# Recommending Library Methods: An Evaluation of the Vector Space Model (VSM) and Latent Semantic Indexing (LSI)[⋆]

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.** The development and maintenance of a reuse repository requires significant investment, planning and managerial support. To minimise risk and ensure a healthy return on investment, reusable components should be accessible, reliable and of a high quality. In this paper we concentrate on accessability; we describe a technique which enables a developer to effectively and conveniently make use of large scale libraries. Unlike most previous solutions to component retrieval, our tool, RASCAL, is a proactive component recommender.

RASCAL recommends a set of task-relevant reusable components to a developer. Recommendations are produced using Collaborative Filtering (CF). We compare and contrast CF effectiveness when using two information retrieval techniques, namely Vector Space Model (VSM) and Latent Semantic Indexing (LSI). We validate our technique on real world examples and find overall results are encouraging; notably, RASCAL can produce reasonably good recommendations when they are most valuable i.e., at an early stage in code development.

## 1 Introduction

Successful software reuse has been to shown to improve software quality and developer productivity whilst reducing defect density [1] and time-to-market [2]. Despite this, reuse has not been adopted widely. Ye et al. [3] identifies the significant cost and commitments required from an organisation to institute a reuse program. To maximise reuse, minimise risk and ensure a healthy return on investment, reusable components should be accessible, reliable and of a high quality. In this paper we concentrate on effective tool support to increase the accessibility and use of reusable libraries.

Poulin [4] suggests that the best libraries contain around 30, but in rare cases up to 250, components. In reality however, it is possible that a library could contain many thousand components; for example the Java 1.4 API library has 2,723 classes. To avail of all the reusable components in such a large library,

[⋆] Funding provided by the IRCSET under grant RS/2003/127.

it is essential that adequate tool support be provided. Indeed it has been the inadequacy of conventional tools that has long hampered reuse. Frequently, the time taken to locate a component in a particular repository and the subsequent integration of that component with existing code will be perceived as too costly and outweighing any potential reuse benefits.

The importance of reuse support tools is reflected in the shift from initial software reuse research which focused on techniques to develop reusable components and component libraries to a focus on supporting reuse through intelligent storage and retrieval strategies, for example [5]. Each solution attempts to assist developers in discovering or locating components in which they are interested. These approaches share a common shortcoming though; the developer must initiate the retrieval process. Pragmatic issues such as time constraints, limited conversancy with the library and lack of developer motivation will determine the likelihood of a developer searching a library. In reality, if a developer believes a reusable component for a particular task does not exist or they do not anticipate the need to reuse such a component, then they are less likely to query the component repository; no retrieval schemes address this important issue.

In our work, we focus on complementing component retrieval with component recommendation. We describe a technique that can produce recommendations and we develop a reuse tool, named RASCAL, to investigate our approach. We believe recommendations will assist and encourage developers in making full use of large component libraries in an efficient manner and in turn will help to promote software reuse. RASCAL is a proactive tool; no additional requirements are placed on a developer and it is applicable to any existing code libraries.

Similar to many commercial recommenders, we produce a set of personalised recommendations for an individual. However, unlike other domains where perhaps a set of books or movies may be presented to a customer, RASCAL recommends a set of task relevant methods to a particular developer. Like most recommendation tasks, RASCAL recommends software components that the developer is interested in. Recommendation in our tool is complicated though because we wish to recommend components which we believe the developer may be unfamiliar with or unaware of. Another interesting distinction between our recommender system and most mainstream recommenders is that we are trying to predict, in order, the next likely items a developer will employ. Many typical recommender systems only predict a vote for items which the user has not yet tried. Our aim is to predict the next library method a developer should invoke; it is quite likely that the developer will have invoked this method previously.

We compare two information retrieval approaches commonly used in text retrieval and explain how these techniques can be adapted to our domain in order to produce recommendations. Firstly, we employ a Collaborative Filtering (CF) [6] algorithm using the popular Vector Space Model(VSM) [7]. We then compare this approach with recommendations produced using CF and the more advanced retrieval technique Latent Semantic Indexing (LSI) [7, 8, 9]. To validate our work we produce over 32,000 recommendations for almost 1500 open-source Java classes.

The remainder of this paper is organised as follows. In the next section we review related works. An overview of RASCAL's implementation is presented in section 3. In section 4 we detail our recommendation techniques followed by a comparative analysis of the experimental results in section 5. Finally we discuss how RASCAL can be extended and draw general conclusions in section 6.

## 2 Related Work

We discuss related research in software reuse tool support and recommender systems using information retrieval (IR), and we describe how IR techniques can be adapted to support software reuse.

### 2.1 Reuse Tool Support

The development of reusable components and component libraries has been an active research area for some time but this alone will not encourage reuse. "A classified collection is not useful if it does not provide a search-and-retrieval mechanism to use it" [10]. Mili et al. [11] classify traditional search and retrieval methodologies into four categories, namely *Keyword Search*, *Faceted Classification*, *Signature Matching* and *Behavioral Matching*. Each of these retrieval schemes has a number of limitations that result in less than adequate retrievals.

More recently, several *Semantic-Based* retrieval tools have been proposed; typically while querying the repository the developer specifies component requirements using natural languages which are interpreted using a language ontology as a knowledge base. Components in the repository will also have a natural language description. Both the developer query and component descriptions are formalised and closeness is computed. A set of candidate components can be ranked based on their closeness value. Unlike the approaches mentioned above, domain information, developer context and component relationships are all considered. Empirical results indicate that such schemes are superior to traditional approaches [12, 13].

Drummond et al. [14] present the use of a *learning* software agent to support the browsing of software libraries. The active agent attempts to learn the component 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 disadvantage 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 components. If they believe a reusable component for a particular task does not exist then they are less likely to search the component repository; none of the above schemes attempt to address this important issue. Thus to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with component delivery/recommendation.

Ye and Fischer [3] 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 which currently they may not do. Active commenting is an additional strain placed on developers which is likely to make the use of *CodeBroker* less appealing. Notably, Ye and Fischer remark that browsing and searching are passive mechanisms because they become only useful when a developer decides to make a reuse attempt by knowing or anticipating the existence of certain components.

## 2.2 IR and Recommenders System

Sarwar et al. [15] describe a collaborative filtering recommender system with Latent Semantic Indexing (LSI). Collaborative filtering works by matching customer preferences to other customers in making recommendations. LSI is a technique commonly used to infer meaning or concepts in texts. Recommendations are produced for two datasets: a movie dataset and a e-commerce dataset. Several limitations of CF algorithms are identified such as sparsity, scalability and synonymy. In an attempt to address these issues, LSI is applied; recommendations using this technique can be performed much faster than pure CF. Recommendations using the LSI approach performed less well than pure CF for the e-commerce dataset but in some cases performed better than CF on the Movie dataset.

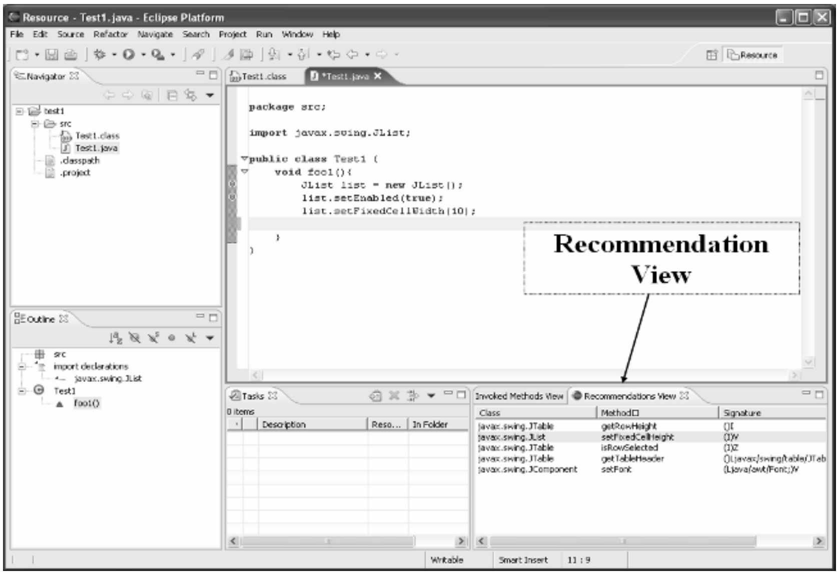
LSI is commonly used in natural language domains though some of the properties of source code, such as comments and identifiers, make it suitable for LSI also. Marcus et al. [16] apply LSI to recover documentation-to-source-code links. LSI is used to extract meanings from documentation and source code, this information is then used to identify traceability links based on similarity measures. The results of this approach are promising; the LSI technique has performed at least as well as the traditional Vector Space Model (VSM) however much less preprocessing of the source and documentation is required. This work follows on from a LSI source code clone detection tool [17].

Our work is similar to a number of the techniques mentioned above. Like *CodeBroker* [3], 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 and LSI, akin to the work of Sarwar et al. [15] and Marcus et al. [16], however in a different context. Like Drummond et al. [14] we use an active agent to monitor the current developer though we are concerned with pro-actively recommending suitable reusable components as opposed to assisting the search process.

## 3 RASCAL Overview

RASCAL is currently implemented as a plugin for the Eclipse IDE, as illustrated in figure 1. As a developer is writing code, RASCAL monitors the methods currently invoked and uses this information to recommend a candidate set of methods to this developer. Recommendations are then presented to the developer in the recommendations view at the bottom right hand corner of the IDE window. At present, RASCAL recommends methods from the Swing and AWT libraries. An important consideration when implementing RASCAL is that recommendations must be produced in a real time environment; we discuss the implications of this in section 5. Below we describe the main components of RASCAL, as shown in figure 2.

We produce personalised recommendations for each individual **Developer**. When producing a recommendation, we only consider the content of the current



```
package src;

import javax.swing.JList;

public class Test1 {
    void foo1(){
        JList list = new JList();
        list.setEnabled(true);
        list.setFixedCellWidth(10);

    }
}
```

| Class | Method | Signature |
|---|---|---|
| javax.swing.JTable | getRowHeight | ()I |
| javax.swing.JList | setFixedCellHeight | (I)V |
| javax.swing.JTable | isRowSelected | (I)Z |
| javax.swing.JTable | getTableHeader | ()Ljavax/swing/table/JTab |
| javax.swing.JComponent | setFont | (Ljava/awt/Font;)V |

**Fig. 1.** Prototype implementation of RASCAL

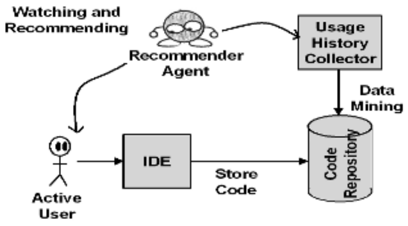


**Fig. 2.** RASCAL Overview

active method which this developer is coding. Recommendations are produced for and based on the current active method. For clarity in future sections, we introduce two terms:

**User.** The current active method which a developer is implementing.
**Item.** A reusable library method, ignoring signature, which is utilised by a user.

The **Code Repository** contains code from previous projects, external libraries, open-source projects etc; in our work we used the Sourceforge [18] repository. This repository will be continually updated as new classes/systems are developed. From such a repository, we can extract information about what reusable items exist and also knowledge about how these are used. The **Usage History Collector** automatically mines the code repository to extract item usage histories for all users. This will need to be done once initially for each user and subsequently when a new user is added to the repository. We extract this information using the *Bytecode Engineering Library* [19]. Item usage histories for all the users are then transformed into an item-user preference database, as detailed in section 4.1, which can be used to establish similarities between two users. Finally the **Recommender Agent** actively monitors the method that the developer is coding. The agent attempts to establish a set of neighbouring users who are similar to the active user; a set of ordered library methods is then recommended to the active user based on the neighbouring users.

## 4 Recommendations

In this section, we describe two information retrieval techniques; namely Vector Space Model and Latent Semantic Indexing. We explain how either of the retrieval techniques can be used by a Collaborative Filtering (CF) algorithm to produce recommendations. CF using the vector space model is commonly referred to as ‘pure’ CF; we will use this terminology in latter sections.

### 4.1 Information Retrieval

**Vector Space Model.** In text retrieval, the Vector Space Model (VSM) [7] is one of the most commonly used methods for representing a document as a vector of terms. A collection of documents is represented as a term-by-document matrix where the $[i, j]^{th}$ element indicates the association between the $i^{th}$ term and $j^{th}$ document. This association reflects the $i^{th}$ term occurrence in document $j$. A term can represent different text units; most commonly a word. Each term can also be individually weighted allowing that term to become more or less important within a document or the entire document collection as a whole. We discuss weighting schemes below. The similarity between any two documents 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 documents are identical whereas -1 denotes no similarities.

In the context of our work, a document is representative of a user and a term represents an item. Table 1 displays an example item-user matrix derived using

**Table 1.** Non weighted Item-User matrix created using VSM

| Item | User | User | User | User | User |
|---|---|---|---|---|---|
| | **U1** | **U2** | **U3** | **U4** | **U5** |
| JButton:getText | 1 | 1 | 0 | 0 | 1 |
| JButton:setText | 2 | 1 | 0 | 0 | 0 |
| JButton:setEnabled | 1 | 0 | 1 | 0 | 1 |
| JPanel:setLayout | 0 | 0 | 3 | 4 | 0 |
| JPanel:grabFocus | 0 | 0 | 1 | 2 | 0 |

VSM. Given such a matrix, we can query this to find the set of users similar to user $x$. Firstly we need to create a query vector representative of user $x$. If a weighted scheme has been applied to the VSM matrix then each non-zero element in the query vector should be a weighted frequency. We can now calculate the similarity between user $x$ and any other user by determining the cosine of the angle formed by their vectors.

**Latent Semantic Indexing.** Latent Semantic Indexing (LSI) is a vector space model approach to conceptual information retrieval. LSI is commonly used to overcome the synonymy and polysemy problem; it captures underlying latent semantic relationships between terms and documents. LSI achieves this by dimension reduction, selecting the most important dimensions from a term occurrence matrix, such as the matrix in table 1, using Singular Value Decomposition (SVD). In the natural language text domain, LSI has outperformed standard lexical retrieval techniques [20], classified texts [22] and been shown capable of extracting significant levels of meaning from words, sentences and documents [23].

SVD [7] is a powerful technique in matrix analysis. Once we have created a item-user $m$ x $n$ matrix $A$, as described earlier, a rank$-k$ approximation of that matrix ($k < min(m, n)$) to $A$, $A_k$ is computed using SVD, as illustrated in figure 3. The SVD of the Matrix $A$ is defined as the product of three matrices; $A = U\Sigma V^T$, where $U$ represents the original row entries as vectors of derived orthogonal factor values, $V$ represents the original column entities in the same

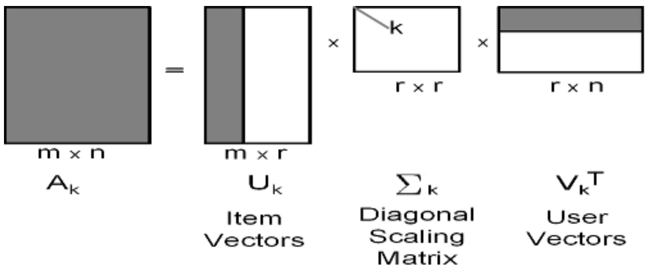


**Fig. 3.** Illustration of SVD. The shaded areas of $U$ and $V$, as well as the diagonal line of $\Sigma$, represent $A_k$, the reduced dimension representation of the original item-user matrix.

way and $\Sigma$ is a diagonal matrix containing scaling values such that when the three matrices are multiplied, the original matrix is reconstructed. As illustrated in figure 3, the first k columns of the U and V matrices and the first (largest) $k$ singular values of $\Sigma$ are used to construct a rank$-k$ approximation of $A$ via $A_k = U_k \Sigma_k V_k^T$.

$A_k$ is the $k-$dimensional approximation of the original item-user matrix. By reducing the dimensionality of this space, semantic relationships between users are revealed and much noise is thought to be eliminated. Thus care must be taken not to reconstruct $A$ when choosing the dimensionality. The optimal dimension is an open question and is usually determined experimentally; in our domain we found 499 to be the most appropriate dimension. Like VSM, we can calculate the similarity between any two users by determining the cosine of the angle formed by their vectors.

**Table 2.** Original Item-User matrix with LSI SVD applied. The matrix has been reduced in dimensionality from 5 to dimension 2 ($A_2 = U_2 \Sigma_2 V_2^T$).

| Item | User | User | User | User | User |
|---|---|---|---|---|---|
| | **U1** | **U2** | **U3** | **U4** | **U5** |
| JButton:getText | 1.32 | 0.69 | 0.16 | -0.11 | 0.53 |
| JButton:setText | 1.73 | 0.91 | 0.20 | -0.16 | 0.69 |
| JButton:setEnabled | 1.04 | 0.54 | 0.48 | 0.40 | 0.42 |
| JPanel:setLayout | 0.01 | -0.05 | 2.93 | 4.05 | 0.06 |
| JPanel:grabFocus | -0.04 | -0.04 | 1.29 | 1.79 | 0.01 |

In table 2 we briefly illustrate the power of the LSI SVD technique. We have constructed a 2-dimensional approximation of the original item-user matrix in table 1. Taking user $u4$ as an example, we see the original item values have changed. Items $setText$ and $getText$ have now both taken on negative values while *setEnabled* now has a value of 0.40. This new value for *setEnabled* can be viewed as an estimate of how many times it would be used by each of an infinite set of users who also use *setLayout* and $grabFocus$. The negative values ensure user $u4$ will be less similar to users $u1$, $u2$ and $u5$ than may have previously been the case.

Queries are performed on the reduced dimension user vector, $V_k$; a smaller dimension can greatly increase query execution time. In the LSI model, queries are formed into *pseudo-documents* that specify the location of the query in the reduced document space [7]. Given a query vector $q$, identical to a VSM query vector, the pseudo-document, $\hat{q}$, can be represented by $\hat{q} = q^T U_k \Sigma_k^{-1}$.

Thus, the pseudo-document consists of the sum of the item vectors ($q^T U_k$) corresponding to the terms specified in the query scaled by the inverse of the singular values ($\Sigma_k^{-1}$).

**Weighting.** Term weighting is frequently applied in natural language processing; we investigate if such weighting is applicable to the source code domain. The most simple weighting scheme is a local weight. For non zero frequencies, this local weight is defined as $tf_{ij}$ (frequency user $i$ employs item $j$) dampened by the log function: local weight $= 1 + log(tf_{ij})$. In text documents, this reflects the fact that a term which appears in a document $x$ times more than another term is not $x$ times more important.

We extend this simple local weighting scheme to log-entropy weighting as recommended by [20]. Log entropy is local weighting times global weighting. Global weighting is defined as $1 - entropy$. The log-entropy item weight for item $j$ by user $i$ is:

$$log(1 + tf_{ij}) * \left[ 1 - \frac{\sum p \in I_j \left( \frac{tf_{pj}}{gf_j} * log \frac{tf_{pj}}{gf_j} \right)}{log(numUsers)} \right] \tag{1}$$

where $I_j$ is the set of all users who use item $j$, $tf_{ij}$ is the frequency of use of item $j$ by user $i$ and global frequency $gf_j$ is the total number of times item $j$ is used in the complete user set.

## 4.2 Collaborative Filtering

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 user's 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 are likely to agree on future items. Collaborative filtering algorithms are used in mainstream recommender systems such as *Amazon* [24]. In our work we use CF to recommend a candidate set of items to a user.

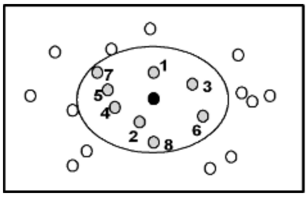


**Fig. 4.** Illustration of the $k$ Nearest Neighbour formation. The similarity/distance between the target user query and all users in the item-user matrix is computed and $k$ closest users are chosen as neighbours. $k = 8$ in this example.

**Recommendation Algorithm.** Recommendations are produced by examining the item-user matrix created using either VSM or LSI. Vote $v_{ij}$ corresponds to the vote by user $i$ for item $j$. The mean vote for user $i$ is calculated as follows:

$$\overline{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j} \tag{2}$$

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

$$cf_{aj} = \overline{v}_a + N \sum_{i \in kNN} sim\,(a,i)\,(v_{i,j} - \overline{v}_i) \tag{3}$$

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 4. 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 $k = 10$. $N$ is the normalising factor such that the absolute values of the weights' sum to unity. From equation 3 we can now predict a users' vote for any item in the user-item preference database. Items are ranked based on their predicted vote and the top $n$ items are recommended to the user. In our experiments, we use a value of $n = 7$.

We can calculate the similarity between the current user $a$ and any user in the item-user matrix, $sim(a,i)$, by determining the cosine of the angle formed by their vectors, as detailed in [25]. If we are using LSI, we can efficiently perform vector similarity on the reduced user space, $V_k$.

## 5 Experiments

### 5.1 Dataset

We produced over 32,000 recommendations for 1410 Java classes taken from over 60 GUI applications mined from Sourceforge [18]. Recommendations were produced at the method level, and not the class level as in previous work [26]; in total there was 3038 methods (users) or approximately just over 2 methods per class. Further to this, each user had originally invoked on average 11 methods (items). The items which we recommended were Swing and AWT methods; in total there was 2407 items. Since we have the complete source code, we can automatically evaluate the recommendations.

For each user, several recommendations were made. For example, if a fully developed method had 10 Swing invocations, then we removed the 10th invocation from that user and a recommendation set was produced for the developer based on the preceding 9 invocations. Following this recommendation, the 9th invocation was removed from that user and a new recommendation set was formed based on the preceding 8 invocations. This process was continued until just 1 invocation remained. Each recommendation set contained a maximum of 7 items.

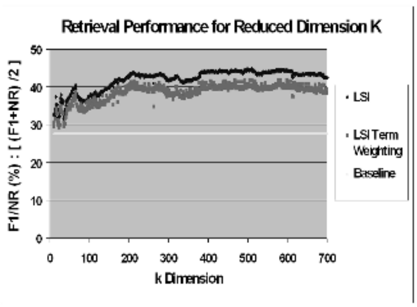
### 5.2 Evaluation

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; $P = n_{rs}/n_s$, 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 item is relevant. An item is deemed relevant if it is used by the user 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 item will be selected. Several approaches have been taken to combine precision and recall into a single metric. The $F1$ measure, initially introduced by van Rijsbergen [27], combines both with an equal weight in the following form: $F_1 = 2PR/\left(P+R\right)$.

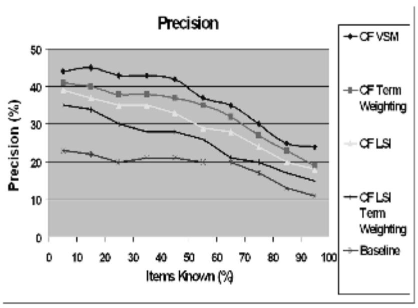
It is particulary important that RASCAL recommends items in a relevant order i.e. the invocation order. We will evaluate this using a simple binary Next Recommended (NR) metric; $NR = 1$ if we successfully predict or recommend the next method a developer will use, otherwise $NR = 0$.

## 5.3 Results

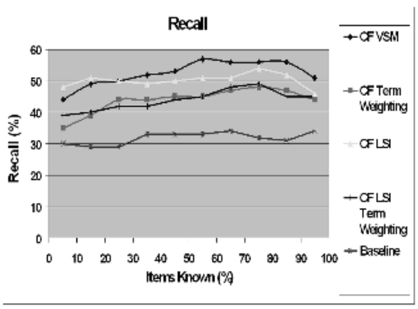
All results are displayed as a percentage value. A baseline result is included; these were produced by recommending the top 5 most commonly invoked items at each recommendation stage. We display the $F1$ metric combined with the $NR$ metric for several different dimensions $k$ in figure 5(a). This is the average $F1$ and $NR$ result for various stages of recommendation, i.e. when $x\%$ of items are known. Without applying LSI SVD, the original dimension of the user-item matrix was 2407. We find that applying relatively low dimensions can produce



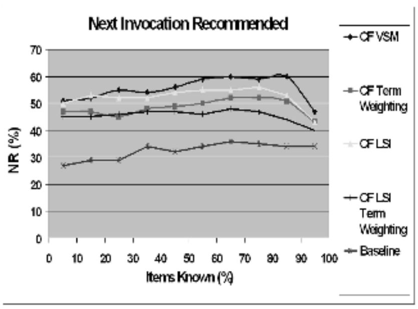
(a)



(b)



(c)



(d)

**Fig. 5.** (a) K dimension (b) Precision (c) Recall (d) Next Recommended ($NR$)

reasonable recommendations. The optimal value for $k$, based on our dataset, is 499; this is the value used in the below experiments. We notice from figure 5(a) that using term weighting has a negative effect on recommendations.

Figure 5(b) displays average Precision for four experiments; collaborative filtering with and without term weighting using either LSI or VSM. The pure VSM CF algorithm performs best, producing better results than the LSI model. Excluding, term weighting, the average CF VSM result is 36% compared with just 30% when using LSI. Precision, for all techniques, decreases as more items are known; we discuss this in the following section. Both LSI and VSM produce significantly better precision values than our baseline technique. We present Recall in figure 5(c). Like precision, the CF VSM produces the best retrieval, averaging at 52%. The result is followed closely by LSI where the average recall value is 50%. Term Weighting performs poorly. Figure 5(d) displays Next Recommended ($NR$). Again, the CF VSM produces the best retrieval, averaging at 55%. LSI performs well here with an average $NR$ value of 52%.

### 5.4 Discussion

We make several interesting observations from these experiments. Firstly we note that applying the log-entropy term weighting scheme to the item-user matrix has a consistent negative effect on the recommendation results. This suggests that items which are used by many users are as important as items which are used by only a small number of users. To verify this, manual experimentation is required. We also find that pure CF recommendations consistently outperform CF LSI recommendations. However, it is important to recognise several other benefits of using LSI; most notably performance efficiency which is crucial in a realtime recommender. Using LSI with reduced dimension $k = 499$, RASCAL initialises twice as fast and produces recommendations approximately three times faster than the VSM approach. This will be important as we scale up our application.

Generally, we notice two different trends in precision and recall. Precision tends to decrease as we know more information about a user while recall tends to increase. This result perhaps requires clarification. Consider a user who uses in total 10 items. When we make a recommendation for that user when they have only used 1 item, there is a set of 9 possible items to recall. The chances of recalling all relevant items is quite low and hence the recall result is low in earlier recommendations. However, when this user has used 9 items and there is only 1 possible item to recall, then the chances of this item being in the recommendation set is quite high. In contrast, the more items we know about the current user, the fewer there are to correctly recommend and hence precision decreases in latter recommendations.

## 6 Conclusions

Just as people can be clustered in terms of their preferences for various items, Java methods may also be clustered based on the methods they invoke. To clusters methods, we investigated and compared two information retrieval techniques, namely the vector space model and latent semantic indexing and found the VSM most effective. Unlike many retrieval schemes, we found that preprocessing or weighting of items negatively impacted retrieval. We also noted some of the limitations with using VSM such as scalability and performance times, and we explained how LSI can overcome these challenges.

Further work is needed to enhance RASCAL. Using LSI, we will investigate significantly increasing the size of the library; we would expect this to improve precision and recall whilst having a small impact on performance times. We will also investigate the use of probability models to produce recommendations. RASCAL offers unsolicited advice and we must be sensitive to this in our delivery of recommendations. We will extend our Eclipse plugin, complementing and extending the existing context-sensitive list of methods recommended by the Eclipse IDE. Our overall goal is to develop a recommender that seamlessly integrates with the Eclipse IDE but more importantly allows reuse to become a natural and convenient part of a developers daily routine.

Recommender systems are a powerful technology that can cheaply extract knowledge for a software company from its code repositories and then exploit this knowledge in future developments. We have demonstrated that RASCAL offers real promise for allowing developers discover and easily access reusable library components. When little information is known about the user we can nevertheless make reasonably good recommendations and it is our belief that future work will strengthen both recommendation accuracy and performance.

## References

1. Mohagheghi, P., et al.: An empirical study of software reuse vs. defect-density and stability. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, (IEEE Computer Society) 282–292
2. Yongbeom, K., Stohr, E.: Software reuse: Survey and research directions. Management Information Systems **14**(4) (1998) 113–147
3. Ye, Y., Fischer, G.: Reuse-conducive development environments. International Journal of Automated Software Engineering **12** (2005) 199–235
4. Poulin, J.: Reuse: Been there done that. Communications of the ACM **42**(5) (1999)
5. Inoue, K., et al.: Component rank: relative significance rank for software component search. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 14–24
6. Sarwar, B.M., Karypis, G., Konstan, J.A., Reidl, J.: Item-based collaborative filtering recommendation algorithms. In: World Wide Web. (2001) 285–295
7. Letsche, T.A., Berry, M.W.: Large-scale information retrieval with latent semantic indexing. Inf. Sci. **100**(1-4) (1997) 105–137
8. Landauer, T., Foltz, P., Laham, D.: An introduction to latent semantic analysis. Discourse Processes **25** (1998) 259–284
9. Deerwester, S., et al.: Indexing by latent semantic analysis. Journal of the American Society for Information Science **41** (1990) 391–407
10. Prieto-Diaz, R., Freeman, P.: Classifying software for reuse. IEEE Software **4**(1) (1987) 6–16

11. Mili, A., Mili, R., Mittermeir, R.T.: A survey of software reuse libraries. Annals of Software Engineering **5** (1998) 349–414
12. Sugumaran, V., Storey, V.C.: A semantic-based approach to component retrieval. SIGMIS Database **34**(3) (2003) 8–24
13. Girardi, M., Ibrahim, B.: Using english to retrieve software. Journals of Systems and Software **30**(3) (1995) 249–270
14. Drummond, C.G., Ionescu, D., Holte, R.C.: A learning agent that assists the browsing of software libraries. IEEE Trans. Softw. Eng. **26**(12) (2000) 1179–1196
15. Sarwar, B.M., et al.: Application of dimensionality reduction in recommender systems–a case study. In: Proceedings of ACM WebKDD Workshop. (2000)
16. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 125–135
17. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, IEEE Computer Society (2001) 107
18. Ebert, J.: Storm - a user story tool. `http://xpstorm.sourceforge.net`. (2002)
19. Apache: Apache software foundation - bytecode engineering library (2002-2003). `http://jakarta.apache.org/bcel/index.html`. (2003)
20. Dumais, S.: Improving the retrieval of information from external sources. Behavior Research Methods, Instruments and Computers **23**(2) (1991) 229–236
21. Dumais, S.: Latent semantic indexing (lsi) and trec-2. The Second Text REtrieval Conference (TREC2), National Institute of Standards and Technology Special Publication 500-215. (1994) 105-116
22. Zelikovitz, S., Hirsh, H.: Using lsi for text classification in the presence of background text. In: CIKM '01: Proceedings of the tenth international conference on Information and knowledge management, New York, ACM Press (2001) 113–118
23. Berry, M.: Large scale singular value computations. Int. Journal of Supercomputer Applications **6** (1992) 13–49
24. Bezos, J.: Amazon.com plc. seattle, wa 98108-1226, usa `www.amazon.com`. (2004)
25. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. In: Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence. (1998) 43–52
26. McCarey, F., Cinnéide, M.O., Kushmerick, N.: Knowledge reuse for software reuse. In: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering. (2005)
27. van Rijsbergen, C.: Information Retrieval. Butterworths, London (1979)