Recommending Library Methods: An Evaluation of Bayesian Network Classifiers

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

Programming tasks are often mirrored inside an organisation, across a community or within a specific domain. We propose that final source codes can be mined, that knowledge and insight can be automatically obtained and that this knowledge can be reused for the benefit of future developments. We focus on reusable software libraries; we wish to learn information about how such libraries are used and then elegantly pass this information onto individual developers.

In this paper we investigate a Collaborative Filtering approach of recommending library methods to a individual developer for a particular task. The central idea is that we find source codes that are the most relevant to the task at hand and use these to suggest useful library methods to a developer. To determine the similarity and relevance of source code, we investigate and compare a number of Bayesian clustering techniques including Bayesian Networks and Naïve-Bayes. We present results and discuss the suitability of Bayesian networks to this domain.

1. Introduction

A healthy knowledge flow between programming peers can positively impact personnel morale, team productivity and the ultimate outcome of a project. Be it a new developer just added to the team or the experienced professional unfamiliar with a particular library, all can benefit from the experiences and skills of others. The tools and techniques used to share such information within organisations can vary greatly; for example, colleagues may hold informal meetings, telephone or email each other or perhaps rely on detailed support materials. Though these techniques may be effective, it is clear that they lack efficiency. For both the requestor and the responder, there is the overhead of task switching; just replying to an email may upset the flow of ones primary task. Similar to Ye and Fischer [28], we propose that much of this knowledge can be shared automatically through the provision of proper tool support.

In this paper we focus on tool support for software libraries. Reuse of such libraries has been shown to improve software quality and developer productivity whilst reducing defect density [20] and time-to-market [29]. It is impractical though to consider that any one individual would be entirely familiar with any one library; for example, the latest version of the Java API library has over 3000 classes while the Java Swing library has over 500 classes. Over a period of time, it is likely that many different programmers will have used a particular library. We suggest that insight can be gained from analysing how particular libraries are used and that this knowledge can be passed onto individual programmers through intelligent support tools; we are currently developing the RASCAL tool.

RASCAL is a proactive recommender that is designed to support library reuse. RASCAL hopes to address several of the pragmatic issues that currently hamper reuse; for example, developer motivation, time constraints, library accessability and lack of conversancy for a particular library. RASCAL currently recommends a set of library methods to a developer which it believes to be relevant to the task at hand. We propose that by identifying and recommending reusable methods from a library and subsequently facilitating quick access to these, we will foster and encourage reuse.

Similar to many commercial recommenders, RASCAL produces 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 methods that the developer is interested in. Recommendation in our tool is complicated though because we wish to recommend methods which we believe the developer may be unfamiliar with or unaware of. Another interesting distinction between our recommender

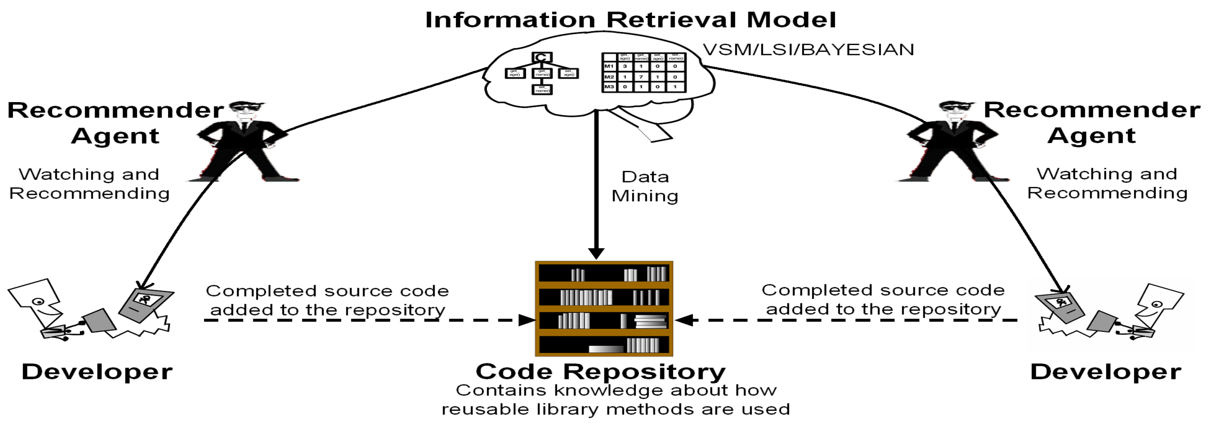

Information Retrieval Model
VSM/LSI/BAYESIAN
Recommender Agent
Watching and Recommending
Recommender Agent
Watching and Recommending
Data Mining
Completed source code added to the repository
Completed source code added to the repository
Developer
Code Repository
Contains knowledge about how reusable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

Figure 1: RASCAL Overview

system and most mainstream recommenders is that we are trying to predict, in order, the next likely method 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.

Recommendations are produced using a Collaborative Filtering (CF) [25] algorithm as explained in section 3. An important aspect of CF algorithms is clustering users; in this paper we investigate and compare a number of Bayesian approaches that can be used to classify how similar source codes are. In particular, we will detail Bayesian Networks, Naïve-Bayes, Tree-Augmented Naïve Bayes, Forest-Augmented Naïve Bayes and finally Bayesian Network Augmented Naïve-Bayes.

The main contributions of this work are:

- A viable solution to domain knowledge sharing, in respect of software reuse libraries.
- A technique embedded in the RASCAL support tool that significantly enhances reuse.
- An investigation of how effectively Bayesian techniques can be applied to source code. We use these techniques to support reuse but in theory several other tasks could be supported such as clone detection, code modeling and categorisation.

The remainder of this paper is organised as follows. In the next section we provide a brief overview of the main components in RASCAL. This is followed by a detail explanation of the recommendation algorithm and a comparison of a number of different Bayesian techniques in section 3. Section 4 presents experimental results with discussion. Related works are reviewed in section 5. Finally we discuss how RASCAL can be extended and draw general conclusions in section 6.

2. System Overview

RASCAL is currently implemented as a plugin for the Eclipse IDE. As a developer is writing code, RASCAL monitors the methods currently invoked and uses this information to recommend a candidate set of methods to the 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. Below we describe the 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. 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. The **Code Repository** contains code from previous projects, external libraries, open-source projects etc; in our work we used the Sourceforge [8] repository. This repository will be continually updated as new classes/systems are developed. From such a repository, we can extract information about what reusable library methods exist and also knowledge about how these are used. We produce an **Information Retrieval Model** by mining the code repository; the actual information retrieval model used can vary as discussed in section 3.2. This model will need to be created once initially and subsequently when a new piece of source code is added to the repository. We extract information from the repository using the *Bytecode Engineering Library* [1].

Finally there will be a **Recommender Agent** for each individual developer; this agent actively monitors the method that the developer is coding. The agent then uses the information retrieval model to establish a set of source codes that are most similar to the code currently being written by the developer and following this, a 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 recommendation technique in full in the following section.

3. Recommendations

3.1 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 [25]. 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*. In our work we use CF to recommend a set of library methods to a developer.

For clarity we describe three terms, specific to this context, that are common terminology in recommender literature. An **item** refers to a reusable library method. We wish to predict a developers preference for an item. A **user** is a Java method in our source code repository. The active user can be considered as the method currently being written or indeed the actual developer of that method. Finally a **vote** represents a users' preference for a particular item. In this context, a vote is simply an invocation count for a particular library method.

3.1.1 Recommendation Algorithm

Breese et al. [3] identify two classes of CF algorithms, namely Memory-Based and Model-Based. In a memory-based approach, a prediction for the active user is based on the opinions of like-minded users. In contrast, model-based CF first learns a descriptive model of user preferences and then uses it for predicting ratings. Employing a memory-based algorithm, vote v_{ij} corresponds to the vote by user i for item j (invocation count in this work). 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 CF for the active user a on item j, cf_{aj}, is a weighted sum of the votes of the other similar users:

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

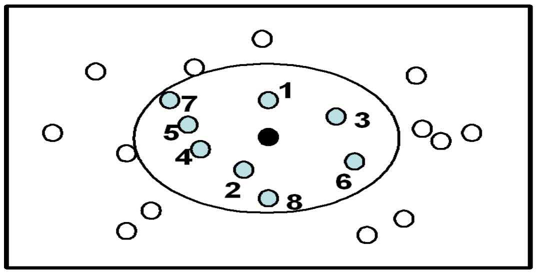


Figure 2: Illustration of the kNN formation. Here we look for the active methods' k=8 most similar source codes.

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 2. 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 2 we can now predict a users' vote for any item. In the context of this work, we can now predict a developers' vote for any library method assuming that there exists at least one snippet of code in the code repository that has 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. The Information Retrieval (IR) model chosen will have a direct impact on which users are deemed relevant and which are not, and thus ultimately impacts the recommendation set. Baeza-Yates and Ribeiro-Neto [2] identify three basic retrieval models; boolean, vector/statistical and probabilistic.

In previous works we have investigated the suitability of vector approaches in the software component recommendation domain [4]; namely we looked at the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) and found VSM to produce the best results. Here we investigate how effective probabilistic approaches are at ranking source code based on similarity. This is equivalent to classification in machine learning; however, we are attempting to classify the top n pieces of code that are most similar to the active method being written. Typically statistical approaches are used for memory-based algorithms while probabilistic techniques are used with model-based algorithms. In this work, we employ a hybrid approach akin to the work of [23]. Like the model-based technique, we construct a Bayesian network though we treat each method as a unique cluster and therefore when making a prediction, we need to consider all methods in the code repository.

3.2 Bayesian Network Classifiers

A **Naïve-Bayes** BN [7, 16] is a simple structure that has the classification node as a parent of all other attribute nodes. 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 would represent a particular piece of code from the code repository, whereas an attribute node represents each reusable library method that can be invoked. The conditional probability of each attribute given the class C is learnt from training data. Classification is then done by applying Bayes rule to compute the probability of C given a particular instance of attributes and then predicting the class with the highest posterior probability. In this work, we wish to determine the top kNN pieces of code that are most similar to the query instances.

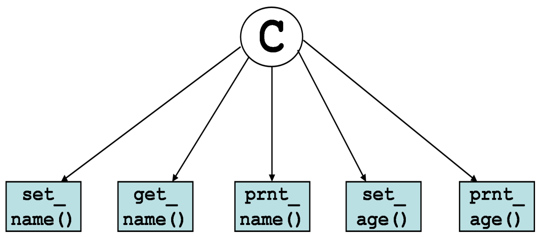


Figure 3: Naive-Bayes Network

Despite the Naïve assumption of probabilistic independence between attributes, Naïve-Bayes classifiers in general work reasonably well; indeed they have been shown to outperform BN [9]. This is surprising given that the attribute assumption rarely holds in real world examples. In our domain we might expect that there would be a relationship between at least some of the methods in the reusable library; we investigate if the Naïve-Bayes BN can effectively classify source code whilst ignoring such relationships. Figure 3 displays an example of Naïve-Bayes Network.

A general **Bayesian Network** (BN) [22] is a much more powerful representation of probabilistic dependencies over a set of random attributes; a BN can effectively model the complex dependencies that exist in most real world problems. More formally, a BN is a directed acyclic graph with nodes representing attributes and arcs representing dependence between relations among the attributes. Probabilistic parameters are encoded in a set of tables (Conditional Probability Tables), one for each attribute node, in the form of logical conditional distributions of a attribute given its parents. Using the independence statements encoded in the network, the joint distribution is uniquely determined by these logical conditional distributions. Figure 4 displays an example of a general BN; unlike Naïve-Bayes the classification node is treated the same as the attribute nodes. As is suggested by Cheng and Greiner [5], this lack of distinction between the classification and attributes nodes is not always desirable in certain domains and thus we introduce Bayesian Networks Augmented Naïve-Bayes shortly.

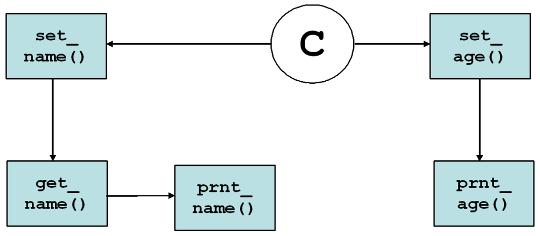


Figure 4: Bayesian Network

Learning a BN based classifier is a computationally challenging problem; if the network is unrestricted then it is a NP-hard problem. We need to find a network that best matches the entire instances in the training data. Using a scoring function we need to evaluate each learnt network against the training data and determine the optimal network.

Several authors have proposed a compromise between the computationally expensive Bayesian network model and the over-simplified Naïve-Bayes approach. The desire is to merge the ability of BN to model attribute dependence with the simplicity and efficiency of Naïve-Bayes BN. Friedman et al. [9] define such structures as Augmented Naïve Bayesian Networks. Each attribute must have a class attribute as a parent and each attribute may have one other parent [15]. From figure 5, it can be seen that it is now possible to model dependency between attributes whilst maintaining the simplicity of the Naïve-Bayes BN. In general, as stated earlier, learning an unrestricted network is a NP-Hard problem. Friedman et al. [9] deal with this by restricting the network to a tree topology; the result is known as a **Tree Augmented Naïve-Bayes** (TAN) as is specifically shown in figure 5. There is an arc from $getName()$ to $setName()$ and thus these two attributes are not independent given the class.

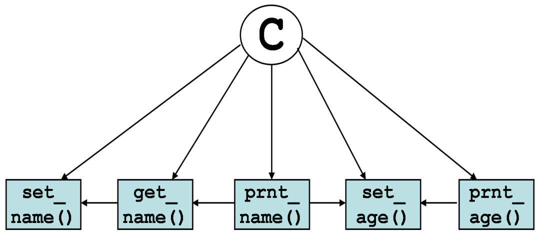


Figure 5: TAN

Keogh and Pazzani [15] present a similar tree augmented network but unlike TAN, which adds $N - 1$ arcs (where N is the number of attributes), they add any number of arcs up to $N - 1$. An arc is only added if it improves accuracy. This same approach is defined by Sacha [24] as a **Forest-Augmented Network** (FAN), as the augmenting arcs form a forest of attributes (or a collection of trees); this is illustrated in figure 6.

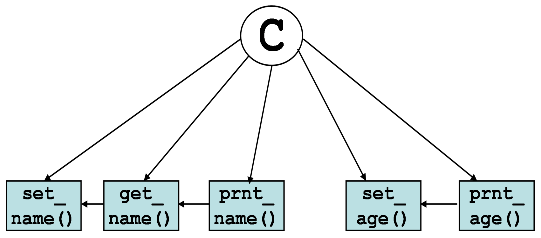


Figure 6: FAN

The final BN we consider is the **Bayesian Network Augmented Naïve-Bayes** (BAN). This extends TAN by allowing attributes to form an arbitrary graph, rather than just a tree, as is shown in figure 7. This is similar to the original general BN but in this case the classification node is treated differently from the rest of the attribute nodes. It is hoped that the BAN will more richly model relationships between attributes but this will likely come at a computational cost. A more detailed comparison of Bayesian networks can be found in [5].

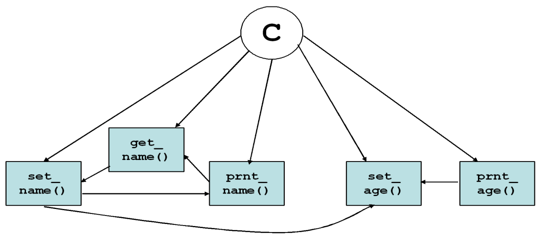


Figure 7: BAN

Excluding general BN's and FAN's, all the above networks were constructed using the popular WEKA [27] machine learning tool. We used a repeated hill climbing searching algorithm (maximum of 5 runs) and the BDeu scoring function. As general BN's do not distinguish between class and attribute nodes, we decided to implement the more efficient BAN instead; the number of parent nodes was limited to 4. For the FAN implementation we used the Java Bayesian Network Classifier (JBNC) toolkit [14]. All training data was normalised and discretised to have 3 values; for example, the method $setName()$ may be invoked either between 0 and .33 times, between 0.34 and 0.66 times or finally between 0.67 to 1 times.

4 Experiments

4.1 Dataset

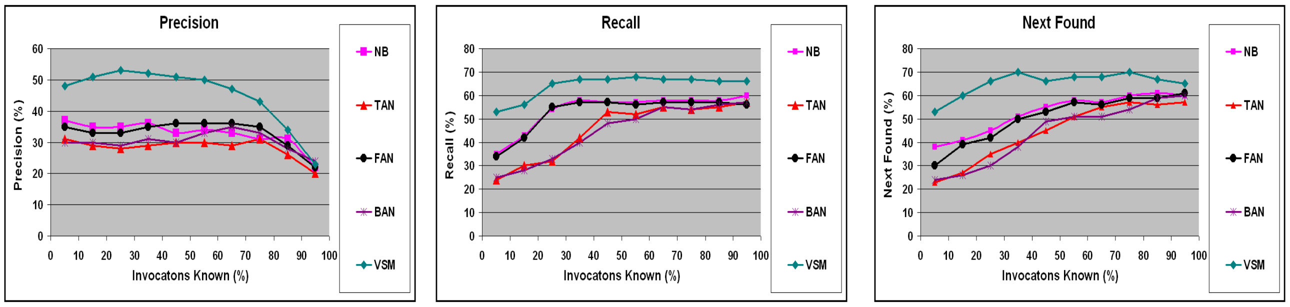
In these preliminary experiments, we used relatively small datasets. We produced almost 6000 recommendations from approximately 350 methods mined from Sourceforge [8]. Recommendations were produced solely at the method level and not the class level as in previous work [18]. Further to this, each method had on average 16 invocations. Recommendations were made for both the SWING and AWT libraries; in total there was 697 Swing and AWT library methods that were invoked at least once in our code repository. Although the data is small for this domain, 697 instances and 350 classes is comparatively large with experiments carried out in machine learning literature. Since we have the completed source code, we can automatically evaluate recommendations for a piece of code by checking whether the recommended method was called subsequently.

For each of the 350 methods, 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 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.

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; $P = n_{rs}/n_s$, 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.

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 $NR = 0$. In these investigative experiments we focused



(a) (b) (c)

Figure 8: (a) Precision (b) Recall (c) Next Found

solely on the above 3 metrics whilst ignoring computational complexity.

4.3 Results

All results are displayed as a percentage value. A baseline result is included; this was produced using the Vector Space Model (VSM) as detailed in previous work [4]. From figure 8, it is immediately identifiable that the VSM baseline result produced the best results in general. While this may not have been the desired outcome, there is still insight to be gained from the results.

Precision is displayed in figure 8(a). VSM vastly outperforms all of the Bayesian techniques; for example, the average FAN precision is 33% which compares poorly with 45% when using VSM. Recall is shown in figure 8(b); again VSM outperforms all other techniques. We notice that Naïve-Bayes (NB) and the Forest-Augmented Network (FAN) produce similar results and that these are both marginally better than the Tree Augmented Network (TAN) and the BN Augmented Naïve-Bayes (BAN).

The next found metric is displayed in figure 8(c). Using NB, there is a 48% likelihood that RASCAL would be able to correctly predict the next library method that a developer would invoke; such a prediction would provide significant help to a developer who was unfamiliar with a particular library. In general, this is an encouraging result yet it is relatively poor when compared with the VSM 64% average.

4.4 Discussion

From this exploratory research on using Bayesian networks to recommend library methods, we can make some interesting observations. Firstly we discover that FAN and NB produce similar results for all metrics. This would suggest that the FAN added very few links as these did not improve classification. We also notice the similarities between TAN and BAN; again this would suggest that the BAN was very similar to the TAN created and that there is no benefit to having multiple parents. In the context of this work, this can be interpreted as there being very few relationships between library methods and hence a Naïve-Bayes network will produce better recommendations. Further work is needed to verify this. In addition to this, further investigation is needed in the area of searching and scoring techniques to ensure they are ideally suited to this domain.

Generally, we notice two different trends in precision and recall. Precision tends to decrease as we know more information about the active method while recall tends to increase. This result perhaps requires clarification. Consider a developer who invokes in total 10 methods. When we make a recommendation for that developer when they have 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.

5 Related Work

Traditional retrieval schemes focused generally on techniques such as *Keyword Search* and *Signature Matching* [19]. More recently several *Semantic-Based* retrieval tools have been proposed [26, 10]; these allow a developer to specify queries using natural languages. Unlike traditional retrieval, the domain information, developer context and component relations are considered. Empirical results indicate that these tools are superior to traditional approaches.

ComponentRank [13] is a promising component retrieval technique which is useful for locating reusable components. Similar to *Google* [21], this approach ranks components 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. Similarly, Hummer and Atkinson [12] 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 have been proposed by Drummond et al. [6]. 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 components or methods in a library. 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 [28] 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 but 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. 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.

Mandelin et al. [17] present an intelligent tool for understanding and navigating the API of a particular reuse library. They suggest developers often know the objects they would like to use but are unaware of how to write the source code to get the object; 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 representation or traversing object schemas.

Another notable tool for finding code examples is Strathcona [11]. 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* [28], 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 CF which is similar to the example based techniques of Holmes and Murphy [11]. Like the *PROSPECTIVE* tool, 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. [6] 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.

6 Conclusions

We have presented a solution that automatically facilitates 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 note the importance of correctly identify the optimal technique for clustering source code; we investigated a number of Bayesian techniques and compared these with our VSM statistical baseline result.

In this work, we discovered conclusively that Bayesian Networks are less useful at clustering source codes than VSM and ultimately have a negative effect of recommendation performance. Further and larger experimentation is needed to generalise this finding though; in particular we need to evaluate more search and scoring techniques. Bayesian techniques do still offer promising opportunities for us; for example, modeling relationships between library methods, classification or clustering of library methods as opposed to classifying entire source codes as is presently done and finally applying the discussed Bayesian techniques to pure model-based CF.

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 proactive and constantly suggests library methods to reuse.

Recommender systems are a powerful technology that can cheaply extract knowledge for a software company from its code repositories and then share this knowledge to the benefit of future developments. We have demonstrated that RASCAL offers real promise for allowing developers discover and easily access reusable library components but that care needs to be taken when choosing the clustering technique.

7 Acknowledgements

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology (IRCSET) under grant RCS/2003/127.

References

[1] Apache. Bytecode engineering library (2002-2003). `http://jakarta.apache.org/bcel`. 2003.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] 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*, pages 43–52, 1998.

[4] F. M. Carey, M. O. Cinnéide, and N. Kushmerick. Recommending library methods: An evaluation of the vector space model (vsm) and latent semantic indexing (lsi). In *9th International Conference on Software Reuse*, Italy, 2006.

[5] J. Cheng and R. Greiner. Comparing bayesian network classifiers. In *Proceedings of UAI*, pages 101–108.

[6] C. G. Drummond, D. Ionescu, and R. C. Holte. A learning agent that assists the browsing of software libraries. *IEEE Trans. Softw. Eng.*, 26(12):1179–1196, 2000.

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

[8] J. Ebert. Storm - a user story tool. `http://xpstorm.sourceforge.net`. 2002.

[9] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[10] M. Girardi and B. Ibrahim. Using english to retrieve software. *Journals of Systems and Software*, 30(3):249, 1995.

[11] 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*, pages 117–125, New York, NY, USA, 2005. ACM Press.

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

[13] 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*, pages 14–24. IEEE Computer Society, 2003.

[14] J.P.Sacha. Java bayesian network classifier (jbnc) toolkit. `http://jbnc.sourceforge.net`. 2004.

[15] E. Keogh and M. Pazzani. Learning augmented bayesian classifiers: A comparison of distribution-based and classification-based approaches, 1999.

[16] P. Langley, W. Iba, and K. Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.

[17] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. *SIGPLAN Notices*, 40(6):48–61, 2005.

[18] 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*, July 2005.

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

[20] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA. IEEE Computer Society.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[22] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[23] D. M. Pennock, E. Horvitz, S. Lawrence, and C. L. Giles. Collaborative filtering by personality diagnosis: A hybrid memory and model-based approach. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 473–480, CA, USA, 2000.

[24] J. Sacha. *New synthesis of Bayesian network classifiers and interpretation of cardiac SPECT images*. Ph.d. dissertation, University of Toledo, 1999.

[25] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *World Wide Web*, pages 285–295, 2001.

[26] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

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

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

[29] K. Yongbeom and E. Stohr. Software reuse: Survey and research directions. *Management Information Systems*, 14(4):113–147, Spring 1998.