_x. To make a recommendation set, we recommended the top n items with the highest predict vote. We also use a value of $n = 7$ in the model-based experiments.

Unlike previously, a users' predicted vote for an item is based on users within cluster c_x and not the top kNN users. We only consider one cluster when making recommendations and thus the probability weighting is not as important as perhaps in other systems were the top s clusters are considered. However the probability value is still useful as a confidence measure. We considered the possibility of examining the top s clusters to perhaps compensate for a cluster which had a low confidence value (probability) or a low user population, however we found that such a solution had a negative effect on overall recommendation accuracy.

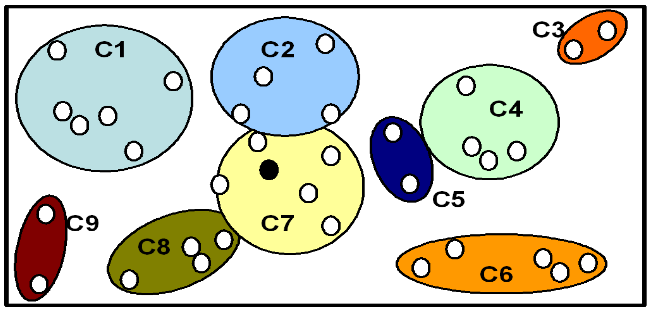


Fig. 5: Illustration of EM clustering. We now compare the active method with 9 clusters as opposed to 34 individual users when using the kNN memory-based algorithm.

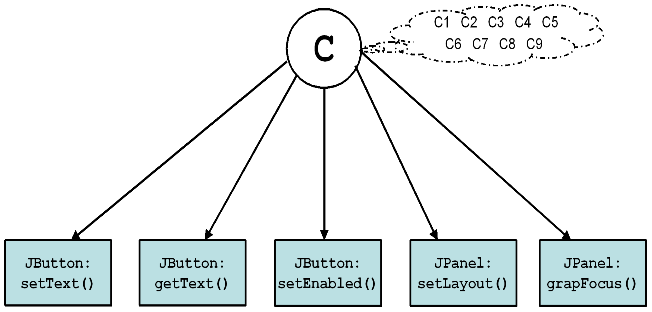


Fig. 6: Naïve-Bayes Example

To implement this approach, we use a simple Naïve-bayes [15] [16] clustering model. For clarity, we can visualise this model as a simple tree structure that has the classification node as a parent of all other attribute nodes. This is illustrated in figure 6. Naïve-bayes is based on the assumption that the attributes values are independent of each other given the class C. In the context of this work, the classification node represents a particular cluster, whereas an attribute node represents each reusable library method that can be invoked. We initially use the Expectation-Maximisation (EM) [17] algorithm to find clusters and hence to learn the structure of the model. The conditional probability of each attribute given the class C is learnt from training data. Classification is then done for a query user, the active user, by applying Bayes rule to compute the probability of C given a particular instance of attributes. Following this, we can establish which class or cluster has the highest posterior probability. The probability that user a belongs to cluster c_x is calculated as follows:

$$P(c_x, a) = P(c_x) \prod_{j=1}^{I_a} P(v_{a,j}|c_x) \tag{4}$$

where I_a is the set of items that user a as voted on and $v_{a,j}$ is user a's vote for item j.

This model was constructed and implemented using the popular WEKA [18] machine learning tool. A major benefit of model-based CF is that we need only keep the parameters of the model in memory to make predictions, which takes much less space than the entire user-preference matrix as is required by the memory-based model, assuming the number of clusters is less than the original number of users. We found the optimal size of the cluster set to be less than 50% of the original number of users. A further benefit is that recommendations can be produced very fast as there will likely be a much smaller set of clusters than users. The average item-vote for each cluster can also be pre-computed. Unlike the memory-based scheme, the model-based technique requires an initial learning phase before predictions can be made and thus it is more difficult to add new users. However it may be possible to add more source codes to our model in a nightly build and thus the availability of the recommender agent would not be adversely affected.

IV. Experiments

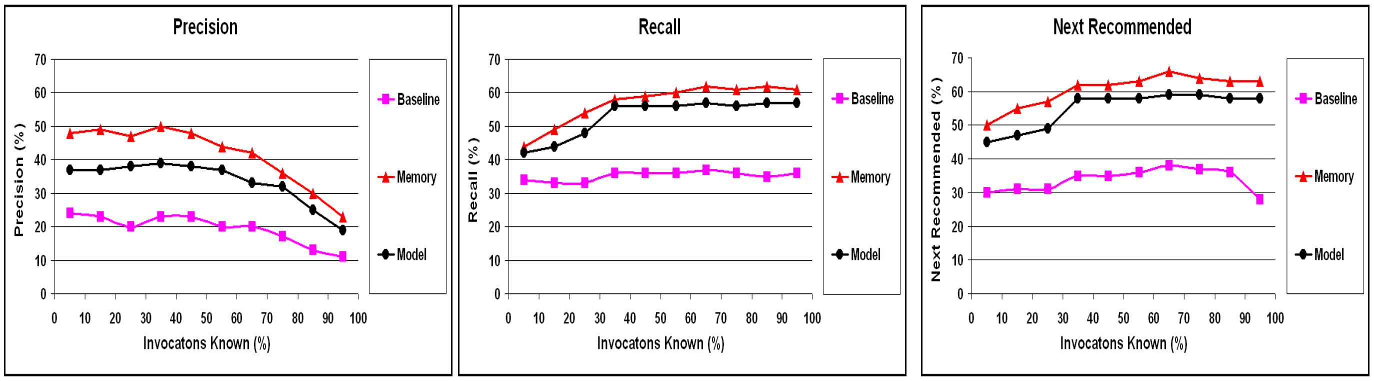
A. Dataset

We produced over 35,000 recommendations from 3481 methods mined from Sourceforge [7]. Recommendations were produced solely at the method level and not the class level as in previous work [19]. Further to this, each method had on average 11 invocations. Recommendations were made for both the SWING and AWT libraries; in total there was 2090 Swing and AWT library methods that were invoked at least once in our code repository. Since we have the complete source code, we can automatically evaluate the recommendations for a piece of code by checking whether the recommended method was in fact called subsequently.

For each user defined method, several recommendations were made. For example, if a fully developed method had 10 Swing invocations, then we removed the 10th invocation from that method and a recommendation set was produced for the developer based on the preceding 9 invocations. Following this recommendation, the 9th invocation was removed from the user method and a new recommendation set was formed for this developer based on the preceding 8 invocations. This process was continued until just 1 invocation remained. Each recommendation set contained a maximum of 7 library methods; we must be careful not to overload the developer with a large recommendation set yet recommend a reasonable number of methods that will allow the developer to better plan their approach to the programming task.

B. 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; $P = n_{rs}/n_s$, where n_{rs} is the number of relevant items selected and n_s is the number of items selected. This represents the probability that a selected library method is relevant. A library method is deemed relevant if it is used by the developer for whom the recommendation is being sought. Recall is defined as the ratio of relevant items selected to the total number of relevant items; $R = n_{rs}/n_r$, where n_{rs} is the number of relevant items selected and n_r is the number of relevant items. This represents the probability that a relevant library method will be selected.



(a) (b) (c)

Fig. 7: (a) Precision (b) Recall (c) Next Recommended

It is particulary important that RASCAL recommends methods in a relevant order i.e. the invocation order. We will evaluate this using a simple binary Next Recommended (NR) metric; $NR = 1$ if we successfully predict or recommend the next method a developer will use, otherwise $NR = 0$. We will also discuss runtime performance issues in subsection IV-D.

C. *Results*

All results are displayed as a percentage value as shown in figure 7. Model-based results are based on 850 clusters. We include a baseline result here also. Baseline recommendations were produced by recommending, at each recommendation stage, the seven most commonly invoked library methods. Both the memory-based and model-based produce significantly better results than our baseline technique. Overall the memory-based approach produced marginally better results as detailed below.

Precision is displayed in figure 7(a). The average precision value while using the memory-based CF algorithm was 42%, this compares with a 34% average when using model-based CF. If the recommendation set contained the maximum seven library methods then, on average, three would be used by the developer based on using memory-based CF. The average value of recall for memory-based was 57% compared with 53% for the model-based approach. As the size of the recommendation set can be between zero and seven, we validate our recommendations against the next seven invocations the developer actually made. Hence it is possible that a single recommended library method was used twice by a developer within this scope and thus would be relevant twice. This would have a positive effect on recall but would likely negatively impact precision as now there may be more incorrect recommendations in the set.

The next recommended metric is displayed in figure 7(c). Again the model-based average of 55% is relatively close to the memory-based average of 60%. When we know very little about the actually method a developer is writing there is still a reasonable good chance that we will be able to correctly predict the next library method that they need to invoke.

D. *Discussion*

We find that memory-based CF consistently outperforms model-based CF recommendations. However, it is important that we recognise several other benefits of using model-based CF and in particular the Naïve-bayes clustering model. Our work is aimed at developing a practical tool to assist developers to reuse software libraries. To gain developer confidence in such a tool, recommendations should be useful and frequent. Likewise it also important to consider performance issues; recommendations should be presented in a timely an efficient manner. The average time taken to make a recommendation was under 0.2 seconds using the Naïve-bayes model compared with over 1 second when using the memory technique. As more source code is added to the repository or as a reusable libraries grows, there will be a much more sever effect on the memory-based performance. However, there is an overhead cost associated with building the original clustering model. In this example where there was 2090 users and 3481 library methods, it took just under 3 hours to create the Naïve-bayes model which had 850 clusters (40% of the original number of users). We experimented with different cluster sizes; generally recommendations were poorer when a smaller number of clusters were used but there was no significant improvement in results when the cluster size was greater than 850.

Generally, we notice two different trends in precision and recall. Precision tends to decrease as we know more about the active method while recall increases. This result perhaps requires clarification. Consider a developer who invokes in total 10 methods. When we make a recommendation for that developer when she has only used 1 method, there is a set of 9 possible methods to recall. The chances of recalling all relevant methods is quite low and hence the recall result is low in earlier recommendations. However, when this developer has used 9 methods and there is only 1 possible method to recall, then the chances of this method being in the recommendation set is quite high. In contrast, the more invocations the developer has made, the fewer there are to correctly recommend and hence precision decreases in latter recommendations.

V. Related Work

Traditional retrieval schemes focused generally on techniques like *Keyword Search* and *Signature Matching* [1]. More recently several intelligent retrieval tools have been proposed. Mingyang et al. [20] employ conversational case-based reasoning technology to help developers locate reusable components. Software components are represented as cases and a knowledge intensive case-based reasoning method is adopted to explore context-based semantic similarities between a users' query and stored components. A conversational approach is used to collect user requirements interactively and incrementally. Another approach for promoting reuse is to assume that all components (ie, packages, methods and arguments) are annotated with semantic labels that indicate opportunities for reuse. A key problem with this approach is that it requires considerable effort to provide the semantic annotations. Hess et al. [21] describe a variety of machine learning techniques for automatically assigning such annotations, and demonstrate their effectiveness on a large collection of Web Services.

Several web-based search and retrieval techniques have been developed such as `Koders.com`, `Kickjava.com` and `Planetsourcecode.com`. Similar to *Google* [22], *ComponentRank* [23] is one such technique. Components are ranked based on analysing use relations among the components and propagating the significance of a component through the use relations. Preliminary results indicate that this technique is effective in giving a high rank to stable general components which are likely to be highly reusable and a lower rank to non-standard specialised components. Hummer and Atkinson [24] have carried out a general study on using the web as a reuse repository; they evaluate several search engines such as *Google*, *Yahoo* and *Koders*. They identify some of the advantages of web based approaches such as scalability and efficiency but also note limitations such as security, legal concerns and implicit classes.

The use of software agents for supporting and assisting library browsing has been proposed by Drummond et al. [25]. An active agent attempts to learn the component which the developer is looking for by monitoring the developers' normal browsing actions. Based on experimental results, 40% of the time the agent identified the developers' search goal before the developer reached the goal. By providing non intrusive advice that accelerates the search, this work is intended to complement rather than replace browsing.

A major limitation with all of the retrieval techniques above is that the developer must initiate the search process. However, in reality developers are not aware of all available methods in a library or may be unable to express their query clearly. If they believe a reusable component for a particular task does not exist then they are less likely to search the component repository. Thus to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with component delivery/recommendation.

Ye and Fischer [2] identify the cognitive and social challenges faced by software developers who reuse and also present a tool named *CodeBroker* which address many of these challenges. *CodeBroker* infers the need for components and pro-actively recommends components, with examples, that match the inferred needs. The need for a component is inferred by monitoring developer activities, in particular developer comments and method signature. This solution greatly improves on previous approaches, however the technique is not ideal. Reusable components in the repository must be sufficiently commented to allow matching and developers must also actively and correctly comment their code. Active commenting is an additional strain placed on developers which is likely to make the use of *CodeBroker* less appealing.

Mandelin et al. [26] present an intelligent tool for understanding and navigating the API of a particular library. They suggest developers often know the objects they would like to reuse but are unaware of how to write the source code to get those objects; for example a developer may wish to create a $IFile$ object from a $ASTNode$ but may not be aware of the code needed to do this. They provide a tool named *PROSPECTOR* which can automatically assist a developer to better understand the library API by providing code snippets relevant to the current task; for example, how to convert between different data representations.

Reuse support through code examples is a popular research theme. Grabert and Bridge [27] present a software tool for examplet reuse. They define examplets to be goal-directed snippets of source code, often written for tutorial purposes, that show how to use program library facilities to achieve some task. The tool allows users to specify their goal in natural text and will subsequently automatically take into account the source code on which they are working. The system combines text retrieval and a semantic net representation of the source code to achieve promising results. Another notable tool for finding code examples is Strathcona [28]. The tool is used to find source code in an example repository by matching the code a developer is currently writing. Similarity is based on multiple structural matching heuristics, such as examining inheritance relationships, method calls, and class instantiations. These measures are applied to the code currently being written by the developer and matched examples from the repository are retrieved and recommended.

Our work is similar to a number of the techniques mentioned above. Like *CodeBroker* [2], our goal is to recommend a set of candidate software components to a developer; however our recommendations are not based on the developers' comments/method signature. In contrast we produce recommendations using collaborative filtering which is similar to case-based reasoning and the example based techniques employed by Grabert and Bridge [27], and Holmes and Murphy [28]. Similar to the *PROSPECTIVE* tool and the work of Hess et al., we are interested in increasing and supporting library reuse though we are attempting to predict in advance what a developer is attempting to code. Like Drummond et al. [25] we use an active agent to monitor the current developer though we are concerned with pro-actively recommending suitable reusable methods as opposed to assisting the search process.

VI. Conclusions

We have presented a software agent that can support and encourage software reuse by facilitating knowledge sharing within a community. We have shown that just as people can be clustered in terms of their preferences for various items, Java source code may also be clustered based on the library methods invoked. We investigated two collaborative filtering algorithms; we found that memory-based algorithms performed better in terms of precision and recall but that a model-based technique was significantly faster and yet produced reasonably good results. In terms of developing a user friendly recommender agent, we believe a model-based approach is most applicable to our domain.

Our recommendation scheme addresses various shortcomings of previous solutions to the library retrieval problem; RASCAL considers the developer context and problem domain but uniquely does not place any additional requirements on existing library components or developers. Unlike many typical reuse tools, RASCAL is pro-active and constantly suggests library methods to reuse. However there is a need to incorporate code examples in our recommendations, similar to the work of Holmes and Murphy [28]. In addition, presently the only information we know about an active method is library invocation frequency. We will examine if it is useful to know further information; for example the classes instantiated by a user or structural information. This information may be useful for improving the clustering technique. Further work is also required in the practical implementation; RASCAL offers unsolicited advice and we must be sensitive to this. The RASCAL plugin should be unobtrusive and work seamlessly with the existing IDE.

Recommender agents provide a low-cost practical solution to the software component retrieval problem. Knowledge about reuse libraries can automatically be gathered and easily distributed among developers, for the benefit of future developments. RASCAL can make fast and reasonably accurate predictions, even when little information is known about the active method and it is our belief that future work will strengthen both recommendation accuracy and performance.

VII. Acknowledgements

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology (IRCSET).

References

[1] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, pp. 349–414, 1998.

[2] Y. Ye and G. Fischer, "Reuse-conducive development environments," *International Journal of Automated Software Engineering*, vol. 12, pp. 199–235, 2005.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Commun. ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[4] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pp. 282–292, 2004.

[5] K. Yongbeom and E. Stohr, "Software reuse: Survey and research directions," *Management Information Systems*, vol. 14, no. 4, pp. 113–147, Spring 1998.

[6] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Reidl, "Item-based collaborative filtering recommendation algorithms," in *World Wide Web*, 2001, pp. 285–295. [Online]. Available: citeseer.ist.psu.edu/sarwar01itembased.html

[7] J. Ebert, "Storm - a user story tool. http://xpstorm.sourceforge.net," 2002.

[8] Apache, "Bytecode engineering library (2002-2003). http://jakarta.apache.org/bcel," 2003.

[9] F. McCarey, M. O. Cinnéide, and N. Kushmerick, "Recommending library methods: An evaluation of the vector space model (vsm) and latent semantic indexing (lsi)," in *Proceedings of the 9th international conference on Software Reuse*, 2006, pp. 217–230.

[10] F. McCarey, M. O. Cinneide, and N. Kushmerick, "Recommending library methods: An evaluation of bayesian network classifiers," in *Proceedings of the 2nd international workshop on Knowldge Collaboration in Software Development in Conjuction with the 21st IEEE ACM on Automated Software Engineering*, 2006.

[11] J. Bezos, "Amazon.com plc. seattle, wa 98108-1226, usa www.amazon.com," 2004.

[12] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm, and J. Riedl, "GroupLens: An Open Architecture for Collaborative Filtering of Netnews," in *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work*, Chapel Hill, North Carolina, 1994. http://www.movielens.org, pp. 175–186. [Online]. Available: citeseer.ist.psu.edu/resnick94grouplens.html

[13] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 43–52. [Online]. Available: citeseer.ist.psu.edu/breese98empirical.html

[14] T. A. Letsche and M. W. Berry, "Large-scale information retrieval with latent semantic indexing," *Inf. Sci.*, vol. 100, no. 1-4, pp. 105–137, 1997.

[15] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. New York: John Wiley and Sons, 1973.

[16] P. Langley, W. Iba, and K. Thompson, "An analysis of bayesian classifiers," in *National Conference on Artificial Intelligence*, 1992, pp. 223–228. [Online]. Available: citeseer.ist.psu.edu/langley92analysis.html

[17] N. L. A. Dempster and D. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society*, vol. 39, pp. 1–38, 1977.

[18] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, 2005.

[19] F. McCarey, M. O. Cinnéide, and N. Kushmerick, "Knowledge reuse for software reuse," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, 2005.

[20] M. Gu, A. Aamodt, and X. Tong, "Component retrieval using conversational case-based reasoning," pp. 259–271, 2005.

[21] A. Heß, E. Johnston, and N. Kushmerick, "Assam: A tool for semi-automatically annotating semantic web services," in *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, 2004.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Tech. Rep., 1998. [Online]. Available: citeseer.ist.psu.edu/page98pagerank.html

[23] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: relative significance rank for software component search," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 14–24.

[24] O. Hummel and C. Atkinson, "Using the web as a reuse repository," in *Proceedings of the 9th International Conference on Software Reuse*. Springer, 2006, pp. 298–311.

[25] C. G. Drummond, D. Ionescu, and R. C. Holte, "A learning agent that assists the browsing of software libraries," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1179–1196, 2000.

[26] D. Mandelin, L. Xu, R. Bod&#237;k, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," *SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, 2005.

[27] M. Grabert and D. Bridge, "Case-based reuse of software examplets," *Journal of Universal Computer Science*, vol. 9, pp. 627–640, 2003.

[28] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005, pp. 117–125.