Mining Software Repositories for Software Change Impact Analysis: A Case Study

Lile Hattori 1, Gilson dos Santos Jr. 2, Fernando Cardoso 2, Marcus Sampaio 2

1Faculty of Informatics – University of Lugano
Lugano – Switzerland

2Departamento de Sistemas e Computação – Universidade Federal de Campina Grande
Campina Grande, PB – Brazil

lile.hattori@lu.unisi.ch, {gilson,fernando,sampaio}@dsc.ufcg.edu.br

Abstract. Data mining algorithms have been recently applied to software repositories to help on the maintenance of evolving software systems. In the past, information about what classes changed together, obtained by mining software repositories, were used to guide future changes. We use this information to measure the possible impacts of a proposed change. In this paper we propose and compare two approaches for sorting impact analysis results that use two different data mining algorithms: Apriori and DAR. Even though Apriori is a classic and largely used algorithm, the case study shows that the approach with DAR is less complex and more suitable for measuring the impacts of a change.

1. Introduction

Source code repositories contain valorous information about the evolution of software systems along their development and maintenance phases. Recently researchers have been exploiting this information for different purposes, such as bug prediction [Panjer 2007] or analysis of the development team behavior [Joshi et al. 2007]. One of the most interesting aspects of the evolution of a software system that can be extracted from the repository is the history of changes. From that history, one can discover what changes occurred in the past, how they were implemented and what classes changed together.

In order to obtain the change history of a system, recent works have applied data mining algorithms to software repositories such as CVS [Tigris 2008] or Subversion [Foundation 2008]. Zimmermann et al. [Zimmermann et al. 2004] applied the Apriori algorithm [Agrawal and Srikant 1998] to obtain information about what entities changed together in the past and use it to guide future changes. Ying et al. [Ying et al. 2004] used the FP-Tree [Han et al. 2000] algorithm for proposing modification tasks.

We apply two data mining algorithms, Apriori and DAR (Disjunctive Association Rules) [Sampaio et al. 2008], to help the prediction of the impact caused by a change to the software. A change in the source code can be originated by a change in the requirements, bug fix, change in the architecture, among others. Before implementing such a change, it is important to measure the impact that it will cause, i.e. what other entities will have to be changed to maintain the software’s integrity. Software change impact analysis techniques [Lee 1999, Ryder and Tip 2001, Arnold and Bohn 1996] that are applied before the implementation of a change are too imprecise to be used for planning and measuring the effort to make such a change.
We affirm that the precision of impact analysis (IA) applied before the implementa-
tion of a change can be increased by using change coupling information extracted from
the repository [Hattori 2008]. Change coupling refers to what classes changed together
in the past and with which frequency [Fluri et al. 2005]. In this paper, we propose two
different approaches for mining software repositories to combine the change coupling
information with the results of a static change impact analysis technique. We compare the
results from the two approaches through a case study. The results showed that DAR was
more suitable than Apriori to address the problem of reducing the number of false IA
results.

The remainder of this paper is organized as follows. Section 2 describes other
works that explored historical change information to aid the evolution of a software sys-
tem. In Section 3, we describe our model for impact analysis that combines a classical
static IA technique with change coupling information. In Section 4, we detail the two
approaches for mining the repositories. The case study is presented in Section 5. Finally,
in Section 6, we draw the conclusions and future work on this field.

2. Related Work

Mining software repositories (MSR) is a recent and growing research field that aims to
extract information from repositories to understand and control the evolution of software
systems. Some of the most important topics that have been under investigation are: bug
prediction, understanding software infrastructure and architecture, understanding team
collaboration and understanding software changes.

Predicting, finding and fixing a bug are challenging activities that are difficult to
estimate. Joshi et al. present an approach that extracts information from bug database
to automatically predict the fixing effort, i.e., the person-hours spent on fixing a bug
[Joshi et al. 2007]. Panjer uses Weka toolkit [The University of Waikato 2008] to pre-
dict the time to fix a bug given only the basic information known at the beginning of
a bug’s lifetime [Panjer 2007]. In this approach, he uses the following algorithms: 0-
R, 1-R, Naive Bayesian Networks, Decision Trees, and Logistic Regression. Addition-
ally, automatic bug finding tools, such as FindBugs [Hovemeyer and Pugh 2004] or JLint
[citeJlint], tend to have high false positive rates: most warnings do not indicate real bugs.
Kim and Ernst prioritize warning categories by analyzing the software change history
[Kim and Ernst 2007].

It is important to continually keep track of the changes to the software’s source
code to be able to understand how its infrastructure and architecture are evolving and
avoid early software degradation. In this sense, Fluri et al. were the first to introduce
the notion of change coupling - classes that changed together in the past [Fluri et al. 2005].
Later this definition was used by D’Ambros et al. to understand the dependencies between
modules and also among a module and classes of another module [D’Ambros et al. 2008].
This approach helps on the identification of intensive coupling, what could mean an error
prone piece of code. Another approach, by Mizuno et al., uses text mining techniques to
detect fault-prone modules in a way that the source code modules are considered as text
files and are applied to the spam filter directly [Mizuno et al. 2007].

Another interesting issue is to understand how developers collaborate to develop
a software system. Weißgerber et al. describe three visualization techniques that help
to examine how programmers work together, e.g. if they work as a team or if they develop their part of the software separate from each other [Weißgerber et al. 2007]. Yu and Ramaswamy present a model to represent the interactions of distributed open-source software developers and utilize complete-linkage hierarchical clustering technique to derive developer roles on software projects [Yu and Ramaswamy 2007]. For a small-size and medium-size project, the developer roles can be basically divided into two types, core members and associate members. Finally, Minto and Murphy propose the Emergent Expertise Locator (EEL) that uses emergent team information to propose experts to a developer within their development environment [Minto and Murphy 2007].

The last and most important issue is related to approaches that investigate the software changes in a fine-grained level. These approaches are strongly linked with approaches that keep track of software changes in a higher level, e.g. architectural level. However, they analyze changes that happened in the past to aid change tasks in the future. Usually, they go into detailed information of the source code, correlating changes in method and field (entity) level. These approaches are shown in the following.

2.1. Understanding Software Changes

According to Fluri and Gall, an efficient way to overcome or avoid the negative effects of software aging is by placing change in the center of the software development process [Fluri and Gall 2006]. By doing it so, it is possible to verify the frequency of changes to the software, detect coupling in the source code, follow the evolution/degradation of its architecture and, consequently, improve the quality of the produced software.

In order to aid the activity of making a change, Zimmermann et al. apply the Apriori algorithm to the software repository to find association rules that informs what entities methods changed together in the past [Zimmermann et al. 2004]. The idea of the solution is to give the following tips to the programmer who is executing a change: “Programmers who changed method A also changed method B”. The solution is implemented as an Eclipse plug-in, called eROSE, and sorts the recommendations according to support and confidence values of each rule.

Ying applies FP-Tree algorithm to CVS in order to find change patterns - files that have changed together with a certain frequency [Ying et al. 2004]. From these change patterns, she recommends the revision of files that follow a certain pattern when the developer is performing a change task. She states that the motivation to apply data mining for recommending changes comes from the fact that, using only static analysis, it is difficult to identify dependencies among modules implemented in different programming languages.

Finally, Kagdi and Maletic believe that combining IA and MSR methodologies can result in an overall improved support for software change prediction. They propose two approaches to combine IA and MSR results: (i) disjunctive approach - measures precision and recall of the union of IA and MSR results; (ii) conjunctive approach - measures precision and recall of the intersection of IA and MSR results [Kagdi and Maletic 2007]. Their approach is very similar to ours in the sense that both combine IA and MSR methodologies to support software change. However, their models are too simple and have not been submitted to experimentation.
3. Software Change Impact Analysis

In this section we present our probabilistic software change impact analysis technique that combines IA with MSR to sort the results provided by IA based on historical change information. The technique is composed of: (1) static change impact analysis technique – identify possible impacted entities given a set of changes; (2) an extractor, responsible for extracting all committed changes from the repository; and (3) a theoretical model, based on Bayes theorem, to combine IA results with information extracted from the repository.

Figure 1 shows the overview of the probabilistic IA technique. The inputs are: change set, software’s current source code version, repository. The output is the impact set sorted from the most probable to the least probable impacted entity. In the following we show how the impact set is calculated, how the change history is extracted from the repository and, finally, how we combine these two pieces of information to calculate the sorted impact set. For detailed information about this technique, refer to [Hattori 2008].

3.1. Impact Analysis Tool

Our impact analysis tool, called Impala, analyzes specifically object-oriented systems implemented in Java. Impala is responsible for identifying possible impacted entities given a proposed change set to the software. Note that it analyzes the software before the execution of the changes.

It takes as input the proposed change set and the software’s current source code to give as output an impact set. The change set is composed of a certain number of entities, i.e. class, method or field, to be changed in a modification task.

In order to search for possibly impacted entities, Impala first produces from the source code a graph that represents the system. In this graph, the nodes are entities and the arcs are relationships between two entities. These relationships can be of the following types: instanceOf, contains, extends, implements, isAccessedBy, isInvokedBy, and their inverse relations.

After producing the graph, Impala searches for dependencies of each change from the change set individually and according to the change type. The change types can be:
add, remove, change or change visibility (public, protected, package or private) of class, method or field, and add inheritance, remove inheritance of class. So, for each change in the change set, Impala exhaustively searches for entities that depend on the entity that will change, according to the change type. For example, if a method will be removed, Impala searches for its callers (inverse relation of isInvokedBy), for the callers of its callers and so on.

The result is a list containing the change set and all entities possibly impacted from each change in the change set. Note that this list contains many false-positives, e.g. entities that will not be impacted by a change. Figure 2 illustrates one example where method ClassD.m4() is removed. Impala puts ClassC.m3(), ClassB.m2() and ClassA.m1() in the impact set, because it does not know what changes will be performed in ClassC.m3() and if these changes will impact in ClassB.m2() and ClassA.m1(). However, after removing ClassC.m4(), the developer only changed the reference of ClassC.m3() from ClassD.m4() to ClassE.m5(). This means that, in practice, ClassA.m1() and ClassB.m2() were not impacted, so they are called false-positives. The main goal of our probabilistic approach is to eliminate these false-positives.

![Figure 2: Example of a method removal; a developer removes ClassD.m4() and changes a method call from ClassD.m4() to ClassE.m5()](image)

### 3.2. Change Extractor

The Change extractor, called **Impala Plug-in**, is an Eclipse plug-in that extracts from CVS information concerning all and every change to the software in the past. Consider “past” the time from the beginning of the codification until the last commit before the software version used for the IA presented in Section 3.1.

Impala Plug-in pre-processes CVS data automatically. It accesses the software’s CVS and, for each class, extracts all structural changes, organizes these changes into commit transactions, and outputs a matrix containing information of what structural changes were committed together in the past.

Structural change is any syntactic change to the source code, e.g. change to a method’s signature, addition or deletion of a class, deletion of an inheritance relationship. Changes in comments, in license term, or formatting changes are not considered structural changes, because they do not modify the functioning of the system. To extract the structural changes, Impala converts every revision of a class into an abstract syntax tree (AST), compares every two subsequent revisions and stores every single structural change that occurred from one revision to another.

After extracting all structural changes, Impala Plug-in organizes them into commit transactions. The definition of a commit transaction is: changes that were committed by the same author, with the same log message, within the same time window.
In this solution we use a sliding time window of 200 seconds. The result is a matrix $H_{[T \times |E|]}$, where:

$$T = \{t_1, t_2, ..., t_n\}, \text{ where } n \text{ is the total of commit transactions;}$$

$$E = \{e_1, e_2, ..., e_m\}, \text{ where } m \text{ is the total of entities in the system.}$$

Matrix $H$ is composed of 0 and 1 values. Given $E = \{e_1, e_1, e_3, e_4, e_5\}$, suppose $t_1 = \{01010\}$. This means that entities $e_2$ and $e_4$ were committed together in commit transaction $t_1$.

### 3.3. Combining Model

The idea of our theoretical model is to calculate, for each entity in the impact set, the probability of its impact based on the frequency that this entity was committed together with the entities from the change set. After calculating the probability, we sort the impacted entity and eliminate from the impact set those entities that have 0% of probability to be impacted.

To calculate the probability, we formally define change set and impact set:

- $H_{[T \times |E|]}$ – matrix defined in Section 3.2;
- $C = \{e_1, e_2, ..., e_i\}$ – change set;
- $I = \{e_1, e_2, ..., e_j\}$ – impact set;
- $C \cap E = \emptyset$;

For each $e_j$ in the impact set, we apply Bayes Theorem:

$$P(e_j|C) = \frac{P(C|e_j)P(e_j)}{P(C|e_j)P(e_j) + P(C|\overline{e_j})P(\overline{e_j})}$$

Interpreting the formula, the nominator is the conditional probability of any entity in $C$ to be committed with $e_j$. The denominator is the probability of any entity in $C$ to be committed regardless if $e_j$ is committed together. If entities of $C$ and $e_j$ were committed together many times in the past, $P(e_j|C)$ will be high. If entities of $C$ were not committed with $e_j$ very often, $P(e_j|C)$ will be low. If every time that $e_j$ was committed, at least one entity of $C$ was committed together, $P(e_j|C) = 1$. If every time that $e_j$ was committed none of the entities of $C$ was committed together, $P(e_j|C) = 0$.

### 4. Approaches for Sorting Impact Analysis

In the last session we presented a theoretical model to calculate the probability of an entity to be impacted by a change, based on historical information about what entities changed together in the past. However, the matrix given by the change extractor (Section 3.2) does not explicitly inform what entities changed together and with what frequency. The information is there, but still needs to be mined. Therefore, we have to apply a data mining algorithm to find these change couplings. In this section, we propose two approaches for mining software history, represented as matrix $H$, and sorting the impact set according to the frequency that impacts were committed with the change set in the past. The first one uses Apriori algorithm and the second uses DAR algorithm.
Before going into the approaches, it is important to understand what change coupling information is important to predict the impact of an entity in the impact set. For example, given a change set containing three entities \( C = \{D, E, F\} \) and an impact set \( I = \{G, H\} \), the first thing that should be clear is that we calculate the probability of impact for each entity in the impact set, individually. To calculate the possible impact of entity \( G \), we need to consider that \( G \) could be impacted by a change in \( D \), by a change in \( E \), by a change in \( F \) or by any combination of changes in the three elements of the change set \( C \). This scenario is mapped to the following disjunctive rule:

\[
D \text{ or } E \text{ or } F \Rightarrow G
\]

In summary, one entity in the impact set can be impacted by any combination of at least one and up to all entities in the change set. This means that to calculate the weight (not probability anymore) of impact of an entity, we need to consider any change coupling involving the impacted entity and at least one entity from the change set. After the calculation of the impact weight for each entity in the impact set, these entities should be sorted according to the confidence of the rule.

4.1. Apriori Approach

The Apriori algorithm, originally used to address market basking problem, is now frequently used to support exploration and analysis of software repositories information. We use it to address the impacts sorting problem, although it does not give a straight forward answer. Apriori algorithm gives as output a list of conjunctive rules. Suppose that, for the change set \( C = \{D, E, F\} \) and impact set \( I = \{G, H\} \), Apriori mined the repository and found the following rules related to entity \( G \):

- \( D \ (5) \Rightarrow G \ (4) \ (0.8) \)
- \( E \ (4) \Rightarrow G \ (3) \ (0.75) \)
- \( F \ (3) \Rightarrow G \ (1) \ (0.33) \)
- \( D \text{ and } E \ (3) \Rightarrow G \ (2) \ (0.67) \)
- \( D \text{ and } F \ (2) \Rightarrow G \ (1) \ (0.5) \)

The first number is support of the left side of the rule, the second number is support of the rule and third is the confidence. Apriori does not generate automatically the disjunctive rule presented in the last section. Hence, we have created the following heuristic to sort the impacts:

For each entity in the impact set, select the rule with maximum number of changed entities in the left side or the rule. If there is more than one rule that conforms to this criterion, select the one with highest confidence and support, in this order of priority. The last step is to sort the impact set according to confidence value of each selected rule.

4.2. DAR Approach

DAR (Disjunctive Association Rules) is an algorithm based on Apriori that also finds association rules. For detailed information on DAR algorithm, refer to [Sampaio et al. 2008]. It differs from Apriori as it can also find disjunctive rules, i.e. rules that associate the union of some elements in the left side with the union of some elements in the right side. Figure 3 shows the pseudocode of DAR algorithm.
To illustrate the use of DAR, consider the change set $C = \{D, E, F\}$ and impact set $I = \{G, H\}$. DAR can directly compute the following rule:

$$D \text{ or } E \text{ or } F \Rightarrow G$$

To calculate the above rule, the DAR algorithm receives as input: matrix $H$ (dataSet), change set as left side of the rule (allowedAntecedents), impact set as the right side of the rule (allowedConsequents), number of possible elements in the right side (maxConsequentSize), minimum support (minimum_support) and minimum confidence (minimum_confidence). For the impact analysis problem the number of possible elements in the right side is always 1. The last step to find possible impacts is to sort the impact set according to the confidence value of each rule.

5. Case Study

In this case study we apply and compare the two proposed approaches for sorting IA results: Apriori and DAR models. It must be clear that we do not evaluate the results from the impact analysis technique. Our goal is to evaluate the two approaches that use data mining algorithms. We apply them in two small Java software projects, shown in Table 1. Both Impala and DesignWizard are internal projects from our research group.
The reason for choosing these projects is that we manually analyzed the source code for every commit transaction to identify what was the cause (change set) and what was the consequence (impact set) of that transaction. For the Apriori model, we use the Weka implementation of the Apriori algorithm, and for the DAR model we use the original implementation of DAR algorithm (also in Java).

Table 1. Projects selected for the case study. LOCs = lines of code, Nc = number of classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>LOCs</th>
<th>Nc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impala</td>
<td>Static impact analysis tool</td>
<td>1,584</td>
<td>45</td>
</tr>
<tr>
<td>DesignWizard</td>
<td>API for automated inspection in Java programs</td>
<td>3,644</td>
<td>44</td>
</tr>
</tbody>
</table>

5.1. Research Questions

The goal of this study is to compare the use of Apriori and DAR algorithms to address the problem of sorting impact analysis results to increase their precision. We are interested in investigating and answering the following two questions:

1. Which algorithm is able to eliminate a greater number of false impacts?
2. Which algorithm has better performance for this problem in terms of time of execution?

5.2. Setup

We used past versions of the two systems to apply the impact analysis technique, collect the estimated impact set and compare it with the real impact set. To run the complete technique, including both proposed models, we follow the steps described below.

1. Identify in the software history some modification tasks and separate their elements into two sets: elements that caused the modification – change set –, and elements that were consequently changed – impact set.
2. Apply the static impact analysis tool on each change set and the corresponding software version to collect the estimated impact set.
3. Extract from the repository the corresponding matrix $H_{|T| \times |E|}$ for each change set. Each matrix $H_{|T| \times |E|}$ is composed of commit transactions that happened before the corresponding change set. This matrix is extracted on class level, i.e., $E$ is a set of all classes in the system.
4. Run the two algorithms and update the impact set. We use a threshold to eliminate elements from the impact set based on the rules generated by the algorithms.

In the last step, we applied different strategies for each algorithm:

**Apriori.** Set minima support and confidence values to a very low value to try to generate as many rules as possible. We need to use this strategy because most of the commit transactions are composed of few elements, which require low support to find some rules. Additionally, the frequency of a certain commit transaction tends to be low, while a large number of different commit transactions appear in the repository, which requires low confidence. Minimum support value used was 0.05, and minimum confidence value was 0.01. After the generation of the rules, we apply the heuristic described in Section
4.1 to select the interesting rules. The threshold we use is the confidence value 0.5. All rules that conform to the heuristic and have confidence value greater than 0.5 are placed in the new estimated impact set. While low minima support and confidence seem awkward for other application, the have been heavily applied by the MSR (Mining Software Repositories) community.

**DAR.** Set minima support and confidence to a low value, for the same reason as above. Minima support and confidence values wet set to 0.001 and 0.01 respectively. After applying DAR to matrix $H$ we select the rules that have the largest number of changed elements in the left side. The last step is to select the interesting rules, for which we use the same threshold as for Apriori: confidence value 0.5.

5.3. Results and Evaluation

We evaluated 4 modification tasks from DesignWizard and 2 from Impala, making a total of 6 modification tasks. We also applied the same setup on method level, where $E$ from $H_{[T \times E]}$ is a set of all method from the system. This makes $H$ grow wider, with a large number of elements, which makes harder for both Apriori and DAR to generate association rules. Because of this, we could only collect the results for 1 modification task of DesignWizard. For Impala we were able to collect the results for the 2 modification tasks.

Table 2 summarizes the results obtained by applying the setup steps for the 6 modification tasks on class level. The modification tasks are enumerated from 1 to 6 in column “Task”. The number of classes in the change set is shown in “Changes” column and the real impacts are shown in “Real I”. “IA” presents the number of impacted classes identified by the static impact analysis tool. For Apriori and DAR we show the new estimated impact set in “I”, the correct estimated classes in “Correct Impact”, and the execution time in milliseconds.

<table>
<thead>
<tr>
<th>Task</th>
<th>Changes</th>
<th>Real I</th>
<th>IA</th>
<th>Apriori</th>
<th>DAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Correct I</td>
<td>Time</td>
</tr>
<tr>
<td>DW</td>
<td>1</td>
<td>2</td>
<td>14</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Impala</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that the number of impacted elements identified by our static impact analysis tool is often much larger than the real impact set. On task 1 this number is seven times larger, with a total of 12 false-results. There are only two cases in which the results from the IA tool were exactly the real impact, tasks 3 and 4. Analyzing the results from task 1, we observe that both algorithms were able to reduce the number of false impacts from 12 to only 1. However, both of them eliminated one real impact erroneously. The same occurs for task 2, while in task 3, both of them reassured the results of IA tool. Task 4 is the only one where DAR did not find real impacts whereas Apriori found 1.

In terms of performance, DAR was superior than Apriori on most of the tasks.
In these tasks the difference in performance was notably high, ranging from approximately 3 times faster (task 2) to approximately 104 times faster (task 3). This great difference in performance evidences that DAR might be more suitable to address the problem of mining software repository for IA purpose than Apriori.

Considering only the outputs of both algorithms, Apriori was more effective than DAR, because it obtained better result in one of the six tasks and equivalent result for the other five tasks. However, there is a need for further investigation because Impala tool finds impacts on method level and the data mining algorithms were applied to the system at class level. What could happen in this case is that Impala finds a possible impacted method m1() of class A for a change in method m2() of class B, but the data mining algorithms find historical change coupling between classes A and B because m3() of A changed together with m4() of B. This means that there is no historical change coupling between A.m1() and B.m2(), but the data mining algorithms erroneously point other relations.

For this reason, we decided to apply Apriori and DAR algorithms at method level for the same modification tasks. As explained before, when we extract matrix $H$ at method level, its number of columns grows large and the number of committed elements in a commit transaction remains almost the same. For this reason, both Apriori and DAR have difficulty in finding association rules for the software history at method level.

Table 3 shows the results obtained for entity level. Both data mining algorithms were not able to find interesting rules for tasks 1, 2 and 4. For this reason, we do not show them in this table. For task 3 we notice that the number of changes and impacts remained unaltered compared to class level. Consequently, both Apriori and DAR found the same results. For task 5, DAR could eliminate all 27 false impacts by identifying exactly the right impacted method, while Apriori could not find any relevant rule. Finally, DAR was able to eliminate 30 false impacts by identifying the correct impacted method and only one false impact, while nothing was found with Apriori.

<table>
<thead>
<tr>
<th>Task</th>
<th>Changes</th>
<th>Real</th>
<th>IA</th>
<th>IA</th>
<th>Apriori</th>
<th>DAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I</td>
<td>Time</td>
</tr>
<tr>
<td>Impala</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>1656</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1656</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>5516</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>4609</td>
</tr>
</tbody>
</table>

In terms of performance, DAR was remarkably superior than Apriori in all three cases: 51.75 times faster than Apriori for task 3, 70.72 times faster for task 5, and 98 times faster for task 6. This case study have shown that, undoubtedly, in terms of performance, DAR is superior to Apriori for the IA sorting problem. With this statement, we answer research question 2.

For research question 1, we have concluded that both Apriori and DAR satisfy the IA sorting problem on class level. However, it is important to remain that the results obtained on class level are subjected to the problem of misleading change couplings. When it comes to method level, again based on the case study, DAR is the right choice.
The results from this case study show that DAR algorithm might be more suitable to address the IA sorting problem, in which we have a set of estimated impacted elements with too many false results and we want to narrow it. Although DAR was more suitable for the cases analyzed in this paper, there is a need for more experiments with a greater number of software systems before we can make an affirmation.

6. Conclusions and Future Work

In this paper, we briefly presented our probabilistic impact analysis technique for increasing the precision of the results obtained from static impact analysis algorithms. Then, we proposed two approaches for applying this technique, one that uses the classic Apriori algorithm and another one with DAR. This new data mining algorithm generates disjunctive association rules, that are closer to the combining model presented in Section 3.3 than the conjunctive rules produced by Apriori.

We conducted a case study that systematically applied both models to a number of modification tasks from two software systems, called DesignWizard and Impala. Two aspects of the models were investigated: ability to reduce the number of false impacts, and time of execution. To address both aspects, we applied Apriori and DAR in the systems’ history (commit transactions) on class and on entity level. The performance of DAR in terms of execution time was far better than Apriori in the great majority of the cases. In terms of reducing the number of false impacts, the algorithms’ results were equivalent for class level, but not for entity level. DAR was superior to Apriori when analyzing the systems’ history in a fine-grained level.

The case study evidenced that the model with DAR was more suitable for reducing IA false impacts than Apriori. However, some aspects still have to be investigated. For example, if the amount of historical data influences the suitability of both algorithms. Other aspects that might influence the algorithms are: size of development team, type of development process, team’s rules and behaviors, project’s size and complexity. Hence, for future work, we will conduct some experiments to consider these aspects. We are currently working to apply the data mining algorithms on repositories of open source software systems. The size and complexity of these systems make it a challenging work, especially when it comes to entity level.

References


