AN APPLICATION OF ASSOCIATION RULE MINING TO UNIT TEST SELECTION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Karl Nils Gunderson

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science and Software Engineering

June 2013

Fargo, North Dakota

North Dakota State University
Graduate School

Title

AN APPLICATION OF ASSOCIATION RULE MINING TO UNIT TEST SELECTION

By

Karl Nils Gunderson

The supervisory committee certifies that this disquisition complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

Supervisory Committee:

Dr. Hyunsook Do
Chair

Dr. Saeed Salem

Dr. Neil Gudmestad

Approved:

June 19, 2013
Date

Dr. Brian Slator
Department Chair

ABSTRACT

Appropriate selection of unit tests during the software development process is vital when many unit tests exist. The developer may be unfamiliar with some tests and non-obvious relationships between application code and test code may exist. Poor test selection may lead to defects. This is especially true when the application is large and many developers are involved. By the application of association rule mining to the unit test selection process and by comparison with extant selection techniques, we will provide a quantitative analysis of the benefits of heuristic and its limit to development where process patterns are stable.

ACKNOWLEDGEMENTS

I would like to acknowledge the guidance of Drs. Hyunsook Do and Saeed Salem, the encouragement of Dr. Dean Knutson, the assistance of PhD. Candidate Jeff Anderson and the support of my family and friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AOS.....AX Object Server – supports server tier software execution

AOT .....AX Object Tree – hierarchical listing of development environment elements

ARM.....Association Rule Mining – technique for finding interesting relationships in data and the heuristic used in this study

AX .....Microsoft Dynamics AX 2012 R2

CIT .....Check in test – tests that are executed for each code change to AX and one of the controls used in this study

IF .....Item Frequency – the frequency with which tests appear in the data and the top N of which is used as a control in this study

LHS.....Left hand side – the antecedent of an ARM rule

RHS .....Right hand side – the consequent of an ARM rule

SCCS .....Source Code Control System – the source code repository and it's management software

SQL Server.....Microsoft SQL Server

1. INTRODUCTION

When software engineers make changes to large applications, they are expected to test those changes to ensure that they work correctly. The initial testing pass is often a unit test. A unit test is usually designed to exercise the functionality in one class of an object oriented language.

Unit testing is a standard practice in many software development organizations [1].

One important question with this approach is how to identify appropriate unit tests. To date, some researchers have proposed various approaches to select or prioritize test cases to improve the effectiveness of regression testing. In [2], clustering was applied to test prioritization. In [3], code patterns were examined to identify errors. In [4], code metrics were used to identify components that were likely to contain errors. In [5], test case prioritization and regression testing are considered with respect to residual defects and the age of those defects.

Most of these techniques depend primarily on code coverage information or code complexity metrics. However, the various phases of software development and maintenance produce numerous software artifacts such as specifications, test harnesses, bug reports, and version control databases. We believe that obtaining data about these relationships must be cheap and easy to obtain to be practical.

Ideally, a unit test will only depend upon the behavior of a single class. Practically, in a large multi-layered software application, there are many interdependencies among application and test classes. Some may be intentional and well understood, but many will be hidden and poorly understood. This is even more the case when the application's lifetime is measured in years or decades.

While not ideal, practically in a large multi-layered software application, there are many interdependencies among application and unit tests. Some interdependencies may be

intentional (also essential) and well understood, but many will be hidden and poorly understood. As the software system evolves over time and its lifetime stretches to years or decades, as many enterprise applications do, the interdependencies among various software artifacts can be very complex. Thus, selecting unit tests by understanding such complicated relationships in massive data is not a simple task.

In this research, we propose to use Association Rule Mining (ARM) to discover relationships that may improve unit test selection. Improved early testing should result in higher quality software at lower cost.

ARM was originally motivated by "market basket analysis" [6]. In market basket analysis, one looks for items in a shopping basket that are frequently found together. The algorithm used is Apriori as proposed by R. Agrawal and R. Srikant in 1994. In the first part of the algorithm, an iterative process is employed to find frequent itemsets. In round K + 1 of the algorithm, the fact that items in round K are already frequent is employed, hence the name. The probability of support (see Equation 1) is to identify useful frequent itemsets. From these frequent itemsets, all combinations of items in the itemset are generated as candidate association rules (see Equation 2). Rules that also meet the minimum confidence (see Equation 4) requirements are the association rules sought.

To evaluate our approach, we conducted an empirical study using an industrial software product, Microsoft Dynamics AX. Our results show, within limits, that selecting unit tests with ARM was able to improve unit test selection when compared to the control techniques.

1.1. Problem Definition

Selecting unit tests to execute when software is changed is difficult. While unit tests are ideally designed to test only a single software element but in large systems they often have complex relationships to many software elements.

We hope to improve the selection of unit tests. Better early testing leads to fewer defects in software at lower cost.

Failure to select appropriate unit tests can lead to bugs that are not found in a timely manner. The earlier a bug is found the less expensive it is to fix.

While some attempts have been made to aid test selection, they depend on information that is not readily available. We hope to use Association Rule Mining in the successful application of unit tests to aid unit test selection with a minimum of effort.

We have found that by using Association Rule Mining we are able to identify appropriate unit tests at a rate significantly better than current practice using only commonly available data as input.

The development and maintenance of a large application involves many software engineers in a number of roles. Chief among these roles is the development and execution of unit tests. By unit tests we mean the tests that directly exercise a single software element. In an object oriented language this will usually correspond to a class.

We have used an actual commercial product with a current lifespan in excess of 14 years in its 6th major release. The product as of this writing is known as Microsoft Dynamics AX 2012 R2.

The default approach to unit testing is often "run everything". As an application matures and functionality deepens, this approach becomes increasingly untenable. With this research we intend to investigate the use of association rule mining to select the most appropriate unit tests to run when a change is made to the application.

1.2. Significance of This Research

Finding bugs early has been identified as the fastest and cheapest time to fix bugs. Finding a bug after delivery can cost from 5 to 100 times the cost of finding and fixing the bug before delivery [7]. Ensuring that the unit tests selected either by the engineer prior to submitting changes to the check in system or by the check in system itself are optimal is vital to ensuring high quality at reasonable cost.

To the extent that we can use association rule mining to anticipate which unit tests are most likely to be useful and appropriate to test an engineers proposed changes, we can raise the level of quality and reduce costs by an even greater amount.

1.3. Organization of This Paper

Section 2 describes the background for this paper and related research. In section 3, we describe how association rule mining was applied to unit tests in our research project. Section 4 covers analysis of the research including comparison to two other techniques used as controls. Section 5 illuminates the boundaries of applicability of this study. In section 6 we take a broader look at the meaning of our results. And finally in section 7 we present our final conclusions and what we think might be fruitful lines for additional research projects.

2. BACKGROUND AND RELATED WORK

Dynamics AX is a large, complex, multi-layered software application which has evolved over many years as designed, built and tested by a diverse set of individuals acting in many different roles. Unit tests are typically developed by software engineers as the first line of defense against software defects. Unit tests are typically designed to test the functionality of a single class of an object oriented language. With many developers and highly interrelated software functionality, unit tests have additional dependencies both intended and unintended. Finding the relationship between application classes and unit test classes using association rule mining has the potential to improve the efficacy of the unit tests by selecting unit tests which have a high probability of uncovering bugs before they become part of the product.

To keep the volume of data to a manageable size, a small sub-system was selected as a source of mined data. Analysis of the data was carried out using the arules [8] package of the R statistical software system, version 3.0.1 [9]. Specifically, the apriori algorithm as implemented by Christian Borgelt [10] was used to find rules that identify relationships between modified application files (usually classes) and unit tests (again, classes). The strength of these relationships is then used to select tests. The set of tests selected by using association rule mining is compared to two controls: tests selected based on historic frequency and those previously identified as “important” by the software engineers themselves.

In this research project the software system is known as Microsoft Dynamics AX 2012 R2. The implementation language for the application code is X++. X++ is an object oriented language proprietary to Dynamics AX. It is in the class of curly braces languages of the C language heritage, borrowing ideas from C++, Java and Pascal. Application unit tests are also written in X++.

Configuration management and check in systems are used to control access to the source code of the application. With many tens of engineers working on the product simultaneously over many years, the check in system ensures that source code changes do not conflict and that

the changes conform to quality guidelines. Part of the process of integrating changes involves executing unit tests that can be directly attributed to the changed classes of the application. Later, a more complete set of tests is executed. This is known as the daily build. The daily build includes execution of all unit tests and a first line execution of functional tests. Later, a weekly or monthly build will include execution of all known automated tests. Periodically, a battery of manual and ad hoc testing will be carried out. If the engineer has "broken" the product as identified by the result of any of this testing or as identified by a customer, a bug report is written. After appropriate investigation the bug will be assigned to the offending engineer or to another engineer if the bug is the result of unintended side effects from other code. This "defect" will then go through the same check in system and battery of tests.

Other researchers have applied ARM or other techniques to testing and test selection. In [11], several techniques are evaluated for regression test selection. In [12], it is asserted that ARM can be applied to the problem of predicting defects. In [13], ARM is used to predict the effort needed to fix a defect based on a number of factors. In [14] and [15], ARM is employed to predict defect location and the effort to correct the defect. In [16], association rules are generated from a classifier based on code metrics that identify modules likely to have defects.

We have been unable to locate research that specifically addresses the use of ARM to select unit tests. We also have not been able to locate research that uses ARM with the history of changes in a manner similar to its use in this study.

2.1. Microsoft Dynamics AX

In this study, we used Microsoft Dynamics AX. It will be described here.

In this research project the software system under test is known as Microsoft Dynamics AX 2012 R2. The implementation language for the application code is X++. X++ is an object oriented language proprietary to Dynamics AX. It is in the class of curly braces languages of the C language heritage, borrowing ideas from C++, Java and Pascal. Application unit tests are also written in X++.

Microsoft Dynamics AX (henceforth referred to simply as AX) is a large business application serving customer in a broad range of sizes and industries.

To keep the volume of data to a manageable size, a small sub-system was selected as a source of mined data. Analysis of the data was carried out using the arules [8] package of the R statistical software system, version 3.0.1 [9]. Specifically, the apriori algorithm as implemented by Christian Borgelt [10] was used to find rules that identify relationships between modified application files (usually classes) and unit tests (again, classes). The strength of these relationships is then used to select tests. The set of tests selected by using association rule mining is compared to two controls: tests selected based on historic frequency and those previously identified as "important" by the software engineers themselves.

A configuration management and check in system are used to control access to the source code of the application. With many tens of engineers working on the product simultaneously over many years, the check in system ensures that source code changes do not conflict and that the changes conform to quality guidelines. Part of the process of integrating changes involves executing unit tests that can be directly attributed to the changed classes of the application. Later, a more complete set of tests is executed. This is known as the daily build. The daily build includes execution of all unit tests and a first line execution of functional tests. Later, a weekly or monthly build will include execution of all known automated tests. Periodically, a battery of manual and ad hoc testing will be carried out. If the engineer has "broken" the product as identified by the result of any of this testing or as identified by a customer, a bug report is written. After appropriate investigation the bug will be assigned to the offending engineer or to another engineer if the bug is the result of unintended side effects from other code. This "defect" will then go through the same check in system and battery of tests.

As seen in Figure 1, the physical model of AX is shown as a 3-tier application with a database tier, an application processing or server tier and a client or presentation tier. The database tier is where the software managing the database executes. The server tier runs

software called the Application Object Server or AOS. The AOS is responsible for executing non-presentation application code and also provides an abstraction layer to the database tier. The presentation tier is where direct interaction with the user occurs. The client has a forms based user interface and is the primary means of presenting information and gathering input.

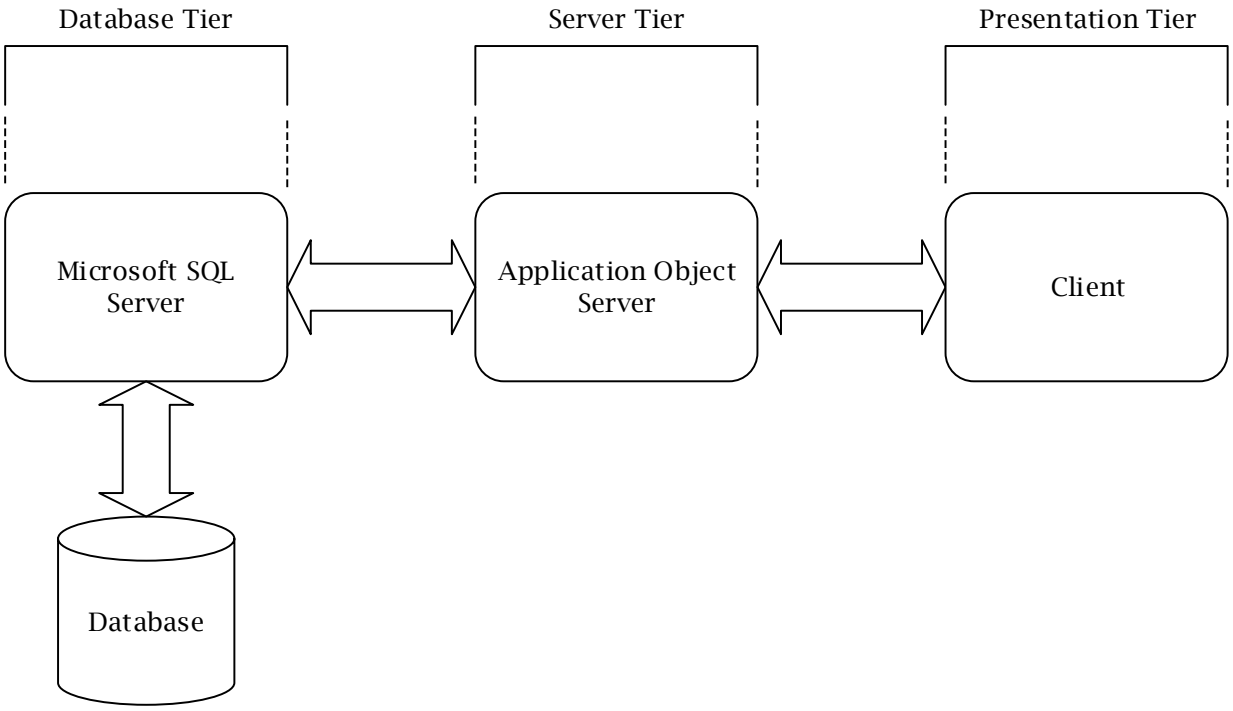

Figure 1. Microsoft Dynamics AX Physical Model

AX can be viewed as a stack consisting of two primary architectural layers: system and application as seen in Figure 2. The system layer is written primarily in C++ and some C#. The application layer is written primarily in a proprietary language called X++. X++ is an object oriented language in the C/C++ family with additional ideas borrowed from Java, Pascal and dynamic languages. X++ code in an object can run on either the AOS or the client. The system layer abstracts the location of objects and provide communications between objects via remote procedure call (RPC).

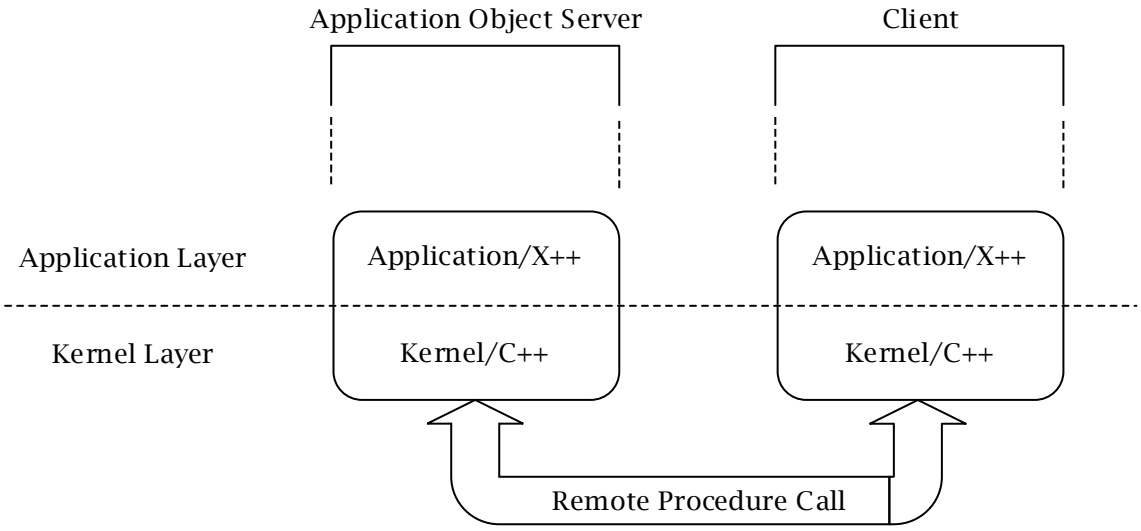

Figure 2. Microsoft Dynamics AX Logical Model

With few restrictions, application developers can choose to run the X++ code on the AOS or the client or "called from" meaning that it runs on the same tier as the caller. Further, while X++ code on the client is always interpreted, X++ running on the AOS can run in native code via compilation into .NET CIL. There are performance implications for running in the AOS vs. the client since the AOS is closer to the database tier and is less likely to be affected by network latency.

Within the application layer, there are two further subdivisions. First, the X++ code is divided into "vertical" layers. There are 8 such layers from SYS to USR as shown in Figure 3. Each layer is divided into a main layer and a patch layer, identified by a last letter of "P" as shown in Figure 4.

There are a large number of element types in the X++ language. Major elements include forms, classes, database tables, enumerated types and reports. The vertical orientation makes the effect of the layers easier to understand. Visualizing the stack from the top (USP) down to the bottom (SYS), any element of the same name at a higher layer "overlays" a similarly named element as a lower layer. In some cases the overlayering is at a granularity finer than the entire

element. For instance, a single method of a class in the SYP layer can overlay and act as a patch for the same method in the SYS layer.

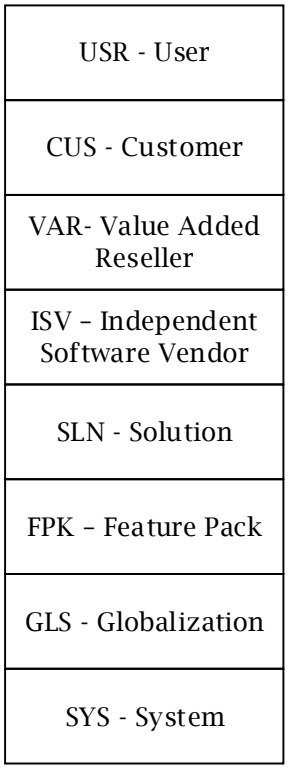

Figure 3. AX Layers

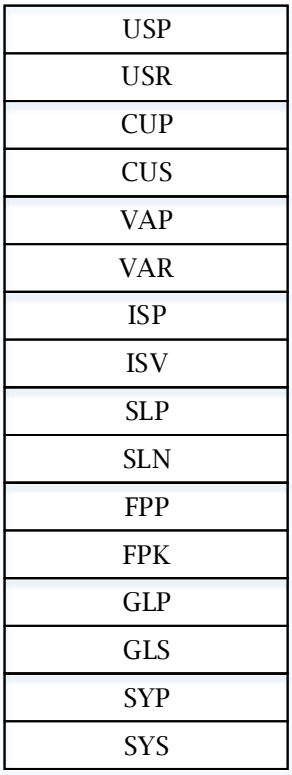

Figure 4. AX Layers With Patch Layers

During the development process, when an element at a lower layer is modified it is copied to the "current" layer. Care must be taken in two areas. First, making a change to the comment in a method will cause the copy operation to occur. If no functional change is intended, the method at the lower layer will be hidden. This can be significant following an upgrade. The method at the higher layer will continue to hide the method in the lower layer, even in the lower layer's patch layer. Second, it is not possible to delete an element from a lower layer. For instance, an obsolete method of a class from a lower layer will still need to exist at the higher layer lest the obsolete code would continue to be "visible".

The second subdivision in application code is the module, see Figure 5. Within each layer it is possible to create modules. A module is a grouping of elements. No duplication of elements is allowed within a layer, so a module does not act as a namespace but rather as a deployment

mechanism. For instance, the original version of the main product as delivered by Microsoft will exist entirely within the SYS layer but the product and upgrade code are in separate modules. New customers do not need to install the upgrade module. Existing customers need to install the upgrade module but the module may be removed once the upgrade is complete. Unit test code will only be deployed internally, it will never be installed on a customer's system.

Just as layers can be visualized as being vertically stacked, modules can be thought of as non-overlapping, horizontal groupings within a layer.

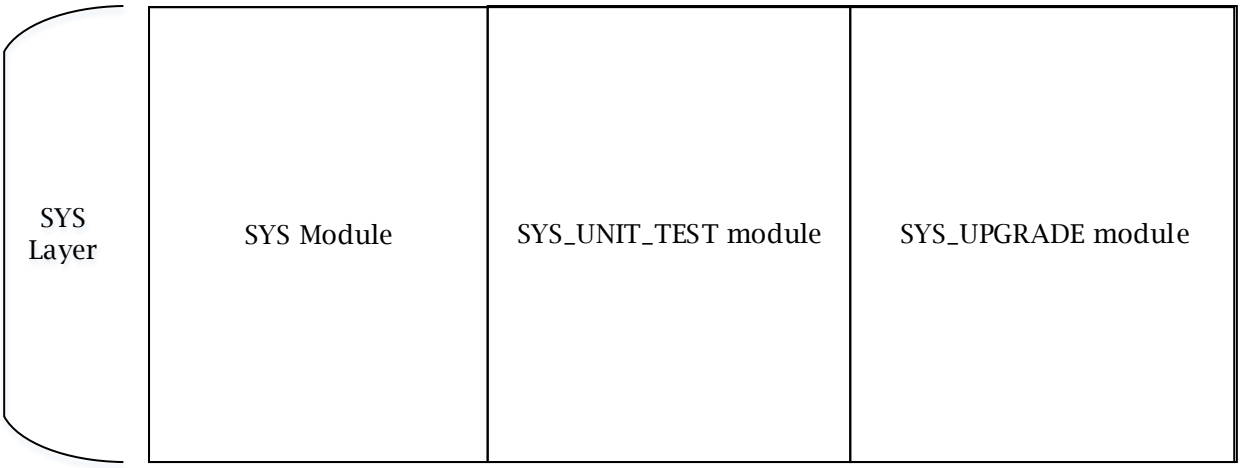

Figure 5. Modules

2.2. Current Practice of Unit Test Selection in Dynamics AX

During development of the main product in the SYS layer there are two main modules. The application code is in the SYS module and unit test code is in the SYS_UNIT_TEST module, Figure 5. The SYS module is delivered to customers, the SYS_UNIT_TEST module is not. The module concept is new, so unit tests are also identified by at least three other mechanisms. First, the configuration management system, at the developer's instruction, places the textual version of the class implementing unit tests in a separate directory hierarchy. Second, unit tests have a naming convention. All unit test classes end in "Test" facilitating identification by a developer. Third, unit test classes inherit from the SysTest class which facilitates automated identification by the use of reflection on metadata. Methods within a test class that implement individual tests either begin with "test" or use a code attribute or both.

By convention, a unit test for a class named Xyz will be named XyzTest. This makes it easy to browse a sorted list of classes and locate both a class and its unit test. Such a facility exists with the development environment in the form of the AX Object Tree or AOT. When a unit test class is selected in the AOT, as identified by metadata/reflection, commands are available to execute it with the unit test execution framework and collect statistics, optionally including code coverage.

When a developer modifies a class, the associated unit test class is also usually modified. As explained above, naming conventions make it easy to locate the associated unit test class by name. The developer will execute the unit test class locally to ensure no regressions have occurred and that cover coverage guidelines have been met. The gated check in system also executes unit tests associated with code elements prior to accepting code changes into the product's code base to ensure quality standards are met. Ideally, if a given class or unit test has an intended, unintended or hidden side effect on another class or unit test, the developer will also execute those tests prior to submission. This does not always occur. The automated check in systems will only execute unit tests for application elements submitted and selected "important" unit tests known as "check in tests" or CIT's.

When code changes are submitted that have side effects, these can escape the automated system and will destabilize the product and/or the unit test bed. Therefore, improvements to the automated unit test selection are valuable. It is also valuable for developers to have a "suggestion" list of unit tests they should run prior to submitting their code changes. Neither are available today but this research could lead to such a system of improved test selection.

2.3. Association Rule Mining

Association rule mining (ARM) is a technique for identifying interesting relationships in large volumes of data. The data is identified as a collection of transactions where each transaction contains items. The typical example is a database of shopping transactions. Each transaction then represents purchased items from the set of all items the store has available for

purchase. A transaction is the contents of one shopping cart. For example, Table 1 shows 4 transactions identified by a transaction identifier and the items associated with each transaction.

Table 1. A Collection of Transactions

Transaction ID	**Items**
ID1	Milk, Bread, Hammer
ID2	Bolts, Nuts, Hammer
ID3	Bread, Butter, Cheese
ID4	Milk, Cheese, Bolts

ARM would be used to find relationships between the items in a shopping cart. Retailers use this information to group items in the store making it easier for consumer to find items purchased together and to increase sales by associative purchases. Consumers would see this as a form of suggestive selling or an attempt to encourage impulse purchases.

A combination (set) of items is called an itemset. For example, { Bolts, Nuts } is an itemset and Transaction ID2 in Table 1 contains this itemset.

Support is a measure of how often a given combination of items appear together. For itemset X in a database of transactions D, the support is calculated as shown in Equation 1 [6].

$$SUPP(X) = Freqency(X, D)/|D| \quad \text{(Equation 1)}$$

Where Frequency(X, D) is the number of transactions containing at least the items in X among all itemsets in the transactions of D. Using the data in Table 1 the support for itemset { Butter, Cheese } is 1 / 4 or 0.25 since Butter and Cheese appear together in one transaction out of four. Support is a criterion for selecting interesting relationships since it is a measure of how often a particular set of items appear together. The higher the support of a given itemset, the more often it is found in the database of transactions.

Relationships are expressed as an antecedent and a consequent. That is to say that if the shopping cart (transaction) contains items identified in the antecedent, then those items

identified by the consequent are likely to be found. For antecedent X, and consequent Y, this is expressed as shown in Equation 2 [6].

$$X \Rightarrow Y \quad \text{(Equation 2)}$$

This is referred to as a "rule" that says if items in X exist in the transaction that implies that items in Y are likely to be found. X is also known as the left hand side or LHS and Y as the right hand side or RHS.

Using the data in Table 1, we will take food items (Bread, Butter, Milk, and Cheese) to be antecedents or the LHS and hardware items (Nuts, Bolts, Hammer) to be consequents or the RHS of ARM rules.

We can also refer to the support of a rule, see Equation 3 [6].

$$supp(X \Rightarrow Y) = (Frequency(X \cup Y, D))/|D| \quad \text{(Equation 3)}$$

The Frequency(X ∪ Y, D) is the number of transactions in D with itemsets that contain all the items in X and in Y. Support of an itemset X and support of a rule can be vastly different.

It is possible that itemset X is frequent while the itemset X ∪ Y is not frequent.

Using the data in Table 1, the support for rule { Bread } ⇒ { Hammer } would be 1 / 4 or 0.25 since there is only one itemset out of 4 that contains at least the items Bread and Hammer.

Confidence measures how often the rule is correct (items from the LHS and RHS appear together) given how often items in the LHS appear in total. Confidence is expressed by Equation 4 [6].

$$CONF(X \Rightarrow Y) = (SUPP(X \cup Y))/(SUPP(X)) \quad \text{(Equation 4)}$$

Where SUPP(X U Y) is the support for itemsets where both items in itemsets X and Y are found together. A greater confidence for a rule means that the LHS of the rule correctly predicts the actual value on the RHS.

Using the data in Table 1, the confidence of rule { Bread } ⇒ { Hammer } would be 1 / 2 or 0.5 since there is one item set that contains at least both Bread and Hammer and two that contain at least Bread.

Once a set of rules has been identified via ARM, those rules can be applied to a new set of transactions. For our purposes, we will refer to the transactions from which the rules are constructed as the "training" transaction set and the transactions to which the rules are applied as the "sample" transaction set.

How well the rules work when applied to the sample transactions will be evaluated with three measures: recall, precision and f-measure [17].

Descriptively, recall in data mining is defined as the fraction of consequents that are retrieved and is a measure of completeness or quantity. Numerically, recall is the number of consequents from rules which are also found in the sample (i.e. overlap or intersection) divided by the number of consequents in the sample. The formula for recall is given in Equation 5 [17].

$$Recall = |T\_R \cap T\_S\ |/|T\_S\ | \qquad \text{(Equation 5)}$$

Using the data in Table 1 as the sample and given a rule { Bread } ⇒ { Hammer }, the recall would be 1 / 2 or 0.5 since one instance of Hammer was selected by the rule but Hammer appears twice in the sample.

Descriptively, precision is the fraction of retrieved instances that are relevant and is a measure of exactness or quality. Numerically, precision is the number of consequents from rules which are also found in the sample (i.e. overlap or intersection) divided by the number of consequents from rules. The formula for precision is given by Equation 6 [17].

$$Precision = \ |T\_R \cap T\_S\ |/|T\_R\ | \qquad \text{(Equation 6)}$$

Using the data in Table 1 as the sample and given a rule { Bread } ⇒ { Hammer }, the precision is 1 / 1 or 1 since there is one consequent in the sample for Bread (namely Hammer) and is the same as the RHS for the rule. If the rule was { Milk } ⇒ { Hammer, Nuts } the

precision would be 1 / 2 or 0.5 since there are two consequents from the rule, Hammer and Nuts, but the sample says that Milk implies Hammer and Bolts so the intersection is 1 (Hammer) divided by 2 (Hammer, Nuts).

Table 2. Symbols in the Recall and Precision Equations

Symbol	**Meaning**
T_S	the set of consequents actually found in the sample data
T_R	the set of consequents found from the application of rules to the sample data (i.e. matching LHS)
$T_R \cap T_S$	relevant consequents, i.e. consequents found in common from the application of rules and the sample data

The F-measure is a measure of accuracy. Here we consider the balanced F-measure which is an evenly weighted average of recall and precision. Numerically, F-measure is two times the product of precision and recall divided by their sum as shown in Equation 7 [17].

$$F_1 = 2 \cdot (\text{precision} \cdot \text{recall}) / (\text{precision} + \text{recall}) \quad \text{(Equation 7)}$$

3. UNIT TEST SELECTION USING ASSOCIATION RULE MINING

When a developer makes a change to the source code for AX, all of the changes to the product code and the unit test code are submitted to the configuration management systems as an atomic transaction. The check in system ensures that all of the unit test submitted pass at 100% and that all of the unit test associated with changed product elements also run with a 100% passing rate. There are several other quality gates that the check in system utilizes that the source code must pass. If all gates are passed, the product changes are accepted, all or none.

On a daily basis, all of the product code is compiled and all of the unit tests are executed. This is part of the daily build process. If any of the unit tests fail, a bug report is generated and a developer is expected to make an appropriate change so that the product code can again pass all of the unit tests. Those changes will pass through the same quality gates and on a subsequent day, through the daily build process.

Through this continuous process of code integration and testing, product quality is kept at a high level.

When a developer modifies a product element such as a class, the corresponding unit test class will usually need modification. In a large complex product like AX, changes to one product element will affect other product elements, either intentionally or unintentionally or unexpectedly. These secondary effects may require additional adjustments to unit test code. New or inexperienced developers and developers not accustomed to working in a particular area may not be familiar with all of the potential secondary effects. Such side effects may not be caught by the gated submission system but will likely be caught by the daily build. When this occurs, the effect is destabilizing and potentially costly as the product quality is degraded and other developers are exposed to the low quality product or test code.

In Figure 6, we show the destabilizing effect of an incorrect or incomplete (bad) code change. The timeline shows a daily build occurring at the same time each day. Shortly after the

build on day 1, bad code gets past the gated system to become part of the product code base. The bad code is not discovered until the daily build on day 2 when side effects of the bad code are seen in a distantly related unit test. At this point, a bug report is generated for the failing unit test. A developer will now examine the failing test and have to determine how to deal with the effects of the bad code:

1) Change the unit test

2) Change the bad code to remove the side effects

3) React to the side effects of the bad code. In this case, the code is not bad but rather the author of the misbehaved code did not understand what other code was effected. This is common in a large, complex product.

On day 3, the bug report is fixed by submitting new code changes. The daily build on day 4 confirms the original, bad code and the reactionary code are OK. The results of the daily build are now stable again. Two daily builds over 3 days have effected other engineers. The cost of destabilization is high.

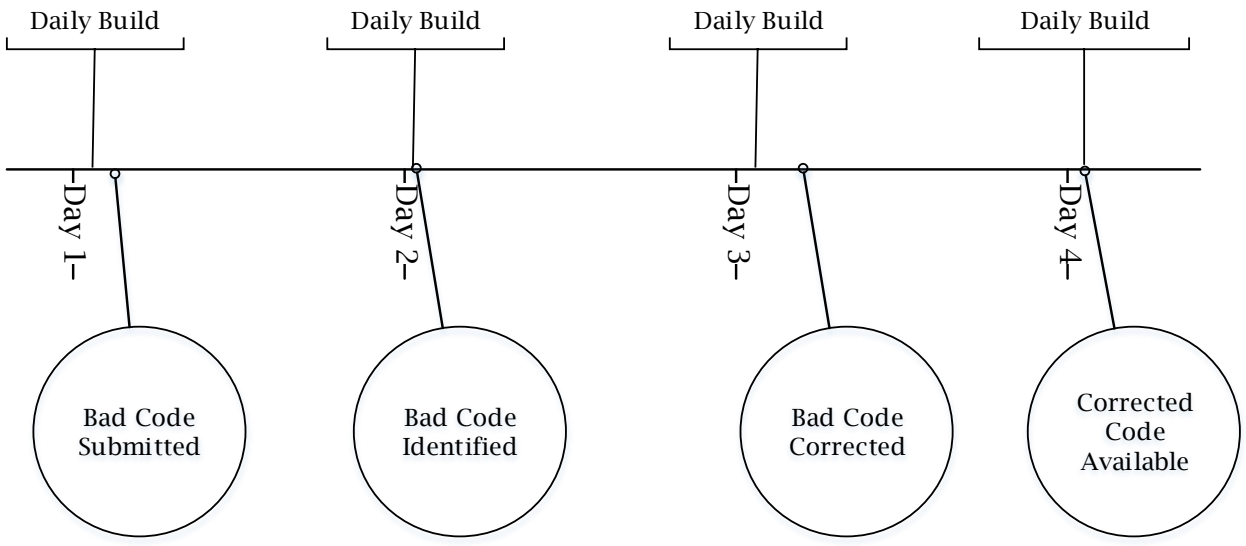

Figure 6. Destabilizing Code

From this discussion, it should be clear that a system that can suggest unit tests to execute prior to submitting changes to the gated system or as part of the gated system could be valuable. This is motivation for the research described here.

Most changes submitted to the gated system do not contain "bad" code. Some changes will reflect efforts to correct prior bad code changes. We hypothesize that ARM can be used to find these relationships between code changes and related unit tests. Further, using the rules mined, we propose that this feedback loop will increase product quality as measured by a reduction in bad code becoming part of the product.

4. EMPERICAL STUDY

In this section we will describe our investigation of association rule mining as applied to the selection of unit tests in Microsoft Dynamics AX R2.

4.1. Research Question

Can association rule mining be applied to changed file history to facilitate prediction of candidate unit tests?

4.2. Object of Analysis

In this study, we used Microsoft Dynamics AX. The system is described in Section 2.1, we summarize the metrics of the SYS layer code in AX 2012:

- The application code in X++ in the SYS module:
    - Has 16,073 class with 5,411,361 lines of text or 336.7 lines per class
    - Has 77,905 code element with 1,218,009,039 bytes of text
- The unit test code in X++ in the SYS_UNIT_TEST module:
    - Has 22,748 class with 5,266,169 lines of text or 231.5 lines per class
    - Has 24,639 code elements with 260,801,655 bytes of text
- The subsystem we used in this research, a subset of the X++ code in the SYS module:
    - Has 104 classes with 51,013 lines of text or 490.5 lines per class
    - Has 424 code elements with 4,419,080 bytes of text
- The subsystem we used in this research, a subset of the X++ unit test code in the SYS_UNIT_TEST module:
    - Has 130 classes with 51,898 lines of text or 399.2 lines per class
    - Has 133 code elements with 2,693,382 bytes of text

During the development of AX 2012 R2, there were:

- 5,566 check-ins (changelists)
- 141 check-ins included code elements from the subsystem used in this research:
    - 108 application code element were modified
    - 135 unit test class were modified

4.3. Variables and Measures

4.3.1. Independent Variable

Our independent variable is unit test selection technique. For our unit test selection technique, we consider three techniques as follows:

1) ARM produces a set of unit tests as the consequent for each antecedent. The antecedent is a set of application elements modified. ARM is our heuristic.
2) CIT's are defined as a preselected set of "important" unit tests. CIT's are our first control.
3) The frequency with which unit tests appear in the training data will provide a set of unit tests. The most frequent unit tests are selected. The mean number of unit tests found in each submission of the training data is N. The N most frequent unit tests in the training data will provide a set of unit tests. Frequent tests are our second control.

4.3.2. Dependent Variables

ARM rules are evaluated using the dependent variables support and confidence, see section 2.3. The quality of unit test selection is evaluated based on recall, precision and F-measure, see section 2.3, and the value of these measures compared to the controls. In each case, a higher value for the measure indicates higher quality unit test selection.

To determine recall, precision and F-measure for a control, the unit tests selected by the control take on the same role as T_R in each equation. For recall see Equation 5 and for precision see Equation 6.

4.4. Experiment Process and Data Collection

We will take the submission file history for Microsoft Dynamics AX R2 for a small subsystem and apply the techniques of ARM to some portion of that history, the training data. The antecedents of each rule will be the application elements and the consequents will be the unit tests.

We will then take the remaining history, the sample data, and apply the rules to application files. The rules that match the application files (the LHS or antecedents) will generate a list of unit tests (the RHS or consequents). We will then apply various measures to the unit tests selected by the rules as well as by two control techniques to those actually found in the sample data. The measures will provide a quantitative means of evaluating which unit test selection technique was the most efficacious.

The source code control system (SCCS) keeps complete history of each group of files submitted. The group is referred to as a changelist. The SCCS was queried for all changelists made for a particular subsystem. The files changed were further classified as either application files or unit test files. Application files were prefixed with "F_" and unit test files with "T_".

The extraction process produced a set of 141 changelists from a population of 243 items, see Figure 7 and Table 3. It was found that 69 of the changelists contained only a single file and were eliminated. Three of the changelists were of a size that was out of the norm. This left us with a transaction database of 69 changelists to mine, see Figure 8 and Table 4.

Table 3. 141 Original Transactions

# of Items	# of Item Sets
1	69
2	19
3	11
4	4
5	6
6	5
7	5
8	4
9	1
10	1
12	1
14	4
15	1
16	1
17	1
19	1
23	1
26	1
30	1
34	1
46	1
76	1
188	1

Table 4. 69 Cleaned Transactions

# of Items	# of ItemSets
2	19
3	11
4	4
5	6
6	5
7	5
8	4
9	1
10	1
12	1
14	4
15	1
16	1
17	1
19	1
23	1
26	1
30	1
34	1

Each changelist was grouped as a transaction and the files as items of the itemset of the transaction. The apriori algorithm as implement in the arules package of the R statistical analysis system was applied to these transaction. Apriori was instructed to restrict antecedents to "F_" items and consequents to "T_" items. This particular implementation of apriori produces only a single consequent in each rule. Apriori was further instructed to produce rules with a minimum of two items, guaranteeing that each rule will have at least one antecedent.

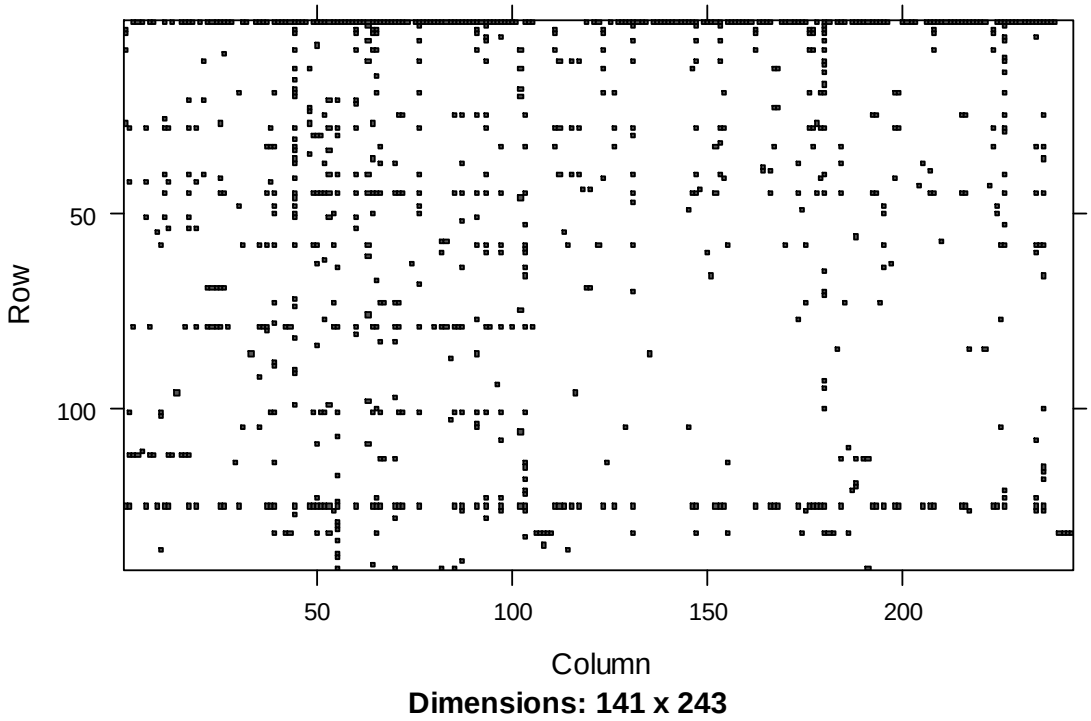

Figure 7. Plot of the 141 Original Transactions

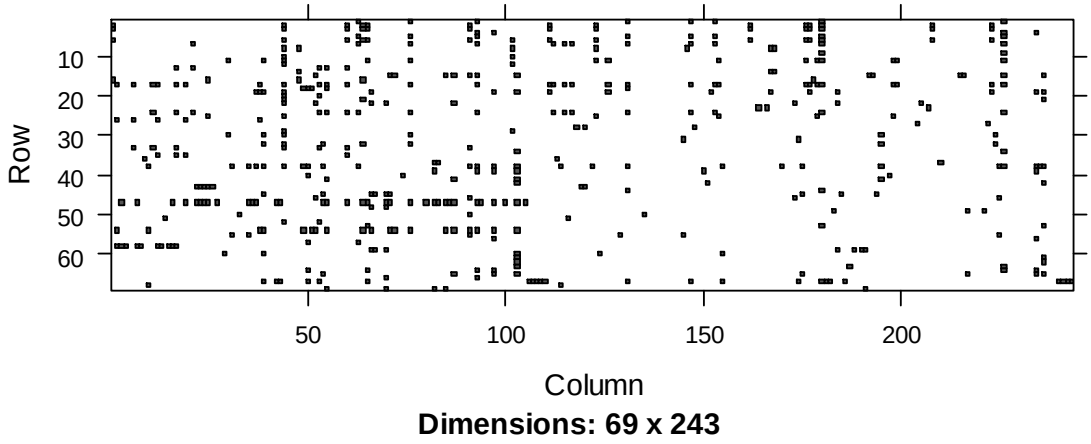

Figure 8. Plot of the 69 Cleaned Transactions

4.5. Data and Analysis

Association rule mining will provide a set of unit tests. CIT's will provide another and frequency will provide yet a third. We will evaluate these using the measures of precision, recall and F-measure.

Current practice is for a developer to run the unit tests for each element modified prior to submitting code to the gated system. The developer may choose to execute additional unit tests but there is no record made of this choice. The gated system repeats the execution of the unit tests for each element submitted and additionally runs check in tests (CIT's) which are preselected tests deemed as important by other developers. Any submission of code failing these tests is rejected and there is no record. A subsequent submission will include modifications necessary to allow all tests to pass. In this way, successful submissions are a record of modified application code and appropriately modified test code.

To demonstrate the analysis used, we will use a small, artificial dataset to explain the steps. The dataset consists of 8 itemsets show in Table 5:

Table 5. Table of Demo Transactions

Changelist ID	Files	Tests
C1	F_A	T_A
C2	F_B, F_C	T_B
C3	F_A, F_C	
C4	F_B, F_C	T_C
C5	F_C	T_D
C6	F_B	T_C, T_D
C7	F_B, F_C	T_B, T_C, T_E
C8	F_C, F_D	T_C

Or represented as a binary matrix as shown in Table 6:

Table 6. Binary Matrix of Demo Transactions

	F_A	**F_B**	**F_C**	**F_D**	**T_A**	**T_B**	**T_C**	**T_D**	**T_E**
C1	X				X				
C2		X	X			X			
C3	X		X						
C4		X	X				X		
C5			X					X	
C6		X					X	X	
C7		X	X			X	X		X
C8			X	X			X		

The transaction ID for a changelist begins with C, an item beginning with "F_" is an application file and an item beginning with "T_" is a unit test file. In the binary matrix, presence of an item in the itemset is indicated by an "X".

Running apriori with the first 4 changelists (C1 to C4) as training data using confidence of 0.3 and support of 0.01 produces the 7 rules found in Table 7.

Table 7. Demo Rules

Rule	LHS	RHS	Support	Confidence
R1	{F_A}	{T_A}	0.25	0.50
R2	{F_B}	{T_B}	0.25	0.50
R3	{F_C}	{T_B}	0.25	0.33
R4	{F_B}	{T_C}	0.25	0.50
R5	{F_C}	{T_C}	0.25	0.33
R6	{F_B, F_C}	{T_B}	0.25	0.50
R7	{F_B, F_C}	{T_C}	0.25	0.50

As binary matrix the rules are represented in Table 8.

Table 8. Binary Matrix of Demo Rules

	F_A	F_B	F_C	F_D	T_A	T_B	T_C	T_D	T_E
R1	X				X				
R2		X				X			
R3			X			X			
R4		X					X		
R5			X				X		
R6		X	X			X			
R7		X	X				X		

A rules such as R1 is { F_A } ⇒ { T_A } with support of 0.25 and confidence of 0.50. When represented as a binary matrix, an X in a column indicates that the item is a member of the rule.

When the sample data (C5 to C8) is applied to the rules the result is shown in Table 9. Each row is a rule and a column represents an itemset from the sample data. The intersection of a row and column indicates whether the items in the LHS of the rule were found in the items of transaction.

Table 9. Binary Matrix of Demo Rule Application

	C5	C6	C7	C8
R1	FALSE	FALSE	FALSE	FALSE
R2	FALSE	TRUE	TRUE	FALSE
R3	TRUE	FALSE	TRUE	TRUE
R4	FALSE	TRUE	TRUE	FALSE
R5	TRUE	FALSE	TRUE	TRUE
R6	FALSE	FALSE	TRUE	FALSE
R7	FALSE	FALSE	TRUE	FALSE

For instance, the LHS of R3 is { F_C } and item F_C was found in transactions C5, C7 and C8 but not in transaction C6.

Or looking at the individual rules and changelists, we see the following for recall and precision in Table 10:

Table 10. Demo Rule Production w/Recall & Precision

Change List => T	**Rule Production => T**	**R**	**P**
C5: { F_C } => { T_D }	{ R3, R5 } => { T_B, T_C }	0 / 1	0 / 2
C6: { F_B } => { T_C, T_D }	{ R2, R4 } => { T_B, T_C }	1 / 2	1 / 2
C7: { F_B, F_C } => { T_B, T_C, T_E }	{ R2, R3, R4, R5, R6, R7 } => { T_B, T_C }	2 / 3	2 / 2
C8: { F_C, F_D } => { T_C }	{ R3, R5 } => { T_B, T_C }	1 / 1	1 / 2

The mean recall = (0 + 1/2 + 2/3 + 1) / 4 = 13 / 6 / 4 = 0.5417

The mean precision = (0 + 1/2 + 1 + 1/2) / 4 = 2 / 4 = 0.5000

F-measure = 2 * (0.5417 * 0.5000) / (0.5417 + 0.5000) = 0.5200

The following tables (Table 11, Table 12, Table 13, Table 14, Table 15 and Table 16) show the results of increasing support (0.03 to.10, see Equation 3) and using a confidence of 0.80 (see Equation 4) with the specified percent training data and the remainder of the data used for the sample. In all cases the mean number of tests per changelist from the training data was 3. Note that tests from CIT's and item frequency (IF) are the controls.

In Table 11, 50% of the transactions (34 out of 69) are used for training and the remaining 35 are used for samples.

In Table 12, 60% of the transactions (41 out of 69) are used for training and the remaining 28 are used for samples.

In Table 13, 70% of the transactions (48 out of 69) are used for training and the remaining 21 are used for samples.

In Table 14, 75% of the transactions (52 out of 69) are used for training and the remaining 17 are used for samples.

In Table 15, 80% of the transactions (55 out of 69) are used for training and the remaining 14 are used for samples.

In Table 16, 90% of the transactions (62 out of 69) are used for training and the remaining 7 are used for samples.

As will be seen in the next set of figures, there is a definite peak at 75% so we felt it was important to show this particular table.

The column heading are:

- support – the support level input to the apriori algorithm
- rules – the number of rules generated
- R – mean recall when the rules are applied to the sample data row by row
- P – mean precision when the rules are applied to the sample data row by row
- F1 – the F-measure from R and P
- R-CIT – mean recall for CIT's when applied to the sample data row by row
- P-CIT – mean precision for CIT's when applied to the sample data row by row
- F1-CIT – the F-measure for R-CIT and P-CIT

- R-IF – mean recall for the 3 most frequent tests in the training data when applied to the sample data row by row
- P-IF – mean precision for 3 most frequent tests in the training data when applied to the sample data row by row
- F1-IF – the F-measure for R-IF and P-IF
- tests – the number of unique tests on the RHS of all rules

Table 11. Training 50% (34 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1308	0.1092	0.1448	0.1245	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	18
0.04	1308	0.1092	0.1448	0.1245	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	18
0.05	1308	0.1092	0.1448	0.1245	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	18
0.06	437	0.0354	0.0376	0.0365	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	12
0.07	437	0.0354	0.0376	0.0365	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	12
0.08	437	0.0354	0.0376	0.0365	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	12
0.09	37	0.0354	0.0376	0.0365	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	10
0.10	37	0.0354	0.0376	0.0365	0.0660	0.0254	0.0367	0.0616	0.0571	0.0593	10

Table 12. Training 60% (41 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1375	0.1097	0.1274	0.1179	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	20
0.04	1375	0.1097	0.1274	0.1179	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	20
0.05	440	0.1008	0.1095	0.1050	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	13
0.06	440	0.1008	0.1095	0.1050	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	13
0.07	440	0.1008	0.1095	0.1050	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	13
0.08	38	0.1008	0.1095	0.1050	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	11
0.09	38	0.1008	0.1095	0.1050	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	11
0.10	10	0.0179	0.0357	0.0238	0.0795	0.0278	0.0412	0.0944	0.0833	0.0885	5

Table 13. Training 70% (48 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1257	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	20
0.04	1257	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	20
0.05	406	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	13
0.06	406	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	13
0.07	38	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	11
0.08	38	0.1344	0.1460	0.1399	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	11
0.09	10	0.0238	0.0476	0.0317	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	5
0.10	10	0.0238	0.0476	0.0317	0.0822	0.0317	0.0458	0.0782	0.0794	0.0788	5

Table 14. Training 75% (52 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1257	0.1660	0.1804	0.1729	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	20
0.04	406	0.1660	0.1804	0.1729	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	13
0.05	406	0.1660	0.1804	0.1729	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	13
0.06	38	0.1660	0.1804	0.1729	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	11
0.07	38	0.1660	0.1804	0.1729	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	11
0.08	10	0.0294	0.0588	0.0392	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	5
0.09	10	0.0294	0.0588	0.0392	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	5
0.10	1	0.0294	0.0588	0.0392	0.0819	0.0327	0.0467	0.0966	0.0980	0.0973	1

Table 15. Training 80% (55 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1232	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	21
0.04	389	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	12
0.05	389	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	12
0.06	29	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	10
0.07	29	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	10
0.08	5	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	3
0.09	5	0.0051	0.0714	0.0095	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	3
0.10	0	0.0000	0.0000	0.0000	0.0638	0.0317	0.0424	0.0816	0.0952	0.0879	0

Table 16. Training 90% (62 transactions)

support	rules	R	P	F1	R-CIT	P-CIT	F1-CIT	R-IF	P-IF	F1-IF	tests
0.03	1229	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	21
0.04	386	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	12
0.05	26	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	10
0.06	26	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	10
0.07	5	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	3
0.08	5	0.0102	0.1429	0.0190	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	3
0.09	0	0.0000	0.0000	0.0000	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	0
0.10	0	0.0000	0.0000	0.0000	0.0561	0.0476	0.0515	0.1633	0.1905	0.1758	0

There is very little significant change in the values from 50% (Table 11) to 60% (Table 12) training. There is a small deterioration of the F-measure for the heuristic and improvement for the controls. There is a significant fall off of precision as the sample set gets smaller and as support raises due to fewer rules available to find consequents (RHS or tests).

At 70% (Table 13) and 75% (Table 14) recall and precision improve markedly for the heuristics while the controls see only minor improvements. At 75% training, the heuristic hits a marked peak in F-measure. We conclude that the distribution of the training data is such that it matches the distribution of the sample data optimally.

At 80% (Table 15) and 90% (Table 16) training data, the falloff in F-measure is pronounced. Recall falls by 2 orders of magnitude and precision by one. To understand the dramatic change, we examine the files and tests that the training and sample data share.

In Table 17, we show the number of files and tests that the training data and sample data have in common at each training percentage. For instance, in the first row of Table 17, at 50%

training data and 50% sample data, there are 46 uniquely named unit tests in the training data and 47 uniquely named unit tests in the sample data. Of these unit tests, there are 12 that are common between the training portion and the sample portion. Likewise, there are 38 uniquely named files in the training data and 65 uniquely named files in the sample data. Of these files, there are 31 that are common between the training and sample portions.

The significance of a common file or test is: if A & B are training files but only A is found as a sample file then any rule that includes B in its LHS will never be used. So the fewer files in common, the fewer rules that will be used regardless of the number of rules generated.

Table 17. Common Files and Tests

percent training	training tests	sample tests	common tests	training files	sample files	common files
50%	46	47	12	38	65	31
55%	55	42	16	44	64	36
60%	57	39	15	45	63	36
65%	61	34	14	50	62	40
70%	61	33	13	62	47	37
75%	66	29	14	64	44	36
80%	67	26	12	64	33	25
85%	70	23	12	68	20	16
90%	71	22	12	69	19	16
95%	72	16	7	69	14	11

In Figure 9 we see that the drop in common file drops precipitously at 80% training data. Since there are so few common files, the rules generated from the training data will rarely match the sample data.

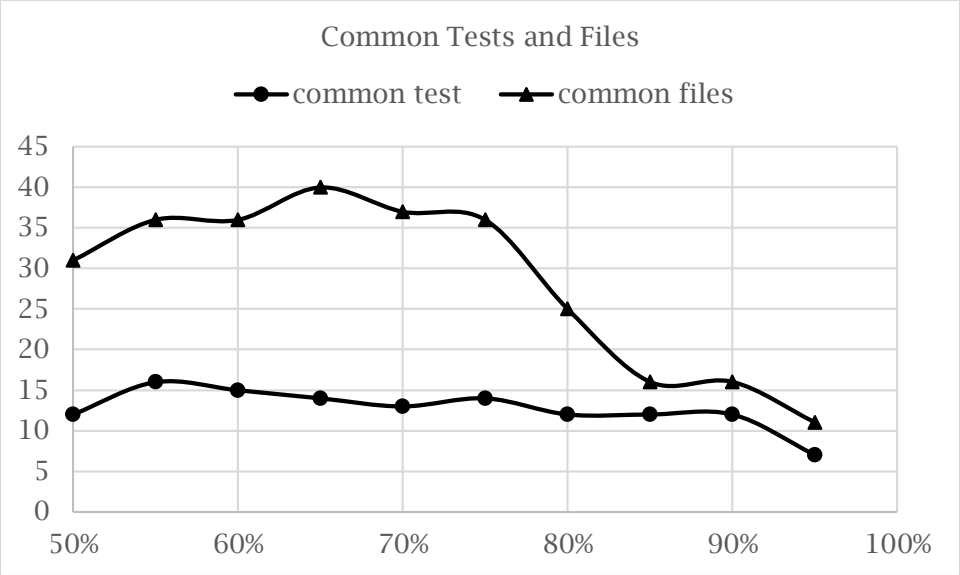

Figure 9. Common Files and Tests

The following graphs (Figure 10, Figure 11, Figure 12, Figure 13, Figure 14, Figure 15, Figure 16 and Figure 17) show the F-measure vs. the % of the data dedicated to training at each support level (0.03 to 0.10).

Figure 10 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.03.

Figure 11 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.04.

Figure 12 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.05.

Figure 13 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.06.

Figure 14 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.07.

Figure 15 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.08.

Figure 16 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.09.

Figure 17 shows the F-measure for each % training data from 50% to 95% when mined using a support of 0.10.

There is very little change in support levels of 0.03 (Figure 10), 0.04 (Figure 11) and 0.05 (Figure 12). In all cases, the F-measure for training data at 50% to 70% for the heuristic (ARM) is significantly above the level of the 2 controls. In each case, the dropped off mentioned above at 80% training is visible. Also, there is a sudden rise for the frequent item control (F1-IF) from 80% to 90% training. Clearly this indicates that the frequently selected tests are in fact being selected at the tail end of the data matching the predictor that the control represents.

At support levels 0.06 (Figure 13) and 0.07 (Figure 14) we see a sudden drop in the F-measure of the heuristic at 50% training. This corresponds to a marked drop in the number of rules generated from over 1300 to under 450. Further, since the drop is only present at 50% training, we must conclude that the sample data after 50% has a distinctly different distribution than the training data before 50%. When the training data goes to 55% the F-measure for ARM recovers its prior, relatively robust value.

At a support level of 0.08 (Figure 15) the heuristic only manages to do better than the controls at training levels from 60% to 70%. At 0.09 (Figure 16) and 0.10 (Figure 17) support, even that small advantage is gone.

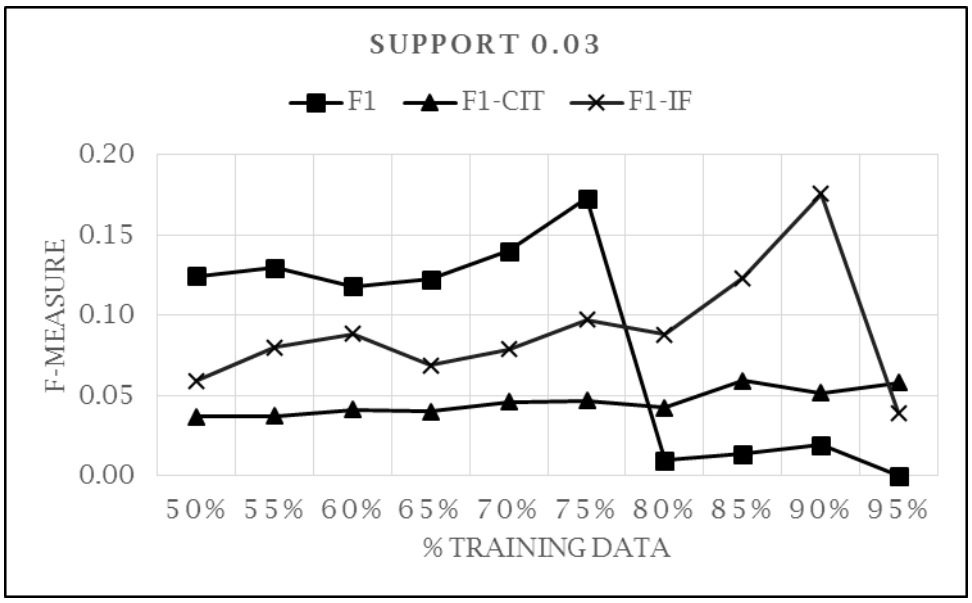

Figure 10. Support 0.03

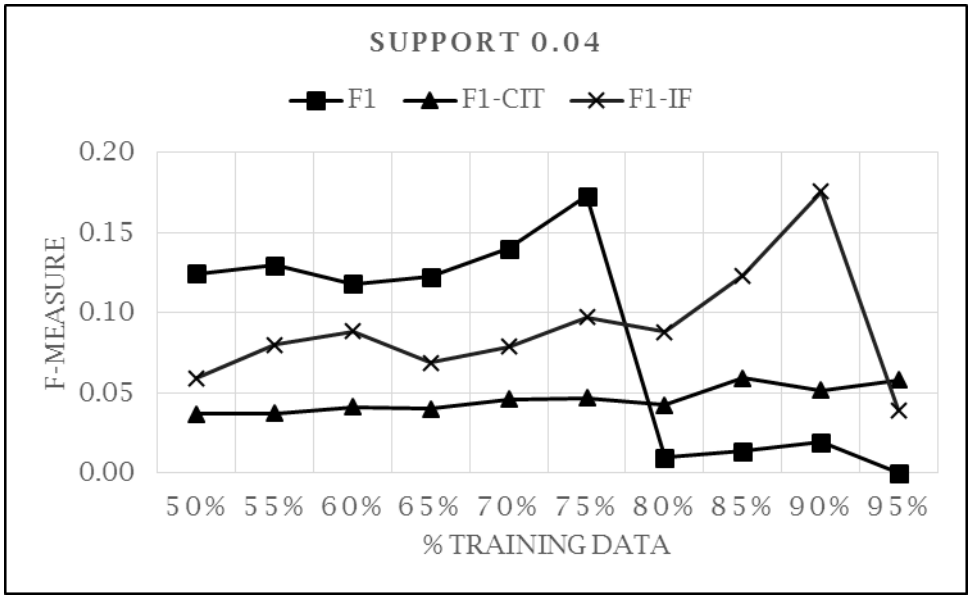

Figure 11. Support 0.04

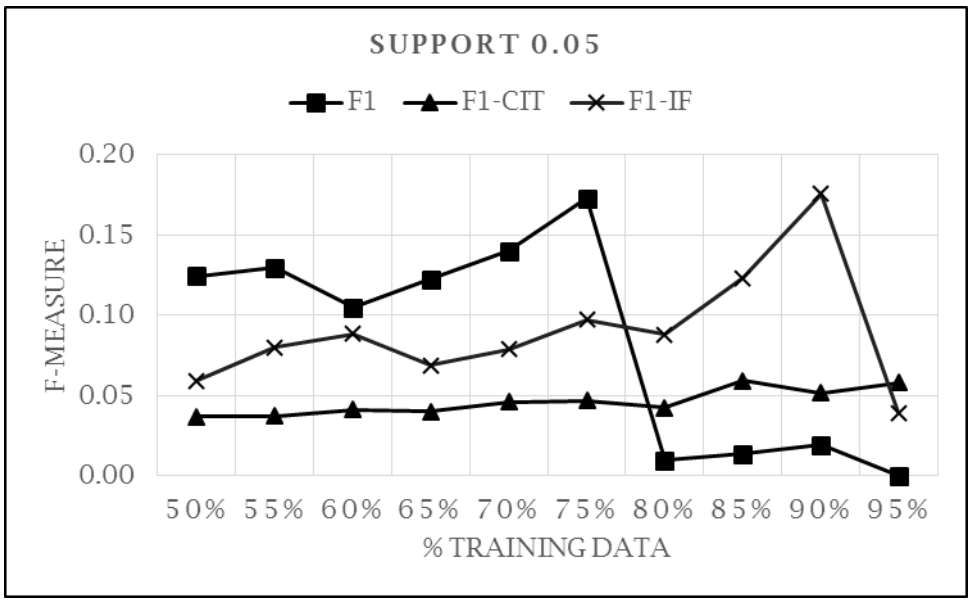

Figure 12. Support 0.05

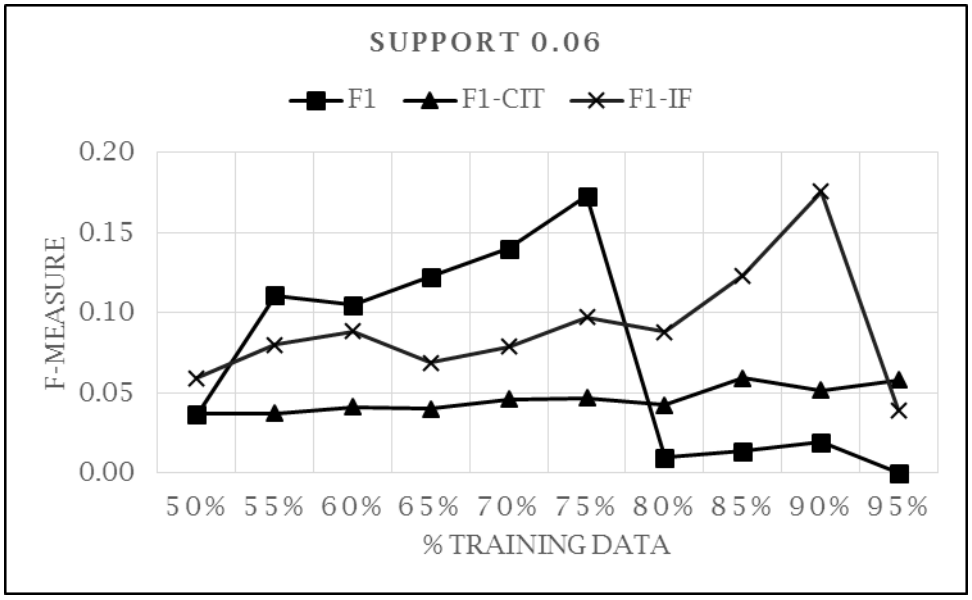

Figure 13. Support 0.06

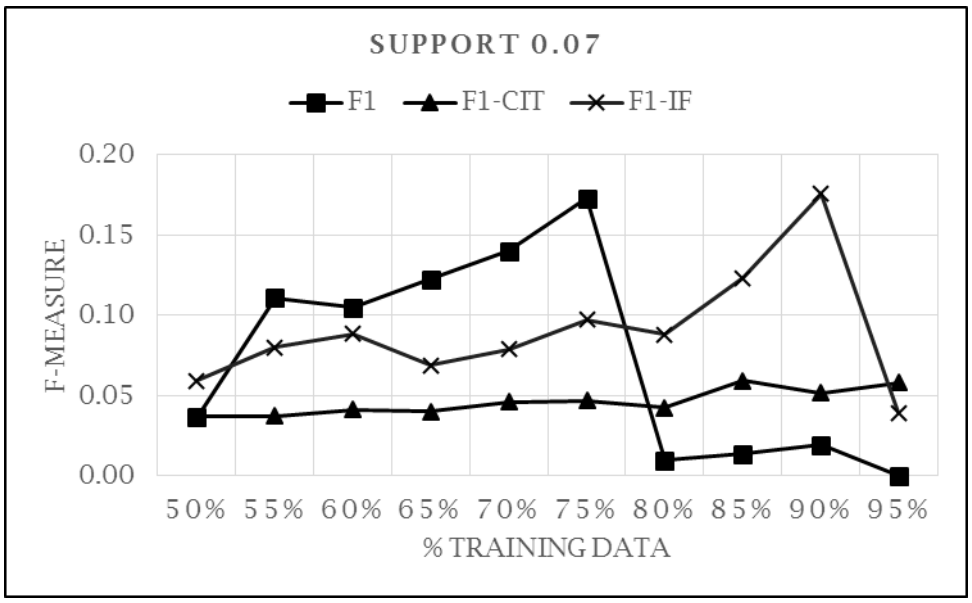

Figure 14. Support 0.07

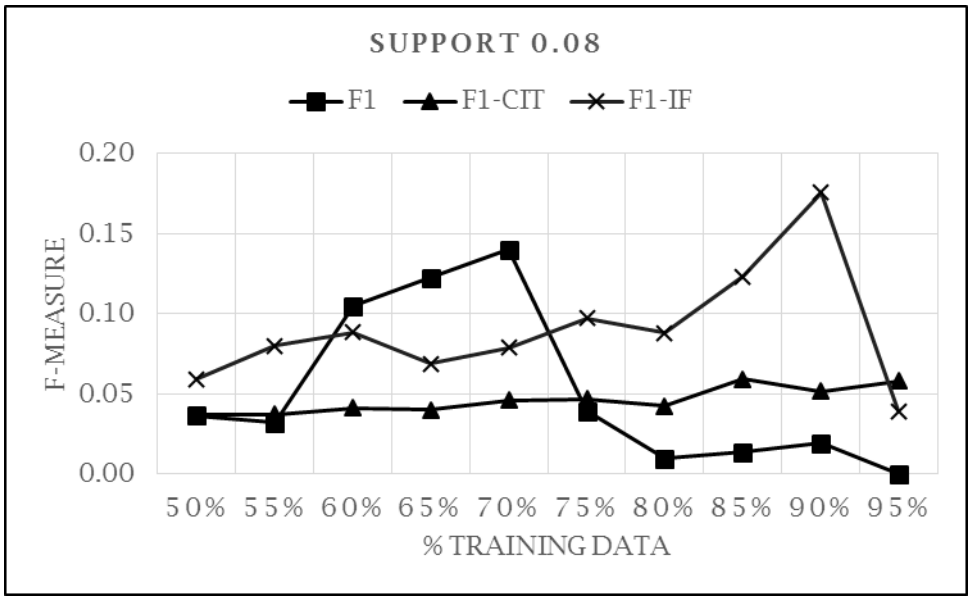

Figure 15. Support 0.08

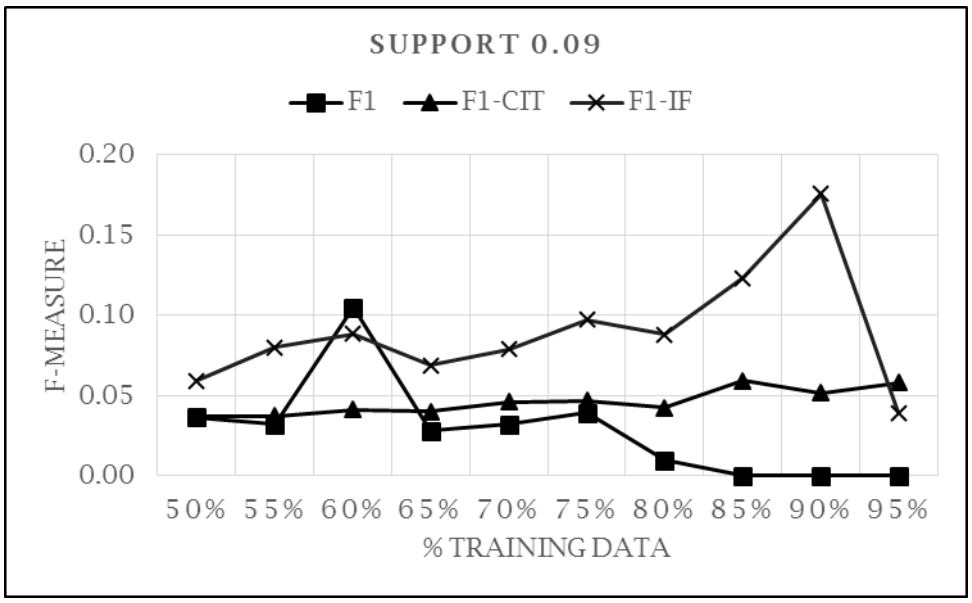

Figure 16. Support 0.09

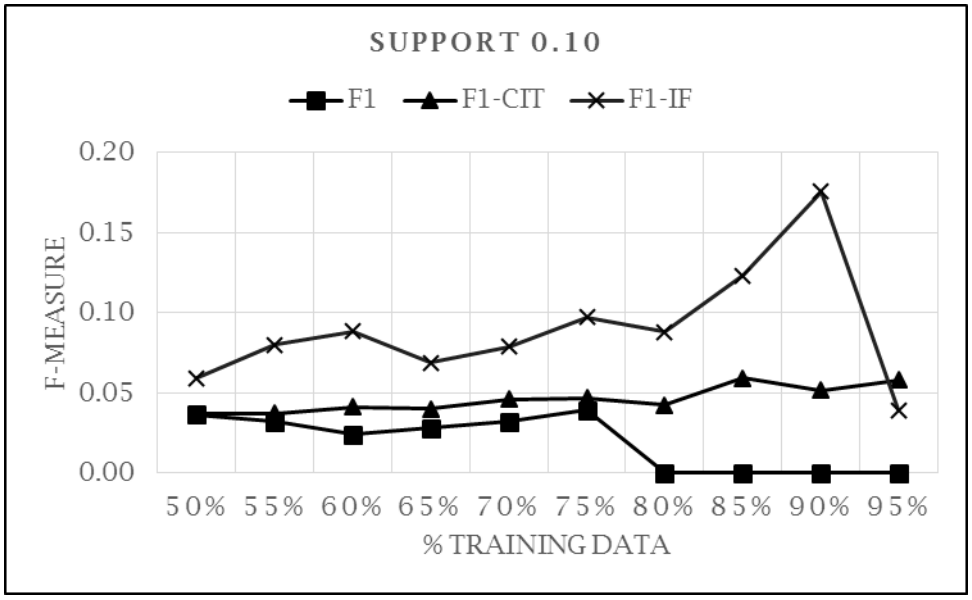

Figure 17. Support 0.10

The preceding tables and graphs show us that training with up to 75% of the data at support levels of no more than 0.7 and ideally no more than 0.05 (for lower percentage level training) are best.

5. THREATS TO VALIDITY

We describe the threats to the validity of our empirical study and its results.

5.1. Internal Validity

We are new to the R system. We have written a moderately sized R application (see sections 1.A.1 and 1.A.2). A small (8 transaction) dataset was used to validate results, and we have been careful to ensure accurate coding of our algorithm.

Since only successful submissions to the gated check in system are recorded, our assumption that past history can predict useful unit test selection in the future may not be correct. Failure information would tell us if the patterns found in the history is based on improved performance in this regard or a change in behavior as time progresses. We believe the former but do not have verifiable data to prove it.

Our assumptions about our "success only history" led us to only examine a relatively high confidence level (0.80) throughout the study. This may be ill advised.

5.2. External Validity

The number of changelists we were able to mine was not large. This makes the study vulnerable to undue influence by data with an unrepresentative distribution. However, some limited testing with a larger dataset showed the same pattern found in this study.

The data were taken from the development of a minor release of the software focused in part on country specific features. It is possible that work on a major release or more general features may have significantly different or more subtle patterns that are not discoverable by the techniques employed for this study.

Since the product is under development by a large number of geographically dispersed teams, it is possible that other teams may have different work patterns or habit that lead to significantly different results.

The fact that only successful check ins are logged and that the relatedness of a “bad” submission with its correcting submission are not tracked means that some potentially valuable data is unavailable. This other data may lead to results that are far more compelling or contrary to those we have found in this study.

Since this study was based on the work of a single large business software application, it may have limited applicability to other types or sizes of software.

Software development styles, processes and cultures may lead to patterns of work that bear no resemblance to those used in this study.

The use of unit tests may be more or less prevalent in other development organizations. Reliance on different testing frameworks and facilities, different languages and different development environments might all affect that ability to mine for patterns or the patterns themselves. As a result, these results may not apply equally in other situations.

6. DISCUSSION

We discuss our results, provide additional insights into the data and describe the practical implications of these results.

6.1. ARM Can Be Used To Improve Unit Test Selection

It would seem from the analysis in Section 4.5, that while ARM can be used to improve unit test selection, it has its limitations.

First of all, support levels are relatively low. This indicates that the rules found from ARM are subtle and not based on frequently encountered patterns. This is the reason one conducts a data mining process: if the rules found by data mining were obvious, data mining would only confirm common knowledge. Still, these support levels seem unusually low. As mentioned in the internal threats to validity (section 5.1), we do not have failure information. Failures can be a significant source of information. Failure can indicate a complete misunderstanding but it can also indicate imperfect understanding. And again, ARM takes this into account by providing us with the ability to use confidence as a discriminator. Since our historic data consisted only of "successful check ins" we did not employ this discriminator in selecting rules.

So, in which situations was ARM successful in improving unit test selection? Primarily when the pattern of behavior was relatively static. In analysis we found that the mix of tests between training and sample data was relatively stable but the mix of files was not. In the software development lifecycle, there is a point near the end game when behavior changes.

As software projects near completion, a common strategy is to limit the changes made to the code base. Such is exactly the situation at Microsoft. Initially during the development cycle, new features and new code are the priority as there is a push to show progress. Then at the mid-point there is an effort to "catch-up" the test code to match the progress of the application code. The effect is less so with unit tests but nonetheless exists. Toward the end of the project, project management intentionally limits the amount of churn in the product code base. This is for two primary reasons:

1) Eliminate changes that can be disruptive. Stability becomes the primary goal and change is antithetical to stability.

2) Allow test work, which lags behind feature work, to finally catch up. Product quality is measured by the volume and frequency of bug reports. By limiting bug work to increasingly higher impact bugs, this measure converges to an acceptable value.

This is borne out by the relatively lower number of files beyond 80% training (see Figure 9). The sample data now represents the last 20% of the software development lifecycle and fewer files are changing. Tests remain pretty consistent right up until the last 5% of the project as represented by the sample data. Note in Figure 9 how both common tests and common files appear to converge. This is exactly what happens in the real software development community at Microsoft.

As behavior changes and the pattern of file and unit test modifications change, the ARM generated rules become less relevant.

7. CONCLUSIONS AND FUTURE WORK

We have presented a technique for selecting unit tests based on check in history using association rule mining. We conclude that ARM is a viable technique with the caveat that the technique is sensitive to changes in submission history.

Given the sensitivity to changing submission behavior, various strategies for further research come to mind. A weighting that favors more recent history might yield more robust results. Another weighting is the direct connection between a unit and its test. The current gated system in use in the system studied always executes the directly associated test for any software element submitted for inclusion in the product. The ideal, comprehensive unit test suggestion list cannot ignore this pattern which is the definition of a unit test.

Other sources of information could be incorporated. Rather than trying to identify interdependencies between software elements solely by their appearance together in a changelist other sources of information should be explored. For instance, traversal of cross reference information (static code analysis) of both application files and unit test files could contribute more direct evidence of dependencies. Better yet would be code coverage information from execution of unit tests. Coverage would show which other software element are touched during testing.

In Section 3, we postulate that improved product quality as a result of better unit test selection will provide a positive feedback loop further improving unit test selection. This is clearly an opportunity for further examination.

As mentioned in Section 1.2, the value of high quality testing to high quality software cannot be expressed strongly enough. Similar research that identifies useful patterns of test selection can return benefits to the entire research and industrial software communities.

8. REFERENCES

[1] I. S. Board, "IEEE Standard for Software Unit Testing," IEEE Computer Society, New York, NY, 1987.

[2] R. C. Carlson, "A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study," North Dakota State University, Fargo, North Dakota, November 2010.

[3] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," in *European Software Engineering Conference/Foundations of Software Engineering*, Lisbon, Portugal, 2005.

[4] N. Nagappan, T. Ball and A. Zeller, "Mining Metrics to Predict Component Failures," Microsoft Researcj, Redmond, WA, 2005.

[5] P. Nagahawatte and H. Do, "The Effectiveness of Regression Testing Techniques in Reducing the Occurance of Residual Defects," in *Software Testing, Verfication and Validation*, Paris, 2010.

[6] J. Han and M. Kamber, "Chapter 5: Mining Frequent Patterns, Associations and Correlations," in *Data Mining: Concepts and Techniques*, San Francisco, CA, Morgan Kaufmann, 2006, pp. 227-283.

[7] B. W. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," vol. 34, no. 1, 2001.

[8] M. Hahsler, C. Buchta, B. Gruen and K. Hornik, "arules: Mining Association Rules and Frequent Itemsets. R package version 1.0-14.," 2013. [Online]. Available: http://cran.rstudio.com/web/packages/arules/index.html. [Accessed 3 June 2013].

[9] "The R Project for Statistical Computing," The R Foundation, [Online]. Available: http://www.r-project.org/. [Accessed 3 June 2013].

[10] C. Borgelt, "Apriori - Association Rule Induction / Frequent Item Set Mining," 8 May 2013. [Online]. Available: http://www.borgelt.net//apriori.html. [Accessed 3 June 2013].

[11] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* pp. 529-551, 1996.

[12] M. P. J. Kaur and M. Pallavi, "Data Mining Techniques for Software Defect Prediction," *International Journal of Software and Web Sciences,* pp. 54-57, 2013.

[13] S. Morisaki, A. Monden, T. Matsumura, H. Tamada and K.-i. Matsumoto, "Defect Data Analysis Based on Extended Association Rule Mining," in *Mining Software Repositories*, Minneapolis, MN, 2007.

[14] Q. Song, M. Shepperd, M. Cartwright and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," vol. 32, no. 2, 2006.

[15] N. Mishra, "Art of Software Defect Association & Correction Using Association Rule Mining," vol. 1, no. 1, pp. 261-271, 2010.

[16] M. Baojun, K. Dejaeger, J. Vanthienen and B. Baesens, "Software Defect Prediction Based on Association Rule Classification," *Information Technology & Systems eJournal,* 03/2011.

[17] J. Han and M. Kamber, "Section 10.4: Text Mining," in *Data Mining: Concepts and Techniques*, San Francisco, CA, Morgan Kaufmann, 2006, pp. 614-628.

APPENDIX

A.1. UnitTestMining.r

```
source("ARMcode.r")

cl.orig <- read.transactions("DataMiningOutputNoEmptyRows2.txt",
           format = "single", cols = c(1, 2))

cl <- cl.orig[size(cl.orig[]) %in% 2:39]

uniqueTestNames = TRUE

cat("\n************ FIXED INCREMENTS ************\n")
analyze1Headings()
for(percent in seq(0.5, 0.99, 0.05))
    for(support in seq(0.03, 0.101, 0.01))
           analyze1(function(data, confidence)
                          { aprioriFT(data, confidence, support) }, cl, percent)

cat("\n************ Windows ************\n")
clRows <- dim(cl)[1]

half <- round(clRows / 2)
tenth <- round(clRows / 10)

firstHalf <- as.pairlist(c(start = 1, end = half))
secondHalf <- as.pairlist(c(start = half + 1, end = clRows))

csv("5050", firstHalf$start, firstHalf$end, secondHalf$start, secondHalf$end)

last5Tenths <- as.pairlist(c(end = 1 * tenth, end = 2 * tenth,
           end = 3 * tenth, end = 4 * tenth, end = clRows - half))

for(iOffset in 0:4)
{
    offset <- tenth * iOffset
    trainStartRow <- tenth * iOffset + firstHalf$start
    trainLastRow <- tenth * iOffset + firstHalf$end
    sampleStartRow <- trainLastRow + 1
    sampleLastRow <- half + last5Tenths[iOffset + 1]$end

    csv(paste("SW", iOffset + 1, sep=""),
                  trainStartRow, trainLastRow, sampleStartRow, sampleLastRow)
}

cat("\n************ Sliding Window 50%/10% ************\n")
analyze2Headings()
for(support in seq(0.04, 0.101, 0.01))
{
    R.SW <- 0
    P.SW <- 0

    # split data based on sliding 50% training and following 10% sample
    for(iOffset in 0:4)
    {
           offset <- tenth * iOffset
           trainStartRow <- tenth * iOffset + firstHalf$start
           trainLastRow <- tenth * iOffset + firstHalf$end
           sampleStartRow <- trainLastRow + 1
           sampleLastRow <- half + last5Tenths[iOffset + 1]$end

           results <- analyze2(function(data, confidence)
                          { aprioriFT(data, confidence, support) },
                          cl, trainStartRow, trainLastRow, sampleStartRow, sampleLastRow)
           R.SW <- R.SW + results$R
           P.SW <- P.SW + results$P
    }

    # Calculate mean of 5 values
```

```
67      R.SW <- R.SW / 5
68      P.SW <- P.SW / 5
69      F1.SW <- fmeasure(P.SW, R.SW)
70
71      # split data: 50% training and 50% sample as a baseline
72      trainStartRow <- firstHalf$start
73      trainLastRow <- firstHalf$end
74      sampleStartRow <- secondHalf$start
75      sampleLastRow <- secondHalf$end
76
77      results5050 <- analyze2(function(data, confidence)
78                      { aprioriFT(data, confidence, support) },
79                      cl, trainStartRow, trainLastRow, sampleStartRow, sampleLastRow)
80
81      csv(trainStartRow, trainLastRow, sampleStartRow, sampleLastRow,
82                      results5050$support, results5050$numRules,
83                      results5050$R, results5050$P, results5050$F1,
84                      R.SW, P.SW, F1.SW, results5050$numTestFromRules)
85  }
```

A.2. ARMCode.r

```
1    if(!interactive()) options(echo = FALSE)
2   library(arules)
3
4   # Items (column #) with labels starting with beginsWith
5   itemsLike <- function(trans, beginsWith)
6   {
7       grep(paste("^", beginsWith, sep=""), dimnames(trans)[[2]])
8   }
9
10  # Item labels starting with beginsWith
11  itemNamesLike <- function(trans, beginsWith)
12  {
13      dimnames(trans)[[2]][itemsLike(trans, beginsWith)]
14  }
15
16  # Average number of tests in a change list
17  meanItemsPerChangelist <- function(data)
18  {
19      ceiling(mean(apply(data[,itemsLike(data, "T_")]@data, 2, sum)))
20  }
21
22  # Print Line
23  pl <- function(...)
24  {
25      cat(..., "\n", sep="")
26  }
27
28  # Comma seperated values
29  csv <- function(...)
30  {
31      cat(..., sep=",")
32      cat("\n")
33  }
34
35  # Apriori with Files on the LHS and Test on the RHS
36  aprioriFT <- function(data, confidence, support)
37  {
38      apriori(data,
39                      parameter = list(supp = support, conf = confidence, minlen = 2),
40                      appearance = list(lhs = itemNamesLike(data, "F_"), default = "rhs"),
41                      control = list(verbose = FALSE))
42  }
43
44  # Rules fired (LHS) when applied to data
45  rulesFired <- function(data, rules)
46  {
47      if(length(rules) == 0)
48              return(list())
49
```

```
50      # find rules fired by sample data, i.e. where subset is not empty for at least one rule
51      which(apply(is.subset(rules@lhs, data), 1, sum) != 0)
52  }
53
54  # Test names selected by applying rules
55  testNamesFromRules <- function(data, rules)
56  {
57      seqRulesFired <- rulesFired(data, rules)
58
59      if(length(seqRulesFired) == 0)
60              return(list())
61
62      t <- rules@rhs[seqRulesFired]
63
64      if(uniqueTestNames)
65              itemLabels(t)[itemFrequency(t) > 0] # Unique set of test names
66      else
67              itemLabels(t)[sapply(which(t[]@data) %% dim(t)[2],
68                              function(x) { ifelse(x == 0, dim(t)[2], x) } )]
69  }
70
71  # Tests in data
72  testsInData <- function(data)
73  {
74      # find the tests that exist in the sample data
75      itemFrequency(data[, itemsLike(data, "T_")]) > 0
76  }
77
78  # Test names in data
79  testNamesFromData <- function(data)
80  {
81      # find the tests that exist in the sample data
82      if(uniqueTestNames)
83              itemLabels(data[, itemsLike(data, "T_")])
84  [testsInData(data)] # unique set of test names
85      else
86              grep("^T_", itemLabels(data)[sapply(which(data[]@data) %% dim(data)[2],
87                              function(x) { ifelse(x == 0, dim(data)[2], x) } )], value = TRUE)
88  }
89
90  # Test names most frequently occurring
91  testsFromItemFrequency <- function(data, top = 10)
92  {
93      names(sort(itemFrequency(data[,itemsLike(data, "T_")]), decreasing = TRUE))[1:top]
94  }
95
96  # Test that are preselected as important
97  testsFromCIT <- function()
98  {
99      if(!exists("CIT.tests"))
100             CIT.tests <- readLines("CITtests.txt")
101
102     CIT.tests
103 }
104
105 # Divsion - returns zero for a zero denominator
106 div <- function(numerator, denominator)
107 {
108     ifelse(denominator == 0, 0, numerator / denominator)
109 }
110
111 # Recall statistic
112 recall <- function(overlap, Ts)
113 {
114     div(length(overlap), length(Ts))
115 }
116
117 # Precision statistic
118 precision <- function(overlap, T)
119 {
120     div(length(overlap), length(T))
```

```
121 }
122
123 # F-measure statistic
124 fmeasure <- function(precision, recall)
125 {
126     2 * (div(precision * recall, precision + recall))
127 }
128
129 # Calculate statistics Recall and Precision
130 calcStats <- function(T, Ts)
131 {
132     overlap <- intersect(T, Ts)
133     R <- recall(overlap, Ts)
134     P <- precision(overlap, T)
135     as.pairlist(c(R = R, P = P))
136 }
137
138 # Calculate statistics over the whole source
139 calcStatsFromSource <- function(rules, data)
140 {
141     calcStats(testNamesFromRules(data, rules), testNamesFromData(data))
142 }
143
144 # Calculate statistics per itemset - mean itemsets
145 itemsetCalcStats <- function(rules, data)
146 {
147     v <- sapply(seq(1:dim(data)[1]), function(row)
148             { calcStatsFromSource(rules, data[row]) } )
149     R <- mean(apply(v, 2, function(s) { s$R }))
150     P <- mean(apply(v, 2, function(s) { s$P }))
151     as.pairlist(c(R = R, P = P))
152 }
153
154 # Calculater statistics using a testset - mean of itemsets
155 testsetCalcStats <- function(tests, data)
156 {
157     v <- sapply(seq(1:dim(data)[1]), function(row)
158             { calcStats(tests, testNamesFromData(data[row])) } )
159     R <- mean(apply(v, 2, function(s) { s$R }))
160     P <- mean(apply(v, 2, function(s) { s$P }))
161     as.pairlist(c(R = R, P = P))
162 }
163
164 # Headings for first analysis output
165 analyze1Headings <- function()
166 {
167     csv("percent", "rows", "support", "rules",
168                    "R", "P", "F1",
169                    "R-CIT", "P-CIT", "F1-CIT",
170                    "R-IF", "P-IF", "F1-IF", "ItemsPerCL", "TestFromRules")
171 }
172
173 # Analyze data using % training and remainder sample data
174 analyze1 <- function(aprioriFunc, data, percent, confidence = 0.8)
175 {
176     # split data between training and sample sets based on percent
177     train <- data[1:round(dim(data)[1] * percent)]
178     sample <- data[(dim(train)[1] + 1):dim(data)[1]]
179
180     rules <- aprioriFunc(train, confidence)
181     numTestFromRules <- length(testNamesFromRules(data, rules))
182
183     stats <- itemsetCalcStats(rules, sample)
184     R <- stats$R
185     P <- stats$P
186     F1 <- fmeasure(P, R)
187
188     stats <- testsetCalcStats(testsFromCIT(), sample)
189     R.CIT <- stats$R
190     P.CIT <- stats$P
191     F1.CIT <- fmeasure(P.CIT, R.CIT)
```

```
192
193     itemsToTest <- meanItemsPerChangelist(data)
194     stats <- testsetCalcStats(testsFromItemFrequency(train, itemsToTest), sample)
195     R.IF <- stats$R
196     P.IF <- stats$P
197     F1.IF <- fmeasure(P.IF, R.IF)
198
199     csv(percent, dim(train)[1], rules@info$support, length(rules),
200                     R, P, F1,
201                     R.CIT, P.CIT, F1.CIT,
202                     R.IF, P.IF, F1.IF, itemsToTest, numTestFromRules)
203 }
204
205 # Headings for second analysis output
206 analyze2Headings <- function()
207 {
208     csv("train start", "train end", "sample start", "sample end", "support", "rules",
209                     "R-5050", "P-5050", "F1-5050",
210                     "R-SW", "P-SW", "F1-SW", "TestFromRules")
211 }
212
213 # Analyze data using training row range and sample data row range
214 analyze2 <- function(aprioriFunc, data,
215             trainRow1, trainRowLast, sampleRow1, sampleRowLast, confidence = 0.8)
216 {
217     train <- data[trainRow1:trainRowLast]
218     sample <- data[sampleRow1:sampleRowLast]
219
220     rules <- aprioriFunc(train, confidence)
221     numTestFromRules <- length(testNamesFromRules(data, rules))
222
223     stats <- itemsetCalcStats(rules, sample)
224     R <- stats$R
225     P <- stats$P
226     F1 <- fmeasure(P, R)
227
228     as.pairlist(c(support = rules@info$support, numRules = length(rules),
229                     R = R, P = P, F1 = F1, numTestFromRules = numTestFromRules))
230 }
```