A Case Study on Recommending Software Components using Collaborative Filtering

Mel Ó Cinnéide  Frank McCarey  Nicholas Kushmerick
University College Dublin - Ireland
frank.mccarey@ucd.ie
May 2004
Introduction

- Software Reuse is increasingly important to enterprises as they invest in developing and maintaining large software systems.

- Reusing software components can help develop better, faster and cheaper software systems [Griss, 1998].
Software Reuse Challenges

- Developers are not always eager to learn reusable component – *The Productivity Paradox*.

- Even if a developer is willing to reuse a component they may not be able to locate it in the component repository.

- As the repository of components grows, it is difficult to remain conversant with all components. Component access needs to be complemented with component delivery.
Motivation

Traditional methods for component search and retrieval can be classified into four categories [Mili et al., 1998]:

1. **Keyword Search**
2. **Faceted Classification**
3. **Signature Matching**
4. **Behavioral Matching**

**Semantic-Based Method Retrieval** [Sugurmaran et al., 2003]: Requirements are specified using natural languages.

If a developer believes a reusable component for a particular task does not exist then they are unlikely to query the component repository. Component delivery is required.
Related Work

- **CodeBroker** [Fischer et al., 2002]: Infers the need for a component (method) based on developer comments and method signature. Relies heavily on the components in the repository being correctly commented and the developer actively commented his/her code.

- [Ohsugi et al., 2002] propose a system for recommending useful functions, to a standard user, in application software such as MS Word which is based on collaborative filtering.
Our Technique

- A Recommender System based on Collaborative Filtering.

- A set of candidate software components (methods) which are likely to be useful to this individual developer are recommended.

- The system allows developers discover reusable software components in a Learn On Demand Fashion.
Collaborative Filtering (CF)

- 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.

- The goal of CF algorithms is to suggest new items or to predict the utility of a certain item for a particular user based on the user’s previous likings and the opinions of like minded users [Sarwar et al., 2001].

- A User refers to a Java class and an Item refers to a software component.
Collaborative Filtering (CF)

Recommendations for the active user, Class C, are based on the existing items used in class C and items used by similar users.
Data Mining

- We need to collect information about user preferences before we can create user clusters.

- Software repositories contain a wealth of valuable information. Usage of software components can be automatically extracted from these repositories of Java classes.

- This information can be used to establish similarities between users.
Repositories Used

- Repositories of open-source Java code, available from SourceForge were mined.

- This consisted of over 40 GUI Swing applications including the following:
  - JHome  
  - JAdmin  
  - TimeTrack  
  - Pooka  
  - Vex  
  - LumberMill  
  - ChordCast  
  - JSurfer  
  - JEdit  
  - JasperEdit  
  - JIV  
  - MDateSelecter
User Similarity

- Users (Java classes) can be clustered by examining the software components they use.

- Each user is treated as a vector; the vector holds a count for all components that the user can invoke.

- Similarity between two users can be computed by determining the cosine of the angle formed by their vectors. The cosine will fall in the range \([-1, 1]\).

<table>
<thead>
<tr>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
<th>Method 6</th>
<th>Method 7</th>
<th>Method 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

User A
Recommendations

1. Establish the components used by the active user.

2. Find the similarity between each user and the active user. Using the \( k \)-Nearest Neighbour algorithm, develop a set of the most similar users, i.e. the active users closest neighbours.

3. Produce a recommendation set based on the active users neighbours. The closer a neighbour is to the active user, the more influence it has on the recommendation set.
System Evaluation

- Experiments were carried out on 343 Java classes from over 40 GUI applications.

- A set of candidate Swing components was recommended for each class at various stages of development.
System Evaluation

Original Class

Class A

66% components known

Button b;
b.setText("Button");
b.setAlignmentX(10);
b.setAlignmentY(10);
}

Remove & Recommend

Class A

33% components known

Button b;
b.setText("Button");
b.setAlignmentX(10);
Get Neighbours
Recommendations
}

Remove & Recommend

Class A

classA{
    void method1(){
        Button b;
        b.setText("Button");
        b.setAlignmentX(10);
        Get Neighbours
        Recommendations
    }
}
System Evaluation

- Precision and Recall are the most popular metrics for evaluating information retrieval systems.

- *Precision*: The ratio of relevant recommended items to the total number of recommended items.

- *Recall*: The ratio of relevant items selected to the total number of relevant items.

- Usually a trade-off between two.
Results

Recommendation Accuracy

![Graph showing recommendation accuracy for Top 100 Classes and All Classes against the number of known components as a percentage. The graph indicates that the accuracy increases with the number of known components, peaking around 60% for Top 100 Classes and 50% for All Classes, and then decreases as the number of known components exceeds 60%. The x-axis represents the percentage of known components, while the y-axis represents precision. The graph also includes a legend indicating the lines for Top 100 Classes and All Classes.](image)
Results

Precision V Recall

![Graph showing the relationship between Precision and Recall](image)
Results

- The recommender system provides promising results.

- Based on top 100 classes; recommendation precision was over 40% when a developer had utilised between 10% and 20% of the total components they would actually use.

- As more users were added to the repository, recommendation precision increased at the expense of system speed. A greater number of users in the repository meant a greater chance of locating a similar user to the active user. However we don’t expect this trend of more users/greater precision to continue indefinitely.
Future Work

- Consider different granularities of similarities between classes. At present we only record method invocations for the entire class. We will extend this to record invocations at the method level.

- Create an intelligent IDE by developing a non-intrusive component recommender as an Eclipse plug-in.

- Provide a feature for explaining recommendations and example use of recommended components by code example.
Conclusions

- Our approach address various shortcomings of previous solutions to the component retrieval problem. Recommendations consider the developer and problem domain without placing any additional requirements on the developer.

- The recommender system extracts knowledge from existing code databases and then exploits this information in future developments.

- As seen, this approach offers real promise for allowing developers discover reusable components with minimal effort.
A Case Study on Recommending Software Components using Collaborative Filtering

Mel Ó Cinnéide  Frank McCarey  Nicholas Kushmerick
University College Dublin - Ireland.
frank.mccarey@ucd.ie
May 2004