Multi-Metric Prediction of Software Build Outcomes

Jacquiline Alannah Finlay

A thesis submitted to Auckland University of Technology in partial fulfilment of the degree of Doctor of Philosophy (PhD).

2012

School of Computing and Mathematical Sciences

Primary Supervisor: Dr. Andy M. Connor

Second Supervisor: Dr. Russel Pears

Third Supervisor: Dr. Jacqueline Whalley

ATTESTATION OF AUTHORSHIP

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Signed:

Jacquiline Alannah Finlay
1st March 2013

ACKNOWLEDGEMENTS

Firstly I would like to thank all the people that have supported me on my journey. I owe thanks to so many!

I am very thankful for the endless encouragement, knowledge and guidance provided by my supervisors Dr. Andy Connor, Dr. Russel Pears and Dr. Jacqui Whalley.

Andy, thank you so much for your insights, creativity, support and advice. I have had much enjoyment in sharing ideas and thoughts with you. Your knowledge in the field of software engineering astounding. *Russel*, your expertise in data mining is second to none. Thank you for sharing your skills and being so patient with me. *Jacqui*, thank you for being so supportive, providing advice when I have needed it most and being a wonderful role model for me.

It has been an honour to have you all as my supervisors.

I would like to thank all the staff, the old and new, in the School of Computing and Mathematical Sciences at AUT. I would also like to thank the people at the Software Engineering Research Lab (SERL), for taking me under their wing and providing a excellent environment for learning and research. Thank you to Gordon G., Ewing C. and Ramon L., for your heart warming conversations and IT support. In addition to this I am thankful for the teaching assistant experience I have had at AUT. I am also very grateful for the financial support provided from the Build IT and the AUT fees scholarships.

I would also like to thank my primary supervisor of my honours dissertation, Dr. Terri Lomax. I would have never envisioned travelling down this path if it was not for you.

Finally I would like to thank my family for all their love and support. Thank you to my parents for their endless love and encouragement. Thank you to my mother, Denise, for giving me strength and being my light. Many thanks to my supportive and loving fiancé, Adam Althouse.

ABSTRACT

This thesis details the design, implementation and evaluation of software prediction models designed to address some of the challenges associated with the identification and mitigation of the risks associated with a software development project. Being able to predict potential failures during a software development project is critical to project success and has been the subject of decades of research. Despite the years of research and its importance to the software domain there is much about software project success and failure that remains unknown. This is partially due to the limited software project data available to researchers and the challenges of capturing the relationships between various software artifacts. It is also partially due to the representation, misinterpretation and lack of data captured and made available within existing software projects. As a result there is very little reported research where an attempt has been made to combine software metrics and social network metrics in order to predict software success and failure.

Software metrics extracted from the source code files of a system during its development are employed to create novel prediction models of software success and failure. The social component of a globally distributed software development team was also investigated using social network metrics. These social network metrics were directly mapped to software metrics in order to predict software build outcomes. This thesis presents the results of the first extensive source code analysis of a live software project (IBMs Jazz repository) using a range of traditional data mining methods. A novel data mining approach is reported in which a combination of both software and social network metrics are used to create software build prediction models. Additionally, data stream mining techniques were used to construct models for software build prediction. It has been found that data stream mining offers a powerful solution for monitoring the evolution of source code metrics and social network metrics over time.

It is found that using aggregated software metrics and social network metrics it is more difficult to predict software build failure than build success. The results also indicated that a combination of software metrics and social network metrics do not enhance prediction accuracy. However, when used in parallel they potentially provide an effective decision making tool to avoid potential failure.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction and Outline

Within technology industries many software projects either fail outright or partially fail. Many projects are unable to meet basic project parameters such as cost, schedule and user requirements. The causes behind such failures widely vary and in many cases are not fully understood (Cerpa & Verner, 2009). Software systems have the tendency to require change over time, during both development and maintenance phases. In some cases managing such change can become a complex task, especially when considering larger systems (Settimi et al., 2004). The collaboration and communication between people strongly influence the decisions made within projects and therefore may have a direct impact on system artifacts (Ebert & De Neve, 2001). This research focuses on investigating the relationships between people (in the form of social networks that exist in an organisation) and various software artifacts (e.g. work items, builds, change sets) within the Software Development Lifecycle (SDLC). This is achieved by using data mining techniques applied to the data available in a software repository. This results in the generation of predictive models as a means of improving the existing ways in which knowledge is distributed within a software project team and to enhance understandings of software artifacts to reduce the degree of failure.

Years of effort from researchers and engineers have been dedicated towards solving these types of problems. Researchers are struggling with a lack of relevant software project data in order to generate new insights about causes of project failure. Such insights could have a significant impact in the way in which project planning is implemented. An area which has been particularly challenging is software developer team communication and collaboration (Nguyen, Schröter & Damian, 2009). There has been much research which examines social networks and their relation to team performance (Bolstad & Endsley, 2003; Cannon-Bowers, 1993; Guimerà, 2005) but as yet there has been little work undertaken in the context of a software development team. Social structures (communication hierarchies, flows and relationships between people) heavily influence knowledge distribution and decision making within a team. As a consequence social-cognitive activities of team members have a direct impact on project success (Sack, 2006).

This study examines what collaboration and practices exist within a software development team and what impact development activities have on project success. This involves an investigation into the negative and positive aspects of social networks, software source code and software quality. The relationships between people (connected via social networks) and various software development artifacts (e.g. work items, software builds and code changes) and how such relationships influence project outcomes is investigated. To facilitate this, a range of software and social metrics are derived from project artifacts and predictive models are built from patterns that exist between the metrics.

By extracting artifacts from software repositories, software metric values can be derived that can be used to characterise the artifacts. Software metrics are commonly used in model-based project management methods and are used to measure the complexity, quality and effort of a software development project (Manduchi, 2002). For example a commonly used, or traditional metric, such as Lines of Code (LoC), measures the size of a software project via the number of executable lines of code within a source code file. Other metrics, such as Cyclomatic Complexity, measure the number of possible (linear independent) paths within source code (Fenton & Neil, 2000). A wide range of software metrics have been proposed over time, each of which is intended to measure some attribute of a software artifact. Software metrics provide an advantage as some can be determined easily or automatically. Some metrics can also be visualised and can be used to measure complexity of a system. However, there are also drawbacks in using such measurements. For example to estimate the amount of time to program certain modules, the relationship between LoC and the amount of effort required is not linear. Some lines of code may take longer to write than others due to the complexity found within the system being developed. In addition the LoC required for a particular function in one programming language, maybe expressed as a single statement in other programming languages. There are many different types of categories of software metrics and an in-depth description of these can be found in the next chapter.

Data mining techniques can be used to identify patterns and exceptions from data and build predictive data models. For this research data is extracted from source code and communication metrics from a real software project repository. Metrics from the repository are extracted using various techniques and a range of data mining classifiers

are used to explore the data. Data mining methods have great potential to generate new insights into software developer's activities by looking at social network and software artifact data for project risk management. In many cases the level of recorded information to perform such tasks is limited. Software project data is either highly summarised or compressed and there is no mechanism for drilling down to greater levels of detail or looking at how projects evolve over time. Fortunately, the IBM Jazz data repository has become available and enables researchers to gain insights into developer communication and activities within the SDLC. Through the mining process a social network may be searched for occurrences of structural holes, project management problems and instances of positive patterns as well. In identifying structural holes it may help to highlight issues related to developer communication aspects of project management. Software source code may also be searched for occurrences of poor quality and potential defects. Insights can be gained from how such metrics influence the outcomes of software artifacts and a range of possible solutions may be built to aid decision makers in mitigating reoccurring knowledge distribution problems.

Software development teams are often responsible for delivering a product within a certain time constraint. To achieve a software release before a deadline the development team uses an integration system to integrate modules of work. This process often requires coordination and collaboration of software development team. This integration system is referred to as a build in IBMs Jazz repository. If a software build fails, it can cost the development team in extra time to diagnose the issue. The study of build failure is timely and the subject of previous research (Wolf, Schroeter, Damian & Nguyen, 2009) and is a specific instance of predicting and avoiding software defects which is the focus of significant research activities. The process of extracting data from the Jazz repository will be discussed later in this thesis, but this process resulted in producing a dataset of 199 build instances. Of those there were 72 builds that failed and it is clearly a concern for an organisation when 36% of its development activities would be considered to have defects that are significant enough to cause a build failure.

A software build is the process of converting source code files into executable code. Depending on the compiler settings unit tests for the build are also executed. The build may either be a success, failure or warning type, depending on its outcome. This build

outcome is used as an indication of successful team coordination in the Jazz repository. Previous studies have shown that it is possible to predict build outcomes by data mining developer communication metrics available in the Jazz repository (Wolf, Schroeter, Damian & Nguyen, 2009). However, no studies have investigated the use of software metrics for the prediction of build outcomes using the Jazz data.

1.1 Research Objectives

The issues raised in this chapter are addressed in more detail in later chapters. The remainder of this chapter sets out the goal of the research and describes the contribution of the work to the body of Mining Software Repositories (MSR) research. In addition this, the outline of the thesis is also presented. The overall goal of this study is to explore the nature of the people and the artifacts, central to a software project and the relationships between them in order to develop models that promote software management practices that help avoid project failure.

The research methodology is discussed in more detail in Chapter 3, however it is important at this stage to note that the research has been exploratory in nature. Whilst a number of initial research questions were formed prior to starting research, the nature of exploratory and constructivist research is such that further questions arose throughout the duration of the research that informed particular trajectories of inquiry throughout the study. The primary research objective that was being addressed at the beginning of the research was to determine whether both software metrics and social network metrics could be combined into a single model to predict the outcomes of a given build. Clearly to address this objective it is first necessary to determine whether there is the ability to predict outcomes using just software metrics or just social network metrics. The value of a combined model can then be determined in relation to the individual models.

> **Research Question:** To what extent can a combination of software and social network metrics extracted from IBMs Jazz repository be used to generate predictive models to determine software build success and failure more effectively than either of the individual models?

1.1.1 Building a Predictive Model

In order to evaluate the performance of a combined model based on both software and social network metrics it is necessary to understand whether software metrics have some potential to predict build outcomes using various combinations of approaches. Therefore the first phase of this research was to extract and propagate software and social network metrics from a range of multi-dimensional software artefacts and find whether any subset of metrics for each type (software or communication) are significant indicators for predicting build success and failure in their own right.

In order to investigate the software repository it is necessary to determine which metrics were appropriate and how metrics from a collection of source code file metrics may be aggregated into values that are at an appropriate level of granularity. When combined with a set of builds over time this produces a multidimensional experimental space as shown in Figure 1.

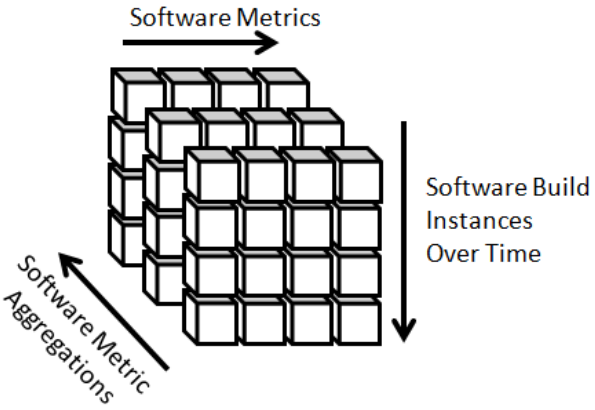

Figure 1 Software Metrics as Multidimensional Data

Similarly it is important to understand which communication metrics can be derived through social network analysis. Just as for the software metrics it was necessary to explore the available communication data as shown in the multidimensional experimental space in Figure 2.

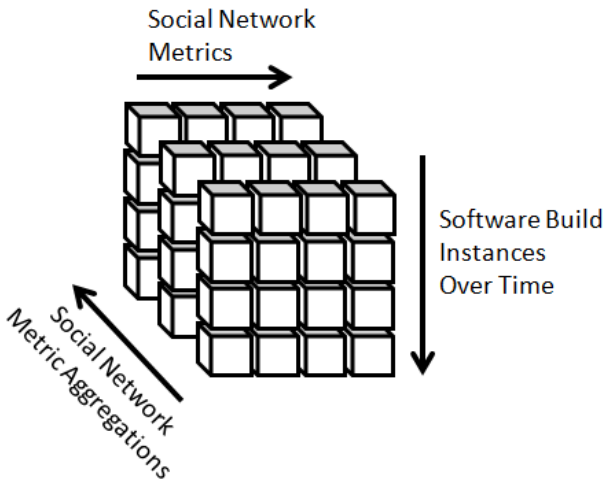

Figure 2 Social Metrics as Multidimensional Data

Assuming that the individual prediction ability of the two sets of metrics is determined; it is then possible to investigate whether the combination of software and social metrics has any value with respect to the predictive power of the individual models produced.

1.1.2 Enhancing the Predictive Model

The early phases of this research resulted in the development of a number of different predictive models as detailed in Chapter 3. The development of these models raised a number of further research directions that were worthy of exploration. Consequently, the next research objective identified was to improve on the prediction accuracy and sensitivity of the preliminary data mining experiments. Traditional data mining methods were explored for a wide range of software artifacts and models generated to predict software build outcomes. A series of feature filter strategies and classification algorithms were applied to combinations of software and social network metrics.

Building on the initial experiments, additional techniques were explored to improve the prediction performance in terms of both accuracy and sensitivity. Three approaches are investigated: 1) Features are selected based on their observed frequency based from the initial experiments. 2) The after state feature filters are applied to the before state software metrics, to see if future filters can enhance past models. 3) Attaining large amounts of real or live software data is challenging, to increase the sample size of the data, a strategy for synthetically generating more instances for the mining processes was also explored. Its impact on accuracy and sensitivity was examined and compared with the first major stage of the data mining experiments.

1.1.3 Evolving a Predictive Model over Time

This next phase of this research was to investigate how software and social network metrics evolve over time. Much of the mining software repository literature treats project metrics as static and not changing over time. This assumption does not accurately reflect reality. Software and communication metrics change over time therefore there is much potential for evolving predictive models to more accurately reflect reality. More specifically the objective of this phase was to build an adaptive time-based prediction model for identifying potential project risks. This would provide the opportunity to operate in real time by integrating the predictive model with the software repository so that prediction events can be managed without having to wait for specific build events, therefore facilitating a risk-based approach to managing development activities.

1.2 Overview of Research Contribution

This thesis contributes to knowledge about success and failure patterns of a software project as well as their implications within a software project repository. This knowledge could enable identification of such patterns and possibly support intervention in cases where failure patterns are observed. A result of this research is a system to classify build success and failure patterns and discover them within a project repository. The predictive models can be used as decision support for planning, diagnosis and rectification of identified or emerging problems within the project.

In terms of data mining this research makes a contribution to the field by exploring a novel application of data stream mining techniques using software project data. Data mining multi-dimensional data is a difficult task even for traditional mining methods. The research objectives and an outline of the approach are presented in 1.1. To further explore these issues the data is also encoded and data mined as a time-series stream to see how metrics evolve over a software projects lifetime. To date there have been no studies that have been conducted for data stream mining multi-dimensional software or social network metrics to predict software build outcomes.

1.3 Thesis Organisation

This thesis consists of seven chapters, followed by appendices (in text and on CD) and references. Chapter 1 has detailed the research problems, questions, objectives and contributions of this thesis. Chapter 2 provides the context and background for this work. In addition, chapter 2 emphasises the existing needs and gaps in knowledge within the software and knowledge engineering domain and the research described in this thesis. Chapter 3 details the research via artifacts methodology that was adopted and its application to this work. Chapter 4 presents a summary of the results of the experimental phases. Chapter 5 presents a discussion of the implications of the results and a summary of the research. Finally chapter 6 presents a discussion of potential future research, limitations of this study and conclusions based on the contributions that have been made by this thesis.

1.4 Chapter Summary

This chapter has provided a summary that forms the foundation of this thesis including an introduction to the research problem and the primary research objective. The next chapter examines the mining software repositories related literature relevant to this research problem.

2 Background and Motivation

2.1 Introduction

There are still many aspects of software management that are not yet fully explored and understood. This is due to lack of knowledge, largely because of the complexity of the domain (Stamelos, 2009). The US Department of Defence lost over four billion dollars a year due to software failures (Dick, Meeks, Last, Bunke & Kandel, 2004). Empirical software engineering research is not yet able to reliably identify critical factors that influence or define a projects' success or failure. Software project management problems are often complex and requires expertise in multiple disciplines (Jørgensen, Faugli & Gruschke, 2007). Both internal and external organisational issues have impact on a projects' success. For example issues occur when people with the wrong skill sets are determining project pathways. In some instances, project managers have taken a certain attitude towards allocating costs and resources. In one instance a project manager may underestimate the effort required for a particular job in order to enhance the performance of their team, while compromising the quality of work (Agrawal, 2007). With project timeframes that are too short software engineers may be more inclined to get something out as soon as possible (a rushed job) rather than taking the time to design, test and adopt good development practices. Poor cost and schedule estimation can also be due to skills of the estimator not matching the job (Barretoa, Barrosb & Wernera, 2008; Chicano & Alba, 2005).

Software engineers and project managers spend much of their time monitoring tasks and activities within changing environments (Denning & Riehle, 2009). It has been found that it is in human nature to be unaware of an environment from one view to the next and people often do not detect large changes to objects and scenes (Simons & Chabris, 1999). In the context of software development projects there can be an inordinate number of complex dynamic events taking place including those events that are influenced by social-cognitive and specific software activities. Reducing the number of software failures within a project was one of the most challenging problems within software engineering research (Mockus & Weiss, 2000). This research focuses on the development of techniques that can be used to analyse development events which are

often hidden (e.g. communication between members of a team), or are not easily detectable (social hierarchies and communication flow), or are ignored, within a software development team.

Software repositories such as source control systems have become a focus for emergent research as being a source of rich information regarding software development projects. The mining of such repositories is becoming increasingly common with a view to gaining a deeper understanding of the development process and building better prediction and recommendation systems for decision support in software development teams. The Jazz development environment has been recognized as offering new opportunities in this area because it integrates the software source code archive and bug database by linking bug reports and source code changes with each other (Herzig & Zeller, 2009). Whilst this provides much potential in gaining valuable insights into the development process of software projects, such potential is yet to be fully realized.

The following survey of literature is broken down into four major themes. To begin the fundamental aspects of empirical software engineering research are covered. In this section the quantitative empirical evidence that can be extracted from source code and used for knowledge discovery processes are introduced. The second section explores the social elements of software projects and presents the concepts of global software development. In this section empirical evidence that can be drawn from social media and interaction is presented in relation to software engineering. In addition to this various software repositories options are explored. This leads to the third section, where a specific software repository for this research and is covered in more detail in relation to the evidence that it provides and previous studies that have utilised it. Finally, to build from the knowledge discovery processes and MSR literature, both traditional data mining and data stream mining techniques are presented.

2.2 Empirical Software Engineering

In modern science empirical studies generate understandings about how and why things work. Empirical research fundamentally comprises of tests that compares what is believed to be true (an expected outcome) against what is observed (an actual outcome). To do this empirical research requires the following steps:

1. Formulating a hypothesis or a question
2. Observing a situation
3. Abstracting observations into data
4. Analysing the data
5. Drawing conclusions with respect to the tested hypothesis

For an empirical study to be a valuable contribution to the body of software engineering knowledge, the context of the research needs to be defined and its terminology explained (Kitchenham et al., 2002). The context of the research can be described by the background in which the research takes place, how the hypothesis is derived and information relevant to the research. Once the context of the problem domain is determined researchers are able to question the nature of software repository artifacts. These questions, in an empirical study, are formalised as hypothesis. Essentially there are two types of hypotheses, abstract hypotheses (high-level) and concrete hypotheses (low level) (Perry, Porter & Votta, 2000). An abstract hypothesis uses natural everyday language for this research, for example: "Successful software builds are crucial part of the development process". A concrete hypothesis is stated in terms of the research design, for example "From a range of software metrics, only a few will be statistically significant predictors of build failure".

The design of a study consists of a detailed plan for generating the data that will be used to test the hypothesis. The design also describes the tools, resources and processes involved. This includes the rationale and methods used for sampling a population to derive data. For researchers to better interpret the data and the results, the threats to the studies validity should also be stated explicitly. Defining limitations aids in clarifying any ambiguity within the results. Results of an empirical study can be analysed using a classical analysis technique or a Bayesian analysis technique. The classical analysis

involves use of statistics to interpret the findings. Whereas the Bayesian analysis method systematically uses prior information from previous experiments used to interpret the results. The classical analysis method is commonly used for software engineering studies (Kitchenham, et al., 2002).

Over the past 40 years there has been extensive use of software metrics in empirical software engineering research (Shepperd, 2011). Applying empirical methods to software metric driven data has remained challenging. Ideally studies that apply the same method, to the same research context, should obtain the same (or similar) results. This is not true for software engineering research, as software metrics can be misunderstood or misused (Fenton and Neil, 2000). There are no single solutions or definitive answers (Tichy, 2000). For example, in some cases, empirical software engineering studies may not scale well (data is collected over a limited time frame) and test results from various software projects (different project contexts) are difficult to compare (Perry, et al., 2000). When reviewing empirical software engineering work, it is important to keep in mind that, any software experiments based from real-world data is not known to be ready for data mining immediately, as the data is often noisy, unpredictable and complicated. Another challenge to empirical software engineering studies is that often the technology that is being studied changes during the research making it difficult (or in some cases impossible) to compare and relate results over time (Pfleeger, 1999).

2.2.1 Software Metrics

Software metrics are used extensively within empirical studies as they represent quantitative measurements of software artifacts. To determine how software artifacts should be interpreted and whether or not such interpretations are valid, it is necessary to cover the software artifacts that are worthwhile measuring for this research. Software source code is a useful software artifact however it is challenging to use towards software project decision making due to the fact that software systems often comprise of massive amounts of code. One way to overcome interpreting large amounts of code is to derive software metrics instead (Subramanian & Corbin, 2001). Software metrics are abstract quantitative measurements derived from software code.

A wide range of metrics can be used during many stages of software projects and are commonly found within planning, cost estimation, effort estimation, quality assurance,

software debugging, project task scheduling and software performance optimization studies. However, software metrics are not commonly used within the industry itself (Fenton & Neil, 2000). A frequent reason for this is because software metrics are easy to misuse or misinterpret as they can be used in a variety of ways and guidance about how to use metrics can be ambiguous. A common problem found within the literature that relates to software metrics is that the terminology is often mixed making it difficult for comparative analysis (Grimstad, Jørgensen & Moløkken-Østvold, 2006). In a multitude of studies software metrics can be extracted and derived in numerous ways and can often have multiple meanings associated with them. The multiple interpretation challenge also becomes apparent when studying software artifacts in general. It is therefore essential to clearly define software metrics and artifacts that are derived and how they will be used in a given context.

When used correctly software metrics can provide powerful insights into the development lifecycle. While the first book about software metrics was published in the 1970s (Gilb, 1977) there is research literature related to software metric that dates back to the 1950s and 1960s. Between then and now software metric measurements have been used to predict software characteristics and development quality issues within a wide variety of studies. The first empirical study of a large software system (the IBM OS 360) utilised software metrics such as the number of modules, the time taken to prepare for software releases and the number of modules between releases (Belady & Lehman, 1976). This study in many ways was ahead of its time as it generated understandings about the evolution of software systems that still hold to be true to this day. These insights were documented as the three laws of software evolution:

1. Law of continuing change
    - Systems that are used undergo change continuously until it is more cost effective to free and recreate it. A system goes under continuous maintenance and development and is driven by changes in capabilities and usage.
2. Law of increasing entropy (complexity)
    - As a system evolves over time the continuous changes that are made need to be managed. Extra recourses are required in preserving a systems (elegant) structure.

3. Law of statistically smooth growth (self regulation)
    - As a program evolves its metrics maybe appear to be randomly distributed in time and space. However, statistically they are self-regulating when trends are mapped over longer periods of time.

The three laws were revisited and expanded on in a later study (Lehman, 1996). Through the use of empirical methods new understandings and insights were generated about software development lifecycles. Then in the late 1970s complexity metrics were used to predict characteristics of programmer performance (Curtis, Sheppard & Milliman, 1979). It was found that as the size of a program grew, Halstead and McCabe metrics were found to be good predictors for measuring psychological complexity. In the 1980s early prediction models were used to assess software cost estimation (Boehm, 1984). In the 1990s it was found through the use of software metrics that 45 to 60 percent of the total cost of a large software project is spent on maintenance (Coleman, Ash, Lowther & Oman, 1994).

To date software metrics have been incorporated in many areas of software engineering research. For example, Nagappan, Ball and Zeller (2006) investigated whether object-orientated metric predictions of component failures made from one software project were applicable to other projects in order to see if mining results were generalisable. Large software projects included within this study included: Internet Explorer 6, Internet Information Services, Process Messaging Component, Microsoft DirectX and Microsoft NetMeeting. For each module a number of traditional source code metrics were computed as well as *Arcs* and *Blocks* metrics that refer to a functions' control flow graph used for computing cyclomatic complexity. In doing so a new metric was introduced that counted the number of instances where a function utilises a global variable (AddrTakenCoupling). Their results showed that metrics proved to be useful in capturing similarity between components. In addition to this predictors were accurate only when obtained from the same or similar projects meaning that project context influenced the selected predictors. There was not a single set of metrics that fits all software projects. In a similar study of software fault-proneness, results showed that it is possible to use software metrics in building statistical models based on historical software project data before testing (Denaro, Morasca & Pezz, 2002). These types of

models provide insights into software project planning, monitoring and testing phases. In another study software metrics were derived from 5ESS® software updates in order to predict change quality to aid towards decision making in code inspection, testing and delivery (Mockus & Weiss, 2000). Using software metrics that reflect software change, size, duration, diffusion, type and developer expertise (e.g. the number of deltas a developer makes towards software code) it was found that the developer expertise metrics alone were a strong predictor of change quality.

Software metrics provide a compressed or abstract view of a systems design, size, stability or maintainability. However, there are various limitations introduced when incorporating software metrics into decision making processes. For example, software metrics do not directly take into account the context of the project. More specifically, the context that surrounds the skills of the developers, expertise of the managers, social networks and communication protocols are not captured. Without project context traditional software metrics and the models that are built may be easily misinterpreted. For example if a large number of defects are found in a software module prior to release, the software metric may also indicate that there will be a large amount of defects after the release. The relationship between software metrics and a project outcome may not be so unambiguous. For example if a person has eaten a large lunch at 1:00pm, is it likely that person will eat a large dinner at 6pm. Now, depending on that person's diet, whether the person eats a lot, the answer may be yes. However, that person may not feel like eating a large dinner as they are full from the lunch (Fenton & Neil, 2000). The same logic applies to that of the software release. For example if a software module has many defects pre-release it does not necessarily indicate that it will have more defects post-release. This is why it is important to make use of a range of metrics, derived from both software and social artifacts, when building decision making models to capture as much project context as possible.

In order to extract insights from a range of software metrics, they first need to be derived from source code from a software project. The next section (2.2.2) presents the concept of software data repositories that can be used as a means to extract metrics. Then in order to find interesting patterns mining methods are used to model the metric data.

2.2.2 Mining Software Repositories (MSR)

To expand on empirical software engineering studies the Mining Software Repositories (MSR) research community analyzes data from software repositories to reveal interesting patterns and information about the development of software systems. MSR has been a very active research area since 2004 (Kagdi, Collard & Maletic, 2007). Until the emergence of MSR as a research endeavour, the data from software repositories were mostly used as historical records for supporting development activities. Analysis of MSR research over the years has shown that the general approach of extracting knowledge from a repository has the potential to be a valuable method for analyzing the software development process for many domains (Poncin, Serebrenik & van den Brand, 2011). Software repositories contain artifacts that are results of software development processes. A software artifact is a term that is used to describe a function of a piece of software. For instance an artifact may be expressed using unified modelling language (UML) diagrams such as classes, use cases, behaviour and interactions (Settimi, et al., 2004). Software artifacts are also used to describe the processes and methods of developing software. For example software documentation, development methodologies, project plans, risk mitigation plans and business plans.

Software artifacts can also hold information about versions of the system, changes made to source code and interactions between people during development phases. These can be accessed via retrieving information from version control systems, bug-tracking systems and communication archives (Kagdi, Collard & Maletic, 2007). Software repositories grant researchers the ability to query developer-level actions such as code transformations and re-factorings, bug fixes and development features (Robbes, 2007). MSR has great potential to generate understandings of how software evolves over time (Hassan, 2006). Combining information that can be retrieved from software repositories and machine learning methods there is potential to improve productivity within software development teams.

An examination of the data sets currently available to researchers is critical as they highly influence the path of this research and subsequently inform the literature review undertaken. The ISBSG and the COCOMO81 NASA data sets are frequently found within the literature (Bibi, Tsoumakas, Stamelos & Vlahavas, 2008; Braga, Oliveira, Ribeiro & Meira, 2007; Lokan, 2005). Researchers found that both data sets have a lack

of evidence to support research into how projects evolve, how changes are handled and how changes can influence a projects success. There is a lack of historical or accurate data for comparative analysis for software project studies (Subramanian & Corbin, 2001). Instead most research employs historical data with artificially added noise to make data appear realistic (Jørgensen & Shepperd, 2007). However, more recently there have been software repositories that have become available which offer a data rich environment for researchers.

There are different types of software repositories available for researchers. These include historical, run-time and code repositories (Hassan, 2008). Historical repositories contain source code, archived communication data that is relevant to the evolution and progress of a software project. Run-time repositories provide information about the execution and usage of an application at a deployment site or sites. Code repositories contain source code for a various number of applications created by a team of developers. When deciding upon which repository to explore it is important to define the scope and aim of the research and whether a particular repository can support its goals. Depending on the complexity of the repository, it may involve a wide or breadth first exploration that focuses on the relationships and dimensions between artifacts to provide a general view and understandings (Kagdi, Collard & Maletic, 2007). Then, depending on the aim of the study, a particular area of the repository is narrowed down for an in-depth analysis.

Even though repositories have existed for many large software projects, the data stored within these repositories have not been of focus within the software engineering research community until recently (Hassan, 2008). This is primarily because of the restricted access to such repositories (containing sensitive and detailed information) and the complexity of the task of extracting the data. There is often limited support for automation of extracting software metrics. Due to these limitations often validating derived models can also be challenging. However, despite these shortcomings, the MSR field has shown that these repositories are able to be data mined to uncover interesting and useful patterns and provide valuable information to software project teams. The aim of the MSR community is to transform these repositories into models to guide decision making during the software lifecycle by detecting interesting patterns from the extracted

data (Williams & Hollingsworth, 2005). The predictive models generated can facilitate developers into making changes to related artifacts to reduce software faults.

Software bugs come in many forms, they can represent an error, a logic or syntax fault or resource issues within source code files. Valuable knowledge for developers and managers, regarding software bugs, include knowing where potential bugs may exist and how much effort it will take to fix them. The ability to predict failure-prone components of software system is valuable knowledge to software developers as they are then able to mitigate the failure-risk. Using an Eclipse bug database (Bugzilla), a combination of complexity metrics were used to predict defects (Zimmermann, Premraj & Zeller, 2007) to a sufficient degree of accuracy. Mining object-orientated metrics were also found to be useful for predicting defect density (Basili, 1996; Subramanyam & Krishnan, 2003).

Predicting how long it will take to fix a software bug can be a difficult topic for developers and managers. From mining data extracted from the JBoss Project bug database, results showed that with a sufficient number of issue reports predictions for effort to fix a bug were very close to the actual time taken (Weiss, Premraj, Zimmermann & Zeller, 2007). Data mining experiments based from the Mozillas bug tracking system (Bugzilla) repository show that during the early stages of a development project little effort was required to fix bugs. However, at later stages of the same project the amount of effort required to fix the same number of bugs increased (Ahsan, Afzal, Zaman, Gütel & Wotawa, 2010). What remains unclear, to date, is how to effectively measure the characteristics of a defect in order to determine where it exists and how long it may take to fix during a specific part of projects lifetime. As mentioned previously software defects come in different forms, therefore extracting metrics for defects at different level of granularity within a bug database will heavily influence the results of the data mining (Moha, Guéhéneuc & Leduc, 2006; Ying, Murphy, Ng & Chu-Carroll, 2004).

Research that focuses on the analysis of metrics derived from source code analysis to predict software defects has generally shown that there is no single code or churn metric capable of predicting failures (Basili, 1996; Denaro, et al., 2002; Nagappan, Ball & Zeller, 2006) though strong evidence suggests that a combination can be used

effectively (Mockus & Weiss, 2000). While software projects can be rated by a range of metrics that describe the complexity, maintainability, readability, failure propensity and many other important aspects of software development process health, it still continues to be risky and unpredictable (Buse & Zimmermann, 2010). In addition to Buse and Zimmermann's (2010) paradigm of software analytics, it is also suggested that metrics themselves need to be utilised to gain insights and as such it is necessary to distinguish questions of information which some tools already provide (e.g., how many bugs are in the bug database?) from questions of insight which provide managers with an understanding of a project's dynamics (e.g., will the project be delayed?). Buse and Zimmermann's (2010) continue by suggesting that the primary goal of software analytics is to help managers move beyond information and toward insight, though this requires knowledge of the domain coupled with the ability to identify patterns involving multiple indicators.

A majority of source code repositories being researched are from a Version Control System (VCS). A VCS records the activities of developers and histories of files of a software project. A VCS allows developers to "checkout" modules so that they are able to retrieve source code files and make changes. Developers are able to update a VCS which retrieves all changes since the last checkout and synchronises by replacing older files with newer versions and removing files that have been deleted. Developers are also able to commit their changes where files are added, removed or updated (Chen et al., 2001). There are also many other potential functions a VCS can provide to aid a team of developers for integrating and backing up their work.

Zimmermann, Weibgerber, Diehl & Zeller (2004) investigated the use of associative rule mining techniques on version histories within a VCS to model specific source code files that were likely to be changed in the future. In another study, also utilising a VCS, models were built to determine source code changes by data mining change history (Ying, Murphy, Ng & Chu-Carroll, 2004). Depending on the mining task, achieving the right level of granularity within a repository may provide better results. For example extracting measurements from smaller units of source code may achieve higher classification accuracies. However, upon refining the level of granularity, it may weaken associations to the "bigger picture".

Depending on the VCS mined, its history may have limited functionality. For example a developer committing code, may not necessarily be the developer who actually wrote the code (Zimmermann, 2007). In addition to this only a certain set of commands may be historically recorded within a VCS. Another issue is that is it may be difficult or impossible to use a VCS for finding projects suitable for particular case studies. It is therefore important to gain understandings about the functionality available from the repository that can be easily utilised for metric extraction.

2.3 Global Software Development

Software development for large systems involves team work, communication and collaboration. It is not uncommon that a large development team is globally distributed. Global Software Development (GSD) systems are having positive impacts on the way products are conceptualised, designed, tested and developed (Herbsleb & Moitra, 2001). In many cases the concept of a GSD team is introduced to optimise the cost of development, via outsourcing and integrate research and development centres from around the globe (Ebert & De Neve, 2001). However, as a total cost solution a GSD team may not be as beneficial as one may suspect. In some case studies it has been found that GSD teams require more manager roles to handle extra coordination requirements. In addition to this it has also been found that in some remote sites it can take a longer time to achieve useful contributions to development (Conchúir, Pär, Olsson & Fitzgerald, 2009). This can be due to the level of expertise of developers or due to collaborating over different time-zones (Herbsleb, Mockus, Finholt & Grinter, 2001). In some cases there has been a negative stigma with GSD as local developers may feel that their jobs are threatened as roles are moved to overseas. As a consequence this can have a negative impact on developers' ability to trust other members within a globally distributed team (Lanubile, Ebert, Prikladnicki & Vizcaino, 2010). To make matters worse it might not be feasible to have any face-to-face meetings.

Despite these barriers engineers with different cultural and educational backgrounds work together to improve and innovate new products (Ebert & De Neve, 2001). Tactical approaches have also been created to alleviate the distance problem for GSD teams. Such tactics include reducing intensive collaboration by increasing modularity in development process as well as the final product and reducing cultural and temporal distances (Carmel & Agarwal, 2001). It is currently uncertain what aspects of a software engineers' behaviour result in an effective team. However, it has been shown that

software engineers who have been optimistic in the past will tend to make optimistic predictions in the future (Jørgensen, et al., 2007). In addition to this, software engineers who were found to be better at solving a task will have less optimistic predictions (as they have a better understanding of the task) and software engineers who have great confidence in estimation are more likely to generate optimistic predictions.

In software projects there are many situations where a group of people are required to work together to collect a wide range of information and reach an agreement about how it is interpreted. Often within software projects there is too much information for a single person to understand and requires the expertise of many people. To make a project even more challenging software projects usually have time-critical tasks and required support for collaborative reasoning. Communication is the primary method used to distribute knowledge within a team (Rus & Lindvall, 2002). Even though research has been conducted on the relationship between communication and coordination in teams, the research that has been applied to the area of software engineering has led to inconsistent results.

2.3.1 Social Networks

In software project management the behaviour of a development team is an influential factor in terms of project success. Group structure and patterns of social relationships have a strong influence on knowledge and information distribution within a team (Warner, Letsky & Cowen, 2005). Therefore taking social network patterns into account during project phases has the potential for identifying success and failure. Current trends within the software development literature focus on gaining insights into how tools are used within development environments to share knowledge and perspectives to teams. It has been found that teams with higher levels of socio-technical congruence have a better chance of success (Herbsleb et al., 2008). Congruence in a team is achieved when people are able to coordinate their capabilities effectively towards the projects goals. Software development environments are now integrating collaboration tools that are used during all phases of the development lifecycle. This aids and captures software developers sharing knowledge and captures project problems and decision making to the extent of what the network is able to capture.

A social network is essentially a coding scheme that is used to aid researchers in extracting coordination data between people (Hanneman & Riddle, 2005) and gain

better understandings of how communication influences an outcome of particular task (Sparrowe, Liden, Wayne & Kraimer, 2001). Social networks are representations of social interactions between people. Networks can be constructed using directed or undirected acyclic graphs. Each node represents a person, an object of interest, or a communication channel and each edge represents the relationship between people (Begel & DeLine, 2009). Social network analysis provides information about positive and negative aspects of the team collaboration and its relationship to a particular artifact (Scott, 1988). In addition to this social networks can be used to analyse how an individual is connected with others and the degree to which an individual's position within a network provides certain advantages or disadvantages (using centrality metrics).

Once a social network is constructed, using social network analysis and mining methods predictive models can be built for finding occurring collaboration problems. Metrics that are derived from social networks can measure the importance of actors within a team. Factors such as communication structure, coordination, harmony and communication frequency are related to project success (Nguyen, et al., 2009; Serce et al., 2009; Stamelos, 2009). Communication plays an important role in determining the success or failure of software integrations for resolving conflicts between components of a software system. In addition to this a combination of network measurements can be combined to create a predictive model.

Social networks and behaviour of people, which can cause projects to fail, need to be accounted for during risk mitigation when planning a software engineering project. Face-to-face meetings, teleconferences, web conferences, emails, forums and online chat are examples of common methods used to distribute knowledge within a team. Some researchers explicitly avoid the concept of collaboration and communication technologies in terms of improving the performance of GSD teams, whilst acknowledging they are the most intuitive approach (Carmel & Agarwal, 2001). It has been found that social networks have great potential towards finding relationships between software build failures and communication problems (Wolf, et al., 2009). Capturing project knowledge and establishing techniques that extract patterns from this data will contribute to the body of knowledge in empirical software engineering research. Even though social influences can have great impact on a projects' success,

there is little research within the software engineering literature which looks at the relationship between the outcome of coordination (successful or not) and the characteristics that lead to coordination failures.

Analysis of the density of a network also provides knowledge on the roles individuals play within a group, task behaviours, accountability and expectations (Hanneman & Riddle, 2005). When direct (face-to-face) communication between project members is not possible, social network analysis techniques can be utilised to identify other people within the project that have the potential to act as communication brokers (Marczak, Damian, Stege & Schroter, 2008; Wolf, et al., 2009). Failures in communication structures may become apparent through the pattern recognition and observation of structural holes, for example where there are missing links between nodes and occurrences of redundancies within the social network when a failed outcome was expected. Other possible anti-social network limitations include communication barriers that global software development teams' experience that local in-house development teams may not experience. For example time zone differences, lack of face-to-face meetings and various corporate cultures (Serce, et al., 2009). These barriers can also result in breakdowns of team trust, sub-optimal team productivity and exchanges of misleading knowledge.

There are similar issues between studies that have used software metrics or social metrics. For example just like software metrics, social networks that have been used within multiple studies are not easily comparable (Wolf, et al., 2009). In addition to this there have been limited research that has focused on the link between people’s positions within a social network and job performance (Sparrowe, Liden, Wayne & Kraimer, 2001). Again it appears that the context in which the metrics are derived have a heavy impact on how decision models are built and interpreted. From social network measurements advice networks can be built to show how resources can be used in a way that can enhance individual job performance (Marczak, Damian, Stege & Schroter, 2008). Such task-based social networks can be constructed from three elements:

1. Project members
    - These include all people who are involved with a specific task (developers, testers and managers etc).
2. Collaborative tasks
    - These are the units of work that project members must collaborate and communicate on.
3. Task-related communication
    - Captures the network between collaborative tasks and team members and represents tasks and project members as nodes and the communication between each node is represented as an edge.

Communication data is a key aspect for constructing social networks, however only a sub-set of this data is captured in software repositories. For example face-to-face meetings are challenging to record and measure without too much interference. However, software repositories have been known to provide a rich source of communication data automatically collected from other communication channels. For example emails, source-code repositories and build systems. Collaborative Development Environments (CDE) provides tools for developers, working in global teams, that facilitate communication and collaboration. Examples of CDEs tools include SourceForge (Howison & Crowston, 2004), Assembla (http://www.assembla.com/), GForge (www.gforge.org), Trac (trac.edgewall.org), Google Code (http://code.google.com), Rational Team Concert (RTC) (Jazz.net/projects/rational-team-concert), GitHub (http://github.com/), Launchpad (launchpad.net), CodePlex (www.codeplex.com) and CodeBook (Begel & DeLine, 2009; Lanubile, et al., 2010). Different CDEs provide different levels of functionality. For example a CDE may or may not provide functionality for source code version control, bug tracking, build tools, planners, knowledge portals, communication and web-based tools. In addition to this different CDEs may or may not provide direct associations between repository artifacts. For example CDEs such as SourceForge, Assembla, Gforge, Trac, Google code, HitHub and Launchpad make use of separate CVS or Subversion (SVN) systems (e.g. Bazaar or Git). Whereas CodePlex and RTC (Jazz) make use of built-in CVS and provide direct links to other artifacts within the repository. In addition to this Codeplex and RTC (Jazz) also offer build tool artifacts while other CDEs may not.

Despite a CDEs ability to cope with geographical distances, many CDEs do not offer support to reduce the socio-cultural distances (Calefato, Gendarmi & Lanubile, 2009). Github combines features from social networking websites using Git. Codebook, a prototype from Microsoft, also aims to develop social networking services over source code. RTC (Jazz), by IBM, is a more recent and full-featured CDE that provides presence and workspace awareness in one environment. For a resource to be considered valuable for this work it will ideally consist of both software and communication data. In addition to this both types of data need to be related to each other and can be used to measure areas of project success and failure.

2.4 Rational Team Concert (Jazz)

IBM's Jazz Repository (RTC) is a fully integrated software team collaboration and development tool that automatically captures software development processes and artifacts (Herzig & Zeller, 2009; The IBM Rational Jazz Project, 2009). The Jazz repository contains snapshot based evidence that provides researchers the potential to gain insights into team collaboration and development activities within software engineering projects (Nguyen, et al., 2009). With Jazz it is possible to extract the interactions between contributors in a development project and examine the artifacts produced. Such interactions are captured from user comments on work items, which is the primary communication channel used within the Jazz project. As a result Jazz provides the capability to extract social network data and link such data to the software project outcomes. What makes the Jazz repository unique is that there is full traceability between a wide range of software artifacts. The Jazz team itself is globally distributed and provides the opportunity to data mine developer communication, bug databases and version control systems at the same time.

2.4.1 Mining the Jazz Repository

The key repository artifacts for mining that are extractable from Jazz include work items, change sets, source code files, contributors, communication and build items (IBM, 2007). Work items are used by contributors for keeping track of tasks and issues within the team. Different types of categories are assigned to work items to indicate their status. A work item may be categorised as a defect, enhancement, plan item, retrospective, story, task, or another category which can be specified by the user (Jazz Work Item Overview, 2009). Work items can be treated as the software systems

requirements. For example a defect work item identifies a possible bug within a software system. When a bug is identified, a new requirement is made in order to fix it. An enhancement work item describes a request for a new feature to be made. Plan items are a high-level description of a unit of with that is for a given development iteration. Retrospective work items record elements of the project that were successful or unfavourable for a completed iteration. A story work item describes part of a use case. Task work items contain information for a specific unit of work and a track build item stores the status of a software build.

Software builds are a key process for converting source code into usable software. During a software build source code is compiled and converted into executable code. Within a development environment a build tool is used to compile and link various files to form a working unit (Buffenbarger, 2005). Simple software programs may consist of a single or small set of files to be compiled. However, for large complex software there may be a vast number of files that are combined in different ways to produce various versions of the software system. Software builds can be manual (compiled on a single or shared computer) or automated (compiled daily). In complex systems continuous integration is performed where small changes are applied and compiled frequently. This is done as a quality control process and aims to reduce time taken to deliver software by applying quality assurance during its development, as opposed to afterwards. Continuous builds provide earlier detection of broken or incompatible code (Duvall, Matyas & Glover, 2007). Metrics generated from automated testing and continuous builds focus aids software developers with delivering quality code (Nguyen, Schröter & Damian, 2009). Software production failures, such as build failures, can delay delivery time by slowing the development cycle (Brooks, 2008). To date no source code analysis has been conducted on the Jazz project data, though some analysis has been conducted in terms of the social network analysis. One of the aims of this research is to perform an in-depth analysis of the repository to gain insight into the usefulness of software product metrics in predicting software build failure.

A change set is a collection of code changes in a number of files. In Jazz a change set is created by one author only and relates to one work item. A single work item may contain many change sets. Source code files are included in change sets and over time can be related to multiple change sets (Jazz Source Control Overview, 2009). People

who are involved with the project are called contributors. Comments are stored communication between contributors of a work item. Comments on work items are the primary method of information transfer among developers. If a work item is complex developers tend to communicate more often (Kwan, Schröter & Damian, 2009).With these artifacts, Jazz provides a rich source of software project data and information that can be extracted for a variety of purposes. Such flexibility combined with the capacity to drill down into many layers of a software project is suitable for the nature of this research.

The Jazz source control repository holds objects that support team processes, system components and work spaces. Team members (contributors) are able to work with different types of artifacts available in multiple work spaces (team areas). Figure 3 illustrates that through the use of Jazz it is possible to visualize members, work items and project team areas. As contributors make changes to a files contents or properties, multiple versions of files are stored within the repository. These file versions provide researchers with the capability to map changes of a software project over time in order to see how software evolves. A team area is a work space that stores items that have been placed under source control. Every team area has an owner and only that owner can make changes to the team area. Changes made by contributors within a team area are kept private until they decide to share changes by "delivering" them to a workspace. When multiple contributors make changes, they can be accepted into their own private work space. In some instances contributors will be required to communicate and collaborate changes made in order to solve potential source code file conflicts.

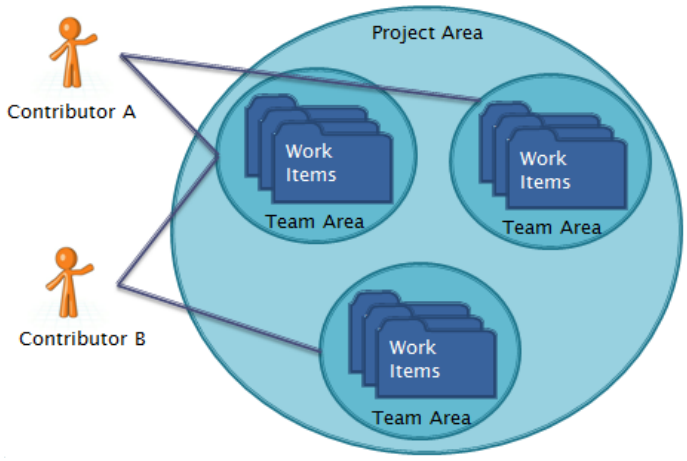

Figure 3 Jazz Repository: Contributors, Project Area, Team Areas and Work Items

Source code change sets and documents are associated with work items so that contributors are able to navigate from requirements to code. Figure 4 shows a change set is a collection of code changes in a number of files. A change set is created by one contributor and relates to one work item. However, a single work item may have many change sets associated with it. Change sets within the Jazz source control system make it possible to combine and synthesise work amongst contributors. If combining changes is problematic it is possible for contributors to collaborate to resolve any issues and this is all captured within one development environment (illustrated in Figure 5). Change sets also have flow targets associated with them that indicate sources and destinations of incoming and outgoing change sets. Change set flow targets aid developers in indentifying which change sets are coming from which team areas and work spaces.

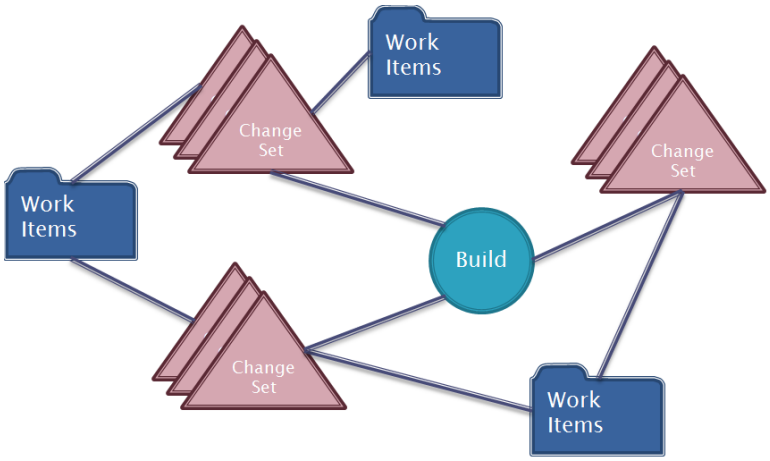

Figure 4 Work Items, Change sets and Software Builds (Kwan, et al., 2009)

Contributor chat sessions are also captured and associated with work items to aid contributors in resolving project issues. Using the Jazz repository it has been shown that, for certain types of software builds, there is a relationship between socio-technical contributors and build success rate (Kwan, et al., 2009). The benefit that Jazz provides is that social networks can be extracted from work items and relationships between work items, source code and software builds are able to be queried (Herzig & Zeller, 2009). Jazz provides the data mining community with an extensive source of software development data.

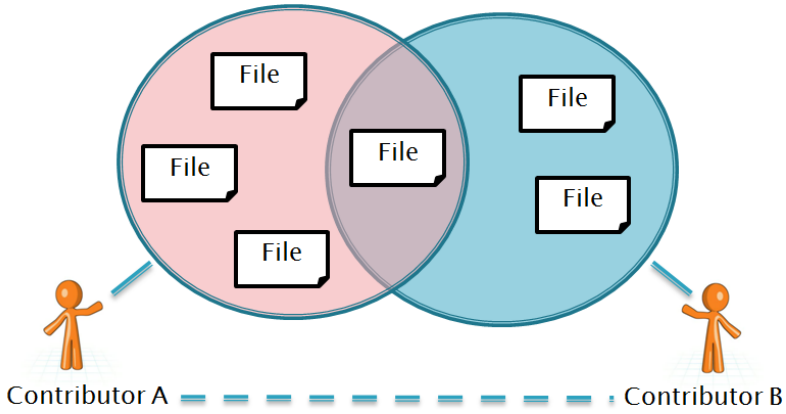

Figure 5 Developer Collaboration Over Source Code Files for Two Change sets (Kwan, et al., 2009)

A build item is compiled software which forms a working unit. The build engine in Jazz stores build information and related items to the repository. For every software build a build definition is provided, indicating if a build is a continuous build (regular user builds), nightly build (incorporating changes from the local site) or integration build (integrating components from remote sites). Builds in Jazz are associated with change sets and work items. Before a build is executed, the latest changes are accepted from a team and snapshots of the files are generated. A repository snapshot is a record of contents (files) in a certain point in time.

A server-side Application Programming Interface (API) is available from Jazz and provides methods for creating, updating and deleting artifacts within the repository. A client-side API is also available and is useful for coordinating client elements such as views and editors of artifacts. An API provides a developer with the capability to run complex queries for stored artifacts based on the artifacts' properties and relationships. If the researcher was to extract the artifacts from the repository database directly, without the support of an API, there are over 200 relations (most of which are cryptic) for the researcher to utilise. There is too great a risk of misinterpreting the databases relations, which would result in errors when extracting artifacts. The only document that describes the database structure is in the data warehouse. However, the only information provided describes only 30 of the major relations relevant to the data warehouse itself. For this reason not using the repository API to gain access to artifacts is considered infeasible. In terms of this research, the client API is a suitable choice for extracting artifacts.

There are a few drawbacks and limitations by adopting the Jazz repository into research. Firstly, the repository itself is highly complex and has fairly large storage requirements for tracking software artifacts. Accessing the data is a challenging endeavour due to the high artifact traceability (linkages) (Herzig & Zeller, 2009). There is also a lack of documentation for the platform for process components and there is no public APIs in the Rational Team Client that allow for easy manipulation of configurations (Cheng et al., 2008). Another issue is that the repository contains holes and misleading elements which cannot be removed or identified easily (Herzig & Zeller, 2009). Such holes within the data occur because the Jazz environment has been used within the development of itself; therefore many features provided by Jazz were not implemented at early stages of the project. As a consequence there is a challenge in dealing with such inconsistency and this research required an approach that delves further down the artifact chain than most previous work using Jazz. In doing so it is argued that the early software releases were functional, so whilst the project "meta-data" may be missing details such as developer comment the source code should represent a stable system that can be analyzed to gain insight regarding the development project. Finally, there is also an issue in the nature of collaborative development environments as there is a lack of support for reducing social-cultural distances that introduce communication barriers in distributed teams.

In terms of the Jazz project, contributors have little or no chances to have face-to-face meetings which can aid in developing values, attitudes and trust (Calefato, Gendarmi & Lanubile, 2009). Despite these limitations there are still many benefits in incorporating the Jazz repository in collaboration and software development research. Data mining of the metrics extracted from the Jazz repository has much potential for improving insights into the software development and team collaboration processes.

2.5 Data Mining Techniques

Researchers adopt the use of data mining methods when they are on a search for valuable pieces of knowledge within large amounts of data and manual analysis of such data is not feasible (Khoshgoftaar, Allen, Jones & Hudepohl, 2001). Data mining in software repositories enable researchers to build predictive models that can provide new and interesting insights and aid decision making for managing software project (Vandecruys et al., 2008). What is challenging within this area is that no single software metric or combinations of metrics have been discovered to predict the quality or success

rate for all types of software projects. However, the high correlation between software metrics and software failure rates indicate that there are relationships which exist between them (Dick, et al., 2004). In order to gain understandings about these relationships a wide range of data mining techniques have been applied to analyse software metrics. The Mining Software Repositories (MSR) literature shows that this it is currently an active research area and that the insights gained from the mining process aids in enhancing software development processes and methodologies (Olatunji, Idrees, Al-Ghamdi & Al-Ghamdi, 2010). Data used by MSR studies are from developer mailing lists, personal archives, issue tracking systems and concurrent version systems (CVS) such as Bugzilla (Amor, 2006). Data extracted from such sources has the potential to provide insights into improving change and source code management (Weiss et al., 2007). Other research areas within the MSR literature focus on software quality (Fenton & Neil, 2000), design (finding reusable components and forming better model solutions) and social processes (Jensen & Neville, 2002).

2.5.1 Data Pre-processing

There are various challenges that arise when adopting data mining approaches. Real life data is not always going to be immediately suitable for the mining process. There is often noise within data, missing data, or even misleading data that can have negative impacts on the mining and learning process (Chau, Pandit & Faloutsos, 2006; Hernández & Stolfo, 1998). Inconsistencies and errors within a system that is being developed are common occurrences within software projects and can result in a project exceeding its budget and time constraints. In terms of this research the project data existing within Jazz was generated during the development of Jazz. Therefore features what would capture certain projects aspects would not exist till later stages (resulting in missing values often appearing at early stages of the project). Such gaps may occur at both social metric aspects of the data and software metrics levels. It is therefore recommended to pre-process or "clean" the data and prepare it in such a way that the highest prediction accuracy and sensitivity possible can be obtained from the data.

Once a data set is extracted from a software repository it is then necessary to do some pre-processing before applying data mining algorithms. Data cleaning will aid in ensuring that any models revealed from the mining are reliable (Hernández & Stolfo, 1998). Data cleaning is the process of detecting and deleting corrupt, misleading or inconsistent records and definitions from the data set. As a result prediction models will

deliver a higher quality representation of the data in terms of completeness, validity, uniqueness and consistency (Ramler & Wolfmaier, 2008). Common methods of data cleaning include, data transformation, duplication elimination and statistical methods (Rahm & Do, 2000).

Data transformation is the process mapping data from one format to another, more suitable, format for mining. A commonly used data transformation method involves normalizing numeric values. Duplication elimination is the detection and deletion of duplicate entries. Removing duplicate values can be achieved by sorting the data set by a key attribute that brings duplicate entries together. Statistical methods can also be applied to analyze the data, focusing on the mean, standard deviation, range of values within the data set. The data analysis phase involves a manual inspection and the use of analysis programs to detect the kinds of instances which should be removed in detail. This aids in identifying values that may be error-prone. Statistical methods can also be applied to handle missing values, replacing them with more suitable values.

Data profiling and data mining are two approaches that can be used in data analysis. In data profiling analysis each attribute of an instance provides information on the data type, length, value range, discrete values, frequency, uniqueness, null values and common string patterns (Rahm & Do, 2000). Data mining aids in discovering patterns within the data set and revealing relationships between attributes. Descriptive data mining models include clustering and sequence discovery. Data mining can aid in cleaning the data by associating "business rules" which can be used towards filling in missing values within the data set.

There are challenges that can arise within the data cleaning process. For instance data transformations need to support any changes in structure and representation of the data. This can become challenging when multiple data sources are integrated (Hernández & Stolfo, 1998). Data quality problems can be found in both schema and instance levels of data sources (Rahm & Do, 2000). Schema level issues will also cause issues in instance levels. For example poor schema design and lack of integrity constraints will affect the uniqueness and integrity of instances within the data set. At instance level data entry errors can occur, which may result in misspellings, redundancy and contradictory values.

There are different types of transactions that can occur within VCS repositories. Transactions may be large in size, that may consist of changes to a systems infrastructure, or they may be simple merge transactions that often consists of small file updates (Zimmermann & Weibgerber, 2004). VCS repositories often do not keep information relating to this aspect of transactions that are made. Depending on the objective of the mining task, a smaller merge-based transaction may introduce noise into a data set as it may consist of unrelated changes and introduce duplicate instances. Defining the level of granularity to extract data has great impact on mining outcomes. However, real world data sets naturally tend to be noisy and ideally the data set needs to reflect reality as much as possible. For mining defects from bug databases, if the number of defect instances are small (where defects are a minority class), then the balance of classes will have a significant impact on prediction accuracies (Sunghun, Hongyu, Rongxin & Liang, 2011). By sampling the false positive and false negative rates it will provide insights into whether or not the defect data is suitable for generating predictions.

2.5.2 Feature Selection

It is often possible to characterise a data set with fewer attributes than originally considered and in doing so the computational expense in processing mining algorithms can be significantly reduced. In addition to this feature selection can also remove further noise from the data set and therefore improve the performance in terms of accuracy and sensitivity of the results. Feature selection is an important part of the data mining process especially in cases where the data is highly skewed (Forman, 2003) and this is often the case with software metric data (Dick, et al., 2004). Feature selection is a process to find a feature subset within a data set that is a good substitute to all features.

This process provides the data miner with knowledge of which features are useful and may lead to better results in terms of accuracy (Forman, 2003). There are a range of subset evaluators that can be adopted and each consists of data wrappers, filters and embedded methods. A wrapper is a search algorithm that is used to search through the data set problem space to identify significant factors using a data model. A filter does the same thing, however it does not apply a model, instead a subset is evaluated by a filter. An embedded method involves the use of embedded logic within a specific model. There are a range of search approaches that can be applied, for example, greedy

hill climbing, greedy forward selection, greedy backward selection, exhaustive, best first, genetic algorithm and simulated annealing searches. There are also an extensive range of filter methods such as, mutual information, consistency-base, correlation-based and separate class identification (Forman, 2003). In terms of this research Information Gain, Subset Evaluation and Principal Components Analysis are covered as they commonly used within the mining literature. The main contribution of this thesis is in the software engineering discipline, so only standard and common techniques are under consideration.

Information Gain as a feature selection algorithm is used to define a sequence of attributes used to investigate the state of an artifact. The sequence that is generated is used by classification algorithms, such as decision trees. Information Gain ranks attributes by their individual evaluators. An attribute with a high Information Gain value is considered to be more significant within the data set. Information Gain generates a list of weighted features that are found to improve classification accuracy of Naive Bayes mining results for software fault prediction models (Menzies et al., 2008). In another study which utilised data from three open source projects (Apache, PostgreSQL and Python), Information Gain was used towards building a model for finding which factors encouraged developers to contribute to development mailing lists. This model also was used to predict which developers were more likely to contribute based on previous contribution behaviour within an 85-89% accuracy (Ibrahim, Bettenburg, Shihab, Adams & Hassan, 2010).

Subset Evaluation as a feature selection algorithm is often used as a data filter or wrapper for classification and clustering algorithms. Within Subset Evaluation each subset is evaluated by two types of criterion. These criterion can be either independent or dependent (Huan & Lei, 2005). Independent evaluation criterion is used within a filtering model to evaluate the goodness of a feature or subset of features by exploiting characteristics of the data without the use of a mining algorithm. Common independent criteria include distance, information, dependency and consistency measures (Huan & Lei, 2005; Trendowicz et al., 2008). Dependent criteria are used in the wrapper model and require a mining algorithm for feature selection and can be more computationally expensive.

Subset Evaluation was applied to the software metrics data extracted from a telecommunications system and from 42 of the original features a subset of approximately 6 features were found be significant for data mining (Gao, Khoshgoftaar, Wang & Seliya, 2011). Finding an optimal feature subset is often a difficult process and in most cases feature selection is used for non-deterministic polynomial-time hard (NP-hard) problems. Subset Evaluation tends to be computationally expensive as the filter iteratively evaluates subsets of attributes by examining the characteristics of the data without learning algorithms.

Many object-orientated software metrics have a high correlation with each other as they are likely to be measuring the same underlying dimension of an object. PCA transforms data components so that they do not correlate to each other and become more orthogonal. PCA is frequently found within software metric studies about effort estimation (Jørgensen, 2007), fault-proneness (Venkatasubramanian, Rengaswamy, Kavuri & Yin, 2003) and software quality (Thwin & Quah, 2005). PCA as a feature selection method has been applied to software metrics extracted from 5 large Microsoft systems and the regression models built could accurately predict the likelihood of post-release defects for new software modules (Nagappan, et al., 2006). As a method PCA treats points within the data set as form a hyper-ellipsoid feature space where there are few large axes and many small axes (Dick, et al., 2004). The algorithm determines the directions and lengths of the axes where each feature forms a vector and a covariance matrix is formed for the data set. Then the eigenvectors and eigen-values are determined. Large eigen-values indicate axes that carry significant amount of information about the data set and smaller Eigen-values represent noise dimensions. The axes are defined by the eigen vectors associated with each eigen-value. Feature reduction is then carried out by forming a matrix of significant eigenvectors then applying a transformation. In a study based on the MIS, OOSoft and ProcSoft data sets it was found that the metric values and failure from these data sets are highly correlated (known as the multi-collinearity problem in data analysis) (Dick, et al., 2004). The advantage of using PCA is that each eigenvector that is obtained is orthogonal to every other eigenvector and attributes within the feature space. As the feature space is reduced and each attribute is therefore statistically independent from one another.

When determining software or communication metric predictors to be used for statistical analysis or machine learning methods it is important to determine which software system will be used. The MSR literature shows that to make studies comparable, it is best focus on prior research that is based on a single software system, or at least systems of similar types (Nagappan, et al., 2006). The next stage is to decompose the system into entities. These entities will represent instances within the data. For each of these instances a set of metric functions are mapped to each entity. From this it is then possible to determine the correlations between all metrics and entities. Using feature selection methods, such as Information Gain, Subset Evaluation or PCA, predictors are built for new entities. It is then important to evaluate the explanative and predicted power of the selected parameters.

One of the limitations of using Information Gain or Subset Evaluation, as well as other standard machine learning methods, is the lack of the ability to integrate business knowledge which can characterise software projects. Instead human input may be required to aid the learning process. For example a smaller set of features may be selected from a data set, however the miner with knowledge of the domain may insert additional features that they think are also important to incorporate within a predictive model.

Another limitation is that the data may still be over fitted to a particular model (Lei & Huan, 2003). Over fitting (or over training) occurs when the search algorithm searches for the best parameters for a model of a limited data set. Not only will an over-fitted model capture the general patterns within the data, it will also model noise. This is because the model is fitted too closely to the training data and new instances are predicted with less accuracy. One solution to the over fitting problem is called early stopping where a data set is divided into a training and test set. The training of a model is then "stopped" when the error rate increases. If the data set is too small, another solution is to use cross fold validation, regularization and other statistical methods to validate derived models from selected features. These methods can be applied to both supervised and unsupervised machine learning methods.

Model-based approaches that use software metrics to predict development outcomes and aid in assessing software quality are described in the research literature. Regression-

based prediction is the most common approach (found within approximately half of the reviewed literature) (Jørgensen & Shepperd, 2007). Commonly used regression techniques include Least-Square Regression (LSR) (MacDonell & Shepperd, 2003), stepwise regression (a variant of LSR) and ordinal regression (Sentas, 2005). However, one of the limitations of using statistical regression models is that it is assumed that predictor attributes exist independently of each other. This is not necessarily suitable for the use of a range of software metrics, as they are often related to each other. This problem is known as the multi-collinearity phenomenon. Machine learning algorithms are able to cater for the multi-collinearity problem and it is for this reason that they are being used in this study over more traditional statistical methods. However, like regression techniques, there are still limitations. For instance where there is a small amount of "linear behaving" data it may be discarded as noise when it may be an important feature or an interesting pattern. However, depending on the from the range of software metrics included in the data, it is fair to say that many of them may be dependent on each other, therefore data mining techniques are suitable for the nature of this research.

2.5.3 Data Mining Methods

In data mining classification algorithms are used to map data into defined classes. A classification tree iteratively partitions data via a learning process and classifies an instance based on a class attribute (that are usually an observed nominal or ordinal data type) (Gray & MacDonell, 1997). Each learned partition forms decision nodes and the final "decided" classification is represented as leaf nodes of the tree (Breiman, 1984). Each decision node contains a split that tests the data based on one or more variables to determine how the data is partitioned to a classification node (leaf). Classification-based modelling for software quality estimation is a confirmed technique in achieving better software quality control (Khoshgoftaar & Seliya, 2002; Khoshgoftaar & Seliya, 2004). CART (classification and regression trees), relational probability trees (RPTs) and C4.5 (j48) methods have also been applied to large social networks and have yielded highly accurate models of relational data (Jensen & Neville, 2002). Decision trees have also been commonly used in software cost estimation (Gray & MacDonell, 1997; Leung, 2002). Decision trees are a powerful tool as they allow the data miner to visualise the new rules and concepts derived from features of a given data set. In addition to this decision tree learning is one of the most commonly used classification methods (Domingos & Hulten, 2000). Software and social metrics have been modelled using a

vast variety of classification methods (forming various hybrids of classification methods). However, there is no single "best" solution that is used to predict software outcomes, regardless of whether they are project specific software metrics or generalised software metrics (taken from a number of different projects).

Amongst the hybrids methods, Bayesian Networks (BNs) are also commonly used within software and social metric studies. A BN is essentially a directed acyclic graph of nodes that are joined by probability functions. BNs convey both qualitative and quantitative information about the relationships which exist between elements of data through conditional probability and have been applied to software metric estimation studies (Fenton & Neil, 2000; Jørgensen & Shepperd, 2007; Pendharkar, 2005). This type of model has great potential for handling the complex sets of relationships between software artifacts and coordination structures of a software team. This is because BN's offer a subjective interpretation of the probability of an event occurring (Yedidia, et al., 2002). Beliefs or rules of a BN are constructed by the philosophy that knowledge is expressed via probability and any inference is conditioned on observed data. Information contained within the observed data can be carried via the “likelihood” function. Knowledge is expressed using Bayes Law:

$$\Pr(\mathbf{B}|\mathbf{A}) = \frac{\Pr(\mathbf{A}|\mathbf{B})\Pr(\mathbf{B})}{\Pr(\mathbf{A})}$$

Each node within a BN has a state and if that node is a child node within a graph, its state is dependent on the states of its parents' nodes. This is also similar in terms of anti-patterns, where one negative practice can cause chain reactions to cause further negative states. If a BN node does not have any parents then it is not conditioned on any other nodes. A BN is most useful if nodes do not have too many direct statistical dependencies as the node itself may become challenging to manage and understand (Stamelos, Angelis, Dimou & Sakellaris, 2003). It is ideal to keep nodes in a singly-connected graph for efficient calculation of probabilities. In some cases the network may form a loop if the graph is undirected.

Within the BN model probabilities are expressed in terms of degrees of belief. Expert systems utilise BNs for various purposes including medical diagnosis, map learning, language understanding and insurance risk. BNs have great potential for use in this

research in order to gain understandings about software metrics, social networks and their relationship to project success. For example BNs have the potential to provide insights into which combinations of metrics (and metric levels) relate to successful software builds.

In study by Wolf, Schroeter, Damian & Nguyen (2009) IBMs Jazz repository was used to build a social network of developers and a Bayesian classifier was used to accurately predict if a software build was going to fail. In order to do this multiple social networks were constructed for a particular focus (in this case software builds). The nodes of the social network, in this instance, were the project contributors. To link the nodes within the network an edge was made where contributors commented or subscribed on the same work items. The weighting of the edge determined the number of comments made (to indicate how heavy the communication flow is). Network analysis measures (k-core) were applied to predict a builds' outcome.

For this research it will be useful to construct social networks based on parameters that have a positive or a negative influence. It is anticipated that as a result patterns might emerge that can be used as a diagnostic tool.

In data mining a clustering algorithm is suitable for seeking sets of categories within large amounts of data. The categories may be mutually exclusive, hierarchical or overlapping. Clustering is a machine learning technique where labels of classes within the training data are unknown. In data mining clustering is unsupervised machine learning. In the software metrics literature clusters are based on observations as opposed to interpolations (estimation) or extrapolations (generalisations) (MacDonell & Shepperd, 2003). With this method the researcher applies background knowledge that can be used in clustering the data.

A widely used and well-known clustering method that minimizes clustering errors is the k-means clustering algorithm and has been applied to software defect and effort estimation (Bibi, et al., 2008), social network analysis (Al-Fayoumi, 2009) and software and system design (Browning, 2001). K-means clustering aims to partition observations within a data set into k clusters or groups (k is specified by the data miner). K-means clustering starts by assigning a distance vector to its closest cluster. Then each

clusters centre is updated to be the mean of its instances. K-means clustering converges to a solution when there are no further changes in assignment of instances to clusters. One of the limitations of k-means clustering is that it uses a local search method which can be detrimental to its performance as it depends on the searches initial starting conditions. For this research, clustering has potential in providing additional insights into classification outcomes.

2.5.4 Synthetic Minority Over-sampling Technique

When working with real work data it is often found that data sets are heavily comprised of "normal" instances with only a small percentage representing interesting findings. As a result the "abnormal" instances have a negative impact on a models' performance as they are have a greater probability of misclassification using data mining methods (Chawla, Cieslak, Hall & Joshi, 2008; Lee & Xiang, 2001). Data instances that introduce noise within the data and are often found within the minority class (Haibo & Garcia, 2009; Jeatrakul, Kok Wai, Chun Che & Takama, 2010). In order to overcome this limitation synthetically under-sampling the majority class may improve a classifiers' performance. However, in doing so a data set may become unbalanced or have an insufficient number of instances for recognising interesting patterns. Another solution is to provide the classifier with more complete regions within the feature space via synthetic and simulation means.

One noteworthy method to do this is to use a Synthetic Minority Over-sampling TEchnique (SMOTE) (Chawla, 2010; Chawla, Bowyer, Hall & Kegelmeyer, 2002). This enables a data miner to over sample the minority class and under-sample the majority class to achieve potentially better classifier performance without loss of data. While other over-sampling methods exist, such as Rippers Loss Ratio and Naive Bayes methods, SMOTE provides better levels of performance as it generates more minority class samples for a classifier to learn from therefore allowing broader decision regions and coverage (Chawla, 2010). An advantage of using SMOTE is that the values derived are interpolated rather than extrapolated, so they still carry relevance to the underlying data set. SMOTE has been utilised within the software research community and compared with other sampling techniques in software quality modelling (random under-sampling, Wilson's editing, one-sided, selection, random oversampling, cluster-based oversampling and Borderline-SMOTE) and has yielded encouraging results (Drown, 2009; Seiffert, Khoshgoftaar & Hulse, 2009). SMOTE has also been applied as a

sampling strategy for software defect prediction where data sets from NASA projects software repository were used (Gray, Bowes, Davey, Sun & Christianson, 2009; Jiang, Li & Zhou, 2011; Pelayo & Dick, 2007) and fault-prone prone module detection using the MIS (Yasutaka, 2007), telecommunication systems and NASA projects (Seliya, Khoshgoftaar & Hulse, 2010) data sets.

In order to avoid the over-fitting problem and expanding the minority class regions within a data SMOTE generates new instances by operating within the existing feature space. For each minority class instance SMOTE interpolates values using a k-nearest neighbours technique and creates attribute values for new data instances (Drown, 2009). Depending on the amount of over sampling needed, synthetic samples are generated by taking the difference between the feature vector that is under consideration and its nearest neighbour, then multiplying it by a random number between 0 and 1 and then adding it to the feature vector under consideration. This causes a random line segment between two existing features (Chawla, 2010) and creates a new instance within the data set. As a result SMOTE generates more general regions from the minority class and decision tree classifiers are able to use the data set for better generalisations.

Most machine learning algorithms work from the assumption that the data is being mined from a stationary distribution. This is not an entirely true when working with software metric data sets. The instances within software and communication metrics data can be gathered over time (months and years). There are a number of algorithms available to researchers that adopt temporal-based learning methods. Of particular interest for this research is mining decision trees from continuously-changing streams of software builds.

2.6 Mining Data Streams

Database management systems (DBMS) are widely used to create, modify and delete data that is modelled within relations. Over time database systems grow and evolve resulting in large volumes of continuous data. The data from this perspective arrives in the form of streams. Data streams are generated continuously and are often time-based. In large and complex systems the data, arriving in a stream form, takes its toll on resources (storage size) and in some cases it is impossible to store the entire steam. This is because streams themselves can be overwhelming (hundreds or thousands of data per second). Often in these cases the data is processed once and then is disposed of, if

storage limitations are of concern. Data Stream Management Systems (DSMS) has spiked much interest within the database research domain (Babcock, Babu, Datar, Motwani & Widom, 2002). A data stream is a sequence of continuous and ordered data elements that arrive in real-time. Data streams have various applications within computer science, including managing network traffic (Babu & Widom, 2001), web searches (Jiawei & Chang, 2002), sensor networks (Madden & Franklin, 2002), ATM transactions (Liu, Lin & Han, 2011) and safety systems (Horovitz, Krishnaswamy & Gaber, 2007). Data stream mining has also been used within the context of social networks, using streams from Twitter for opinion and sentiment mining and analysis (Bifet & Frank, 2010). The concept of data streams and data flows are not new, however this remains a very active area of research.

The implications of data stream mining in the context of real-time software artifacts is yet to be explored. Currently there is no research that has explored whether or not stream mining methods can be used for predicting software build outcomes. In large development teams software builds are performed in a local and general sense. In a local sense developers perform personal builds of their code. In a general sense the entire system is built (continuous and integration builds). These builds occur regularly within the software development lifecycle. As there can be a large amount of source code from build to build, the data and information associated with a build is usually discarded due to system size constraints. More specifically, in IBMs Jazz repository, while there are thousands of builds performed by developers; only the latest few hundred builds can be retrieved in total. Data stream mining offers a potential solution to provide developers real-time insights into fault detection, based from source code and communication metrics. In doing so it potentially enables developers to mitigate risks of potential failure during system development and maintenance and track evolutions within source code over time.

During the development of code, an IDE compiles regularly to check for errors. However, there are no tools that exist which provide information to a developer about how their current development work has impact on the overall build. Software and communication metrics can be extracted any time during development, providing a continuous stream of data. These streams can be data mined and as a result provide real-time models for predicting the outcome of a build.

DSMSs aim to provide support for stream-based processing tools. These tools provide functions for keeping the data streams moving, querying streams, handling noise within streams, integrating stored and streamed data, generating stream-based predictions and instant processing and responding (Stonebraker, Çetintemel & Zdonik, 2005). Figure 6 illustrates the abstract concept of a DSMS. In real world applications streams arrive for processing in real-time. A DSMS requires an allocated memory space that contains a portion of the latest streams and disposes of older data upon newer data arrival. In many cases online stream tools are restricted to only look once at the data stream (Liu, et al., 2011). A permanent storage is used for maintaining data that is regarded as important from a stream and also stores queries and indexes.

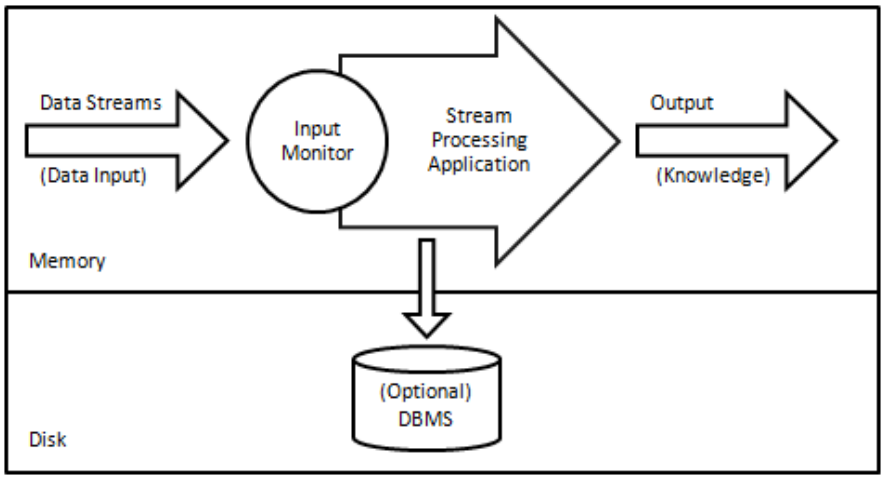

Figure 6 Abstract Concept of DSMS (Stonebraker, et al., 2005)

A stream is composed of two elements, one element being the data and the second being the increasing timestamp, representing the arrival time of the actual data. The data can be encoded as a form of a single relational instance or group of relational instances from a relational-based stream. It can also be object orientated where data is associated and derived from methods. In addition to this, filtering mechanisms can be put in place to process streams that contain particular attributes or items of interest. The timestamp element can be encoded as a traditional timestamp or a sequenced number.

2.6.1 Pre-processing For Data Stream Mining

Task-based techniques are used to reduce computational challenges found in mining and processing data streams. There are three major task-based technique categories: 1) approximation algorithms, 2) sliding windows and 3) algorithm output granularity

(Gaber, Zaslavsky & Krishnaswamy, 2005). Approximation algorithms can be used to provide approximate solutions for one-pass methods with error bounds. While approximation algorithms are designed for computationally hard problems, they do not address the problem of data arrival rates. Therefore this technique may be used in conjunction with another algorithm to adapt to available resources. Sliding windows performs' analysis with the most recent data segments from a stream.

Since entire streams that are large in size are not stored within in a DSMS only a subset of a stream becomes available. The subset of a stream is composed of discontinuous elements which form sub-streams. These sub-streams are used by window models, where a window is based on a certain range of time of a stream. A window over a stream, which is a continuous sub-stream, can be presented by timestamps of the previous data with the newest data. Elements that are inside a window are temporarily stored and can be scanned multiple times. Once older data is outside of a window it can no longer be retrieved by the window model (Babcock, Datar & Motwani, 2002). There are many different window models; these can be based on window sizes, update intervals and window closure constraints. Window size can be time based or count based (fixed number of elements). A windows update strategy depends on the when older elements of data are expired upon the arrival of new data. Such strategies include sliding windows, jumping windows and tumbling windows and are illustrated in Figure 7. With a sliding window the window "slides" across the time series one streaming instance at a time (Patroumpas & Sellis, 2006). With a jumping window, a window may "jump" over instances, streaming a small set of instances at a time (Golab, DeHaan, Demaine, Lopez-Ortiz & Munro, 2003). A tumbling window stores instances until it is full and then flushes all stored instances before starting over from scratch with a new window.

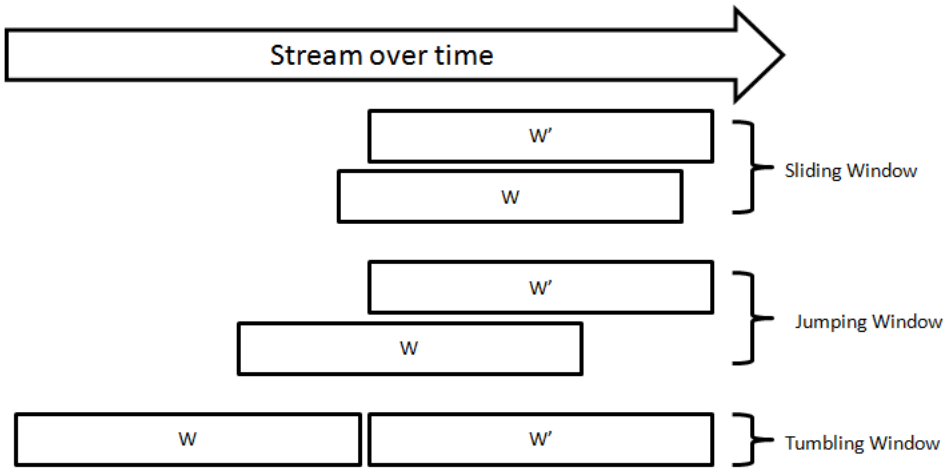

Figure 7 Window Update Strategies (Tao, 2011)

Finally, algorithm output granularity provides a resource-aware approach that can cope with fluctuating high data rates taking into consideration available memory and processing speed. With this technique mining is followed by the adaption of resources and data arrival rates. Before the available computational memory is filled the generated knowledge is merged into existing structures.

2.6.2 Data-Based Techniques

The arrival rate of windows maybe very high and processing occurs in real-time. To ensure that the data is mined efficiently within a timely manner there are three main data-based techniques which can be adopted. Data-based techniques refer to summarizing a data set or selecting a subset of data from a stream and the three main methods for this are 1) sampling, 2) load shedding and 3) sketching (Gaber, et al., 2005). Sampling is the process of selecting data via a probability function. Boundaries of error rate are set by a sampling size that is determined by Very Fast Machine Learning (VFML) methods using the Hoeffding bound. The sampling method is not recommended for use if data rates fluctuate or if sample rates and error bounds contain anomalies. Load shedding is the process of dropping a sequence of data streams. This method has similar issues to sampling. If parts of a data stream are dropped then they are not modelled and they may have introduced a new pattern of interest. The Sketching method is the process of randomly selecting a subset of features. This can have a negative impact on accuracy and is challenging to use in the data stream mining context. Other methods include synopsis data structures and aggregation. These techniques are

not used with traditional mining methods, however can be valuable with stream mining context

2.6.3 Distribution Changes in Data Streams

In a traditional DBMS data is seemingly static and represents a static distribution. This idea does not bode well for many real-world applications. Data is often continuous and is accumulated over long periods of time. In addition to this, data becomes more dynamic as it changes and evolves. This phenomenon is referred to as data evolution, dynamic streaming or concept drifting (Baena-García et al., 2006). Changes in a stream can evolve slowly or quickly and both types of rates of change can be queried within stream-based tools. These data distribution changes from streams have a direct impact on DSMS. This is because a DSMS needs to adjust its model to reflect such changes so that new predictions can be generated for the data under the new distribution. There are two types of changes that can occur within data streams 1) data distribution change and 2) concept drift change. Change detection techniques aid in finding such distribution changes and then provide new information for users to understand the causes of such changes. These changes are modelled into the stream processing application. By analyzing data distributions over time there is then potential for predicting future distribution changes.

Window-based change detection triggers when a window moves and is dependent on a window movement strategy and a statistical test. For each change detection method a reference window (the current or older data) and an observation window (newly arrived data) are used. There are two major techniques used for detecting changes. One of these techniques looks at the nature of the data set, to determine if it has evolved, whereas the other looks at the models to see if they are still suitable for newer data. The latter of the two makes use of concept drifting. The optimal change detector and predictor system consists of high accuracy, fast detection of change, low false positive or negative ratios, low computational cost and no parameters needed.

When using window models the simplest rule is to keep each window the same fixed size. A small size window reflects accurately the current data distribution and a large size window provides more examples to work with, increasing accuracy and stability. The window size can be determined by the user or by using a decay function that

measures the importance of instances according to how long they have been within the window.

A technique called ADWIN is a parameter-free adaptive sliding window strategy that compares all adjacent sub-windows to a partition window that contains all the data (Bifet, 2009). This method is recognised to generate the best accuracy, however may have a time cost with larger streams. This method dynamically adjusts the size of a window and derives efficient variations using Hoeffdings' bound. A window will become larger when the data is stationary to maintain better accuracy. A window will become smaller when change is taking place as it will discard stale data. This eliminates the need for the user to determine the best window size. In addition to ADWIN there are many algorithms available for addressing change detection. Three noteworthy methods available from MOA (a Massive Online Analysis software environment) (Bifet et al., 2011) include OLIN (On Line Information Network), CVFDT (Concept-adapting Very Fast Decision Trees) and UFFT (Ultra Fast Forest of Trees). MOA is a software tool for running online data stream mining experiments.

2.6.4 Data Stream Mining Methods

When capturing data streams the amount of data accumulated can increase rapidly. Traditional data mining methods are tailored for static and structured data, where the data is captured and stored. These methods may not be suitable for time-based streamed data due to storage constraints. Commonly used data stream mining methods include classification, clustering, time series analysis and frequency counting. These are examples of incremental learners. Any discrete search learner can be made capable of processing a data stream. A range of both clustering and classification methods are commonly used in relational data mining. A clustering framework for data streams is called HP Streams, where clusters are found for high dimensional data streams. Other methods available under the MOA framework (Bifet, et al., 2011) include StreamKM++, CLUStream, CLUSTree, DenStream and CobWeb. Classification techniques available via MOA include Bayesian classifiers, decision trees, meta-classifiers, function classifiers and a drift classifier called SingleClassiferDrift. Decision trees are among the most commonly used classifier models. For data stream mining the Hoeffding tree is ideal for Very Fast Decision Tree (VFDT) learning.

2.6.5 Hoeffding Tree

The Hoeffding tree (or VFDT) is an incremental time inducing method. Using the Hoeffding bound, it looks at the number of instances that are needed to predict an outcome within a certain precision that can be predetermined (Hulten, 2001). This method has potential in terms of predicting future outcomes of a software build with high accuracy while working with real-world data. Rather than using training and test sets, instances are represented as streams. Data streams provide unique opportunities for evaluation as the amount of available data with the time-based property can be examined.

The Hoeffding tree is commonly used for classifying high speed data streams. An induction algorithm generates a decision tree from data incrementally by inspecting each instance within a stream without the need to store instances for later retrieval. The tree resides in memory during each iteration and stores information in its branches and leaves, potentially growing from "learning" from a new instance. The decision tree itself can be inspected at any time during the streaming process. The Hoeffding tree is a type of decision tree that is grown by a batch learning process. The quality of the tree itself is comparable to that used by traditional mining techniques, even though instances are introduced in an iterative approach.

Just like traditional decision tree learners, the Hoeffding tree is easy to interpret, making it easier to understand how the model works. In addition to this decision tree learners have proven to provide accurate solutions to a wide range of problems that are based on multi-dimensional data. For Hoeffding trees each node of a decision tree contains a test which splits instances, sending them down various branches of the tree depending on their values from a particular set of attributes. To create a decision in the context of data stream mining, a Hoeffding bound (also known as a Chernoff bound) is used. The Hoeffding bound is expressed as:

$$\epsilon = \sqrt{\frac{R^2 \ln\left(\frac{1}{\delta}\right)}{2n}}$$

Where R is the random variable range and n is the number of independent observations made overall. The bound holds true for the distributions generating the values and only depends on a range of values, number of observations made and a split confidence level.

The confidence parameter for the Hoeffding tree is $(1 - \delta)$. With the probability close to one this parameter is a small value. An additional parameter is tie-breaking (τ) and this is called upon within the decision split method. Tie-breaking is used in cases where the Hoeffding bound value is considerably small where its evaluation values for two attributes may be very close. As a consequence this will prevent the decision tree to grow. Instead of waiting for an additional window which, in some cases, can be a waste of resources, the VFDT forces a split. For example if the Hoeffding bound is less than τ a node split is forced based on the best attribute. Ties are more likely to occur for numeric and nominal types of attributes.

To begin building a tree via a data stream it is preferred to inspect the decision tree after a few instances have been learned from. This is known as a grace period and is an addition parameter to the Hoeffding tree. Other options for the Hoeffding tree include the criterion used to perform splits (e.g. Information Gain), the maximum memory that is to be consumed by the tree, the type of numeric estimator to use (e.g. Gaussian approximation), the type of nominal estimator to use, pre-pruning, the method for leaf prediction (e.g. Adaptive Naive Bayes) and the threshold for the number of examples a leaf should observe before leaf prediction.

2.6.6 K-Means Clustering

Clustering data streams using the k-means technique iteratively generates groups of subsets within the data and indentifies centres of clusters from the records observed. In data stream mining this is referred to as Very Fast K-Means (VFKM). The algorithm starts by calculating the sample size bounded by an error rate of ε. The k-Means algorithm passes instances once through $O(Tkn)$, where T is the average instance size, n is the number of instances and k is the number of centres. The algorithm updates the centres and weightings iteratively during input from streams. Like the Hoeffding tree, clustering methods for data streams make use of the Hoeffding bound and it determines the learner loss as a function for each step of the algorithm. This approach is called Very Fast Machine Learning (VFML). More specifically the Hoeffding bound is used to determine the number of examples needed in each step of a k-means algorithm. More data records are executed with each run until the Hoeffding bound constraint is satisfied.

2.6.7 Mining Data Streams: New challenges

Many of the issues and challenges found from data stream mining are also applicable to relational data mining. These include catering for unbounded data sets, noisy data, efficiency and data evolution (Gaber, et al., 2005; Stonebraker, et al., 2005). Data streams can be unbounded in size and may not be stored entirely in memory all at once; therefore approaches are required to handle the flow of continuous data. To compensate this stream processing tools generate estimations for results when complete data is not available. Larger streams have an impact on computing performance and storage and therefore impact the efficiency of the way a stream is processed. Online stream processing tools in a real-time environment require high efficiency and this sometimes costs the level of accuracy in the overall performance.

Parameters for data stream mining techniques are required to be specified before execution of the mining phase. While a few parameters may be adjusted during processing time, there is no means to adjust other parameters while the mining process is running (Jiang & Gruenwald, 2006). In some cases it may not be possible to stop the mining process to make adjustments to the parameters because it may take a long time for the algorithm to process. Other issues include minimizing the amount of energy and bandwidth of streaming data (Gaber, et al., 2005).

Despite these limitations the dynamics of data streams and displaying changes and trends of the knowledge structures generated has great potential for benefiting temporal-based analysis. The outcomes of the analysis may improve real-time decision making processes as developers work on source code.

A data stream mining application is required to provide time efficiency, resource efficiency, handling noisy data, ability to detect changes and be deterministic. The application must be time efficient because as stated previously, streams often arrive in real-time and be comprised of large amounts of data per second. Stream mining techniques need to have a balance between accuracy of results and response time. Unbounded streams of data are dependent on the amount of available memory and computational speed. Therefore resources are required to be efficiently allocated. This can be achieved through memory management, scheduling and data-based techniques. Just like traditional mining methods, noise within data also needs to be catered for.

Data within streams may be delayed, missing or arrive in unordered segments. In addition to this a data stream mining application needs to automatically detect changes within data so that its mining strategy is adjusted as data evolves over time. For robustness the stream miner needs to be able to generate the same output regardless of the time of stream execution.

2.7 Chapter Summary

This chapter has described the motivations and challenges behind the work presented in this thesis. Within the literature there is a wide range of software and social metrics, ranging from simple count metrics to complex hybrid metrics. There are also a wide range of methods applied to explore the metrics. Despite all this there appears to be no single set of metrics, or combination of methods, that are able to generate predictive models to develop a software project diagnostic tool. In addition to this there are few studies which attempt to map social metrics of the software development process to software metrics.

Predicting the outcomes of processes within the software development life cycle can take considerable effort. For example to accurately predict which software entities are likely to fail, older historical records of previous outcomes are needed. Sources of software failure can be found within bug databases, program code (complexity metrics), or a combination of approaches. There has been much research done within the data mining of software repositories domain and various models have been built to predict change and fault-proneness in modules, however, there are still challenges that need to be addressed. These challenges include focusing on the quality of the data used to generate models, in order to provide reliable and useable explanations for the correlations of features within them (Christensen & Albert, 2007). This is because these models should aim to provide insights into development practices in order to release more reliable software.

The mining software repositories community faces a number of challenges, namely; 1) how to represent and accurately interpret software and social metrics that may be extracted from a repository, 2) how to use this information and relate it to a software process and deliver results about a specific type of project and 3) how to build a predictive model that uses this information about historical development processes so that it can be used as a knowledgebase for decision making.

There are also challenges in querying software and social metrics such as 1) which metrics should be included in the extraction, 2) how should these metrics be ranked or selected as important features and 3) when should metrics be automatically, interactively or manually selected. Finally data mining methods have challenges, 1) how to eliminate "noisy" data and deal with unbalanced data, 2) how to validate models when they are built from small training sets (limited information) as they could behave differently when new instances are introduced (over fitting problem), 3) if the purpose of the mining process is to improve predicting a development process success, will it take too much time to make decisions based from the knowledge derived from the model? For example if a decision tree is complex (consisting of many branches and levels) or not easily coherent (duplicate nodes), there are various methods that can be explored to "simplify" a model.

This chapter has also presented the concept for data stream mining and its potential application to software and communication metrics for software builds. Data stream mining methods provide ways to investigate, design and implement solutions for real-time decision making during the software development process.

As the Jazz repository is relatively a new source of data for researchers there is no literature that has derived software metrics from change sets of software builds, other than those articles that have arisen from this research. Nor is there a great deal of literature which combines the elements of both software metrics and social network metrics for time series analysis. While decision trees classifiers have been used extensively in the software metrics literature, there has been virtually no application of the data stream mining methods to analyse how such software artifacts change over time. The next chapter presents the experimental methods and tools used to evaluate a combination of software and social network metrics and how they relate to software build outcome prediction.

3 Research and Experimental Design

This chapter describes the overall structure of the research conducted and goes into greater depth regarding the experimental design, methods and tools used to evaluate which combinations of software and social metrics give the best prediction indicators of whether a software build will succeed or fail. This chapter is composed of two major sections 1) the Research Methodology and 2) the Experimental Design. In the first section the research via artifacts and the Knowledge Discovery in Databases (KDD) methodologies are introduced. The second section describes how these methods are applied in the context of mining the Jazz repository for this thesis.

Researchers adopt the use of data mining methods when they are on a search for valuable pieces of knowledge within large amounts of data and manual analysis of such data is not feasible (Khoshgoftaar, et al., 2001). Data mining in software repositories may enable researchers to build predictive models that can provide new and interesting insights and aid decision making for managing software projects (Vandecruys, et al., 2008). What is challenging within this area is that no single software metric or combinations of metrics have been discovered to predict the quality or success rate for all types of software projects. However, the high correlation between software metrics and software failure rates indicate that there are relationships which exist between them (Dick, et al., 2004; Menzies, et al., 2008). In order to gain understandings about these relationships a wide range of data mining techniques have been applied to analyse software metrics.

To reiterate, the emergence of Mining Software Repositories (MSR) literature shows that this it is currently an active research area and that the insights gained from the mining process has aided software development processes and methodologies (Olatunji, et al., 2010). Data used by MSR studies are from developer mailing lists, personal archives, issue tracking systems and concurrent version systems (CVS) (e.g. Bugzilla) (Amor, 2006). Data extracted from such sources has the potential to provide insights into improving change and source code management (Weiss et al., 2007). Other research areas within the MSR literature focus on software quality (Fenton & Neil,

2000), design (finding reusable components and forming better model solutions) and social processes (Jensen & Neville, 2002).

As well as informing the direction of the research itself, the review of the MSR literature has also informed the development of a suitable methodological framework in which to conduct the work that combines concepts from both qualitative and quantitative research paradigms. Whilst the work is primarily quantitative in nature there is a degree of interpretation required to translate the meaning of the quantitative outcomes into a real world scenario. As such the research methodology employed is both exploratory and constructivist in nature.

The research via artifacts process (Nguyen, et al., 2009) is used in conjunction with the design science paradigm (Hevner, March, Park & Ram, 2004) as this research requires the construction of new ideas, frameworks and technology using the Jazz repository. Researchers learn from experience when observations are made from prototype behaviours, therefore systems change as new knowledge is developed (Nunamaker & Chen, 1990). The aim of the experiments within this study is to provide insights into a range of solutions to enhance developer communication networks, productivity and gain new understandings about the relationships between developers and software artifacts within the SDLC. From the design science perspective, prototype iterations incorporate knowledge from models that are built from a range of metrics extracted from the Jazz repository and new frameworks are applied from new observations from the mining and simulation processes. For example the first prototypes focuses on software build artifacts. The second prototypes focus on enhancing the prediction accuracy. The third iteration will incorporate work item artifacts and developer communication. Within this stage comments that have been made by contributors on work items that are related to software builds are mined. Insights gained from these prototypes lead to new theories and models about social anti-pattern criteria and optimisation methods (new representation of the problem, fitness function and interpretation of the artifacts).

Data mining and leveraging of software artifacts from IBMs Jazz repository (illustrated in Figure 8) are core components to this work. There are two main challenges in using such an approach: the complexity of the data mining task and the validity of the new

understandings gained from the software artifacts (Nguyen, et al., 2009). This mining process has been found to be complex and in some cases impossible.

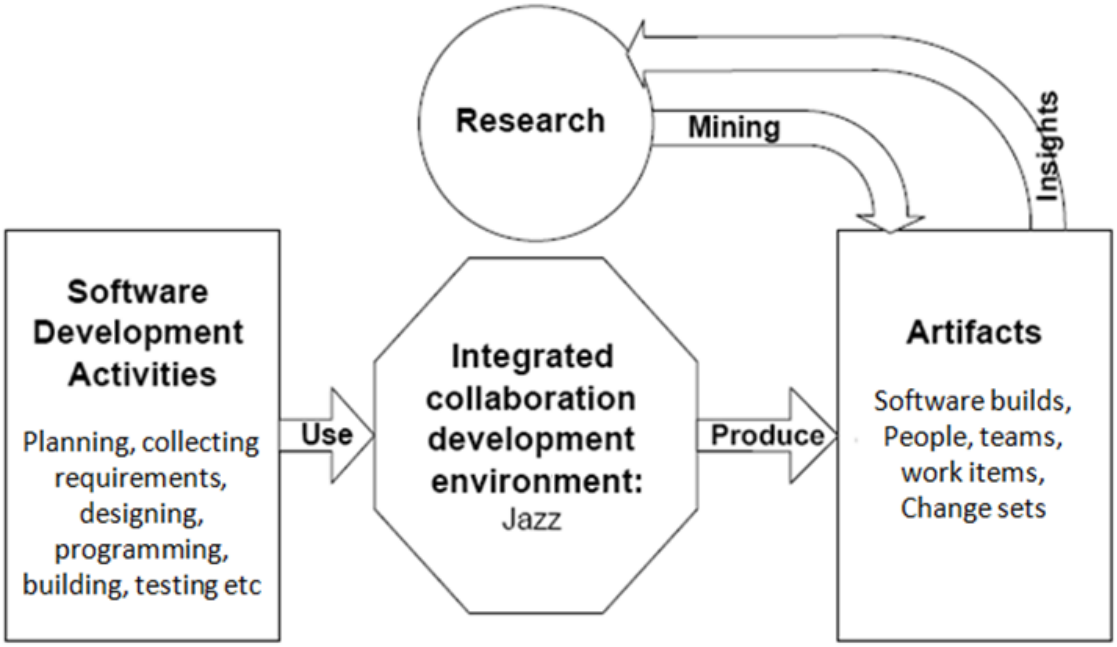

Figure 8: The research via artifacts process (Nguyen, et al., 2009)

Knowledge is gained from the artifacts extracted from the repository as opposed to devoting long periods of time for querying developers of the technology via a series of interviews and surveys. Therefore the approach provides a degree of practical separation between the research and the day-to-day activities of the development team. The research via artifacts process is in essence a structural view of research and therefore requires a degree of control at different levels of abstraction to ensure that a given research goal is addressed. In Figure 8 the process labelled “mining” is at a lower level of abstraction and it is here that knowledge Discovery in Databases (KDD) methodologies is used to control the data mining process. The Design Science paradigm is an iterative process that sits at a high level of abstraction. With particular reference to Figure 8, the Design Science paradigm applies to how insights gained from the artifacts are used to iteratively refine the research and data mining activities.

Even though there are limitations in only utilising a data repository, as not all aspects of the project are captured, it still consists of a rich source of information. The prototypes are developed within an iterative and agile process. Data is extracted from the Rational Team Concert (RTC) client API. The extraction tool is extended appropriately when

adding new queries to the data query set within each development iteration. Due to the large size of the repository, each time an extraction is made it updates earlier data sets as opposed to extracting all the data each time (Nguyen, et al., 2009). In doing so it significantly reduces computational work load.

This research is both exploratory and constructivist in nature and therefore utilises the system development research methodology (Nunamaker, Chen & Purdin, 1991) and the design-science research guidelines (Hevner, et al., 2004) to design, develop and implement a framework. A systems development research process involves: constructing a framework, developing system architecture, analysis and design, building a system, observation and evaluating phases. In the constructing a framework stage, a clear definition of the research problem is vital in defining the research process, this is defined in the research question for this work. In the development of the systems architecture stage, system components (modules), functionalities and interactions between components (relationships) are defined via the available artifacts within the Jazz repository. The architecture then goes through an analysis and design stage, where the design of data structures, databases or knowledge bases is created (Harman & Tratt, 2007). Insights into such structures and knowledge will be gained from the previously mentioned data mining methods and a set of evaluation criteria that measures both the accuracy and sensitivity of the results. To create research constructs that reflect the reality of the software project a quantitative research approach has been adopted under positivist paradigm. Quantitative research revolves around being able to find answers by physically measuring the reality of observations and by creating formulas to aid prediction for the future. This enables the possibility of measuring cause and effects of what goes wrong within software projects. An implementation of the design stage is made during the system building stage. Once a model is built and tested, observations provide insight into the accuracy of the built system and evaluations can be made for further development of enhancements. This provides researchers with insights into the advantages and disadvantages of concepts, frameworks, designs and knowledge discovery.

At a lower level of abstraction this work is also experimental in nature. In the context of this thesis, these experiments are attempts to extract knowledge from the data repository using a data mining approach. The experimental design deployed as part of the research

is discussed in more depth in section 3.2. For this research, the experiments conducted attempt to construct different understandings of the underlying data as a means to develop an appropriate approach for use in a real world scenario.

3.1 Data Mining Software Repositories

Data mining, KDD and machine learning techniques have been used in previous work to generate models of network data (e.g. social networks, web pages, relational databases and data on people, places, things and events extracted from textual documents). Data mining techniques have been developed largely for use with relational databases, where records have dependencies between one another (Jensen & Neville, 2002). The KDD process consists of identifying valid, useful and understandable patterns within data. In this instance, data is a set of values and a pattern is a subset with the data that describes a particular model. Before any mining processes take place it is first necessary to gain an understanding of the application domain. This will aid the data miner in making decisions within the KDD process. For example the researcher will need to identify the goals and generate hypotheses for the KDD process. This will aid towards decision making regarding the selection of particular data mining models (categorical or numerical) such as classification, regression and clustering as well as the selection of a models' parameters. The goal of the mining may be to understand a particular model or use a model for prediction. KDD is an iterative process and fundamentally involves the following steps:

1. Creating a target data set
    a. This may involve creating a subset of data from the available samples from the repository that are suitable for the data mining goals.
2. Cleaning and pre-processing a data set
    a. Removing noise, collecting relevant information about the model and deciding strategies for handling missing data and accounting for any other changes to the data set.
3. Data reduction and projection
4. Finding significant and useful features within the data depending on the goal of the mining task. This step may involve reducing or transforming the representation of the data.
5. Data mining

6. The searching process for patterns of interest
7. Representation of patterns from classification, regression and clustering
8. Interpreting mined patterns

During this process the data miner may result in re-iterating steps 1-8 as new information is revealed about the data. This is because new knowledge may resolve conflicts or change previously held beliefs. Additional information is provided from visualisations of the extracted data and patterns which can aid in forming actions from discovered knowledge. Actions from using newly discovered knowledge include incorporating it into another systems' implementation or documentation. The basic flow of steps and when potentially multiple iterations can occur is illustrated in Figure 9.

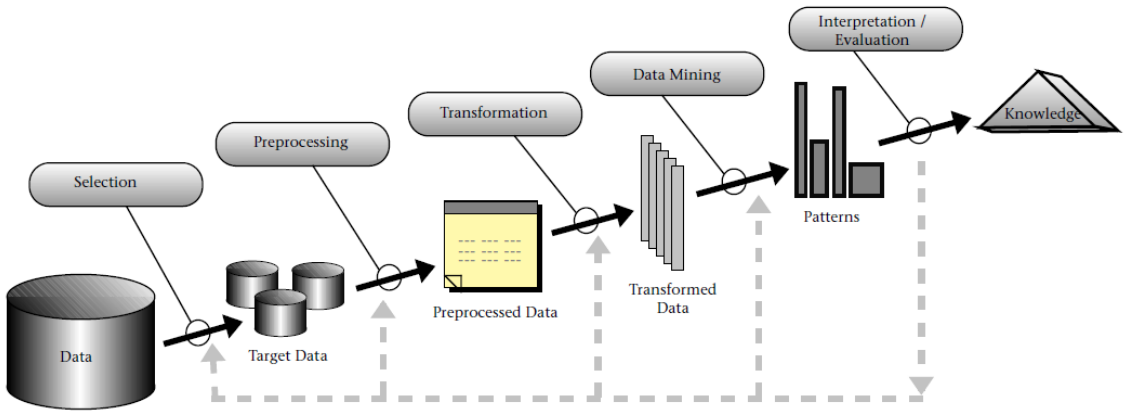

Figure 9 Overview of steps within the KDD process
From data mining to knowledge discovery in databases (Fayyad, Piatetsky-Shapiro & Smyth, 1996)

Before applying data mining methods, data characteristics such as size, the degree of connectivity among elements and how comparable elements within the data are, need to be analysed for estimating computational effort and result accuracy. It is also necessary to look at the number and types of links that exist between the research artifacts of interest and the type of mining task that will be undertaken (e.g. supervised learning or unsupervised learning) (Sebastian, 2002). In order to construct models for predicting software success and failure it is first necessary to distinguish a useful set of parameters that incorporate aspects from both social and software artifacts. These parameters can either be empirical or derived and are heavily dependent on the data that is available.

The research via artifacts process refines the design science approach making it applicable to the knowledge development process towards experimental design.

3.2 Experimental Design

This research is broken down into 3 major experimental phases, each with their own set of sub-stages and activities that ensure that the enquiry is undertaken in the context of the overall research methodology. Phase 1 involves an exploration of Jazz and its client API. To do this an application has been developed by the researcher that extracts the software artifacts of interest. Additional tools are developed to derive software (see Appendix A: Software Metrics) and social metrics (see Appendix B: Social Network Metrics) from the extracted artifacts. From this various data sets are constructed representing different software metric aggregations. The data sets are then pre-processed, where observations with missing data and noise are removed. The data sets are then filtered using Subset Evaluation and Information Gain methods for identifying significant predictors for both software and social metrics. Data mining methods are then applied to provide initial insights into predicting build success and failure. Mining methods explored include the j48 classification decision tree, Naive Bayes and Bayesian Networks. The main purpose of this experimental phase is to take a coarse exploration through all of the possible permutations and combinations of mining approaches and data aggregations in order to highlight the most promising candidates for further investigation. This phase is required because to conduct an exhaustive exploration of the experimental space to the depth required in later phases is simply not feasible in a realistic timescale.

Phase 2 builds from the analysis of the preliminary mining experiments with the aim of increasing classifier results for the promising methods in terms of both accuracy and sensitivity. During this phase a new feature reduction method is explored, where the features selected from the previous phase experiments are counted and are selected based on how frequently they appeared. In addition to this the before and after state metrics are also explored in terms of feature selection, where features selected from the after state are used to filter before state metrics. Towards the end of phase 2 the application of SMOTE is explored to generate more instances of minority classes within the data.

Experimental phases 1 and 2 aim to provide insights by first exploring the software metric data by using a variety of data mining methods. These experiments were the starting point of the research to address the overall research goal defined in Chapter 1. More specifically, phase 1 provides initial insights into which methods are best for predicting successful and failed builds. Phase 2 seeks additional insights by focusing on whether it is possible to improve classification performance on predicting failed builds by exploring a variety of techniques. The need to do this emerged during the research itself as would be expected in the natural cycle of build-test-refine that is associated with constructivist research. This phase builds from the results of the first phase, by taking the best performing data sets adopting additional methods (SMOTE and Frequency Feature Selection) in an attempt to enhance the classification results in terms of both accuracy and sensitivity criteria.

Phase 3 includes an extraction and exploration of communication metrics from the project contributors. This phase aims to provide insights into the confidence of the ability to predict build outcomes from communication metrics. This phase of research has much overlap of the work conducted by Wolf et. al. (2009). Replication of work was necessary to maintain consistency. This experimental stage combines the use of software metrics and communication metric data sets to see if they enhance the prediction accuracies further. This was considered from the early stages of the research, but following this line of enquiry was dependent on being able to gain some degree of prediction from just the software metrics. During this phase the best models are taken from previous mining experiments, are implemented and are executed.

This work revolves around the use of classification methods for the analysis of software metrics. For this purpose the Waikato Environment for Knowledge Analysis (WEKA) (Hall et al., 2009) machine learning workbench is used.

There are various challenges that arise when adopting data mining approaches. Real life data is not always suitable for the mining process. There is often noise within the data, missing data, or even misleading data that can have negative impacts on the mining and learning process (Chau, et al., 2006). The project data that is extracted from Jazz was gathered during the development of Jazz. As a consequence features that automatically capture project processes did not exist until later development stages of Jazz (gaps

would often appear at early stages of the project data set). Excluded from the data set were instances that had no work items associated with a build (as no social network metrics could be extracted for those instances), build results that had no source code files (as no software metrics could be derived from those instances).

Software metrics from continuous builds were used to construct the main data sets, however in doing so there were more instances of successful builds than failed builds. In order to balance the data set failed builds were injected from nightly and integration builds. This option was preferred over removing successful builds from the data set, thus decreasing the possibility of model over-fitting. In total, approximately 200 builds were included in the generation of the metrics.

In the early stages of this research an explorative style is adopted in the first instance as an attempt to discover the implications of the ratio of features to instances. To that end, the early experiments utilizing the Jazz metrics have been designed to explore the different ways that metrics can be extracted from Jazz. This consists of approaches intended to reduce the number of features in the data set as well as to calculate metric values differently.

3.3 Build Prediction with Software Metrics

The first stage of experiments systematically filters the available metrics using a variety of methods to simplify the problem space and determine the best classification trees. The term "best" is open to interpretation which is a reflection of the constructivist nature of this research. In reality, the best classification is normally a subjective interpretation of the trade-off between conflicting goals. For example, this research aims to maximise overall classification accuracy, classification accuracy of the minority class and sensitivity of the results. It may be challenging to satisfy all three criteria. In an attempt to improve classification accuracy, clustering accuracy and sensitivity results a range of feature selection methods are applied to each data set. However, before any data-mining activities take place it is first necessary to become familiar with the Jazz Repository for extracting various software project artifacts. The Jazz repository is complex and can be extracted and interpreted in many different ways. The following details the interpretation of the objects within the repository that have been utilised for extracting software metrics from software build change sets and work item contributor communication.

3.3.1 Extracting Software Artifacts

For extracting the artifacts software has been developed that has utilised the Jazz client API (application programming interface). Unfortunately at the time of extraction, there was little API documentation available for this process, however documentation about the API can be found in the online Jazz forums. For software metrics only Java source code files are extracted from builds' change-sets, as these are the only file types that are useful for deriving all required metrics types. There are many different ways change sets and communication can be extracted from the repository it is therefore essential to be unambiguous how artifacts are queried and interpreted. All software development done in this research is in Java using IBMs Rational Team Concert, which provides an eclipse-based client IDE (Integrated Development Environment). In addition to this IBMs DB2 database management system (and server) is used for retrieving software artifacts from Jazz. At no point during the experimental phases is the data within the repository modified or is new data created within the repository. To begin the extraction process a connection needs to be established. This is achieved by using the Jazz client API and more specifically requires the use of the objects shown in Figure 10, where IloginHandler is an interface from ITeamRepository.

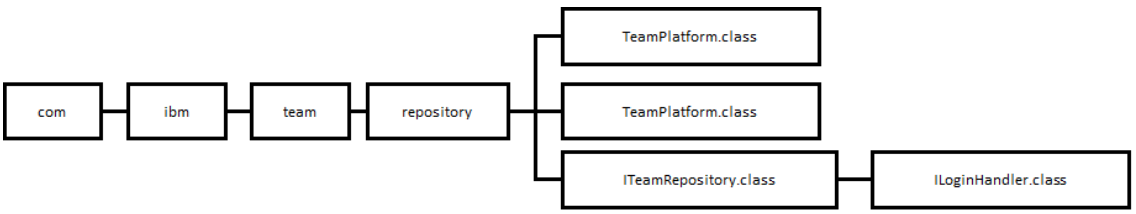

Figure 10 Jazz API, Creating a connection

The repository allows the retrieval, creation and update of software development artifacts. Complex queries can be used to retrieve objects based on their relationships with one another or their attributes.

A build item represents a software build. A build, in this sense, is a compilation of software to form a working unit. In Jazz a build contains the work from one or more work items and these work items are not necessarily unique from build to build. The Jazz database does not keep a complete record of every build over time and as a result the number of builds available to access is limited. Nevertheless builds are an essential

component to this work as their relationship with change sets allow for the extraction of source code files that will be used for extracting software metrics. The attributes of a build item includes:

- Build ID
    - Unique Identifier of a particular software build
- Label
    - Identifier of a particular build (viewed by end user) and provides information on what type of build it is.
- Start Date
    - The builds' start date
- Duration
    - The duration of the build
- Build Status
    - If a build state is "OK" it has been a successful build and if a build state is an "ERROR" the build has failed to compile. However, if a build status is "WARNING" it means that even though the build has compiled (unless the compiler set to treat warnings as errors), there may be potential problems with the source code based.

Jazz builds are, in this case, a vital component for extracting both software metrics and constructing social network metrics to form data sets for mining. The Jazz objects that are used for extracting build instances are shown in Figure 11. All build instances are retrieved from the repository regardless of their connection to other existing (parent) artifacts.

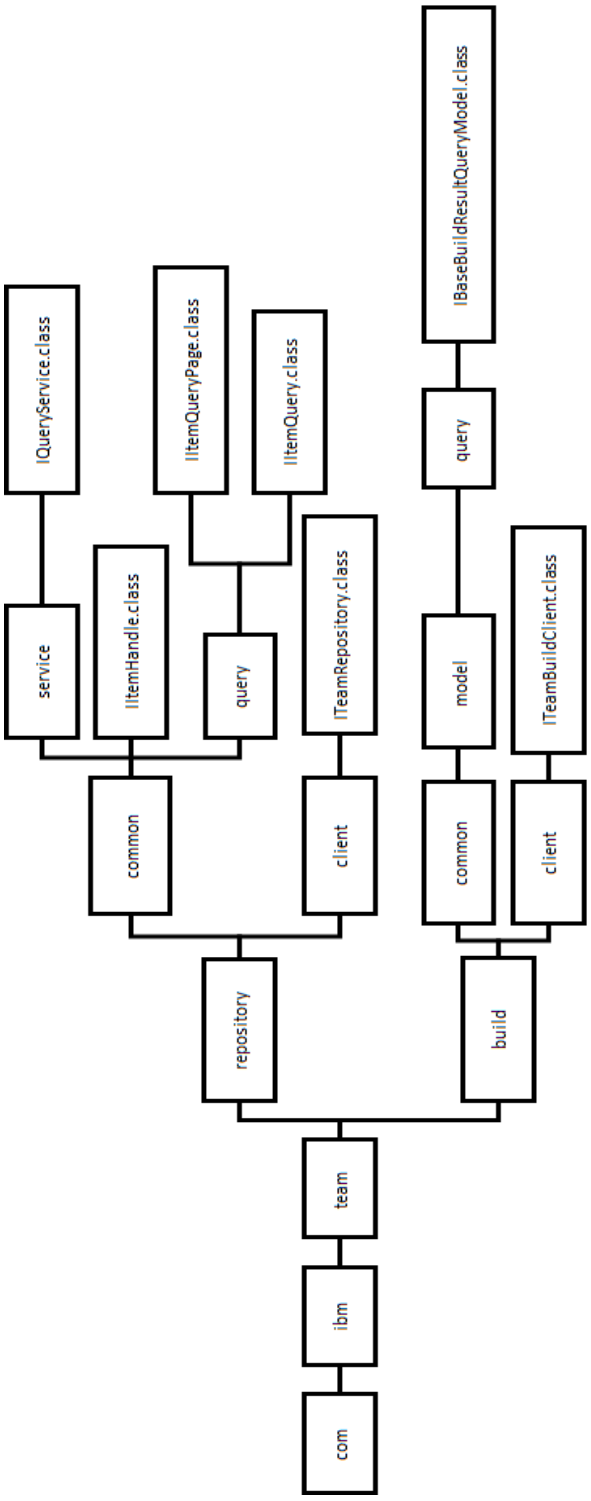

Figure 11 Navigating the Jazz API for Retrieving Software Builds

The Jazz repository consists of various types of software builds. Included in this study were continuous builds (regular user builds), nightly builds (incorporating changes from the local site) and integration builds (integrating components from remote sites). Builds included in the final data sets were chosen from a range of build definitions within Jazz.

These build definitions are applicable to both the before and after state change sets. The following is a summary that is extracted directly from the repository, of the build components that the software metrics for this research were derived from:

- Beta and Weekly Builds
    - A Beta build generally begins when the software is feature complete. However, builds during a beta phase tend to have more software bugs than "completed" software, as well as potential speed and performance issues.
- Continuous Builds
    - Within this snapshot of the repository, continuous builds are the most common build type and are used for building major components of the Jazz project. These include
    - Analysis components:
        - The Analysis Component provides an extensible toolkit for integrating static analysis tools and frameworks, like SAFE (IBM Research project) and the CodeReview framework (Eclipse TPTP project), into the Jazz platform. It allows to perform analyses interactively inside the IDE and non-interactively from within a batch team build.
    - Team build components:
        - The Jazz Team Build component integrates a team's build system into Jazz and provides build awareness to the team. This is accomplished with build progress monitoring, build alerts, build result viewing and linkage of builds with other artifacts in Jazz, such as change sets and work items.
    - Connector components:
        - This component provides interoperation and synchronization between Jazz SCM and other SCM systems
    - Core file system components:
        - Components that are responsible for the server-side architecture, including the server programming model, server extensibility, repository item modelling and storage, database interactions and object persistence, web services dispatch and security.

- Service components:
  - The Jazz REST Services components implement "RESTful" web services for Jazz. This web services provides stable long-term programmatic web-based "APIs" for directly accessing the facilities and data offered by the various Jazz components. These web services use an arrangement of URIs, HTTP methods and standard representation languages such as XML and JSON, that work like the rest of the web.
- Process components:
  - The Team Process component provides Jazz's process support foundations. Team Process is a kernel component, so its facilities are available to other components in all client and server configurations. In this context, process refers to the collection of practices, rules, guidelines and conventions used to organize work.
- Reporting components
  - The Reports component gathers data about the Jazz repository and presents it to the user in a readable format. It manages a data warehouse, in which facts about the repository are stored at periodic intervals. It also integrates with the BIRT reporting engine to render reports and charts based on the information in the data warehouse.
- Source Code management (SCM) components:
  - The SCM component manages source code and other digital assets that a team creates. It is able to recreate earlier configurations in order to maintain previous versions of a product. It may also be used to prevent unauthorized access to assets or to alert the appropriate users when an interesting asset has been altered.
- Web user interface components:
  - This component provides frameworks, APIs and infrastructure for creating web browser-based user interfaces. Also develops the Admin Web UI and the Jazz Team Server Setup Wizard.

    - Work item components:
        - Provides support for managing defect reports, feature requests and other development tasks.
- Nightly Builds
    - Nightly build for the ClearQuest Connector. This build, unlike the continuous build, runs tests that require access to ClearQuest-installed component.

Each build may be linked to zero or more work items. A work item is a description of a unit of work. The attributes of a work item include:

- Work Item ID
    - A unique numerical value which represents the identification of the work item
- Type
    - Defect, Task, Enhancement, Story, Build Item, plan item or Other
- Summary
    - Summary of what the work item is about
- Description
    - Description of what the work item is about
- Severity
    - Unclassified, Minor, Normal, Major, Critical, Blocker
- Priority
    - Unassigned, Low, Medium or High
- Due Date, Creation Date, Modified Date, Resolution Date
- Has History, Is Complete
- Creator UserID, Modifier UserID, Resolver UserID, Approvor UserID, Subscriber UserID
- Comments
    - Discussions by contributors about a work item

During the iterative process of extracting builds, each builds' set of work items are also queried. The objects associated with querying work items that are used within this

research are shown in Figure 12. At this repository level, it is possible to start extracting social network elements which will be used towards later experimental stages.

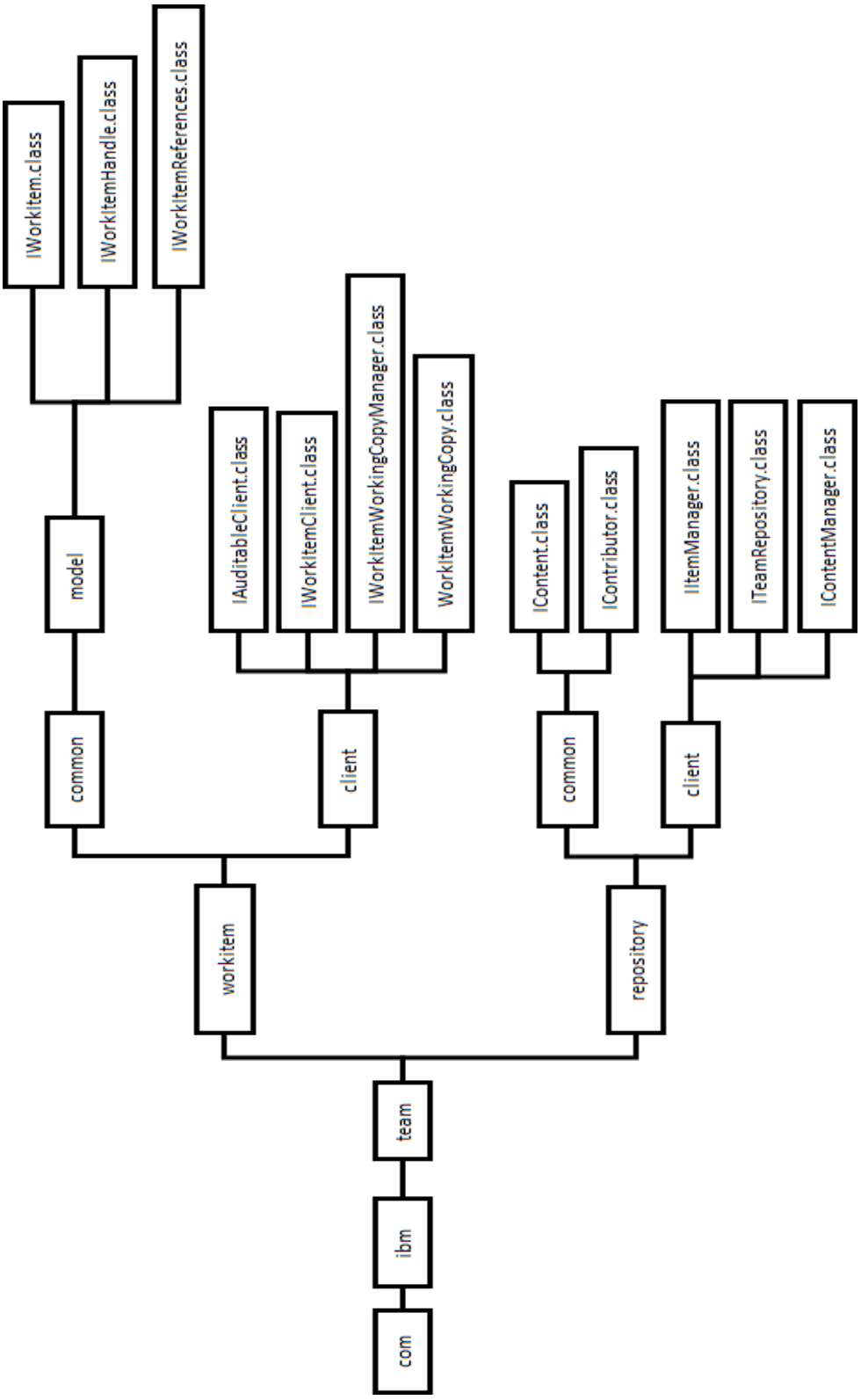

Figure 12 Navigating the Jazz API for Retrieving Work Items

Each work item or software build that is extracted potentially has a change set which is linked to it. A change set is a related group of files that are modified during a projects' lifetime and is the fundamental element of source control within the Jazz repository. A change set may consist of changes to individual files (modifications), deletion of files or addition of new files. A small change set may modify only a few lines of code within a single Java file, whereas larger change sets consist of changes to multiple files and folder contents and structures. Each change set has a record of two states, the before and after state. Each state has versionable files and folders. The before state of a change set is the state that is recorded before any changes have been made. Whereas the after state of a change set represents the collection of files and folders after all the changes have been made and is marked as "complete". For each build software metrics will be extracted that correspond to the before and after states of that build. These will be used to construct data sets for data mining. The before and after software metric data set states are not merged and are mined separately. This is undertaken to determine whether an early prediction of build outcome can be achieved. Once this stage is complete a comparative analysis is carried out to determine the best performing data sets in terms of highest number of correctly classified instances and sensitivity measurements. The classification trees of the best performing mining experiments are also evaluated in terms of making rational sense and their degrees of complexity.

In the Jazz data set a given build contains change sets that indicates the actual source code files that are modified during implementation. These change sets consist of change objects which are used for extracting project files. The objects that are utilised for extracting source code are shown in Figure 13.

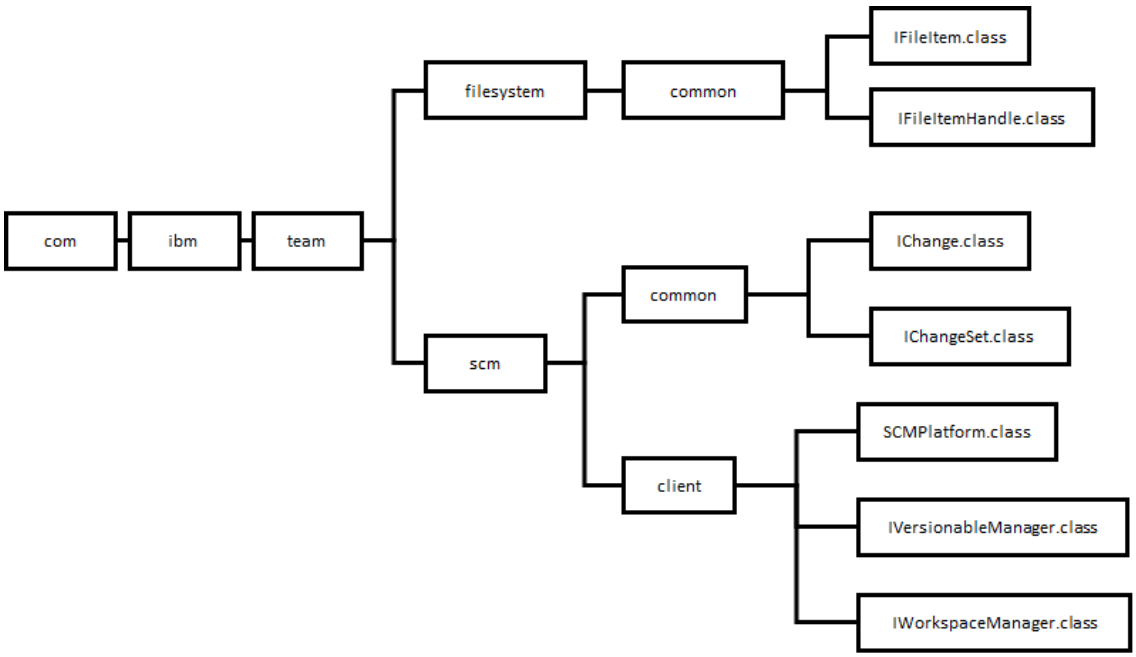

Figure 13 Navigating the Jazz API for Retrieving Source Code

During the extraction process a systematic file hierarchy is set up, where the IDs and results of the builds are preserved as folder names and Java source code stored under the correct folders, respectively. Preserving these extra details about the builds in one form or another is important for constructing the data sets. For this study the build result attribute will be the classifier for all data mining experiments.

3.3.2 Software Metric Aggregations

Forty-two software metrics are calculated for the source code of each build using the IBM Rational Software Analyser. As the repository is developed and used by an IBM team, the IBM software analyzer is a tool that would be readily available for the team to use to analyze their source code metrics. Aggregation is the process of computing statistical measures such as averages, medians, or maximums that summarise the result for a particular set of software metrics. Aggregated software metric data can be processed by mining algorithms and the accuracy and sensitivity between such aggregations can be measured and compared.

In order for IBM's Rational Software Analyzer tool to calculate the metrics for builds, each build needs to be presented as its own Java project. To construct Java projects for the extracted Java files a separate application has been developed which takes Java files and constructs an eclipse project for them (by constructing all the necessary files and packages of a Java project for the Eclipse IDE).

The builds as Java projects are then loaded into an Eclipse workspace. Since there are approximately 200 build projects (some of which are quite large in size) the memory size allocated to Eclipse needed to be increased from its default value. Another alternative is to load the projects into separate workspaces if computer memory becomes an issue. To automatically generate metric reports a batch file that references the analyzer workspace and a metrics rule file is used. The metric rule file is exported from the software analyzer, containing all metrics that are to be processed by the analyzer. With the snapshot of the Jazz repository provided by IBM for this research, it can take 2-3 days of processing using a computer with the following specifications: AMD II triple-core 2.20 GHz, 8 GB DDR3 ram and a Windows 7 64-bit operating system.

The Extensible Mark-up Language (XML) metric reports generated are stored with the build ID and result preserved within the folder system. An additional Java application has been developed that reads the xml reports and generates the desired software metric data set from them. The resulting data set is stored as a plain comma-separated text file (which can then be imported into excel and Weka). The metric data sets are then used to initiate the data mining processes. To ensure that the metrics extracted were valid, sample programs, with expected metric outcomes where created by the researcher.

To begin this work investigates how to aggregate software metrics from collections of source code files: To begin the software metrics are generated for the available software build projects and are stored in an individual report document in XML format. The typical layout for metric data within the report is as follows:

<Category Name>
    <Rule Name>
        <Result Type>
        <Result Type>

The <Category Name> tag is the category assigned for the software metric rules types. For example software metrics such as "Average block depth", "Weighted methods per class" and "Maintainability index" are all rules that are under the "Complexity Metrics"

category. The <Result Type> tag represents the various levels that the metrics are calculated from. Metrics can be at package, class or method level depending on their rules. At the <Rule Name> and <Result Type> levels of the report hierarchy metric values are assigned. In the work to date various options for aggregating the metric values to produce representative metric data sets for each complete build is explored. The metrics extracted at the <Rule Name> level of the report hierarchy will construct the "Rational Software Analyzer" metrics data set, which will be coined as the RSA data set. The Max, Median, Mean and Total metrics data sets are aggregated from the <Result Type> level. At these levels the value for each metric for each source code file is calculated individually, then the average, maximum, median and total sum is propagated up to the build level. A minimum data set is not included as most instances within the data had zero values for metrics. In total 10 data sets are constructed: RSA, max, median, mean and total software metric values for both the before and after states of Jazz builds.

Source code files are extracted for each build within the repository. Subsequently software metrics were generated by utilizing the IBM Rational Software Analyzer tool. As a result the following traditional, object orientated and Halstead software metrics were derived from the source code files for each build, for each aggregation. In total there are 42 software metrics explored in this study. The metrics are derived for the formulas and definitions summarised in Appendix A: Software Metrics. All 42 software metrics are included in this research:

- Complexity Metrics:
    - Average block depth, Weighted methods per class, Maintainability index and Cyclomatic complexity
- Dependency Metrics:
    - Abstractness, Afferent coupling, Efferent coupling, Instability and Normalized Distance
- Halstead Metrics:
    - Number of operands, Number of operators, Number of unique operands, Number of unique operators, Number of delivered bugs (estimated), Difficulty level, Effort to implement (estimated), Time to implement

(estimated), Program length, Program level, Program vocabulary size and Program volume

- Cohesion Metrics:
  - Lack of cohesion 1, Lack of cohesion 2, Lack of cohesion 3
- Basic Metrics:
  - Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average number of comments, Average lines of code per method, Average number of methods, Average number of parameters, Number of types per package, Comment/Code Ratio, Number of constructors, Number of import statements, Number of interfaces, Lines of code, Number of comments, Number of methods, Number of parameters and Number of lines
- Jazz Metric (classifier):
  - Build Result, this attribute indicates whether a software build was a success, failure or resulted as a warning. This metric is used as the classifier for the data mining experiments.

A more directed set of metrics was not defined despite there being some evidence in the literature that certain metrics may have limitations or deficiencies. Similarly, no new metrics were included if they could not be calculate by the Rational Software Analyser tool. The decision to maintain the complete list of available metrics was an attempt to simulate the range of options that would be open to the Jazz development team in their day-to-day activities.

3.4 Data Mining Methods

Feature selection methods used in this study include Subset Evaluation, Information Gain, principal components analysis (PCA) and no feature selection. The "no feature selection" method acts as a control and provides a benchmark for each experiment as it will indicate whether or not feature selection methods are adding any value to the data mining. Classifier methods used for this study include the j48 tree, Naive Bayes and Bayesian network. Simple k-means clustering is also adopted for further exploration of the nature of the data. Default Weka values are used for all classification and clustering methods except for 1 parameter for the Bayesian network. The parameter that is changed is the number of max parents within the K2 search algorithm and is increased from 1 to the total number of features included for the mining.

The j48 mining algorithm in WEKA is an open source implementation of the C4.5 decision tree learner. These trees are represented as connected acyclic node-edge graphs and are fundamental for displaying data structures. Recursive partitioning tree models can be used for prediction and has become a popular alternative for regression and other algebraic methods. Branches within a decision tree indicate rules that have been learned from the data. Leaf nodes within a decision tree indicate a class prediction should that node be reached. To interpret a decision, displayed via WEKA, the trees' leaf nodes will contain the predicted classification followed by two sets of numbers. The first set of numbers is the total number (weight) of instances assigned to that node and the second number is the number (weight) of instances that have been misclassified. Fractional numbers occurring within leaf nodes indicate missing attribute values.

The Naive Bayes classifier is used as another probabilistic classification algorithm and is based on Bayes Theorem. Naive Bayes is robust in the sense that it has the ability to ignore or cope with missing data and smaller data sets. The Bayesian network, also known as a belief network, is an additional probabilistic classification algorithm and, unlike Naive Bayes, constructs conditional dependences and relationships between features and a classifier. The Naive Bayes, Bayesian Network and j48 algorithms have been selected due to their potential for handling complex sets of relationships, which according to much of the research performed using software and social metrics is a necessity. Given the relatively small size of the data sets 10-fold cross validation is utilized in order to make the best use of the training data. The relative optimism of cross-validation is acknowledged and will be addressed in future work when more data becomes available from the Jazz project.

For each mining activity the total number of correctly classified instances and incorrectly classified instances are captured. The confusion matrix (also known as a contingency table) is also documented where the total number of correctly classified successful builds and total of incorrectly classified successful builds are presented. The total number of correctly classified failed builds and number of incorrectly classified builds are also recorded from this matrix. For example in Table 1, under the # Successful builds correct (incorrect) heading, the first value 109, is the total number of

correctly classified successful builds. The value within the brackets (18) under the same heading is the total number of incorrectly classified successful builds.

Table 1 Example of Recorded Accuracy Results

# Successful builds correct (incorrect)	# Failed builds correct (incorrect)	Correctly Classified Instances	Incorrectly Classified Instances
109 (18)	49 (22)	79.798%	20.202%

Performance data that is captured for both successful and failed builds includes TP (True Positive) Rate, FP (False Positive) Rate, Precision, Recall and F-Measure. When precision, recall and F-Measure values are closer to 1.0 it is considered to be a desirable result and values closer to 0.0 are poor. The TP Rate is the rate of instances that are correctly classified as a successful or failed build amongst all instances. Presented in Table 2 the TP rate for successful builds is calculated by 109/(109+18) and the TP rate for failed builds 49/(49+22). For this example 198 build instances were processed in total. The FP Rate is the proportion of instances that are classified as a success (or fail), but belong to a different class, amongst all the instances that are not of that class. For example the FP rate of successful class is 49/(109+49) and for the failed class is 18/(109+18). The precision measurement is the proportion of instances that truly have a class among all instances that were correctly classified of a class. For example for successful builds the precision is calculated by 109/(109+22) and for failed builds is 49/(18+49). The recall value is the total number of correctly classified instances divided by the total number of correctly classified plus the total number of incorrectly classified instances of a class. For example the recall for successful builds is given by 109/(109+18) and for failed builds is 49/(49+22). The F-Measure is a combined measurement for precision and recall values and the formulae is 2*Precision*Recall/(Precision + Recall). The F-measure for successful build instances is 2*0.832*0.858/(0.832+0.858). Table 2 presents an example of how the sensitivity measurements are documented along with weighted averages for both successful and failed build measurements. In addition to classification accuracy and sensitivity measurements, the j48 decision trees are also examined in terms of intuitiveness.

Table 2 Example of Recorded Sensitivity Results

Class	TP Rate	FP Rate	Precision	Recall	F-Measure
Successful Build	0.858	0.31	0.832	0.858	0.845
Failed Build	0.69	0.142	0.731	0.69	0.71
Weighted Average	0.798	0.25	0.796	0.798	0.797

For this thesis an experimental hypercube is used to determine each data mining configuration and is illustrated in Figure 14 and Figure 15. In Figure 14 the n-dimensional cube is comprised of six factors, this includes the 42 software metrics, the 5 software metric aggregations, the 199 build instances that are captured over time, the before and after change set states of a build, the 4 feature selection methods and 3 data mining methods.

Various experiment setups are executed over 4 major phases. The first phase explores the software metrics and various aggregations (Total, RSA, Mean, Median and Max) using traditional feature selection and data mining methods. From the insights gained from the first phase, the second phase attempts to boost the classifiers' performance (increasing correctly classified instances and sensitivity ratings). The methods explored for the second phase are presented in section 3.5. From the first and second phases the relationships between the best performing software metric data set and social network metrics data are explored in the third phase (using methods described in section 3.6). Finally, the fourth phase explores how software metrics and social metrics evolve over time by encoding the best performing metric aggregation as data streams. In doing so the final phase moves away from traditional data mining methods, where data is treated as static and introduces the application of data stream mining methods. The experimental methodology for the final phase is covered in more detail in section 3.7

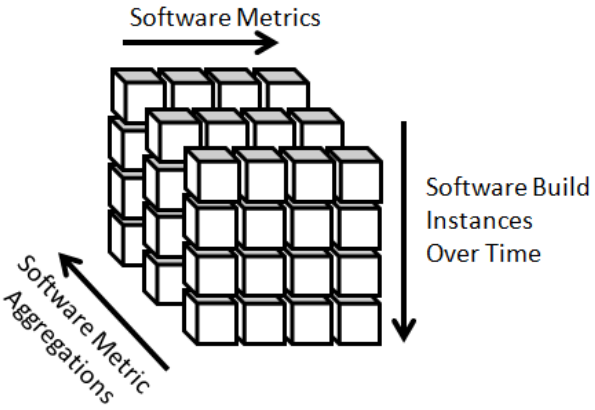

Figure 14 Software Metrics as Multidimensional Data

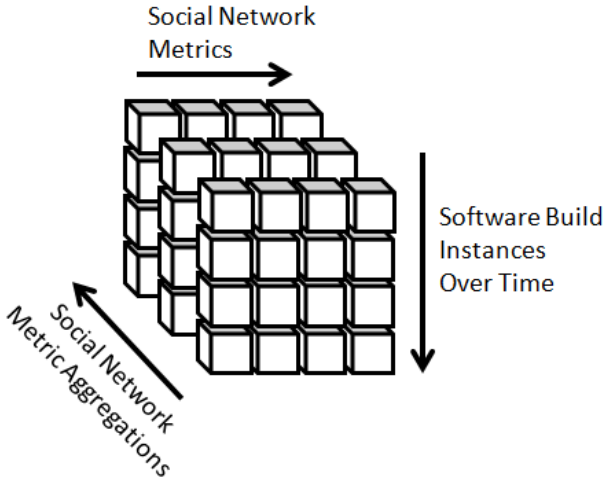

Figure 15 Social Metrics as Multidimensional Data

3.5 Increasing Prediction Accuracy of All Failed Builds

The second experimental phase involves a deeper focus on improving classifier performance of the initial data mining experiments by looking at potential alternatives for feature selection and introducing synthetic data instances. This phase was defined during the research as initial results were generated that helped refine the research direction rather than defined a-priori. This phase can be broken down into three main experimental stages. The first stage focuses on the frequency of selected features amongst the before and after states of the build metrics. The second stage explores the impacts of applying the features selected from the after state to the before state data, in order to see if future metric feature selection can enhance past metric data models. Finally, the third stage utilises SMOTE for synthetically generating more instances of any minority classes within the data for potentially improving the percentage of correctly classified instances. Mining experiments in stage 1 and 2 of this phase utilise

10 cross-fold validation. Mining experiments in stage 3 make use of a supervised learning method where test and training data sets are generated.

3.5.1 Frequency of Features Selection

From the initial experimental phase all features that have been selected from all experiments are counted. For example, if Cyclomatic Complexity was found to be selected from the mean metric data set using Subset Evaluation and in another experiment was found from the max metric data set using Information Gain, then Cyclomatic Complexity will have a frequency of 2. For this stage features are iteratively selected, based on their frequency and are then applied to the pareto-optimal data sets from the initial phase. This process is carried out separately on the before and after state data sets. During this iterative selection phase features are reduced by increasing the count threshold. For example from the various experiments if metrics were selected more than once, they are included within the feature set. In the next iteration a metric is selected if it appears more than twice and so on, until no more metrics are left to select. From the first experimental phase the resulting pareto-optimal data sets will have the iterative frequency feature selection method applied to them.

3.5.2 Applying After State Features To Before State Data Set

The next stage begins with the application of features selected from best performing feature selection methods, from the best performing after state data sets and applying each set of features to the respective best performing before data sets. In addition to this the frequency feature selection iterations performed on the best performing after state data sets is also applied to the best performing before state data sets. In order to manage risk of a failure build it is useful to revise and review risk exposures during software build cycles. In this case the before state metrics are used to determine if build failure is based on changes prior to any changes made in the source code. This will characterise source code that is about to be changed in terms of it being modified successfully or if there are patterns in existing code that may lead to failure.

3.5.3 Application of SMOTE

In stage three SMOTE is applied to the best performing data sets. To do this the data sets are broken down into individual training and test data sets for the exploration of supervised learning methods. The training set data consists of 70% of the original data set instances and will reflect the naturally occurring ratio of successful builds and failed builds. The remaining 30% of instances are then used as the test set. This creates a

stratified data set that will ensure that there is an unbiased estimate of classification performance.

This phase of experiments, utilising SMOTE, is broken down into two stages. For the first stage feature selection is applied to the data that is of the same change set state. For the second stage features from the after state are applied to the before state data. For both stages SMOTE will be iteratively applied to the training sets by incrementally increasing the number of generated instances. For instance the percentage of instances to generate ranges from 100% to 500% (inclusive) and is incrementally increased in steps of 100%.

3.6 Build Prediction Using Social Metrics

This phase focuses on the communication between developers to predict build success and failure and exploring its impacts on development. Within the repository each software build may have a number of work items associated with it. These work items represent various types of units of work and can represent defects, enhancements and general development tasks. Work items provide traceable evidence for coordination between people as they can also be commented. In addition to this they are one of the main channels of communication and collaboration used by contributors of the Jazz project. That being said there are, of course, other channels of communications which are not captured by work items, these include email, on-line chats and face-to-face meetings. Even though these elements are not captured, exploration of communication on work items offers a non-intrusive means to explore much of the collaboration that has occurred during the Jazz project. The Jazz team itself is fairly large, with 66 team areas for approximately 160 contributors that are globally distributed over various sites across the United States of America, Canada and Europe.

This aspect of the research is necessary despite there being previous work that has investigated social network analysis using Jazz (Wolf et al., 2009). Part of the overall goal of this research has been to consider whether the combination of software and social network metrics provides better prediction of build outcomes in comparison to the predictions on a single set of metrics only. To ensure the consistency between the two sets of metrics, it was necessary to generate social network metrics for the same builds from the Jazz data for which there are software metrics.

3.6.1 Extracting Social Artifacts

To explore the communication between contributors involved in builds, social network metrics are derived from the communication networks that are present within work items. Each work item is able to be commented on and this is the main task-related communication channel for the Jazz project. This enables contributors to coordinate with each other during the implementation of a work item. There are many elements, in regards to contributors, of a work item to consider. This makes the process of constructing a social network a little more challenging. In doing so some basic assumptions about the data is been made. Work items can have various contributors assigned to various roles, for example there are creators, modifiers, owners, resolvers, approvers, commenters and subscribers. For the purposes of this work a social network is constructed similar to the work presented by Wolf et al. (2009). For each social network constructed, nodes represent contributors involved with a build. A series of directed edges represent the communication flow from one contributor to another. A build can have zero to many work items associated with it. Therefore the social networks generated at the work item level are required to be propagated to the build level for analysis of its impact on build success. To do this if a contributor appears within multiple work items that are associated with a single build, only one node is created to represent that contributor (there are no duplicate nodes). Additional edges are added to reflect entirely new instances of communication that takes place between contributors. All edges within a network are treated as unique (there are no duplicate edges). This is because it would threaten the validity of metrics such as density. If a network is fully connected it a density of 1. If there are edges which represent each individual flow of communication the density metric would no longer be valid (potentially being greater than 1), which would make comparisons between networks metrics challenging.

For this research roles which are used to construct the network nodes include, committers of change sets, creators, commenters and subscribers. Committers (modifiers and resolvers) of change sets for a build are presented as a node, as they have a direct influence on the result of the build. Creators of a work item are communicating the work item itself with other members of the team. Commenters are contributors that are discussing issues about a work item. Subscribers are people who may be interested

on the status of a work item as it has impact on their own work and other modules. In order to generate the edges between nodes, the following rules have been implemented to establish connections between people:

a) The work items' creator is linked to its commenters [creator → commenters]. This connection is made because the creator of the work item has communicated the work item itself to the contributors who are collaborating on it.
b) The work items' creator is linked to the subscribers [creator → subscribers]. This link is made because the creator of a work item has given the subscribers something to subscribe to and has therefore communicated with them also.
c) Work item commenters are linked to subscribers [commenters → subscribers]. This link is established because commenters pass on information about the work item to subscribers. Comments on a work item may have direct impact on other work items which other contributors are working on within a build.
d) Work item commenters are linked to commenters [commenters → commenters]. Assuming that all commenters of a work item read all other comments on that work item. For example if commenters c_1 and c_2 comment on the same work item the assumption is made that $c_1$has read all comments made by c_2 ($c_1 \rightarrow c_2$). Another connection is made vice versa where it is assumed that $c_2$has read all comments by c_1 ($c_2 \rightarrow c_1$) on the same work item. Even though this may not reflect what happens in reality. There is no sure way to capture what a contributor may or may not read. It is therefore best to capture these types of relationships than to completely negate them.
e) Change set committers of a work item are linked to commenters [committers → commenters]. Committers have a direct impact on a build status and its work items by committing source code and documentation and therefore will affect the comments made on the work item itself. Assuming that change requires group collaboration.
f) Committers are also linked to subscribers [committers → subscribers]. Where committers affect and influence the state of one work item, collaborations may need to be made with its subscribers who work on other work items.

From these elements constructing the social networks for each build, the metrics are calculated are:

- Social Network Centrality Metrics
    - Group In-Degree Centrality, Group Out-Degree Centrality, Group InOut-Degree Centrality, Highest In-Degree Centrality and Highest Out-Degree Centrality
    - Node Group Betweenness Centrality and Edge Group Betweenness Centrality
    - Group Markov Centrality
- Structural Hole Metrics
    - Effective Size and Efficiency
- Basic Network Metrics
    - Density, Sum of vertices and sum of edges
- Additional Basic Count Metrics
    - Number of work items the communication metrics were extracted from
    - Number of change sets associated with those work items
- The classifier value for the mining experiments will, again, be by build result.

Centrality metrics are used to calculate a nodes' importance within the network, by its number of connections within the network. In general terms the degree of a node relates to the number of its connections to neighbouring nodes. The Out-Degree of a node is the number of its outgoing connections $C_{oD}(c)$. The In-Degree of a node is the number of its incoming connections $C_{iD}(c)$. The InOut-Degree of a node is the sum of its In-Degree and Out-Degree metrics $C_{ioD}(c)$. To add to the collection of network metrics the highest In and Out degree metrics are captured for each builds' work items. Doing so may prove to be useful as metrics that represent extreme values are likely to be either wanted or undesirable and therefore many relate directly to success and failure. To calculate the Group Degree Centralization index for a builds' network the following formulae is used (using In-Degree metric focus):

$$C_D = \frac{\sum_{i=1}^{g}[C_D(c^*) - C_D(c_i)]}{(g-1)^2}$$

Where g is the number of nodes, $C_D(c^*)$ is the largest node degree index in the network, $C_D(c_i)$ is any of the degree centrality measures of a node c_i. In addition to group degree centrality metrics the group betweenness centrality is also added to the collection of social network predictors.

Betweenness centrality indicates a nodes' importance in overall connectivity within a network. In this case individual nodes' centrality is derived from the number of shortest paths from all nodes "passing" through it, divided by the total number of shortest paths within the network. It is also possible to calculate betweenness centrality for edges. Betweenness centrality metrics provide insights into how a network is linked and the traffic-directing capabilities of a node or edge in a graph. To calculate the Group Betweenness Centralization metric, using Freeman's formula, for an entire network of a software build:

$$C_B = \frac{\sum_{i=1}^{g}[C_B(c^*) - C_B(c_i)]}{(g-1)}$$

Where $C_B(c^*)$ is the largest betweenness index of all contributors within the network.

The Markov Centrality metric makes use of a global "objective" ranking function, where if a node (C_M) is considered to be "central" (is positioned closer to the centre of the network mass), it will have a higher ranking than nodes which are less "central". Using the same formula as above and substituting C_B with C_M, the Group Markov Centrality is obtained. Where $C_M(c^*)$ is the largest ranking node. This metric is not explored in previous work by Wolf et al., (2009), but has potential as to identify nodes of relative importance of a node by "travelling" random paths within a network (White & Smyth, 2003)

Structural Hole Metrics included in this study include the effective size and network efficiency metrics. The effective size of a node is the number of its neighbouring nodes, minus the average degree of those in c_i's ego network. This is not counting their connections to c_i.The efficiency metric normalizes the effect size of a node (c_i) by dividing its effective size by the number of its neighbouring nodes.

The density metric is calculated as a percentage of all possible connections within a network. A network that is fully connected has a density of 1 and a network with no connections has a density of 0.

To implement the rules for these metrics an application has been developed in Java by the researcher that establishes the social network that utilises the Jazz API and calculate network metrics using the Java Universal Network Graph Framework (JUNG http://jung.sourceforge.net/). If provided with a sample network that is illustrated in Figure 16 the betweenness and Markov vertex rankings for nodes are shown in Table 3 and the overall metrics extracted for such a network are shown in Table 4. Nodes U-Z are representations of contributors and the links are the flows of communication. In this study all networks are composed of directed edges, as per the connection rules above.

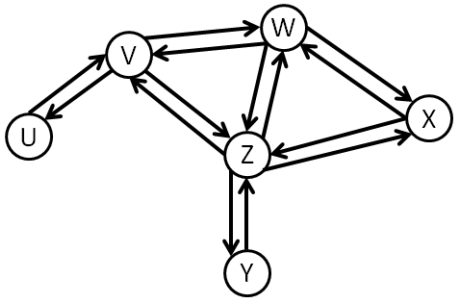

Figure 16 Social Network Example

Table 3 Example Node Rankings for Betweenness and Markov Centralities

Node	Betweenness	Markov
U	0	0.0567
V	8	0.2056
W	2	0.2237
X	0	0.1327
Y	0	0.0620
Z	10	0.3190

Table 4 Example of Network Metrics

Network Metric	Value
Group In-Degree Centrality	0.080
Group Out-Degree Centrality	2.000
Group InOut-Degree Centrality	0.384
Highest In-Degree Centrality	0.800
Highest Out-Degree Centrality	0.800
Node Group Betweenness Centrality	8.000
Markov Centrality	0.1828
Edge Group Betweenness	1.538
Effective Size	9
Efficiency	4.333
Density	0.466
Sum of Vertices	6
Sum of Edges	14

Similar to the software metrics exploration phases, where success is represented by build result, these build results are also used to represent coordination outcomes. From this perspective build success is regarded as coordination success. Just like the first stages of exploring the software metric data sets, the same data mining techniques are applied to the social network metrics data set.

The additional basic count metrics, including the number of work items and the number of change sets associated with a builds' work items, provide an additional size measurement that is directly related to a builds' communication network. While these two metrics are not derived directly from the network itself they provide an additional summary about the amount of work the network is associated with.

3.6.2 Social Network Metric Aggregations

Similar to software metrics, social network metrics can be captured over time. In Jazz each work item comment has a date and timestamp. To generate better understandings about how social metrics evolve over time this phase of experiments introduces a build related time interval that is used as input for building prediction models. The communication metrics are extracted from builds via time intervals (to generate four social metric data sets). More specifically the social metric data sets are constructed by

selecting the first 25%, 50% or 75% and 100% of the communication that occurred since the previous build. This is a similar approach explored by Wolf et al. (2009) which also utilises the Jazz repository. An illustration of this approach is shown in Figure 17.

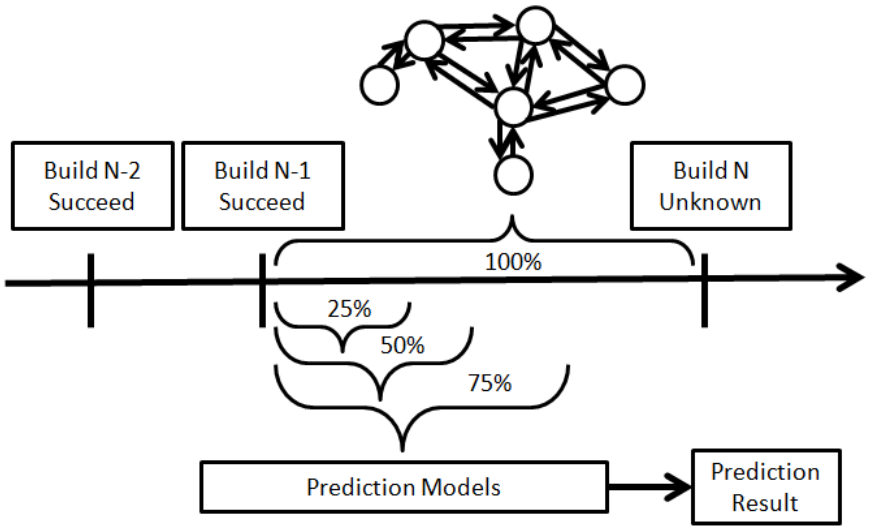

Figure 17 Social Metrics Instance Samples (Wolf et al., 2009)

Once this data set is pre-processed the Subset Evaluator and Information Gain feature selection methods are again used to filter the social network metrics from insignificant predictors (the "no feature selection" is also applied as a benchmark control). Then the j48 classifier, Naive Bayes and Bayesian Network methods are applied using 10 cross-fold validation.

SMOTE may still have potential for improving classification accuracies and sensitivities. For this the data set is broken down into a training and test set, again using the same technique as previously 70% of social network instances will compose the training set (reflecting the naturally occurring ratio of successful and failed builds) and the remaining 30% will be used as the test set.

Propagating social networks that originally exist at work item level, to build level will also prove to be useful for the next stages of this phase. During this next sub-stage the best performing software metric data sets (before and after states) and the social network metric data set are combined. The number of features will increase to 57, as there will be 42 software metrics with the addition of 14 social network metrics plus the classifier (build result). The consequence of this is not an ideal scenario from a data

mining perspective, as the ratio of features to instances is not favourable. However, feature selection methods are again applied to reduce the numbers of predictors of the feature space. This process will also determine if significant predictors are selected from a combination of social network and software metrics.

3.7 Mining time-changing data streams

In order to explore the time-series component of the data a combination of WEKA and MOA APIs are used. The software metrics and social metrics data sets are presented as individual cached instance streams. Instances within the stream are sorted via the date and time of the software build started (oldest to newest) to simulate a software project build process. Using the Hoeffding tree, the model is trained using the first 20 instances. The remaining instances are used for both prediction and additional training. For each instance predicted results are measured again in terms of their overall accuracy and sensitivity. A concept drift detection method called ADaptive sliding WINdow (ADWIN) (also available via the MOA tool) is also utilised to detect and estimate degrees of change within the metrics. Used in conjunction with the Hoeffding tree, drift detection keeps an adaptive list of recently seen instances and measures the overall change in the average values of recently seen items. This is of particular interest for this research as it can be used to detect significant metric values that change over time in less predictable ways. This introduces an additional measurement criteria including the total number of drifts that occur over time, variance, estimation and width(T) values.

In addition to this a linear k-NN method using Euclidean distance is also adopted to predict build outcomes within the time-series based framework. For this research the k-NN offers an additional and separate experimental setup to the Hoeffding tree. The k-NN approach is not used strictly in a data stream mining sense as it requires full access to historical data to perform classification. However, each instance can be classified as it becomes available. In this case the classifier works at instance level rather than group level where each instance will have its own unique blend of software metric bias and communication metric bias. Weka's API offers support for this approach. A Java application has been implemented which utilises this for processing instances. To calculate probability: p1(correct prediction | k nearest neighbours of training instances). Training instances are used to search for near neighbouring instances for each test instance. The nearest neighbouring instances have the correct and known class classification. A class label is then assigned based from the nearest neighbours' overall

probability. For example each nearest neighbours' outcome is counted, then the class label from is assigned based on which outcome is the most common amongst the neighbouring nodes. Due to the limited number of instances 5 nearest neighbours (where k=5) of the training instances are searched. For this research the k-NN approach is defined by the following pseudo-code:

```
kNearestNeighbours(int k, Instance testInstance, Instances trainingInstances){
    Foreach (trainingInstance in trainingInstances){
        o Calculate the Euclidean distance of the test vector to the current
          training vector
        o Map training point if it is within k nearest neighbours
    }
    Label = majorityVote(training map of k)
}
```

Where “k” is the number of nearest neighbours, “testInstance” is the latest instance that has become available for classification; “trainingInstances” is the number of historical instances that have been recorded (to be classified against). The “Label” is the classification outcome based on a “vote” of nearest neighbours (using their actual labelled values). Once the latest test instance has been classified it is then added and stored with the training instances.

The Hoeffding tree, ADWIN and k-NN methods will be applied to the optimal performing (most accurately modelled) software metrics data set found from the previous sets of experiments and will also be applied the communication metrics data set.

The SMOTE method is also applied to perform a what-if analysis to see what might occur if data stream mining ran against a larger data set of software and social network metrics. For the simulation to be realistic the naturally occurring distribution of successful and failed build instances is maintained. To do this SMOTE is applied twice to each data set increasing the number of instances by 900% per application. In the first instance SMOTE will increase the instances of the minority class (failed builds). In the second instance the minority class is changed (to successful builds) so SMOTE then

increases the number of instances of other class. Although this is not the way SMOTE is used in traditional data mining, this has the potential to provide additional insights into how important data sample size is for stream mining software and social network metrics. Furthermore SMOTE generates values via interpolation, rather than extrapolation, therefore the new instances remain relevant to the context of software builds for the Jazz project.

3.8 Chapter Summary

This chapter has presented the overall research methodology including details of the experimental methods and tools used for this research project. There are three major experimental phases that were designed to test the research question constructed to find which combinations of software and social network metrics provide the best predictors for a software builds' success within the Jazz project. The extraction process, metric aggregations interpretations, data mining methods and evaluation criteria have also been detailed. In the next chapter the results of the data mining experiments are presented and analysed.

4 Experimental Results

Chapter 3 described the methodology used to define and extract the data required to complete four experimental phases, namely an initial exploratory mining of software metrics, boosting mining performance, combining software and social network metrics and finally demonstrating how the emergent models can be deployed in practice. The objective of the initial mining stage is to gain insights into which combination of feature selection, mining algorithms and software metrics works best for classifying successful and failed builds. This initial phase is required to narrow down an extensive search space into a smaller number of feasible avenues of investigation. The objective of the second stage is to introduce different ways to boost classifier performance through exploration of novel feature selection tactics and adoption of SMOTE. The objective of the third stage is to see if the models developed using software metrics can be combined with social metric data to provide better insight into the classification of software builds. This analysis is undertaken using the best performing methods from previous experimental phases.

Initial analysis of the Jazz repository indicated that it consisted of approximately 360 builds. However, from these builds software metrics could only be extracted for 199 build instances change sets, for the before and after states. This is because there is a limit defined by the Jazz team to only keep the latest 200 builds. Build instances were extracted from 28th June, 2007 to 16th June 2008. From these instances there are 127 successful builds and 72 failed builds. Early explorative research utilised software metrics from continuous builds to construct the metrics data set, however in doing so there were more instances of successful builds than failed builds. In order to balance the data set, failed builds were injected from nightly, integration and connector builds. This option was preferred over removing successful builds from the data set, as it reduced the possibility of model over-fitting by having too small a data set. Software metrics instances were derived from 15 nightly builds (incorporating changes from the local site), 34 integration builds (integrating components from remote sites), 143 continuous builds (regular user builds) and 7 connector Jazz builds. These builds were included in the data set whether or not they had associated work items. This increased the number of build instances for each data set to 200. This presents a situation where the number of

features is very much less than the number of instances available for analysis, which is a far from desirable scenario from a data mining perspective. To increase the number of instances another possible solution is to include more builds from additional snapshots of the repository, but more data was not forthcoming from IBM at the time that the research was executed. Various strategies for reducing the number of metrics are used to classify the relative number of builds in the data set are investigated, to combat the number of features to instances ratio. The Weka machine learning (Hall, Frank, Holmes & Pfahringer B., 2009) workbench is used for all data mining experiments.

In terms of data pre-processing, build instances that have a warning build result have been classified as failed build instances. During initial explorations of the software metrics it was found that after implementing the classification model (the resulting j48 decision tree), that was built from using only successful and failed build instances, all warning builds instances were classified as failed instances. Warning builds as failed builds also increase the number of the minority instances which will benefit the mining processes in potentially detecting interesting patterns. This is a different approach than that used by Wolf, Schroeter, Damian and Nguyen (2009), where warning builds were treated as successful builds. In their study it is assumed that warning builds required no further actions from developers. Though it is not discussed in this thesis, an initial classification model was created excluding the warning builds and this model was used to attempt to classify the warning builds as either successful or failed. The outcome of all builds being classified as failed builds supports the decision to override the classification and assign the warning builds to the failed class. It can also argued that a warning is a bug in waiting (Spinellis, 2006). While warnings may be acceptable for debugging build types, for release builds it is better practice to treat warning builds as failed builds as some of them may indicated potential problems (Miller, 2008; Subramaniam & Hunt, 2006).

For each data set if a build had source code regardless to the connection to work items or other objects within the repository metrics are extracted and added as an instance. Any instances of a build that had no software metrics (missing values), for all features, are removed. This data pre-process is important to reduce the amount of noise within the data, missing data, or even misleading data that can have negative impacts on the mining and learning process. The project data that is extracted from Jazz was gathered

during the development of Jazz. As a consequence features that automatically capture project processes did not exist until later development stages of Jazz (gaps would often appear at early stages of the project data set).

For the before and after state change sets metrics of builds, Subset Evaluation (CfsSubst (SE)), Information Gain (Infogain (IG)) and Principal Components Analysis (PCA) feature selection methods are used to filter each data sets features. Then for each filtered set the selected mining algorithms (j48 Classifier (j48), Naive Bayes (NB) and Bayesian network (BN)) for this study were ran separately on each set of features for each data set using 10 cross fold validation. In addition to this the mining algorithms are also applied to the full data set without any feature selection (No FS) to serve as a benchmark for observing any increases or decreases of performance from using feature selection methods.

4.1 Initial Software Metric Data Mining Experiment Results

Mining experiments are performed on both the before and after software metric build state data sets. There are multiple dimensions to the results that need to be considered when deciding on which methods and data sets are best to use for further exploration. For this reason the results from the initial data mining are broken down into 3 sections. Each section is based on the data set, the feature selection and the mining algorithms components of this study. For each data set the ranges of correctly classified instances for all mining experiments performed, on each data set, are presented. This includes the overall percentages of correctly classified and incorrectly classified instances. These values are also broken down to show the percentage of correctly classified successful and failed build classes derived from the classifier confusion matrix to ensure that a high overall accuracy is not derived just from identifying one type of build. For each of the feature selection methods used the results present how often Subset Evaluation, Information Gain, PCA and no feature selection (benchmark) performs the best in terms of correctly classified instances. Finally for each mining algorithm adopted the experimental results are presented in terms of correctly classified instances and sensitivity ratings including true positive (TP), false positive (FP), Precision, Recall and F-Measure values for successful, failed build classes and weighted averages.

The initial mining stage requires 120 individual experiments in Weka. Broken down this is 5 data metric aggregations (Rational Analyzer Metrics, Total Metrics, Max Metrics, Median Metrics and Mean Metrics), by 4 feature selection methods (Subset Evaluation, Information Gain, PCA and No Feature Selection), by 3 classifiers (j48 Classification Tree, Naive Bayes and Bayesian network), by 2 software build states (Before and After build states).

The combination of data set, metric aggregation, feature selection method and classification method creates an experimental hypercube. The following sections present results that populate that hypercube with the goal of obtaining some insight into which slices through the hypercube offer the most potential for further study.

4.2 Before State Results

Using the software metrics derived from the before state of each build, classification experiments show that the RSA and Max data sets have performed the best in terms of their overall percentage of correctly classifying instances. Table 5 shows the minimum and maximum classification accuracies achieved from the initial 60 data mining experiments on the before state software metric data set aggregations. Results are presented to one decimal place, so some rounding has occurred. According to these initial figures it is apparent that failed build instances are more difficult to predict than successful build instances. This pattern has occurred across all data sets indicating that there is a strong overlap of success and failed builds within the feature spaces explored. The variation of correctly classified instances occurs largely due to the different types of software metrics aggregations.

Table 5 Summary of Data Mining Results for Before State Metrics Data Sets

	RSA	Total	Mean	Median	Max
% of Correctly Classified Instances	65.2 - 79.3	62.6 - 76.3	60.6 - 75.3	46.0 - 66.2	64.1 - 78.8
% of Incorrectly Classified Instances	20.7 - 34.9	23.7 - 37.4	24.8 - 40.0	33.8 - 54.0	21.2 - 35.9
% of Correctly Classified Successful Builds	80.2 - 95.2	72.2 – 100.0	48.4 - 92.1	29.4 - 100.0	88.9 - 78.6
% of Correctly Classified Failed Builds	12.5 - 70.8	9.7 - 72.2	32.0 - 82.0	0.0 - 75.0	29.2 - 69.4

4.2.1 Data set Performance

The histogram presented in Figure 18 shows the overall percentages of correctly classified instances, of successful and failed builds, per mining experiment for the before state metrics, using 10 cross fold validation. From this perspective each mining scenario is represented by the feature selection method and mining algorithm used. Figure 18 illustrates that in each data mining scenario the median and mean metric data sets have generally performed not as well as the total, max and RSA metric aggregations. For this set of experiments the max data set has on average performed the best across all methods in terms of generating the best percentages of correctly classified instances for 6 out of the 12 scenarios explored. This occurs when the j48 classifier and Naive Bayes methods are used. The RSA software metrics data set has also performed well, generating the best classification accuracies instances for 4 out of the 12 experiments.

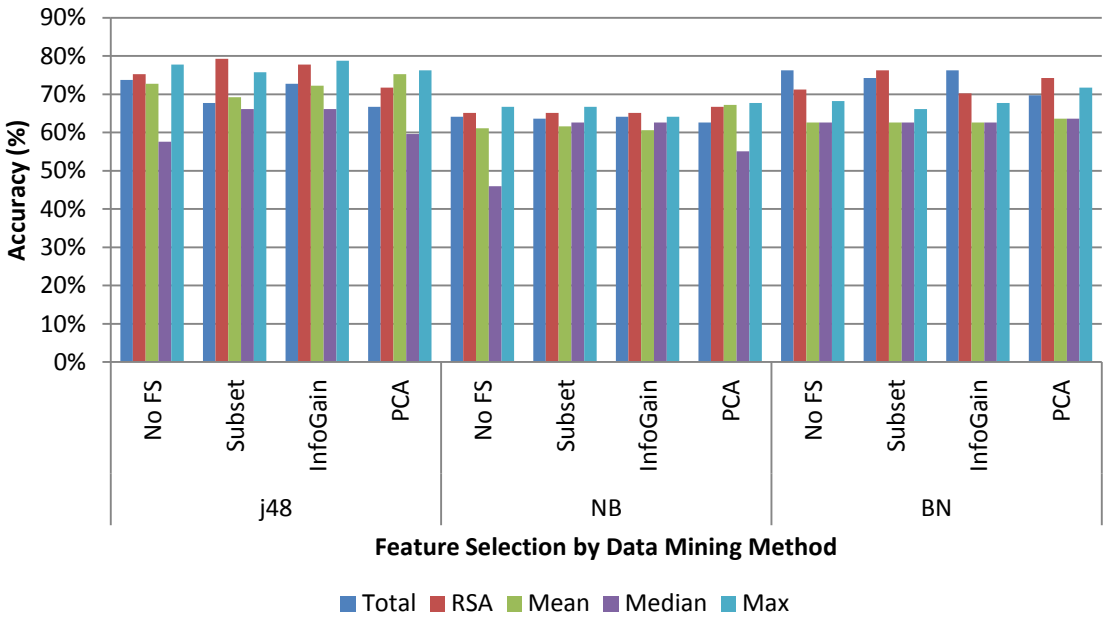

Figure 18 Before State Mining Results by Data Set

From the RSA metrics data set the best result was produced from the metrics selected via Subset Evaluation. Using the j48 tree 79.3% of builds was correctly classified and 70.8% of failed builds were also correctly classified. Using the Bayesian network 76.3% of builds were correctly classified, however only 52.8% of failed builds were correctly classified. From the total metrics data set the best results, in terms of correctly classified instances, were obtained using a Bayesian network with no feature selection and Information Gain. Both experiments were able to correctly classify 76.3% of build instances. With no feature selection the number of correctly classified failed instances was slightly higher (at 72.2%) using the Bayesian network than the Information Gain (at 69.4%).

For the max metrics data set Information Gain produced the best percentage of correctly classified instance at 78.8%. This result was only slightly higher than the 77.8% generated via no feature selection method. With Information Gain applied the number of correctly classified failed builds was at 65.3%, whereas with no features selection this value was minutely increased to 66.7%.

After running the three (j48, Naive Bayes and Bayesian Network) mining algorithms without any feature selection, on each data set, the max metric data set performed the best, with correctly classified instances at 77.8% using the j48 classifier. However, only

66.7% of failed builds were correctly classified. The max data set, with no feature selection methods applied produced the best number of correctly classified instances 2 out of 3 times when compared to the other data sets mining results (without feature selection). In addition to this the max data set generated the highest number of correctly classified instances using the j48 tree and Naive Bayes methods.

4.2.2 Feature Selection Performance

Of particular interest are the results of applying the feature selection algorithms from Weka, as these selection strategies are based around finding significant impact arising in the data. This differs from the more heuristic based filtering approaches that are based on the classification of the metrics rather than arising from the data. A number of the available metrics are selected when applying both the Infogain and CfsSubset algorithms, possibly indicating that these are stronger indicators of build failure. It is observed that the number of correctly classified failed builds heavily varies from experiment to experiment, when changing both the feature selection method and data mining method, on the same data set.

When the Subset Evaluation is applied to each data set it was found that the RSA data set produced the best result using the j48 classifier with 79.3% correctly classified instances. In addition to this 70.8% of failed builds were also correctly classified. The RSA data set produced the best results in terms of correctly classified instances when using the j48 classification tree and Bayesian network methods when compared to all other data sets. From the Bayesian network the rational analyzer data set generated 76.3% overall accuracy. However, only 52.8% of failed builds were correctly classified. From applying Information Gain the max data set produced the highest number of correctly classified instances at 78.8% using the j48 tree. However, in this case the max data set did not produce the best results when using other mining methods. When running Naive Bayes the best performing data set with Information Gain feature selection was from the RSA with an overall accuracy of 65.2%. From the Bayesian network method the best performing data set, with Information Gain feature selection applied, was from the total metrics data set with an overall accuracy of 76.3%.

When PCA is applied to each data set the max data set produced the best results when running both the j48 classification tree (76.3% overall accuracy with 63.9% of correctly classified failed builds) and the Naive Bayes methods (67.68% overall accuracy with

30.6% of correctly classified failed builds). For the Bayesian network the RSA data set produced the best result with an overall accuracy of 74.2% with 55.6% of failed builds correctly classified. Figure 19 shows the percentage of correctly classified instances by feature selection method for the before state metrics. From this perspective each mining scenario is represented by its mining algorithm and data set used. It is illustrated that for 7 out of the 15 feature selection scenarios PCA has proven to be the most effective, particularly when using Bayesian Network and Naive Bayes methods. There are also 4 cases where feature selection methods have tied with each other producing identical levels of accuracy. This can be observed when looking at No FS (No Feature Selection) and the Infogain (Information Gain) and the Subset (Subset Evaluation) and on the RSA, mean and median data sets in several of the scenarios. In a few instances it is also observed that data mining results have not improved with addition of feature selection methods. For example there are cases where no feature selection has outperformed Subset Evaluation (SE), InfoGain or PCA. This indicates that these feature selection methods may not be crucial for improving accuracy in the context of predicting software build failure for this repository instance. However, the role of feature selection itself is to not only to improve classifier performance but also to improve model comprehensibility and run time performance. In this respect feature selection methods have a significant impact on the final model.

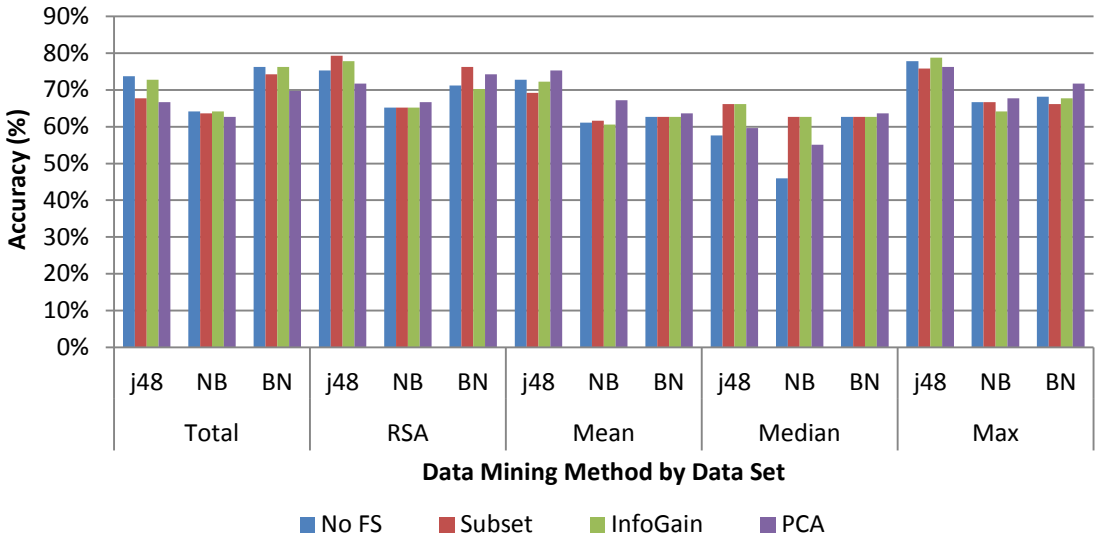

Figure 19 Before State Mining Results by Feature Selection

These results indicate that there is no single best solution when it comes to choosing a feature selection method for mining various software metric aggregations. However, it does show which data sets are worthwhile for further exploration. From the 4 feature

selection methods and 3 mining algorithms the max data set produced the best results 6 out of 12 times. The RSA produced the best result 4 out of 12. The Total data set produced the best result for 2 out of 12 mining experiments.

4.2.3 Classifier Performance

For each mining activity the TP, FP, Precision, Recall, ROC and F-Measure are measured for both successful and failed build classifiers. In addition to this a weighted average is also measured. All these criteria range from 0 to 1. When values are closer to one, this indicates better performance. Ranges of classification accuracy are provided for each mining algorithm adopted and summarised in Table 6.

Table 6 Summary of Data Mining Results for Classifiers on Before State Metrics

Mining algorithm	j48	Naive Bayes	Bayesian Network
Correctly Classified Instances	57.6% - 79.3%	46.0% - 67.7%	62.6% - 76.3%
Incorrectly Classified Instances	20.7% - 42.4%.	32.3% - 54.0%	23.7% - 37.4%

Figure 20 illustrates the percentage of correctly classified instances by data mining algorithm for the before state metrics. Each mining scenario is represented by its data set and feature selection method. From this perspective it is observed that the Naive Bayes method has performed the worst out of the three mining algorithms explored, in terms of generating highest percentages of correctly classified instances. The best performing algorithm in this context is the j48 classifier, where it generated the highest result for 13 out of 20 mining scenarios. This is then followed by the Bayesian network, generating highest correctly classified instances for 7 out of the 20 scenarios.

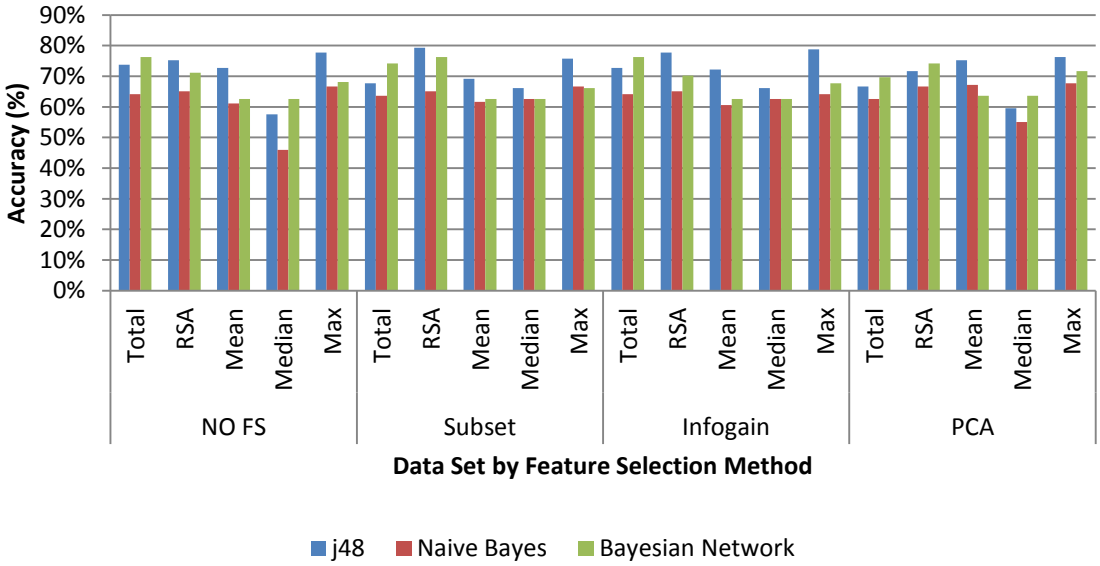

Figure 20 Before State Mining Results by Mining Algorithm

4.2.4 Best Performing Models for the Before State Metrics

This section presents the top performing models from the mining experiments of the before state metrics from phase 1. Models of particular interest will contain high levels of overall classification accuracy and high levels of accuracy for predicting failed builds. Figure 21 presents the overall classification accuracy and failed builds classification accuracy for 60 data mining experiments that were based from the range of data set aggregations, feature selection methods and data mining methods for the before state software metrics. For the full set of results for this section refer to Appendix C: Before State Software Metrics Results.

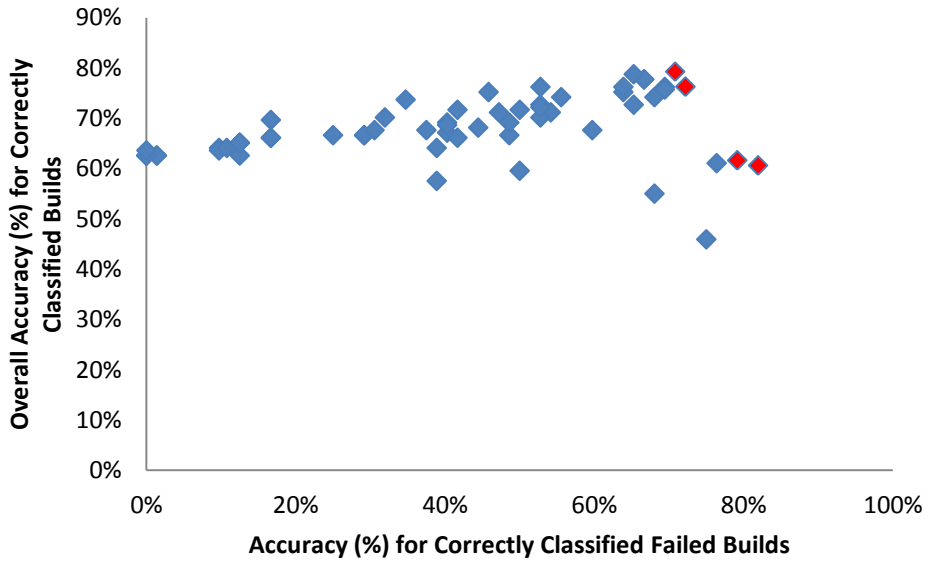

Figure 21 Before State Results: Overall Classification Accuracy of Builds Versus Classification for Failed Builds

Figure 21 allows a simple visualisation of the outcomes of the experimentation. The nature of the experimentation is to find the models that best satisfy the goals of maximising overall accuracy as well as maximising the accuracy on classifying failed builds. This can be considered a multi-objective optimisation problem and the builds highlighted in Figure 21 represent the pareto-optimal set of non-dominated solutions. It is not possible to say that any one of these models is "better" than the rest as each presents a unique solution that performs better in one objective than any of the others.

When maximizing the overall classification accuracy and the accuracy of classifying failed builds, the first model from the pareto-optimal set has been taken from the RSA data set, when the Subset Evaluation feature selection was applied. This model correctly classified 79.3% of the provided instances and is detailed in Table 7. Out of the data mining algorithms the j48 classification tree performed the best. The second model from the pareto-optimal set was derived from using the max data set using Information Gain and the j48 classification tree. This model correctly classified 78.8% of the instances and is presented in Table 8. The third model from the pareto-optimal set was also taken from the max metric data set, with no feature selection, using the j48 classification tree. This model correctly classified 77.8% of the provided instances and is presented in Table 9. From all the best performing models, at this stage, the j48 classification tree has performed generally better than the Naive Bayes and Bayesian

Network methods. It is observed that the Bayesian Network has outperformed Naive Bayes method. This appears to be the current trend across all data sets for this stage. The pareto-optimal set (presented in Table 10) was obtained from using the total sum data set, with no feature selection filter and the Bayesian Network classifier. From this experiment 76.3% of instances were correctly classified and out of those, 72.2% of failed build instance were correctly classified.

Figure 22 presents the decision tree (via j48) using the RSA Data Set with Subset Evaluation filter applied and the classification accuracy and sensitivity values are presented in Table 7. Most of the metrics within the tree are essentially measures of size rather than complexity. Starting at the top of the tree, various branches are followed depending on specific metric values. Once an endpoint is reached a prediction has been made. End nodes (the predicted value) that are separated by a '/' indicate the observed total number of builds that have been classified and the second value (if present) is the number of builds that have been misclassified. For example if a node has [Successful build 5.0/1.0] then 5 builds in total have been classified as a success and 1 build has been incorrectly classified as a success.

In this model the number of types per package, at the top of the tree, is a strong predictor. If the number of types per package is below 27 the build is classified as successful. It is observed that a degree of confusion is present within this tree and rather appearing near the bottom leaf nodes, it appears near the top of the tree. For instance there is a duplicate of the number of unique operators metric that appears on the higher branches (right hand side) of the tree. As a decision tree grows in size, the chance of over-fitting the data increases. Each split within the tree is a representation of a subset of rules from the previous level. This combined with the degree of confusion nodes makes it difficult to create generalisations for classifying failed builds. Ideally the majority of failed builds will appear within higher levels of the tree.

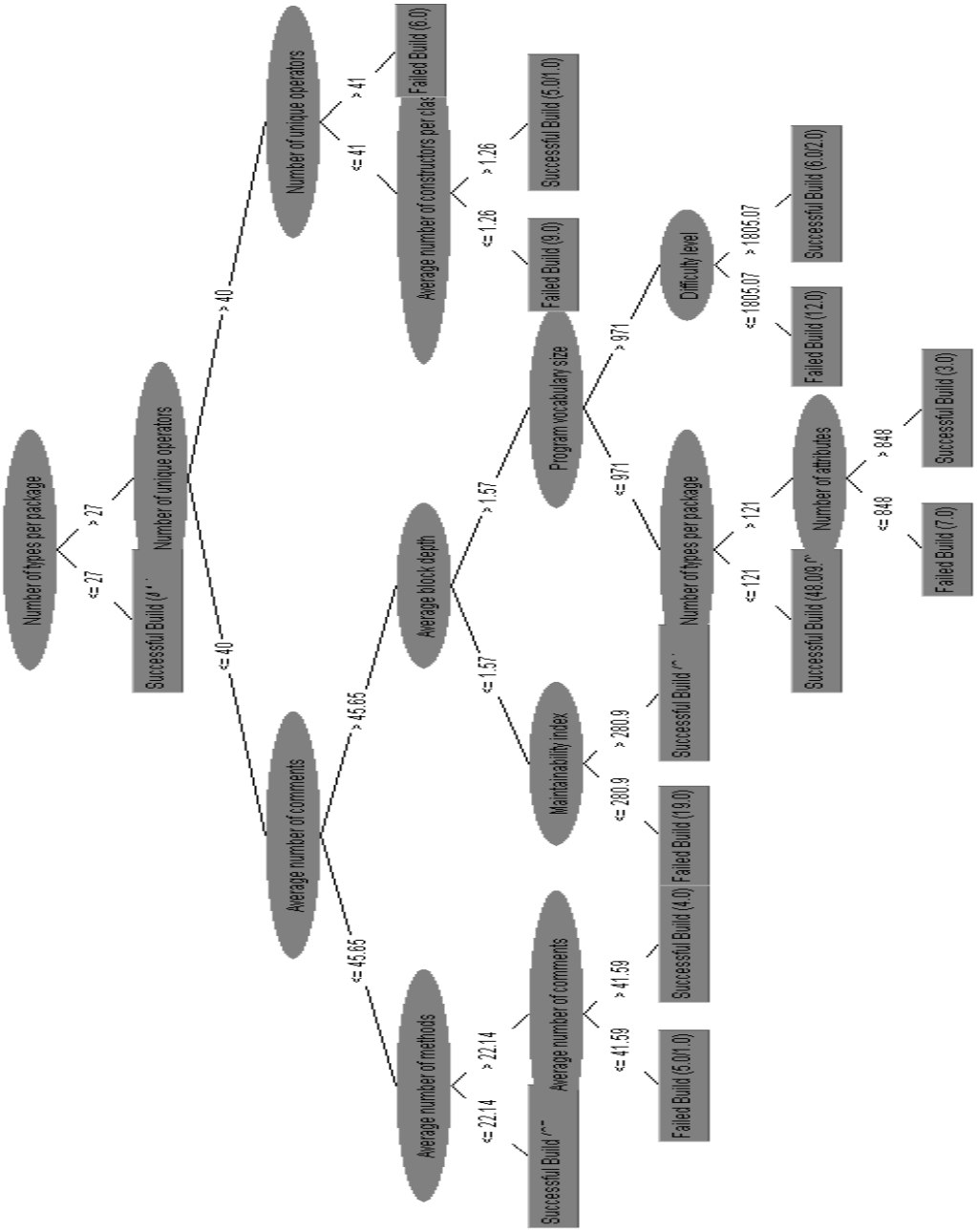

Figure 22 RSA Before State (CfsSubst and j48)

From this experiment approximately 71% of failed builds are correctly classified, whereas approximately 84% of successful builds are correctly classified. It is also observed that the sensitivity measurements were slightly better for successful builds than failed builds.

Table 7 Results from RSA Data Set Using Subset Evaluation and j48 Classification

# Successful builds correct (incorrect)		**# Failed builds correct (incorrect)**		**Correctly Classified Instances**	**Incorrectly Classified Instances**
106 (20)		51 (21)		79.3%	20.7%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.841	0.292	0.835	0.841	0.838
Failed Build	0.708	0.159	0.718	0.708	0.713
Weighted Average	0.793	0.243	0.792	0.793	0.793

The second best result obtained at this stage was with the max data set, using Information Gain as a filter and the j48 mining algorithm. The decision tree for this result is presented in Figure 23. The classification accuracies and sensitivity values are presented in Table 8. It is observed that the comment/code ratio is a significant metric for predicting build success and failure. In this tree failure is predicted in nodes which are placed in higher levels of the tree. For example if the comment/code ratio metric is >115.56 then out of the 20 builds classified as failure, only three have been misclassified. Another strong indicator of build failure appears to involve not only comment/code ratio, but also maintainability index, average lines of code per method and afferent coupling, where 10 failed builds were correctly classified. Again, within this tree, there is also a degree of confusion with rules which are indicated by the duplicate nodes. This is particularly noticeable with average number of methods and average lines of code per method metrics.

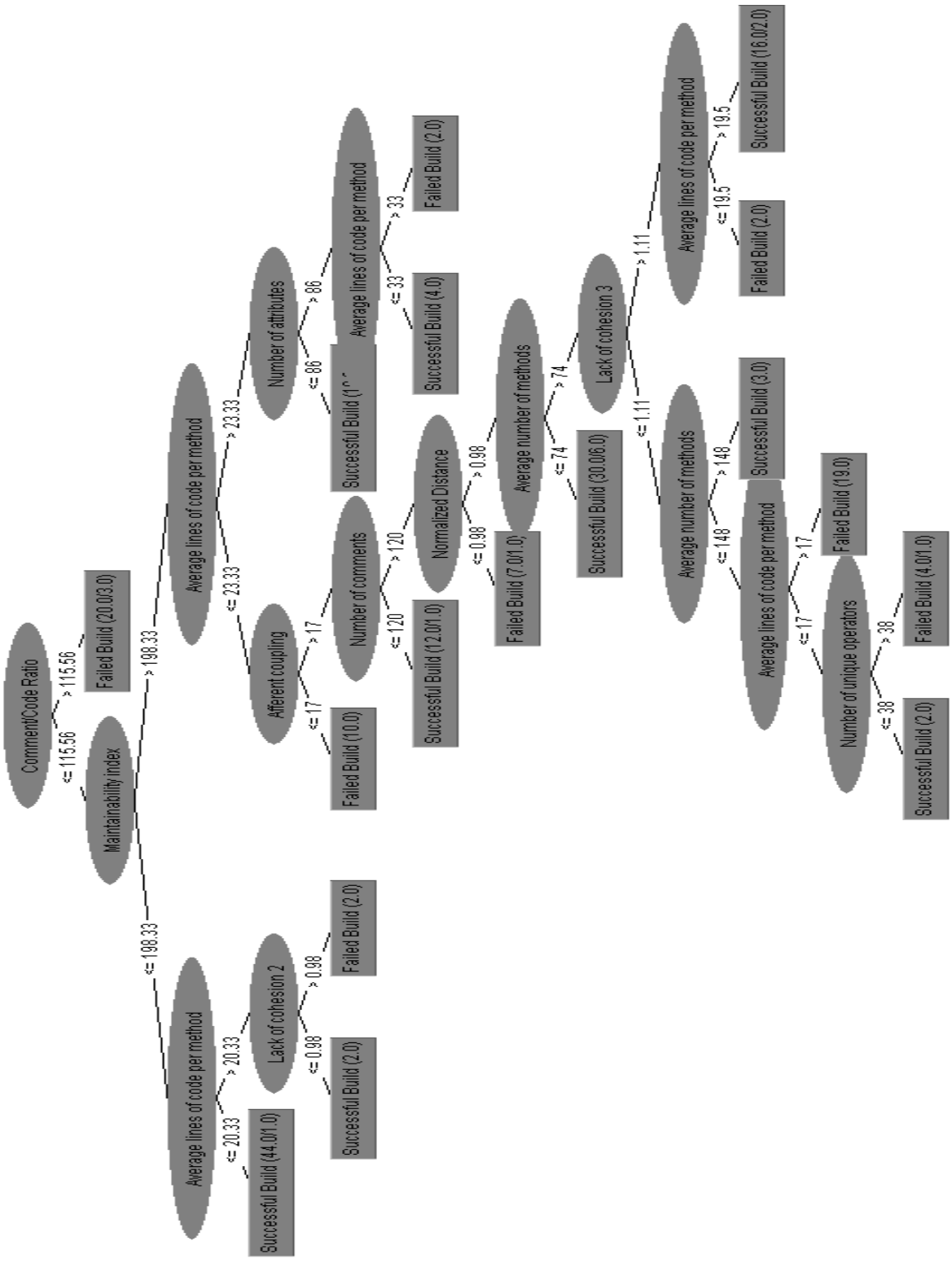

Figure 23 Max Before State (InfoGain and j48)

Table 8 Results for the Max Data set using Information Gain and j48 Classification

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
109 (17)		47 (25)		78.8%	21.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.865	0.347	0.813	0.865	0.838
Failed Build	0.653	0.135	0.734	0.653	0.691
Weighted Average	0.788	0.27	0.785	0.788	0.785

With the max data set a maximum value is calculated for all source code files. This means that the metric value used to represent the *after* state of the code may relate to an entirely different source code file than gave rise to the maximum value for the *before* state. Therefore the above classification can at best be used to interpret trends. Despite this, examination of the tree can provide some insight into what may be occurring during the build cycle. The top node of the tree classifies build instances on the basis of increasing size (as represented by the comment/code ratio). For instance both Figure 23 and Figure 24 decision trees show that when the size of the comment/code ratio increases there is a higher probability of build failure.

Figure 23 is the final decision tree for the results presented in Table 9. The classification accuracies presented in Table 9 shows that approximately 84% of successful builds are correctly classified, whereas only 66.7% of failed builds are correctly classified, for the max before state data set. This result is also reflected by the sensitivity measurements with the average of 0.74 for successful builds and 0.60 for failed builds. Presented in Table 10 are the classification accuracies for the decision tree Figure 24, where 78.5% of successful builds are correctly classified and 72.2% of failed builds are correctly classified.

Table 9 Max Before State (No Feature Selection and j48)

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
106 (20)		48 (24)		77.8%	22.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.841	0.333	0.815	0.841	0.828
Failed Build	0.667	0.159	0.706	0.667	0.686
Weighted Average	0.778	0.27	0.776	0.778	0.776

In Figure 24 even though there is no feature selection applied it is observed that comment/code ratio is a strong indicator of build success or failure. Inspection of the classification tree indicates that generally the first few nodes are intuitive. When navigating through deeper levels of the tree repeated metrics are observed and indicates some confusion found within the classification (e.g. Lack of cohesion 3). Resolution of this uncertainty requires further research, however it may be related to the use of the maximum metric values in the data set. The maximum value may potentially obscure results.

Initial insights indicate that elements within data are highly dependent and the objects that are being explored should not be considered independent of each other. In addition to this there is much variability between the models generated, again, this may be largely due to the various software metric aggregations that have been explored. This aspect will be explored further in the discussion section of this thesis.

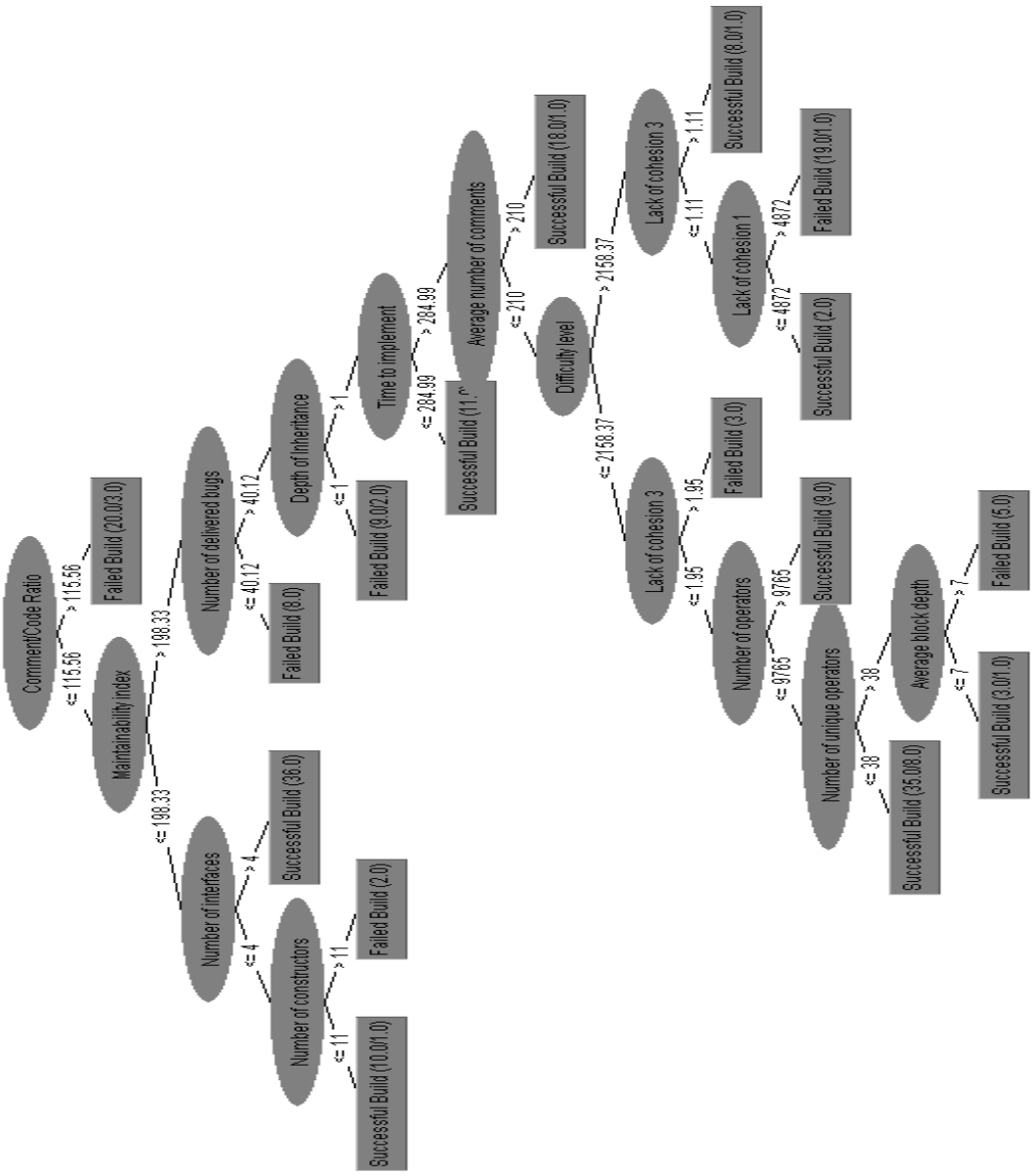

Figure 24 Max Before State (No Feature Selection and j48)

Table 10 Total Before State (No Feature Selection and Bayesian Network)

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
99 (27)		52 (20)		76.3%	23.7%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.786	0.278	0.832	0.786	0.808
Failed Build	0.722	0.214	0.658	0.722	0.689
Weighted Average	0.763	0.255	0.769	0.763	0.765

4.3 After State Results

For each build the after state metrics are also examined. It is found that the after state results show similarities to the before state metrics, in terms of the features selected and classification accuracies. The mining results of the after state have also shown that in most cases failure is more difficult to predict than success. The number of correctly classified failed builds varies from experiment to experiment on the same data set from using the range feature selection and mining methods. The max and RSA software metric data sets, again, outperformed the total mean and median data aggregations. The summary of the ranges of accuracy and sensitivity for each data set is presented in Table 11. It is observed that the experiments performed on the total data set generated the same range of overall accuracies from the before state. For this reason, as well as it not performing as well as the RSA and Max data sets, the total data set is not included in the further experimental phases.

Table 11 Phase 1 Data Mining Results for After State Metrics

	RSA	Total	Mean	Median	Max
% of Correctly Classified Instances	62.8 - 80.4	62.8- 76.3	56.3 - 75.9	52.8 - 63.8	62.6 - 79.8
% of Incorrectly Classified Instances	19.6 - 37.2	23.7 - 37.4	24.1 - 43.7	36.2 - 47.2	20.2 - 37.4
% of Correctly Classified Successful Builds	80.3 - 93.7	74.0 - 100.0	45.7 - 93.7	29.4 – 100.0	81.1 - 94.5
% of Correctly Classified Failed Builds	15.3 - 66.7	6.9 - 72.2	12.5- 75.0	0 .0- 70.7	19.7 - 69.0

4.3.1 Data set Performance

From the RSA metrics data set the best result was produced using Subset Evaluation feature selection (reflecting similar results to the before state). Using the j48 tree 80.4% of builds are correctly classified. However, from this outcome, only 34.7% of failed builds are correctly classified. For the max metrics data set produced the best percentage of correctly classified instance at 79.8% when no feature selection method is applied and using the j48 classification tree. This result was only slightly higher than the 78.3% generated via the PCA method. With no features selection applied the number of correctly classified failed builds was at 69.0% and with PCA this value was lower with 59.2% accuracy. In terms of overall accuracy the RSA and max data sets continue to perform the best out of all metric aggregations, reflecting the same pattern as the before state metric results.

For the total metrics data set the best results produced in terms of correctly classified instances were obtained using no feature selection using the Bayesian network with 76.9% of build instances correctly classified. The number of correctly classified failed

build instances is at 72.2%. With Information Gain the number of correctly classified failed instances was slightly lower at 76.4% using the Bayesian network method and the number of correctly classified failed build instances at 70.8%.

The mean metrics data set generated the best number of correctly classified instances when using PCA as feature selection with a result of 75.9%. Similarly to the before state, when Subset Evaluation and Information Gain was applied to the median metrics data set, the same set of features were selected. As a result the mining results are identical for these two feature selection sets. Again, these results indicate that there is no single "best" solution when it comes to choosing a feature selection method.

Figure 25 illustrates the percentage of correctly classified instances by data set for the after state metrics. Each mining scenario is represented by a feature selection method and mining algorithm. Again, similar to the before state the max data set generated the highest number of correctly classified instances for 5 out of 12 scenarios. This is then, again, followed by the RSA data with best classification percentages in 4 out of 12 scenarios. The same pattern is also detected where the median and mean data sets did not provide valuable evidence in predicting software build success or failure.

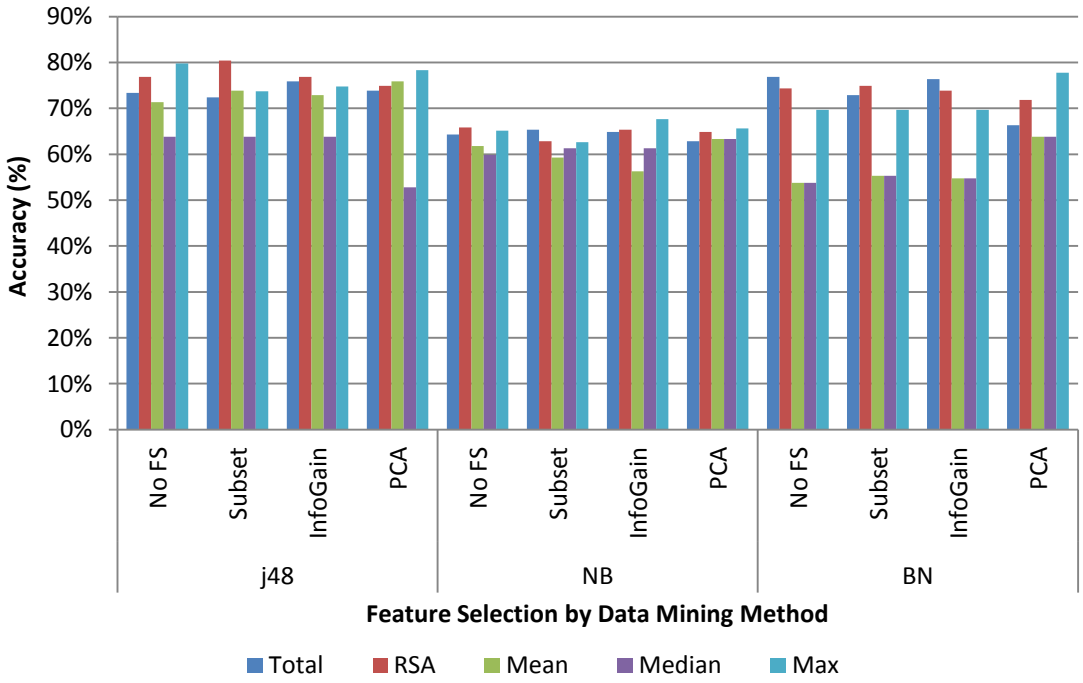

Figure 25 After State Mining Results by Data Set

4.3.2 Feature Selection Performance

After running the three selected mining algorithms without any feature selection, on each data set, the max metric data set performed the best, with correctly classified instances at 79.8% using the j48 classifier with 69.0% of failed builds correctly classified. When the Subset Evaluation is applied to each data set it was found that the RSA data set produced the best result using the j48 classifier with 80.4% correctly classified instances. However, only 34.7% of failed builds were correctly classified, which is very low.

Similar to the before state data set, the rational analyzer produced the best results in terms of correctly classified instances when using the j48 classification tree and Bayesian network methods when compared to all other data sets. From the Bayesian network the rational analyzer data set generated 74.9% overall accuracy with 64.0% of failed builds correctly classified. From applying Information Gain as a feature selection method the RSA data set produced the highest number of correctly classified instances at 76.9% using the j48 tree. However, in this case the max data set did not produce the best results when using other mining methods. When running Naive Bayes the best performing data set with Information Gain feature selection were the max metrics with an overall accuracy of 67.7%. From the Bayesian network method the best performing data set, with Information Gain feature selection applied, was from the total metrics data set with an overall accuracy of 76.4%. When PCA is applied to each data set the max data set produced the best results when running both the j48 classification tree (78.3% overall accuracy with 59.2% of correctly classified failed builds) and the Naive Bayes methods (65.7% overall accuracy with 25.4% of correctly classified failed builds). For the Bayesian network, again, the max data set produced the best result with an overall accuracy of 77.8%, however only 47.9% of failed builds were correctly classified.

Figure 26 illustrates the percentage of correctly classified instances by feature selection for the after state metrics. Each mining scenario is represented by a mining algorithm and data set. Again showing similar patterns to the before state the PCA method gave the best scenarios for generating the highest number of correctly classified instances in 6 out of 15 scenarios. This is primarily observed from the mean, median and max data sets using a combination of all mining algorithms explored. The second best feature

selection methods were derived from Subset Evaluation and when no feature selection was applied.

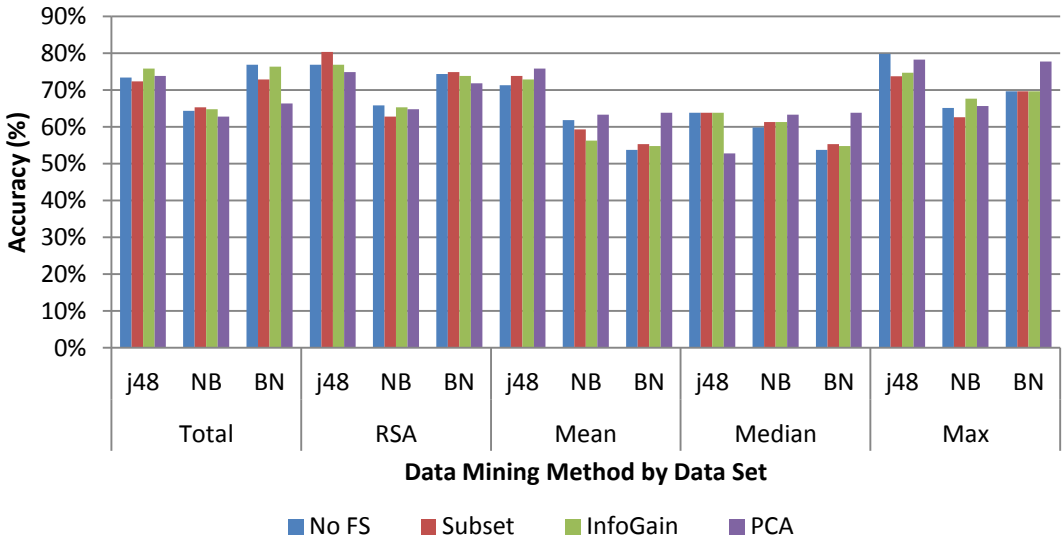

Figure 26 After State Mining Results by Feature Selection

It is observed that the Information Gain method classified a greater number of significant metrics than the Subset Evaluator. This is to be expected as the CfsSubset method does not assume that all metrics are dependent of each other. Instead, it looks for inter-connected relationships between metrics to identify significant associations. From initial mining of the before and after state metrics, the results indicate that there is no single best solution when it comes to choosing a feature selection method for mining various software metric aggregations. However, it does show which data sets are worthwhile for further exploration. From the 4 feature selection methods and 3 mining algorithms the max data set produced the best results 5 out of 12 times. The RSA produced the best result 4 out of 12, followed by the total data set, producing the best result 3 out of 12 times.

4.3.3 Data Mining Performance

Table 12 presents a summary of the results from the data mining methods perspective so that they can be compared for analysis. Presented are the ranges of classification accuracies. From this it is observed that the j48 classification method generates the levels of accuracy with fairly similar levels of sensitivity to other methods explored.

Table 12 After State Result Ranges by Classification Method

	J48	Naive Bayes	Bayesian network
Correctly Classified Instances	52.8% to 80.4%	56.3% - 67.7%	53.8% - 77.8%
Incorrectly Classified Instances	19.6% to 47.2%	32.3% - 43.7%	22.2% - 46.2%

Figure 27 illustrates the percentage of correctly classified instances by data mining algorithm for the after state metrics. Each mining scenario is represented by its data set and feature selection method. In relation to the before state similar patterns are observed in the after state metrics from this context. It is observed that the j48 classifier obtained the highest percentages of overall correctly classifying instances for 17 out of the 20 scenarios. This pattern is observed across all data sets and feature selection methods. This is then followed by the Bayesian Network that performed well for 3 out of 20 scenarios. These instances are shown within the total data set using Information Gain and no feature selection. In addition to this and similarly to the before state metrics, it is shown that the Naive Bayes method did not produce any best performing instances.

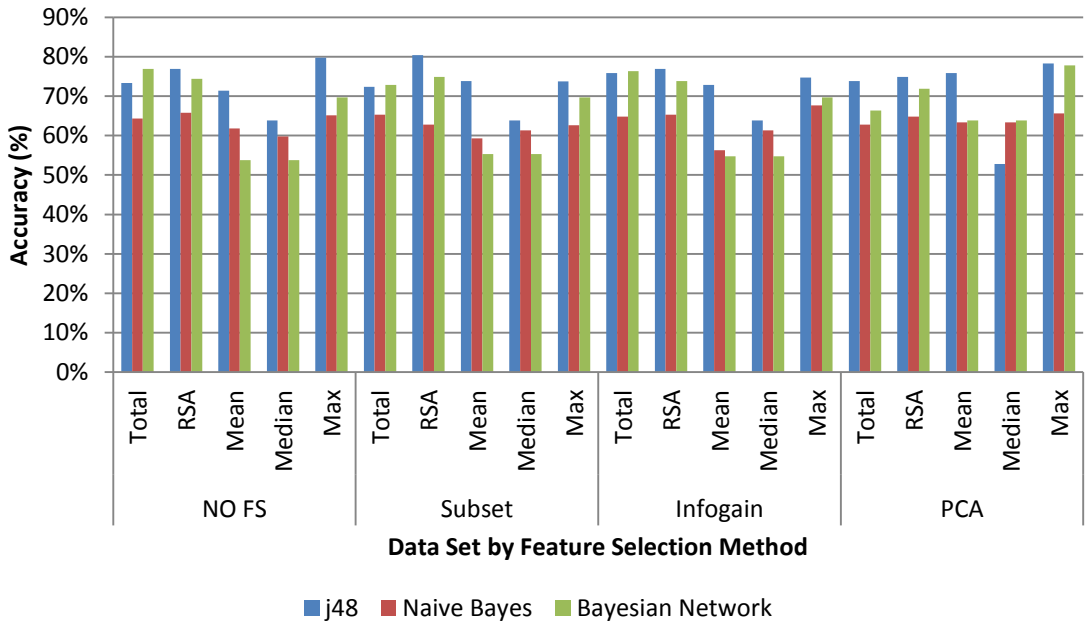

Figure 27 After State Mining Results by Mining Algorithm

4.3.4 Best Performing Models for the After State Metrics

The after state produces very similar results to the before data set with minor observed differences. Figure 28 presents the overall classification accuracy and failed builds classification accuracy for 60 data mining experiments that were based on the range of data set aggregations, feature selection methods and data mining methods for the after state software metrics. Again, models of particular interest have maximum overall classification accuracy and accuracy for predicting failed build instances and the pareto-optimal solutions are highlighted. For the full set of results for this section refer to Appendix D: After State Software Metrics Results

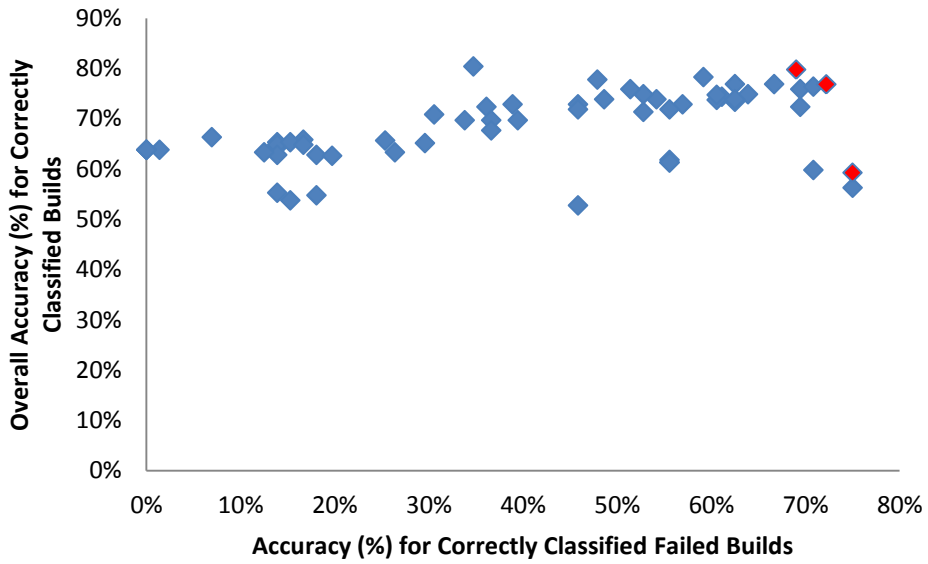

Figure 28 After State Results: Overall Classification Accuracy of Builds Versus Classification for Failed Builds

The first model from the pareto-optimal set is taken from the RSA metrics data set, using Subset Evaluation and the j48 classification tree. The best model correctly classifies 80.4% of instances and is presented in Table 13. The second best result was produced from the max data set using no feature selection and the j48 classification tree. The model correctly classified 79.8% of the instances and is presented in Table 14. The third best model, again, was from the max data set using Information Gain and the j48 classification tree. This model correctly classified 74.7% of the instances and is presented in Table 15.

From the after state results it observed that the prediction of failed builds is generally more challenging than the classification of successful builds. So far the best classification results are primarily from the RSA and max data sets.

Table 13 details the classification accuracies and sensitivity values for the RSA data set with the Subset Evaluation filter applied using j48 decision tree. The classification tree for this result is illustrated in Figure 29. Here it is observed that the number of attributes metric serves as a main predictor for classifying builds. In this result, in addition to the number of attributes metric, the weighted methods per class (successfully classifying 17 failed builds), program vocabulary size and average block depth (successfully classifying 20 failed builds) within this model are strong predictors. So far it has been noticed that size metrics, rather than complexity metrics, are having more influence on the classification of builds. Again there are nodes which add confusion to the decision tree. In this case there are duplicates of the weighted methods per class (which appears on the second level of the tree), number of unique operators metric, program vocabulary size (which appears 3 times).

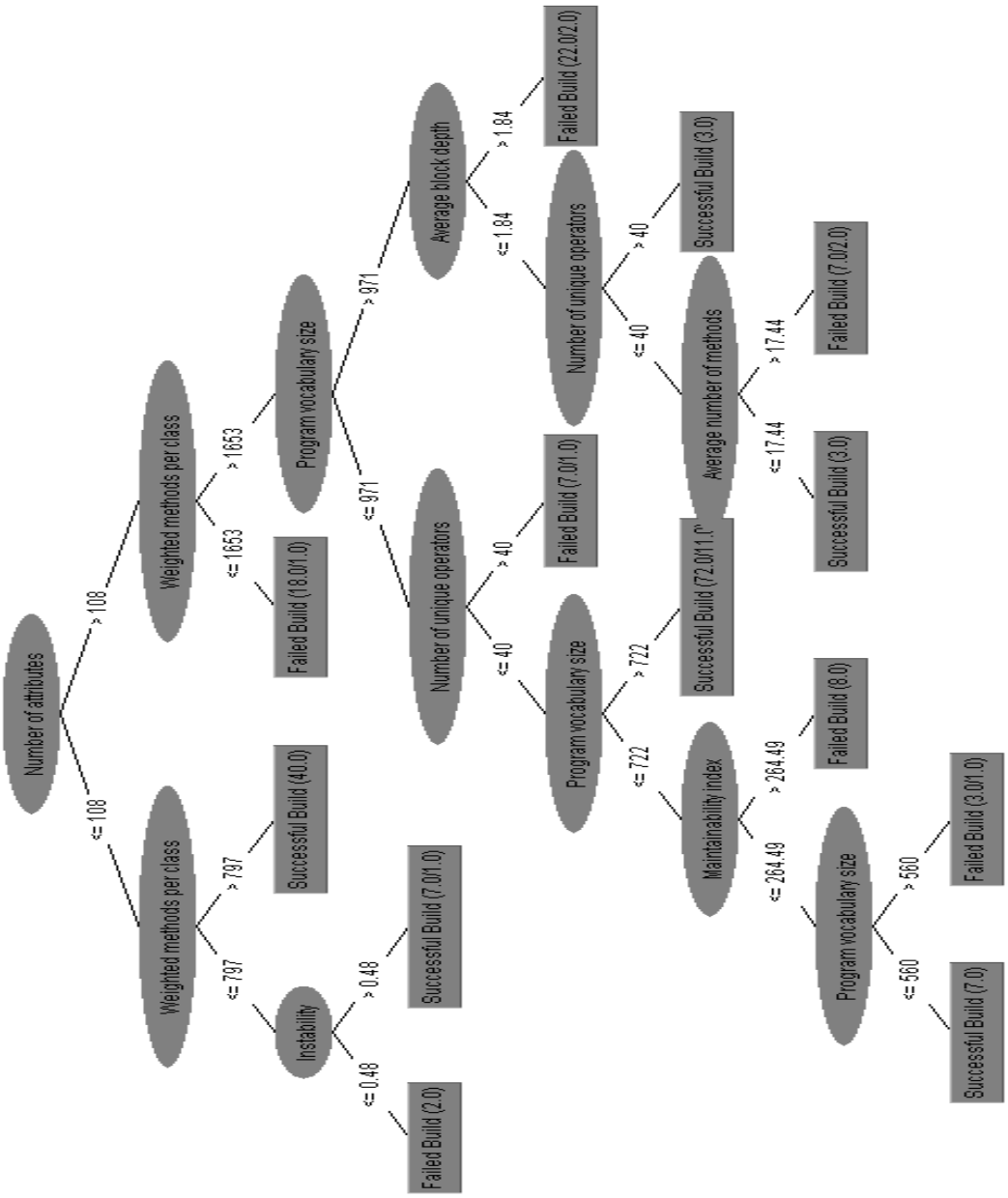

Figure 29 RSA After State (CfsSubst and j48)

This results shows that approximately 89% of successful builds are correctly classified, whereas only 65% of failed builds are correctly classified. In regards to sensitivity, the successful builds have an average rating on 0.76, whereas failed builds have an average of 0.60.

Table 13 Results for the After State of the RSA Data Set Using Subset Evaluation and the j48 Classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
113 (14)		47 (25)		80.4%	19.6%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.89	0.347	0.819	0.89	0.853
Failed Build	0.653	0.11	0.77	0.653	0.707
Weighted Average	0.804	0.261	0.801	0.804	0.8

Table 14 shows again that successful builds are easier to predict than failed builds, with approximately 86% instances correctly classified with a TP rate of 0.858. This experiment shows a slightly higher classification accuracy for failed builds with nearly 70% correctly classified instances, however the TP rate is lower at 0.69.

Table 14 Max After State (No Feature Selection and j48)

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
109(18)		49 (22)		79.8%	20.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.858	0.31	0.832	0.858	0.845
Failed Build	0.69	0.142	0.731	0.69	0.71
Weighted Average	0.798	0.25	0.796	0.798	0.797

Similar to the before state the classification tree with no feature selection was one of the best performing. In the after state however the contents of the tree are different. This is presented in Figure 30 and is based on the results presented in Table 14. Here the initial node metric is the same as the before state where the comment/code ratio, if greater than ~115, classifies a failed build. This tree however also displays elements of confusion with duplicate metrics (average number of methods and comment/code ratio) which

appears at lower levels of the tree. In addition to this metrics also are duplicated on the same branch (number of operators).

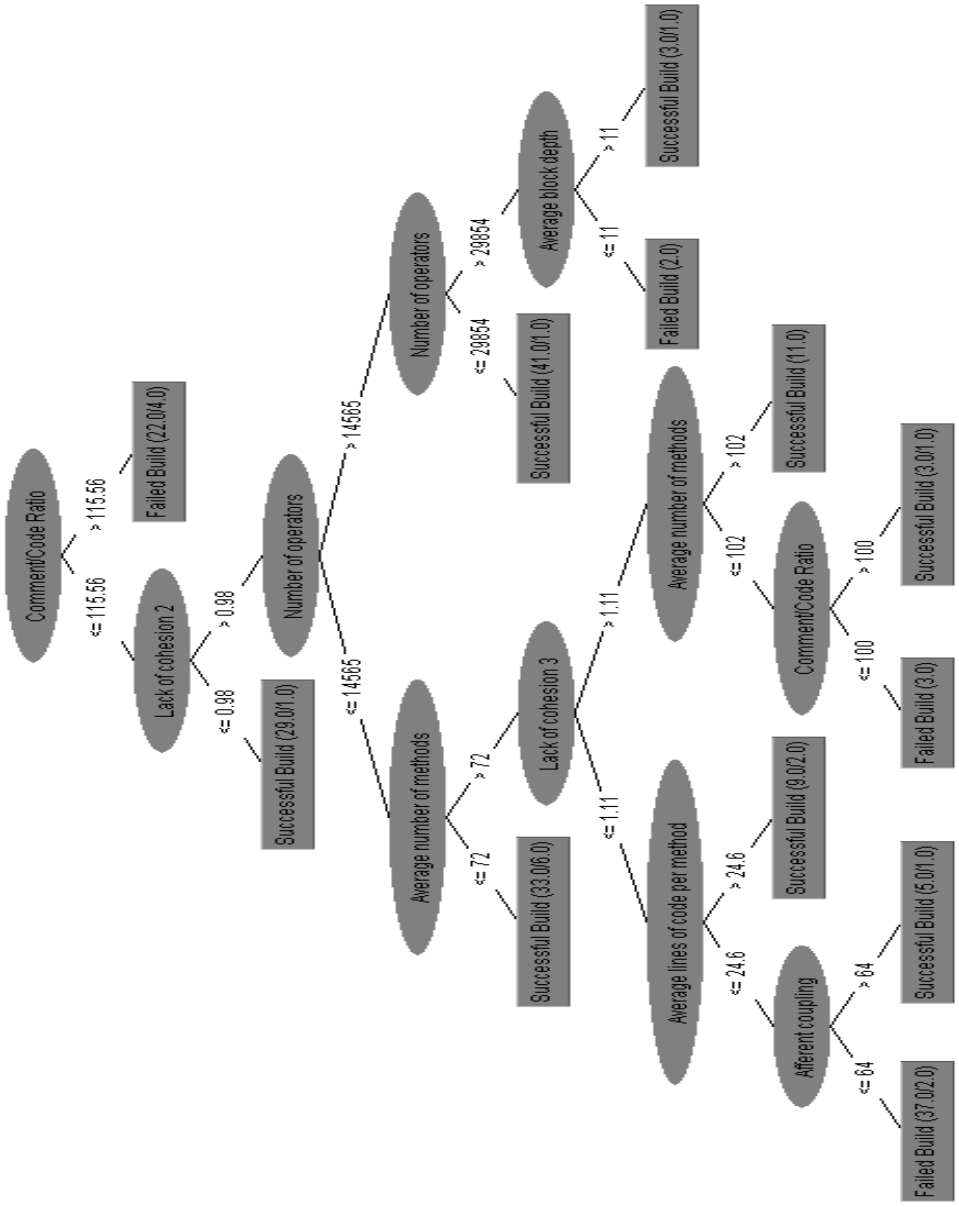

Figure 30 Max After State (No Feature Selection and j48)

The third best result was obtained again from the max data set filtered using Information Gain and mined using the j48 algorithm. The decision tree shown in Figure 31 that presents the results from Table 15 begins by taking on a similar form to Figure 30. Similarly the comment/code ratio is considered a significant metric when trying to predict failed builds. Another evident trend is that size metrics at method level are appearing to be common influences with the decision trees in general (i.e. average

number of methods and average lines of code per method). This tree again displays confusion via appearance of duplicate nodes, namely from such method size metrics.

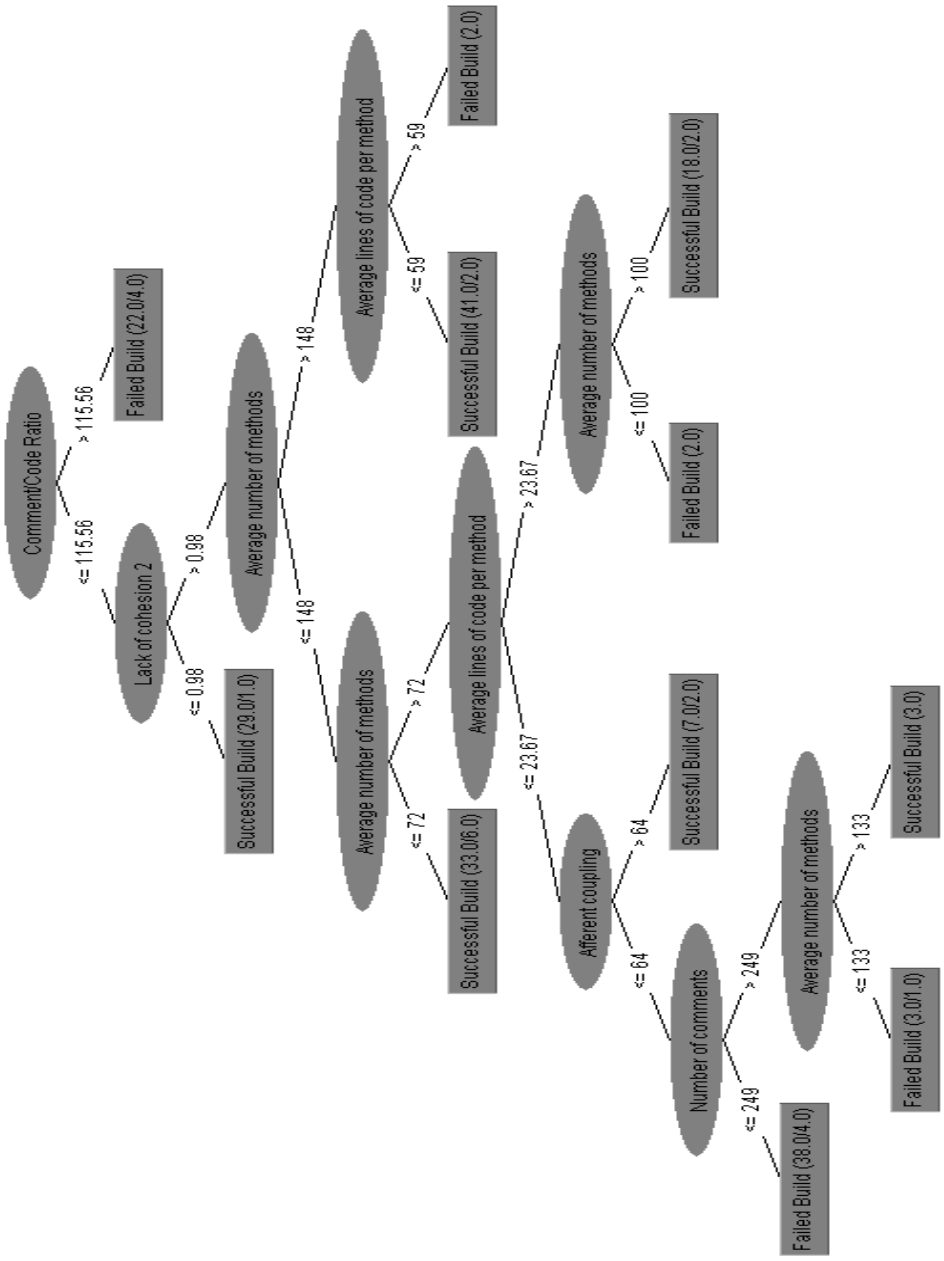

Figure 31 Max After State (InfoGain and j48)

In this experiment approximately 83% of successful build instances were correctly classified with an overall sensitivity average of approximately 0.73. Results for failed build instances did not perform so well, with approximately 60% correctly classified and a lower TP rating of 0.606.

Table 15 Max After State (InfoGain and j48)

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
105 (22)		43 (28)		74.7%	25.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.827	0.394	0.789	0.827	0.808
Failed Build	0.606	0.173	0.662	0.606	0.632
Weighted Average	0.747	0.315	0.744	0.747	0.745

It is not entirely surprising that the maximum value data set yields optimal results in terms of classification accuracy. Again this shows that extremes of values for many metrics are likely to be either desirable or undesirable, depending on the metric and have the effect of being able to find "good" or "bad" code that will contribute towards failure or success. For example if the maximum value of the comment to code ratio taken across all software builds exceeds a certain threshold then one would expect the build to be vulnerable to failure. A key difference in the results is that the overall accuracy can be traded off against an improved ability to predict failed builds. For example, using the RSA data set gives a very large number of correctly classified successful builds that leads to the highest overall accuracy. Meanwhile, using other data sets (e.g. median, mean and total) can give more accurate classification of failed builds at the cost of a reduced ability to classify successful builds.

Given the challenge in identifying failed builds the best classification is again with the RSA and Max data sets, which provided the best trade-off between overall accuracy and correct classification of failed builds. Classifications do not lead to a significant overall difference in prediction accuracy, perhaps indicating that many combinations of metrics have some scope to predict whether some builds are likely to be successful or fail. This certainly supports other literature that has shown that there is no single code or churn metric capable of predicting failures though evidence suggests that a combination can be used effectively (Denaro, et al., 2002).

After mining the before and after state software metric data sets it is observed that the RSA and max data sets perform better, in terms of accuracy, when compared to the

mining results obtained from the median, mean and total aggregations. As a result the RSA and max data sets are carried forward to the next stage of experiments. In regards to feature selection methods, while PCA performed well it will also not be included in the next experimental stages. This is because there is little or no readability of the output of PCA, which becomes a vital factor when interpreting decision trees. Both Subset Evaluation and Information Gain methods for feature selection are examined further. All three data mining methods will also be examined further in the next stage attempts to improve the classification accuracy of failed build instances.

At this stage, it is important to re-iterate that the purpose of this initial phase of experimentation is not to find a definitive prediction model, but instead to identify the most promising approaches. It is acknowledged that the results presented in this phase show a high degree of variability, possibly indicating that the size of the data set is limiting the ability of some methods to perform adequately. The variability in the results will be discussed in more depth in Chapter 5.

4.4 Enhancing Performance of Experiments

The results presented in sections 4.1, 4.2 and 4.3 revealed that while the before and after state metrics are different (consist of different values for metrics) they also have much commonality in relation to the mining results produced. It was also found that prediction of failed builds is more difficult to achieve than the prediction of successful builds. This is in despite of the data set aggregations, feature selection methods and mining algorithms utilised. This sections' objective is to investigate approaches for developing more accurate models by boosting the number of correctly classified failed builds. This is attempted using three main methods 1) feature frequency selection, 2) after state features as a filter to before state metrics and 3) application of SMOTE.

For the first method all the features selected from section 4.1 are counted and are filtered based on their frequency. The frequency is determined by the number of times they were selected using Subset Evaluation and Information Gain. This will provide insights into whether or not the features selected using Information Gain and Subset Evaluation can be further filtered based on the common occurrence features regarded as "significant". In some respects, this is an approach to combat the variability observed in the prediction models by reducing the number of potential predictor parameters.

The second method mentioned explores filtering the before state metrics using features selected from the after state. This provides insights into whether or not features selected from a future change set will increase the accuracy of predicting older metric values. This approach is being considered because it is possible that patterns that lead to failure may be identified in source code before the cause of failure fully emerges.

Finally, for the third method, SMOTE is applied at various levels to the RSA and Max data sets. In doing so the number of instances for the minority class (failed builds) will be increased through a process of creating synthetic data. This will provide insights into how much overlap there is in feature space of successful and failed build metrics.

For this phase the features considered for the before state are taken from all features selected between all before state data sets from the Subset Evaluation and Information Gain selection methods. These features are then counted and are iteratively dropped based on the total number of times they appeared. This iterative frequency selection method was then applied to the best two performing data sets, being the RSA and max metrics data sets, from the first experimental phase. There is a relatively high degree of variability in the features selected from each of the data sets presented in phase 1 and the resulting classification experiments presented in the best performing models section. All mining during the frequency of selection stages are done with 10 cross-fold validation. Overall this phase requires a total of 178 mining experiments in Weka.

4.5 Frequency of Features Selection for Before State Metrics

For this stage there are a total of 36 individual data mining experiments. To break this down it comprises of 2 data sets, by 3 classifiers (j48, Naive Bayes, Bayesian network), by 1 build states, by 6 frequency feature selection threshold tests.

The histogram shown in Figure 32 indicates the frequency of selection for each of the metrics under consideration for the before state. Metrics are omitted from the experiments if their overall ranking was low (they were not counted for more than once). There is a clear indication that certain metrics were insignificant as they were not selected at all, irrespective of the feature selection algorithm used. In contrast, metrics such as the maintainability index, number of unique operators, difficulty level, number of attributes, average number of constructors per class and number of types per package

were selected frequently, suggesting that they are stronger “code quality” indicators for the prediction of either build failure or success.

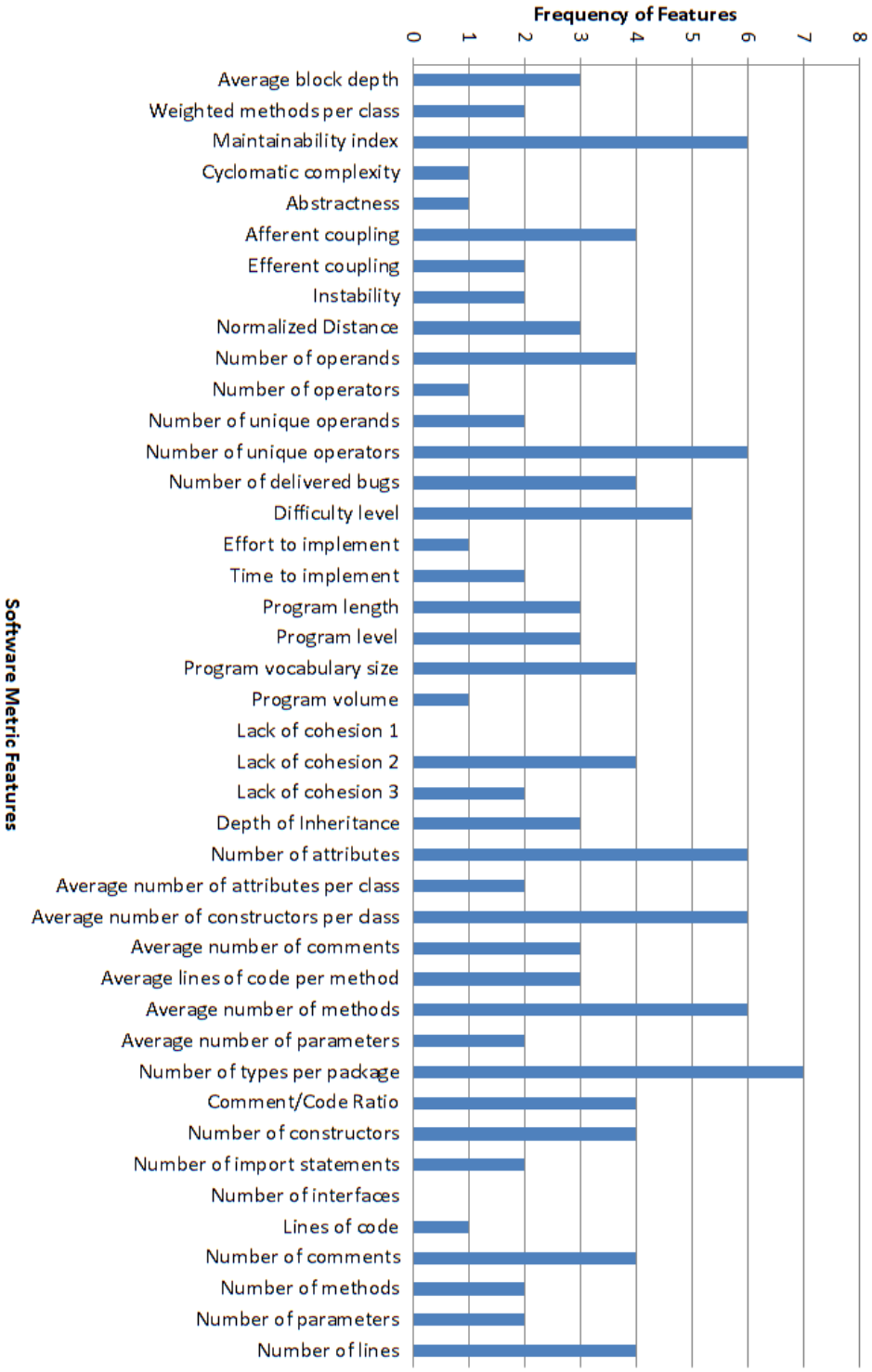

Figure 32 Before State Metric Feature Selection Frequency

In total there are 6 thresholds used for filtering features for each of the before and after state data sets (12 thresholds tested in total). For the full set of features used refer to Appendix E: Frequency Feature Selection Thresholds.

For the before state metrics, using j48 tree, Naive Bayes and Bayesian network methods and iteratively increasing the number of features included, the range of correctly classified instances for the RSA data set is between (and inclusive of) 62.1% and 75.3%. Correctly identifying failed build instances the accuracy ranges from 11.1% to 75.0%. Compared to the initial mining experiments from the RSA data set this method has not improved overall accuracy for correctly classifying instances. From the first stage the accuracies were between 65.2% and 79.3%. In addition to this the upper range of correctly identifying failed builds has slightly dropped from the previous experiments that are between 12.5% to 70.8%. Generally there is no significant change in accuracy from the results presented in section 4.1.

Figure 33 shows the percentage of correctly classified instances by mining algorithm for the before state metrics when applying the various thresholds of the frequency feature selection method. Each mining scenario is represented by the best two performing data sets from the previous experimental phase and 5 selection iterations. In this mining scenario the j48 classifier produced the best results, in terms of correctly classifying build instances, for 11 out of 12 scenarios. Secondly the Bayesian Network method produced the best result in 1 out of 12 scenarios; however the result was not overly more significant than the j48 classifiers (on the RSA data set, with the frequency greater than 2). From these results it is again observed that the j48 decision tree classifier performs better, in most cases, in terms of accurately classifying software builds. In terms of feature selection, the frequency feature selection method compared to Information Gain and Subset Evaluation produced very similar results in terms of correctly classified instances and sensitivity values for the before state metrics.

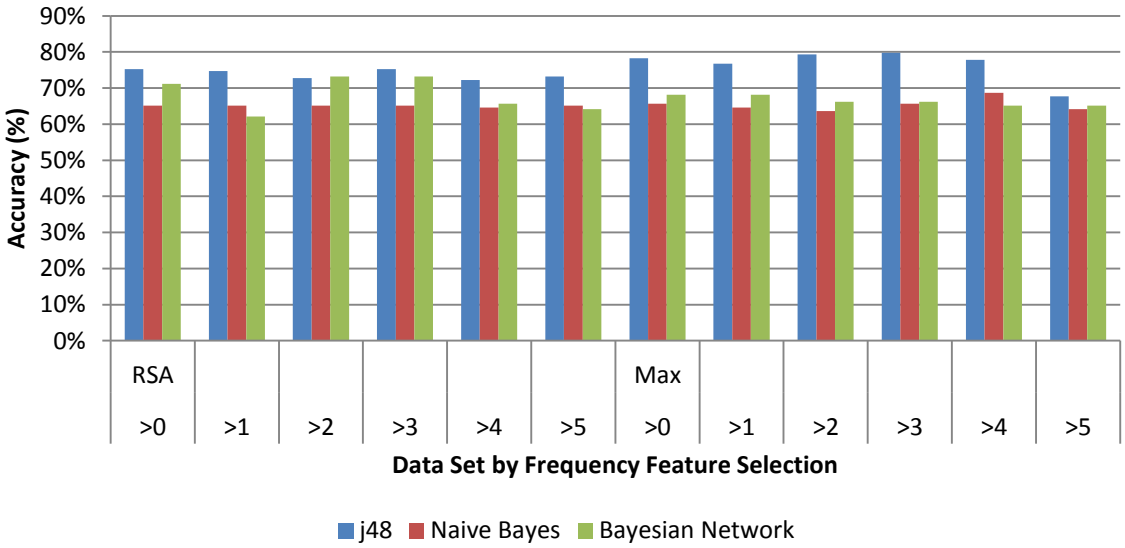

Figure 33 Before State Frequency Feature Selection Results by Mining Algorithm

Using the frequency feature selection method for the max data set the overall correctly classified instances ranges from 63.6% to 79.8%. For correctly identifying failed builds the accuracy ranges from 26.4% to 70.8%. This result is not a significant change from previous mining of the max data set where an overall accuracy generated ranged from 64.1% to 78.8% and for correctly classifying builds ranged from 29.2% to 69.4%. Figure 34 illustrates the percentage of correctly classified instances by frequency feature iteration for the before state metrics. Each mining scenario is represented by a mining algorithm and the best performing data sets from section 4.1 mining experiments. From these sets of mining scenarios feature selection iterations produced very similar results across both data sets.

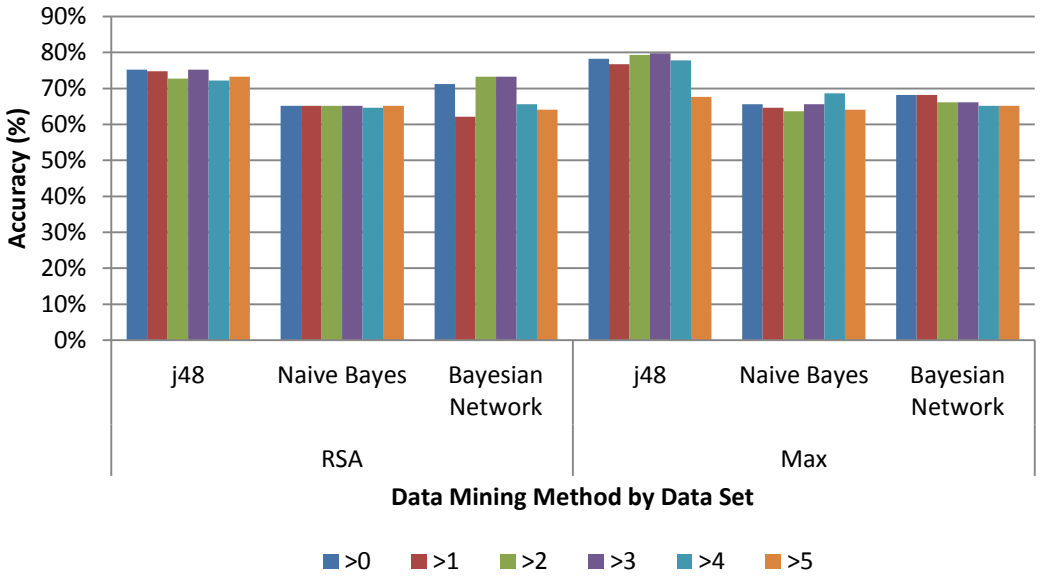

Figure 34 Before State Mining results by Frequency Feature Selection

Figure 35 illustrates the percentage of correctly classified instances by data set for the before state metrics using frequency feature selection. Each mining scenario is represented by a mining algorithm and feature selection threshold. From this perspective the max data set performed the best, in terms of correctly classified instances, for 8 out of 18 instances and produced the highest percentages over this set of experiments. When compared to section 4.2.4 the max data set also performed well when using the j48 classifier.

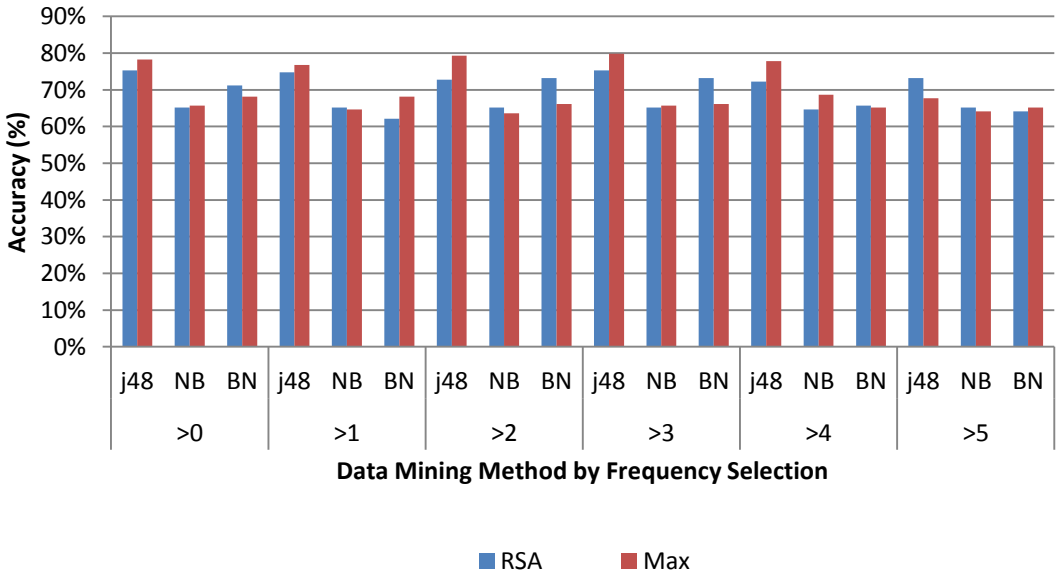

Figure 35 Before State Mining Results for Frequency Selection by Data Set

4.5.1 Best Performing Model for the Before State

The best performing model for the before state metrics using the feature frequency method is produced using the max data set when frequency threshold is greater than 3 (selected more than 3 times from previous experiments). Results for this experiment are presented in Table 16. The percentage of correctly classified instances is 79.8%. This is a very similar result when compared to the results presented in section 4.2.4 where the best model derived from the before state data, using the max data set had an overall accuracy of 78.8% and also had 4 additional correctly classified failed build instances. This is considered to be an increase in overall accuracy; however, it is not a significant improvement. For the full set of results for this section refer to Appendix E: Frequency Feature Selection Thresholds, Features Selected for the Before State.

Figure 36 shows the decision tree for the max before state data set where frequency features filter is greater than three. The classification table for this tree is presented in Table 16. Once again comment/code ratio appears to be a significant metric when predicting build failure (20 classified, with 3 incorrectly classified). Other metrics which also appear to have a higher degree of impact for predicting failed builds includes the maintainability index, number of delivered bugs (correctly classifying 8 failed builds) and difficulty level (correctly classifying 9 failed builds). Within this tree there is still a degree of confusion with duplicate nodes, this time occurring with the difficulty level and number of unique operators metrics. It is also noteworthy that there is still a strong presence of sizing metrics in general.

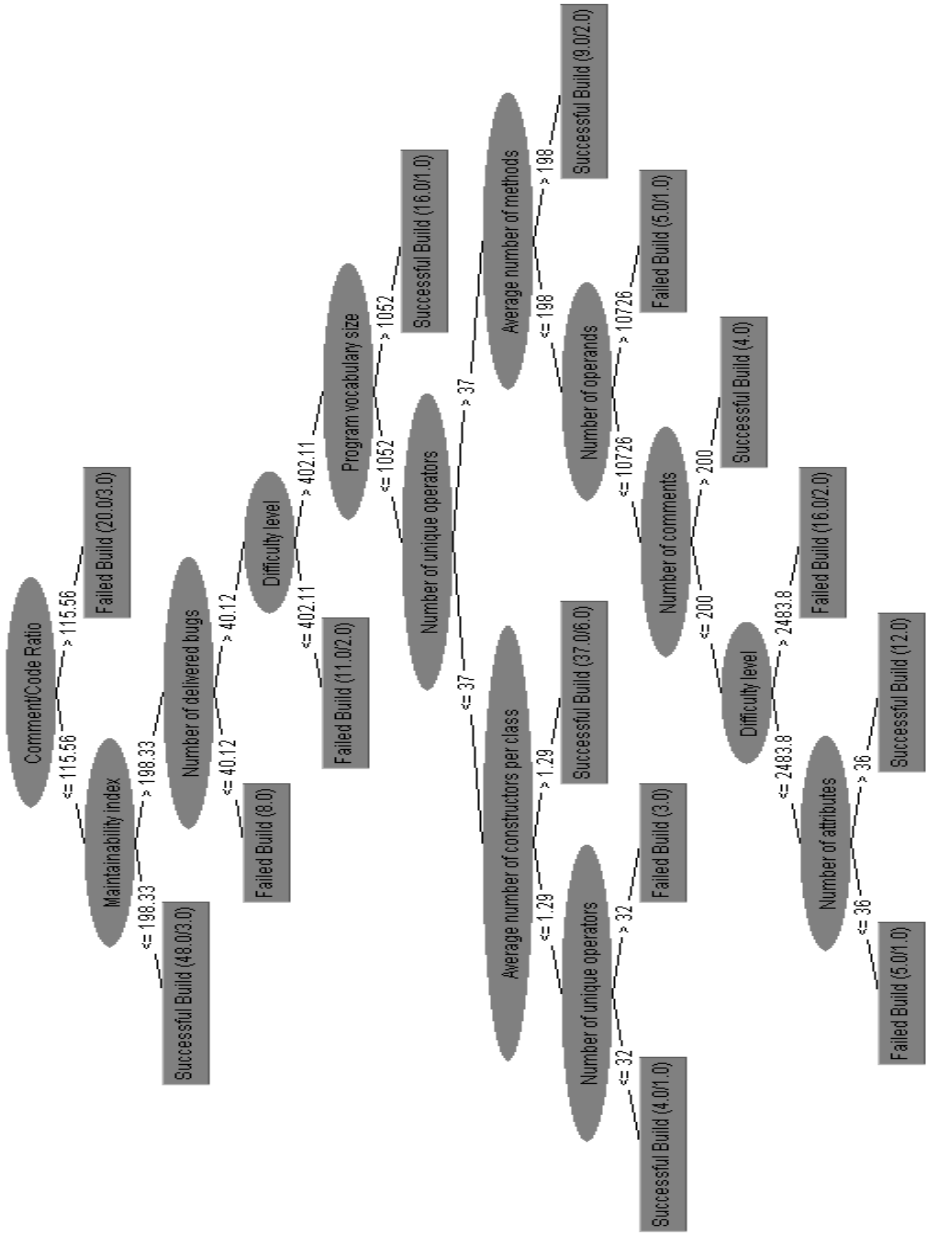

Figure 36 Max Before State (Frequency Features > 3 and j48)

From this experiment approximately 85% of successful builds were correctly classified with a TP rating of 0.849. Approximately 71% of failed builds were correctly classified with a TP rating of 0.708.

Table 16 Results for Before State of the Max Data Set with >3 Frequency Feature Selection and the j48 Classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
107 (19)		51 (21)		79.8%	20.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.849	0.292	0.836	0.849	0.843
Failed Build	0.708	0.151	0.729	0.708	0.718
Weighted Average	0.798	0.24	0.797	0.798	0.797

Interestingly, as the number of selected metrics is reduced by applying a higher frequency threshold, the overall accuracy does not change significantly, yet there is a trend towards better classification of successful builds. This again indicates that some metrics are very strong indicators of success whereas others are weak indicators of failure. This will be discussed in more depth in Chapter 5.

4.6 Frequency of Features Selection for After State Metrics

The histogram shown in Figure 37 indicates the frequency of selection for each of the metrics under consideration for the after state. Similarly to the before state metrics were omitted from the experiments if their significance was low. Software metrics such as abstractness, depth of inheritance, number of attributes, average number of constructors per class, average number of methods and number of comments were selected frequently, suggesting that they are stronger "code quality" indicators for the prediction of either build failure or success.

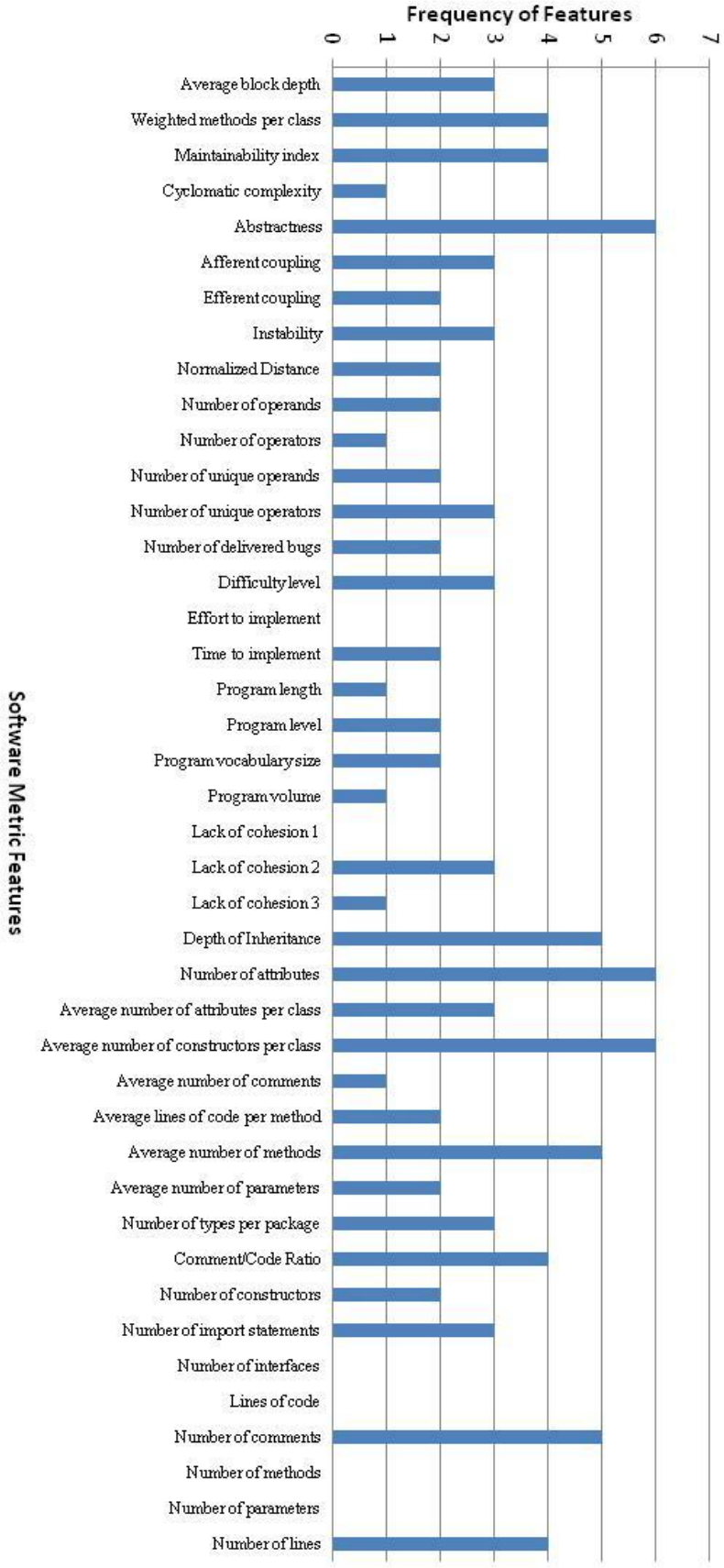

Figure 37 After State Metric Feature Selection Frequency

For the after state metrics, using j48 tree, Naive Bayes and Bayesian network methods and iteratively increasing the number of features included, the range of correctly classified instances for the RSA data set is between (and inclusive of) 58.8% and 76.9%. In regards to correctly identifying failed build instances the accuracy ranges from 4.2% to 66.7%. Comparing to the initial experiments on the after state metrics for this data set this also not an improvement on the results where the range of correctly classified instances is from 62.8% to 80.4%. There is also not a great improvement for classifying failed build instances, where previously accuracies ranged from 5.3% to 66.7%. Figure 38 presents the percentage of correctly classified instances by mining algorithm for the after state metrics using frequency feature selection. Each mining scenario is represented by a data set and feature frequency iteration. The mining algorithm that produced the best results across most (11 out of 12) scenarios was the j48 classifier. This is a very similar result again to the before state metrics.

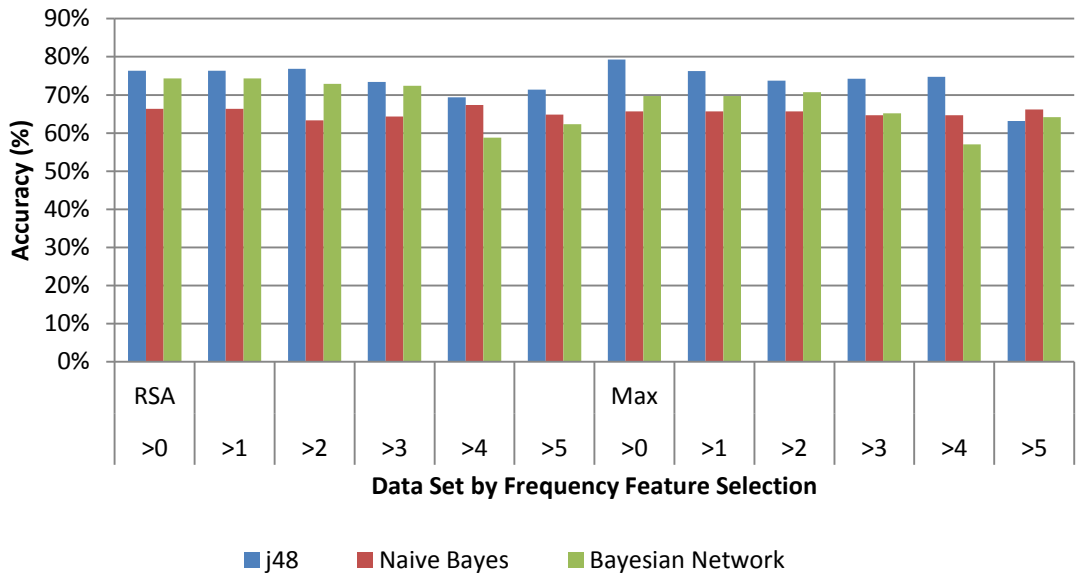

Figure 38 After State Frequency Feature Selection Results by Mining Algorithm

For the max data set the overall correctly classified instances ranges from 57.1% to 79.3%. For correctly identifying failed builds the accuracy ranges from 0% to 68.06%. This again shows no improvement of results, where previous experiments of the max data set had correctly classified instances ranging from 62.6% to 79.8%. Correctly identifying failed builds also has not improved, where previously the accuracy ranged from 19.7% to 69.0%.

Figure 39 presents the percentage of correctly classified instances by mining algorithms for the after state metrics using frequency feature selection. Each mining scenario is represented by a data set and feature frequency threshold. In these sets of experiments it is observed that the RSA data set generates the highest number of correctly classified instances 10 times out of 18 scenarios.

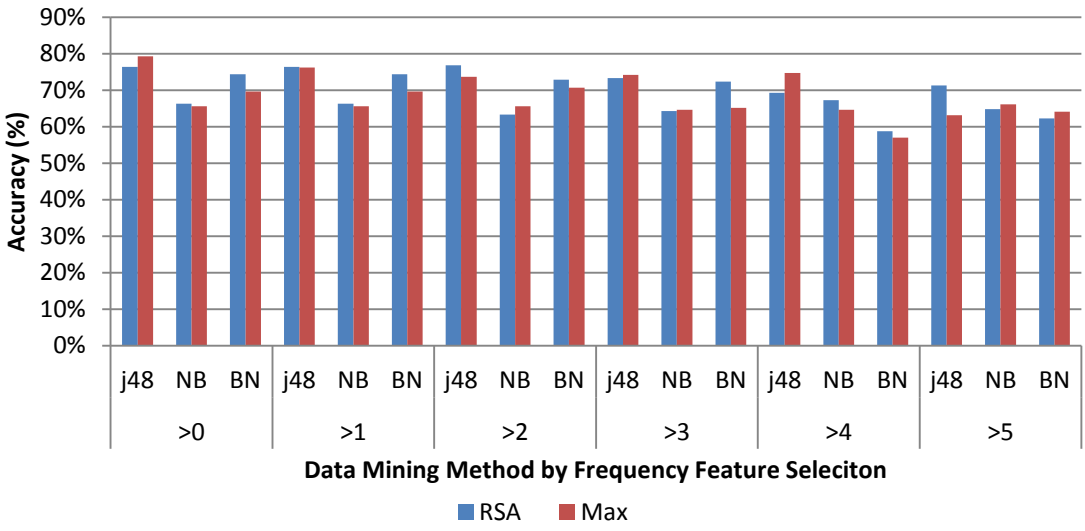

Figure 39 After State Mining Results for Frequency Selection by Data Set

Figure 40 presents the percentage of correctly classified instances by feature selection iteration for the after state metrics using frequency feature selection. Each mining scenario is represented by the mining algorithm and data set that is used. It is observed that as the number of iterations increases, the level of classification accuracy slightly decreases. This shows that even if a few metrics appear to be common influences, exclusive use of such metrics are still not enough to determine success. This again shows that there is no single "best" software metric and there is no single "best" feature selection method, of the metrics investigated.

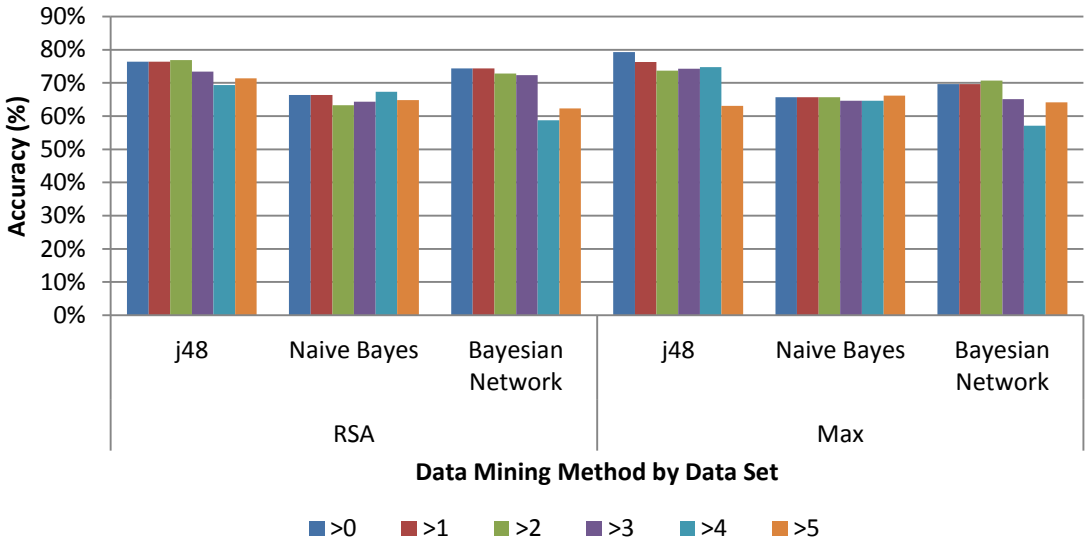

Figure 40 After State Mining results by Frequency Feature Selection

4.6.1 Best Performing Model for the After State

From this stage of experiments the best result produced is derived from the Max data set where >0 feature frequency selection is applied and is presented in Table 17. For this feature selection iteration if features were counted at least once, they were included in the overall features selected. This result has not made significant improvement from the first phase of mining experiments, where the best overall correctly classified instances for the max data set was 79.8% using no feature selection and the j8 classifier. What is a noticeable trend is the j48 classifier has been generating the best mining results across all experiments in both phases for both before and after software metric states. This again is followed by the Bayesian Network. For the full set of results for this section refer to Appendix E: Frequency Feature Selection Thresholds and Features Selected for the After State.

Table 17 Results for the Max Data Set using >0 Frequency Feature Selection and the j48 Classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
108 (19)		49 (22)		79.3%	20.7%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.85	0.31	0.831	0.85	0.84
Failed Build	0.69	0.15	0.721	0.69	0.705
Weighted Average	0.793	0.252	0.791	0.793	0.792

Figure 41 presents the decision tree for the results presented in Table 17. Once again comment/code ratio is a significant predictor for classifying failed builds (18 correctly classified). However, in this case a majority of failed builds are predicted in lower level branches of the tree. Regardless of this the other metrics which impact a majority of classifying failed builds include, in addition to comment/code ratio, lack of cohesion 2, number of operators, average number of methods, lack of cohesion 3, average lines of code per method and afferent coupling to correctly classified 35 failed builds. Duplicate nodes are also present, namely the comment/code ratio, number of operators and average number of methods.

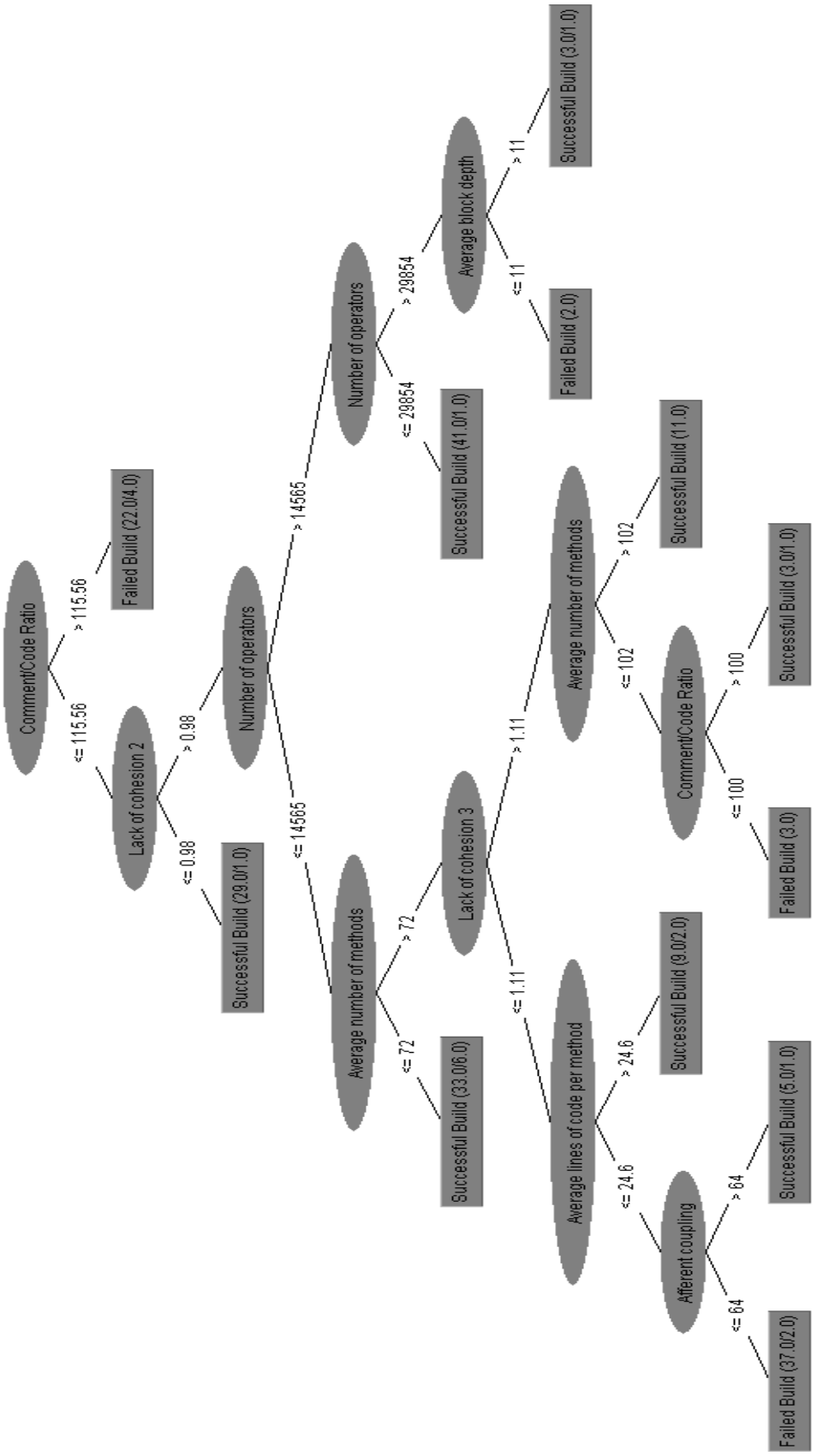

Figure 41 Max After State (Frequency Features >0 and j48)

4.7 Applying After State Features To Before State Data Set

This section comprises of a total of 48 data mining experiments. Broken down this is 2 best performing data sets, by 3 classifiers (j48, Naive Bayes, Bayesian network), by 6 frequency feature selection iterations. This also includes the addition of the 2 initial Subset Evaluation and Information Gain feature selection methods, by 2 best performing data sets and by the 3 previously mentioned classifiers.

Prediction accuracies in the 70% range are not overly high, but may be sufficient to provide some indication of risk at the beginning of the build cycle. A build that is considered to be higher risk could be identified and more diligence placed upon the processes used during the development as a means to mitigate that risk and increase the likelihood of the build being successful. A key aspect of being able to successfully manage risk in this way would be having the means to incrementally revise and review risk exposure during the build cycle. To that end, this experimental stage investigates whether there is any improvement in accuracy achieved when the after state feature selection is applied to the before state data.

For this stage the features selected from Subset Evaluation, Information Gain and all frequency feature tested thresholds from the after state metrics are applied to the before state RSA and max metrics data sets. The before state is used to determine whether build failure can be predicted prior to any changes in the source code being made. This is an attempt to characterise source code that is about to be changed in terms of its likelihood to be modified successfully or whether there are patterns in the existing code that have the potential to lead to failure. In doing so the ranges for correctly classified instances for the RSA data set is between (and inclusive of) 62.1% and 78.3%. The range of correctly classified failed instances ranges from 2.3% to 66.7%. The overall correctly classified instances for the max data set is between 38.4% and 81.8%. The range of correctly classified failed instances ranges from 0% to 76.4%.

Figure 42 illustrates the percentage of correctly classified instances by mining algorithm for application of after state features to the before state data sets. Each mining scenario is represented by a data set and the feature selection method used. From this perspective the best results are again obtained by using the j48 classifier. In this testing phase the j48 classifier produced the best results for 15 out of 16 scenarios.

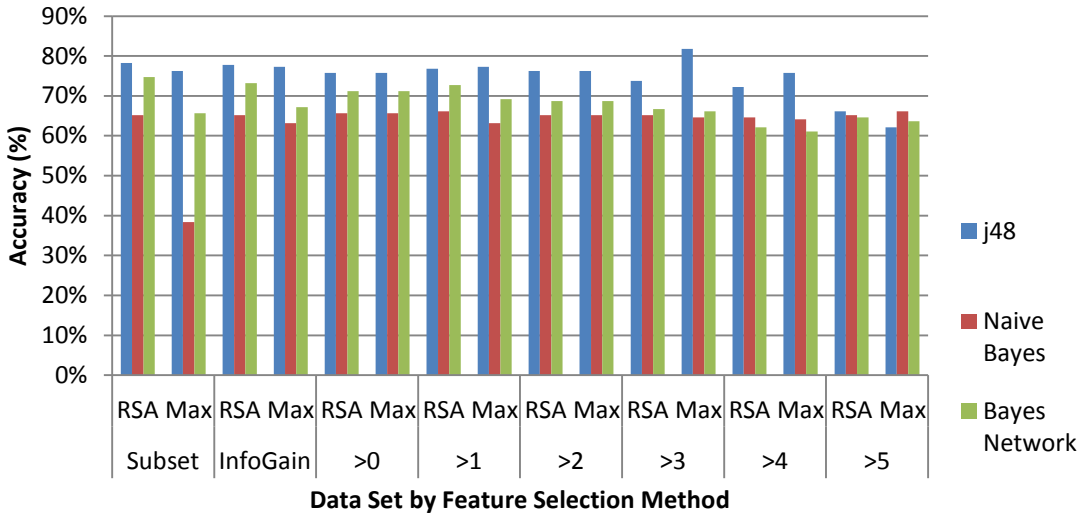

Figure 42 After State Features Applied to Before State Data by Mining Algorithm

Figure 43 shows the percentage of correctly classified instances by feature selection method for application of after state features to the before state data sets. Each mining scenario is represented by a mining algorithm and data set. From this viewpoint it is not clear which feature selection method is best for determining the best models, despite which mining algorithm is used.

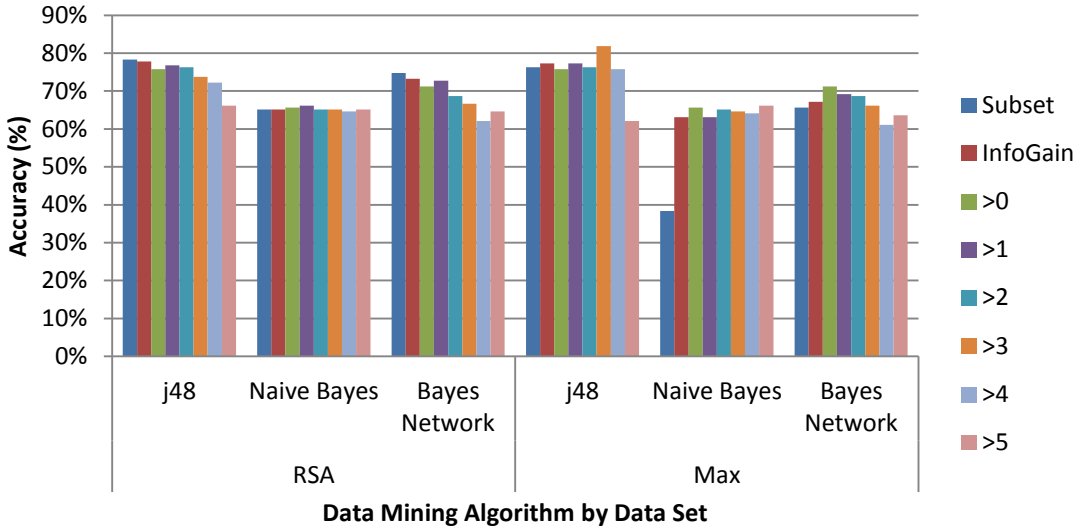

Figure 43 After State Features Applied to Before State Data by Feature Selection Method

Figure 44 shows the percentage of correctly classified instances by the before data set when applying after state features. Each mining scenario is represented by a mining algorithm and feature selection methods used. From this angle the RSA data set has often produced the best results in terms of overall correctly classified instances for 14 out of the 24 scenarios.

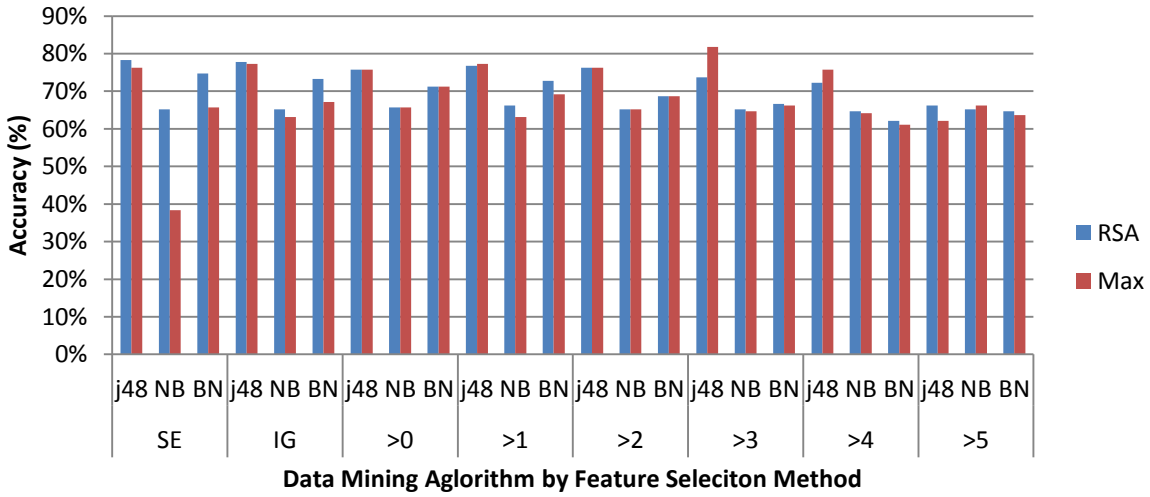

Figure 44 After State Features Applied to Before State Data by Data Set

4.7.1 Best Performing Models

Again models of particular interest contains high levels of overall classification accuracy and high levels of accuracy for predicting failed builds. Figure 45 presents the overall classification accuracy and failed builds classification accuracy for 72 data mining experiments that were based on RSA and Max software metric aggregations, frequency feature thresholds and data mining methods for the before state software metrics.

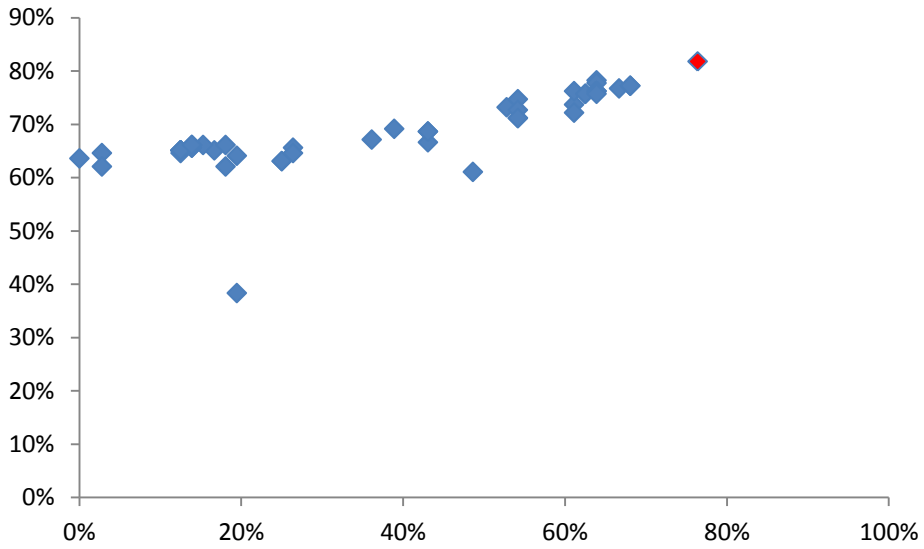

Figure 45 Frequency Feature Threshold Results: Overall Classification Accuracy of Builds versus Classification for Failed Builds

When the after state max data set has the before state >3 feature frequency selection method applied it produced the pareto-optimal solution for this set of experiments and is presented in Table 18. The classification tree for this result is illustrated in Figure 46. This experiment has produced the best result thus far from all experiments performed. There were 85% of successful builds were correctly classified with a TP rating of 0.849 and approximately 76% of failed builds were correctly classified with the TP rating of 0.764. The classification tree exhibits an amount of confusion, particularly in the mid level nodes. Unlike previous trees where comment/code ratio alone was indicating a significant factor for predicting failed builds, other rules for predicting failure are presented in higher levels of the tree. Confusion within this model is again defined by the presence of the same classification attribute at various levels in the same branch. For example the "weighted methods per class" metric appears twice within the same branch. Average number of constructors per class and depth of inheritance was also duplicated.

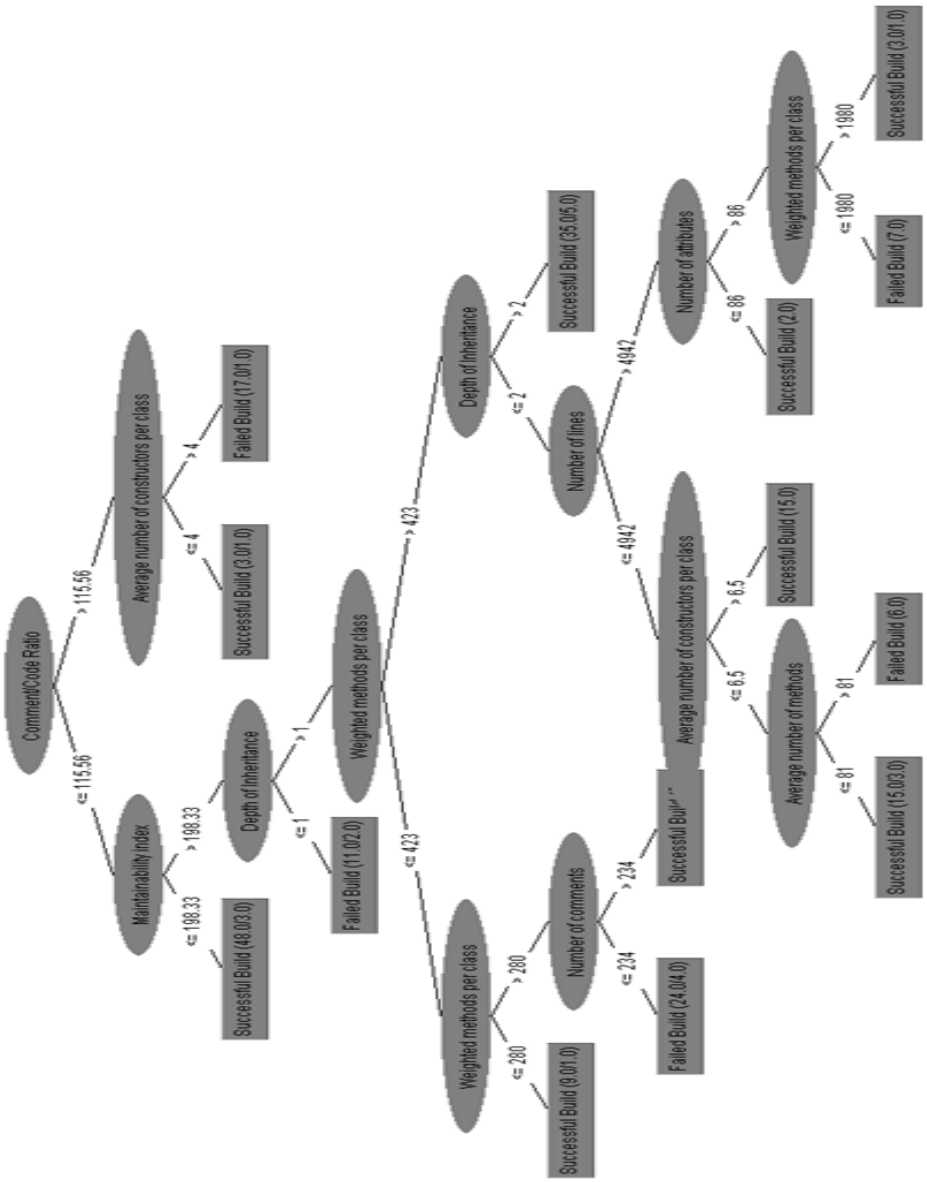

Figure 46 j48 classification tree of the max after state data set, with before frequency features that are greater than 3

Table 18 Results for the After State of the Max Data Set using >3 Frequency Feature Selection From the Before State and the j48 Classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
107 (19)		55 (17)		81.8%	18.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.849	0.236	0.863	0.849	0.856
Failed Build	0.764	0.151	0.743	0.764	0.753
Weighted Average	0.818	0.205	0.819	0.818	0.819

4.8 Application of SMOTE

It has been observed repeatedly that predicting failure is more challenging than predicting success regardless of feature selection and data mining methods applied. In addition to this not predicting failure does not mean that success has been predicted. This is potentially due to the fact that the build successes and failures overlap in feature space and "failure" signatures have a greater degree of fragmentation than their "success" counterparts. Evidence for this is most apparent in the very different classification trees that have been discussed to date. Each shows a different set of software metrics that can be used to gain roughly the same overall prediction accuracy. As a result, the next aspect of this work is to develop a deeper understanding of what source code characteristics are most related to build failure and develop a set of indicative metrics that can provide development teams with the opportunity to proactively manage risk exposure throughout a development project even if they cannot categorically predict build failure or success.

It is observed that one of the reasons why build failure is difficult to predict is because of the naturally occurring skewed class distribution within the data. SMOTE is a supervised class-imbalance learning method. It works by generating virtual minority-class instances. In order to apply SMOTE to the data sets explored within this study, it will require a total of 40 mining experiments. Broken down that is 2 best performing data sets, 2 feature selection methods, by 5 SMOTE iterations, by 2 change set states.

4.8.1 Before State Data Sets

For the before state metrics, the RSA data set generated accuracy ranges between (and inclusive of) 47.5% and 79.7% for correctly classified instances. For correctly identifying failed builds accuracies ranged from 4.6% to 81.8%. This is a minor improvement for correctly classifying failed builds, when comparing the highest levels of accuracy, from the set of experiments from Phase 1 (12.5% - 70.8%). The correctly classified instances from the max data set ranged between 44.1% and 69.5%, from these, correctly classified failed instances range from 9.09% to 100%. This is again a minor improvement from Phase 1 where correctly classified builds ranged from 29.2% to 69.4%. Figure 47 illustrates the percentage of correctly classified instances for the before state data. Each mining scenario is represented by a mining algorithm, feature selection method and by each SMOTE percentage increase (100% to 500% in incremental steps of 100). Where SMOTE at 100% increases the number of instances by 100%. The RSA data set produced the best result for 20 out of 30 of these data mining scenarios.

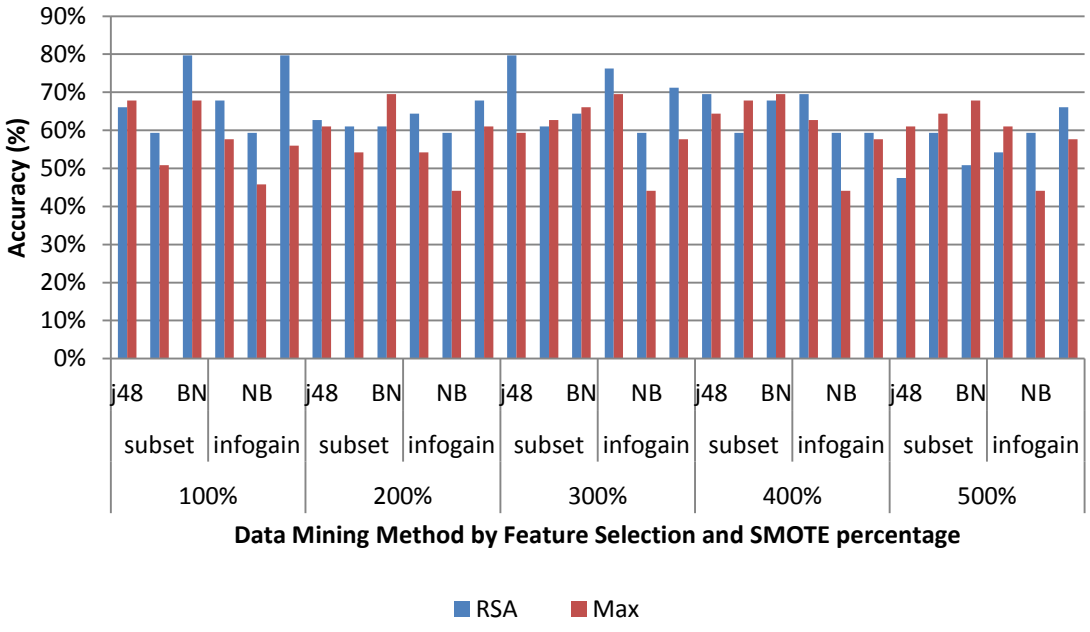

Figure 47 Overall Classification Accuracies using SMOTE Results for Before State Metrics by Data Set

From the feature selection focus, the Subset Evaluation method generated accuracies from 47.5% to 79.7% with correctly classified failed builds ranging from 9.1% to 100%.

The Information Gain method generated overall accuracies from 44.1% to 79.7% with 4.6% to 81.8% failed builds correctly classified. Figure 48 shows the results of applying SMOTE to the before state data by feature selection method. From this angle the best performing feature selection method when used with SMOTE is the subset evaluator, producing the best results for 17 out of 30 scenarios. Information Gain produced the best result for 7 out of 30 scenarios. There were also instances when the feature selection methods tied, this occurred for 6 out of the 30 scenarios.

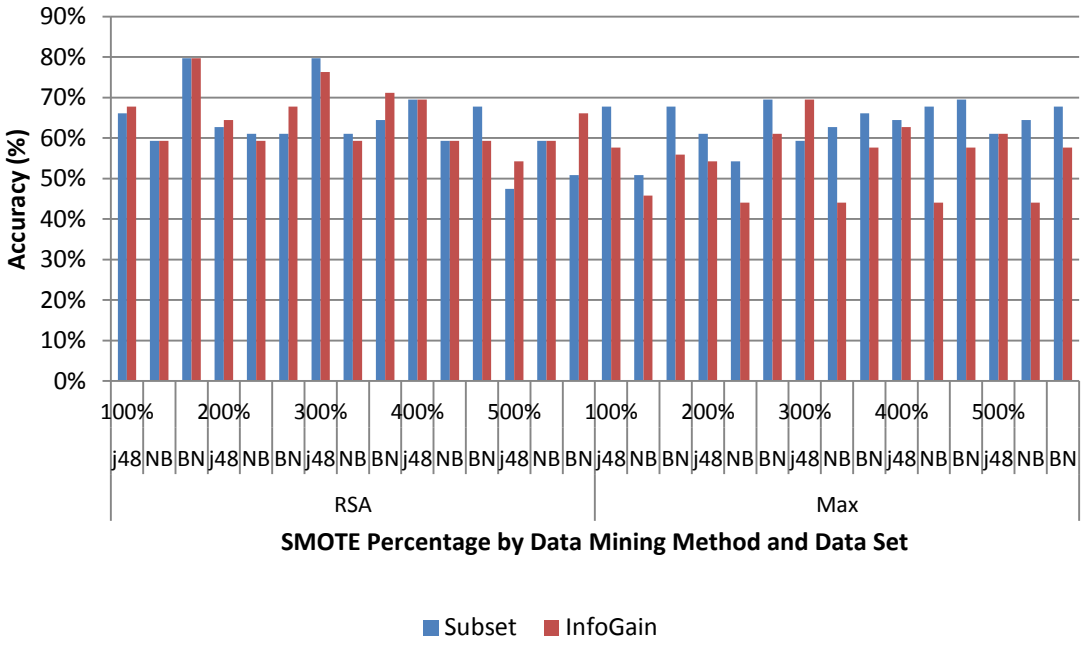

Figure 48 Overall Classification Accuracies using SMOTE Results for After State Metrics by Feature Selection Method

The mining algorithm focus for the j48 classifier generated correctly classified instances ranging from 47.5% to 79.7% with correctly classified failed builds ranging from 40.9% to 81.8%. Using the Naive Bayes method correctly classified instances range from 44.1% to 67.8% with ranges of 4.6% to 86.4% of failed builds correctly classified. Finally the Naive Bayes method generated overall accuracies from 50.9% to 79.7% with correctly classified failed builds ranging from 40.9% to 100.0%. Figure 49 presents the results of the before state data when SMOTE is applied from the feature selection method view. Each mining scenario is presented by its SMOTE percentage increase, feature selection method used and data set. From this perspective both the j48 classifier and Bayesian network methods performed well each producing the best result for 9 out

of the 20 scenarios. Again it is observed that the Naive Bayes method did not produce significant results.

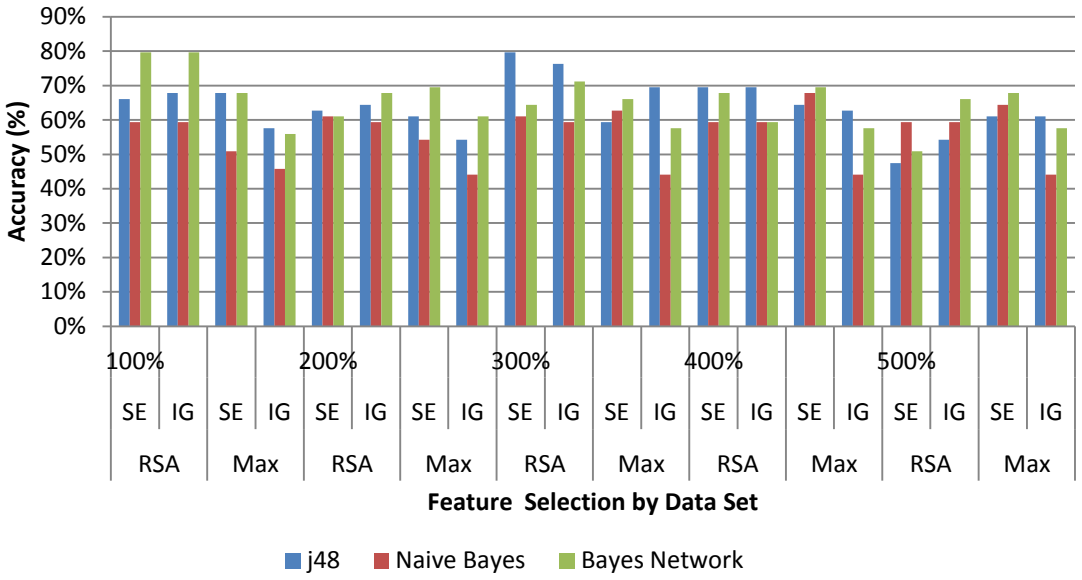

Figure 49 Overall Classification Accuracies using SMOTE Results for Before State Metrics by Mining Algorithm

4.8.2 Best Performing Models for the Before State Metrics

The best performing models from this stage of experiments is derived from using SMOTE at 100% with the subset evaluator and BN (Table 19) and SMOTE at 300% using Subset Evaluation and the j48 classifier (Table 20). An additional pareto-optimal result was also obtained using Information Gain and BN on the RSA data set (Table 21) with SMOTE at 100%. These top three performing methods generated identical percentages of correctly classified instances 79.7% and the same number of correctly classified failed and successful builds. In terms of sensitivity metrics the TP Rate, FP rate, Precision, Recall and F-measure are also identical for both successful and failed builds.

The results presented in Table 19 shows that when using SMOTE the percentage of correctly classifying failed builds increases significantly. In this case approximately 82% of failed builds were correctly classified, with a TP rating of 0.818. For successful builds there were approximately 78% correctly classified instances with a TP rating of

0.784. Identical results obtained are shown in Table 20 and Table 21 where different levels of SMOTE are applied using the j48 and Bayesian Network classifiers.

Table 19 Results for the Before State of the RSA Data Set with 100% SMOTE using Subset Evaluation and a Bayesian Network

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
29 (8)		18 (4)		79.7%	20.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.784	0.182	0.879	0.784	0.829
Failed Build	0.818	0.216	0.692	0.818	0.75
Weighted Average	0.797	0.195	0.809	0.797	0.799

Table 20 Results for the Before State of the RSA Data Set with 300% SMOTE using Subset Evaluation and j48 Classification.

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
29 (8)		18 (4)		79.7%	20.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.784	0.182	0.879	0.784	0.829
Failed Build	0.818	0.216	0.692	0.818	0.75
Weighted Average	0.797	0.195	0.809	0.797	0.799

The classification tree presented in Figure 50 is a representation of the results from Table 20. This tree also displays degrees of confusion within its nodes. However, unlike most of the previous trees which begin with the comment/code ratio, this one begins with the number of types per package as its root node. In this instance duplicate nodes can be found within the same tree path, for example number of unique operators, number of delivered bugs and average number of constructors per class metrics are duplicated.

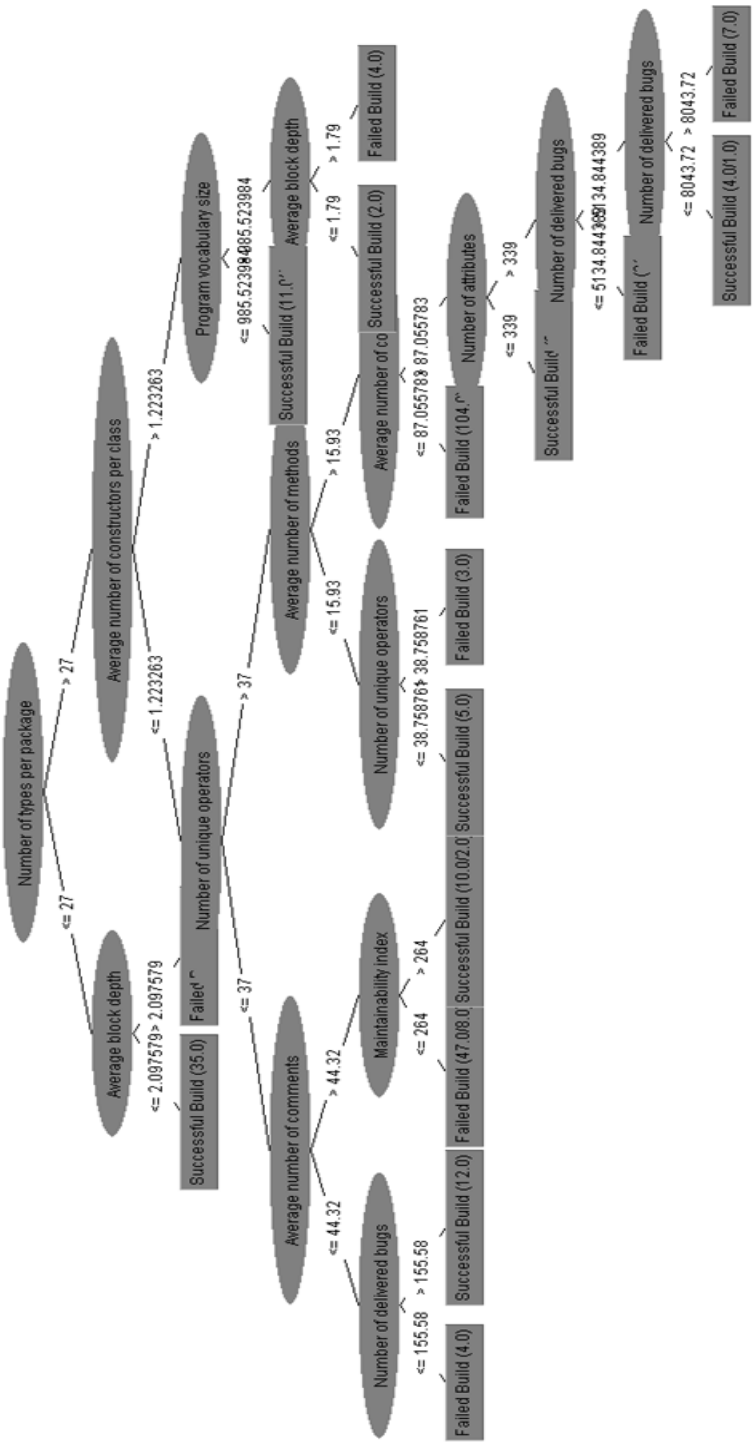

Figure 50 j48 Classification tree using Subset Evaluation on the RSA Before State data set with SMOTE at 300%

Table 21 Results for the Before State of the RSA Data Set with 100% SMOTE using Information Gain and a Bayesian Network

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
29 (8)		18 (4)		79.7%	20.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.784	0.182	0.879	0.784	0.829
Failed Build	0.818	0.216	0.692	0.818	0.75
Weighted Average	0.797	0.195	0.809	0.797	0.799

4.8.3 After State Data Sets

The RSA data set (after state) obtained classification accuracies that range between 56.7% and 71.7%. For correctly classifying failed builds, accuracies are between 13.6% and 90.9%. For the max data set the ranges of correctly classified instances are between 36.7% and 70.0%. Correctly classified failed build accuracies were between 18.2% and 90.9%. Figure 53 presents the results for the SMOTE experiments on the after state data set. Each scenario is presented by a mining algorithm, feature selection method and SMOTE percentage. Similar to the SMOTE experiments on the before state the RSA data set produced the best results for most scenarios (in this case 25 out of 30).

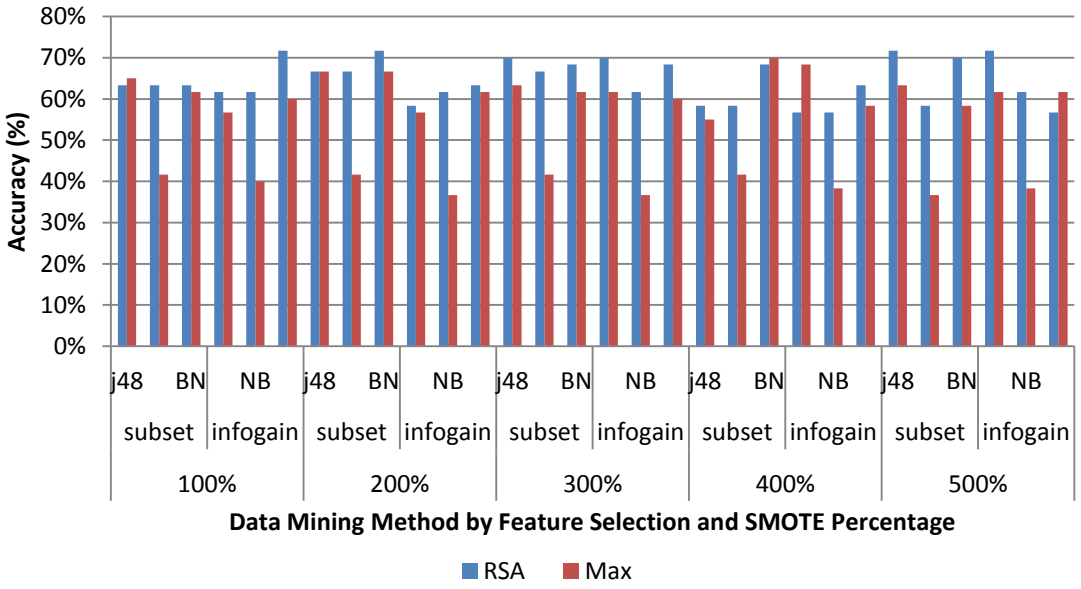

Figure 51 Overall Classification Accuracy using SMOTE for After State Metrics by Data Set

In regards to the feature selection methods used, the Subset Evaluation method overall correctly classified instances between 36.7% and 71.7% with correctly classified failed builds ranging between 18.2% and 90.9%. When using Information Gain the overall correctly classified instances generated identical values to the Subset Evaluation method, ranging again between 36.7% and 71.7%. For correctly classifying failed builds the range was 13.6% and 86.4%. Figure 52 presents the correctly classified instances for the after state data set after the application of SMOTE, with a focus on the feature selection method used. Scenarios from this perspective are presented by a SMOTE iteration, mining algorithm and data set. Results from the SMOTE activities on the before state data showed that the Subset Evaluator produced the best results for 22 out of 30 scenarios.

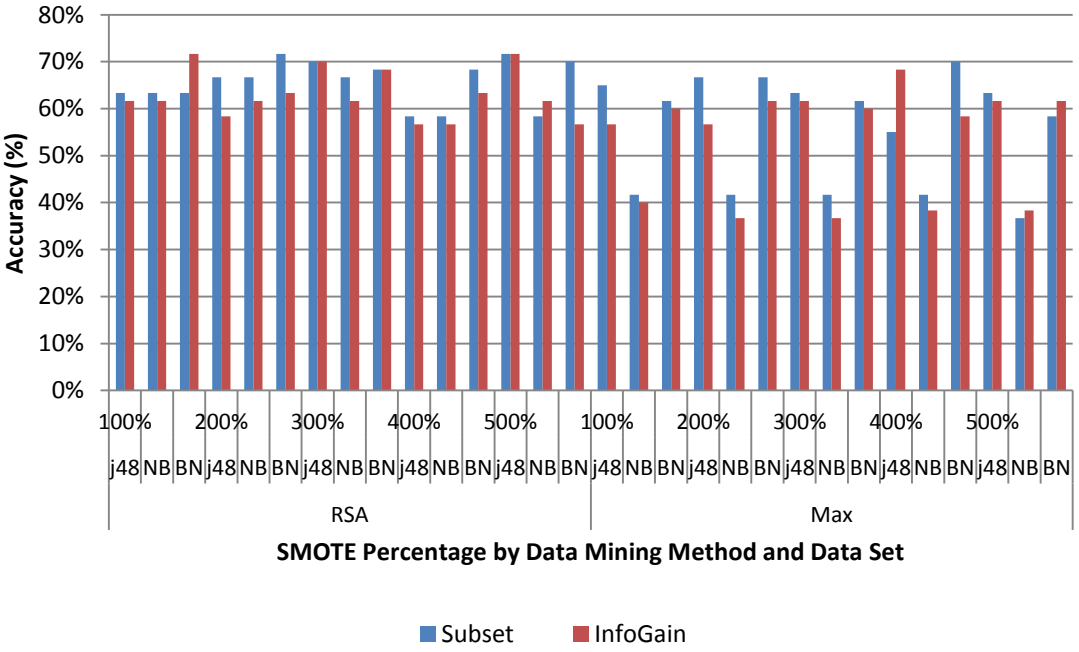

Figure 52 Overall Classification Accuracy using SMOTE for After State Metrics by Feature Selection

In terms of the mining algorithm level for the j48 classification tree, overall correctly classified instances ranged between 56.7% and 71.7% and correctly classified failed instances ranged between 36.4% and 81.8%. From using Naive Bayes the overall correctly classified instances range between 36.7% and 66.7% with correctly classified failed builds ranging from 13.6% and 77.3%. Finally, for the Bayesian Network method the correctly classified instances ranged between 56.7% and 71.67% and correctly classified failed instances ranging from 18.2% to 90.1%. Figure 53 presents SMOTE results for the after state data set based on mining algorithm. Each instance is presented by a SMOTE iteration, feature selection method and data set. From this view, the j48 classifier produced the best result for 9 out of 20 scenarios. This is closely followed by the Bayesian network which produces the best results for 8 out of 20 scenarios.

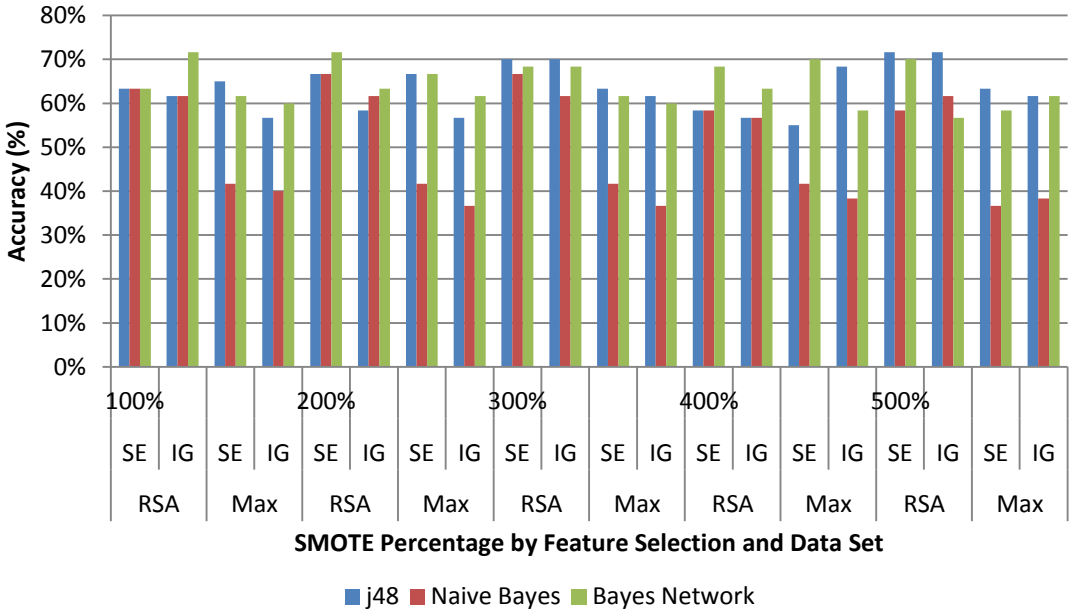

Figure 53 Overall Classification Accuracies using SMOTE for After State Metrics by Mining Algorithm

4.8.4 Best Performing Models for the After State Metrics

Similar to the before state metrics the best performing after state metrics also generated identical results in terms of overall correctly classified instances, however the number correctly classified failed and successful builds vary between experiments. From these mining experiments the best performing results were generated when:

- SMOTE was set to 200% (presented in Table 22), using Subset Evaluation and a Bayesian Network.
- SMOTE was set to 100% (presented in Table 23) when using a Bayesian Network.
- SMOTE was set to 500% (presented in Table 24), using Information Gain and the j48 classifier.

Again these best performing experiments were derived from the RSA data set. Similar to the before state when looking at the sensitivity metrics there is no significant variation between each experimental result. When SMOTE is applied at 200% the sensitivity metrics slightly increase in accuracy for classifying failed builds.

Presented in Table 22 are the results based on the RSA after state data set, with SMOTE applied at 200% and using Subset Evaluation and a BN. In this scenario approximately 90% of failed build instances were correctly classified, with a TP rating of 0.909. This is higher than correctly classified successful build instances (60%), with a TP rating of 0.605.

Table 22 Results for the After State of the RSA Data Set with 200% SMOTE using Subset Evaluation and a Bayesian Network

# Successful builds correct (incorrect)		**# Failed builds correct (incorrect)**		**Correctly Classified Instances**	**Incorrectly Classified Instances**
23 (15)		20 (2)		71.7%	28.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.605	0.091	0.92	0.605	0.73
Failed Build	0.909	0.395	0.571	0.909	0.702
Weighted Average	0.717	0.202	0.792	0.717	0.72

The features selected using Information Gain is identical for the results presented in Table 23 and Table 24. In this case the Bayesian Network and j48 classifier generated identical classification accuracies with slightly varied sensitivity values. The Classification tree for the results presented in Table 24 is illustrated in Figure 54.

Table 23 Results for the After State of the RSA Data Set with 100% SMOTE using Information Gain and a Bayesian Network

# Successful builds correct (incorrect)		**# Failed builds correct (incorrect)**		**Correctly Classified Instances**	**Incorrectly Classified Instances**
24 (14)		19(3)		71.7%	28.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.632	0.136	0.889	0.632	0.738
Failed Build	0.864	0.368	0.576	0.864	0.691
Weighted Average	0.717	0.221	0.774	0.717	0.721

Similar to the decision trees generated in section 4.1 the classification tree displays degrees of confusion within its nodes. In this instance duplicate nodes can be found within the same tree path and involve the weighted methods per class, number of attributes and number of unique operators metrics. In summary the frequency feature selection methods did not greatly improve classification outcomes in terms of the overall accuracy and sensitivity of the results from the first phase of experiments.

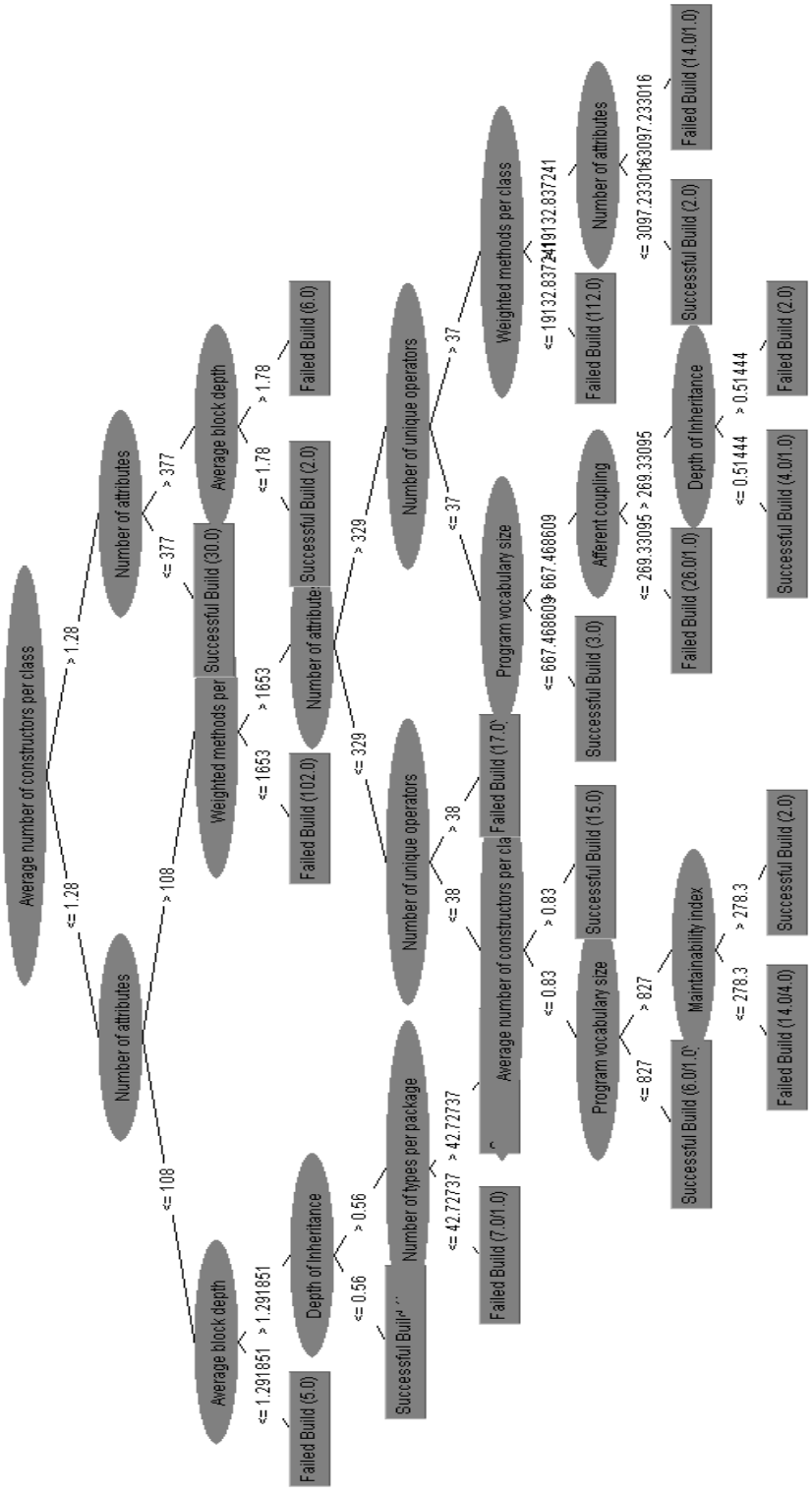

Figure 54 Classification tree of SMOTE at 500% on the RSA After State Data Set using Information Gain

Table 24 Results for the After State of the RSA Data Set with 500% SMOTE using Information Gain and the j48 classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
26 (12)		17 (5)		71.7%	28.3%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.684	0.227	0.839	0.684	0.754
Failed Build	0.773	0.316	0.586	0.773	0.667
Weighted Average	0.717	0.26	0.746	0.717	0.722

4.8.5 After State Metrics Applied to Before State SMOTE Data Sets

To further explore the application of SMOTE on software metrics part two of this stage applies the feature selection method from phase two, stage two. In this instance the features selected from Subset Evaluation and Information Gain from the after state are applied to the before state SMOTE experiments. This requires an additional 60 mining experiments. That is 2 best performing data sets, by 2 feature selection methods, by 3 data mining methods, by 5 SMOTE iterations. Figure 55 presents the classification results for the before state metrics with SMOTE applied and after state features applied. Each scenario is represented by a SMOTE percentage, a feature selection method and a data set. In this series of experiments the Bayesian Network method produced the best results (10 out of 20 times), closely followed by the j48 classifier (8 out of 20 times) and produced the same results twice (from the Max data set using SMOTE at 200% with the subset evaluator).

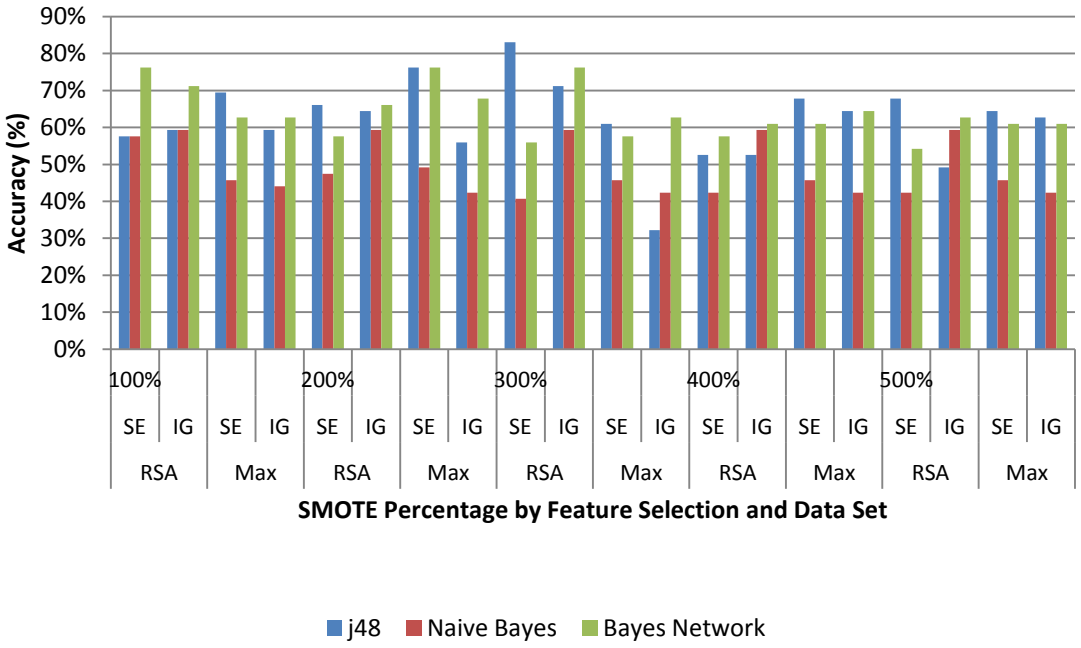

Figure 55 Overall Classification Accuracies using SMOTE when applying After State Features to The Before State by Mining Algorithm

Figure 56 shows the application of SMOTE on the before state metrics with features selected from the after state metrics. Each scenario is represented by a SMOTE percentage, a mining algorithm and a data set. From this perspective the feature selection that produced the best results from these scenarios is the subset evaluator, generating 15 out of 20 best cases.

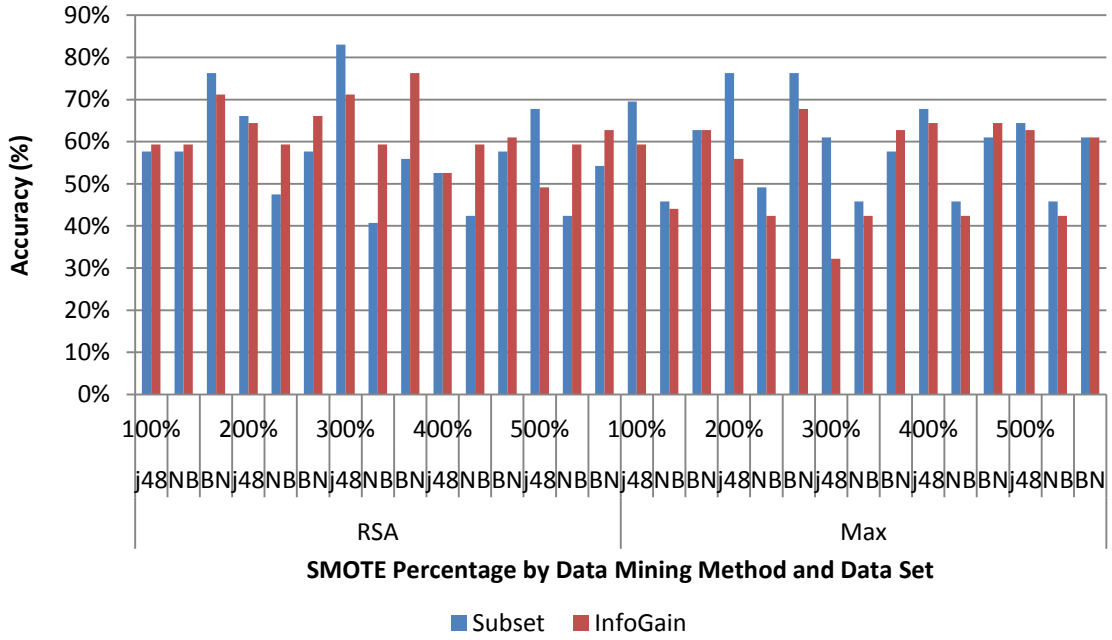

Figure 56 Overall Classification Accuracies using SMOTE when applying After State Features to The Before State by Feature Selection

Figure 57 illustrates the correctly classified instance when applying after state features to the before state data set with SMOTE applied. Each scenario is presented by its mining algorithm, feature selection method and SMOTE percentage. It is observed that the Max and RSA data set give similar levels of performance across all scenarios. More specifically the Max data set generates the best scenario 15 out of 30 times, the RSA 14 out of 20 times and generate the same result once when SMOTE is 100%, using Information Gain and the j48 classifier.

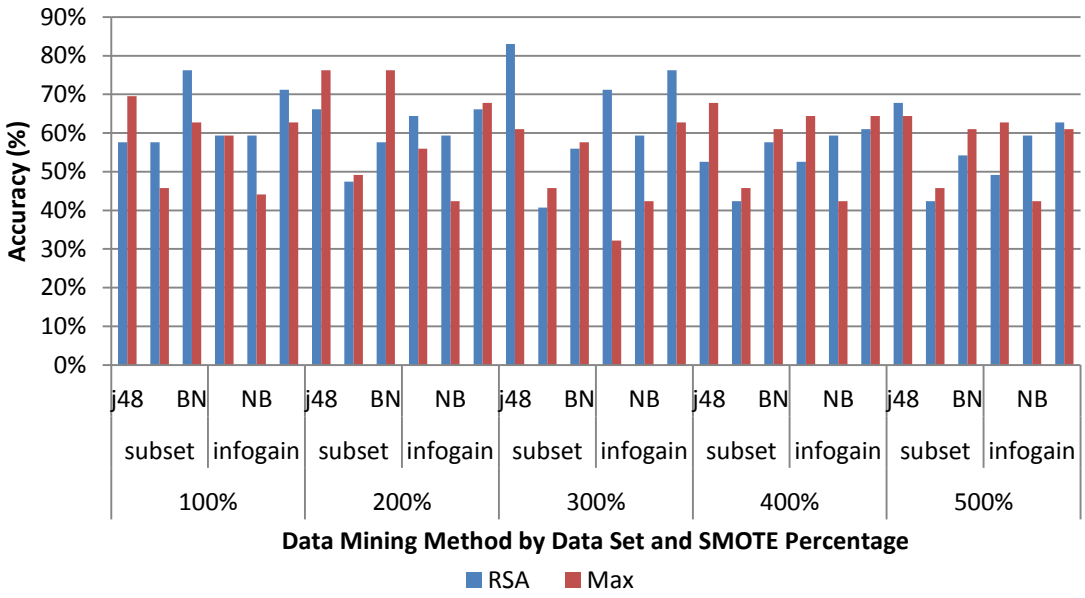

Figure 57 Overall Classification Accuracies using SMOTE when applying After State Features to The Before State by Feature Selection

In this scenario, using the RSA data set, the features selected using the subset evaluator after SMOTE is applied at 300% and 100% are identical. The mining accuracy and sensitivity results are shown in Table 25 (from the j48 classifier) and Table 26 (from the Bayesian Network) respectively. The decision tree for the results presented in Table 25 is illustrated in Figure 58. When SMOTE is applied at 300% the level of accuracy is the best generated this far from all experiments (83.1%). Approximately 84% of successful builds are correctly classified with a TP rating of 0.838. For failed builds 82% of instances are correctly classified with a TP rating of 0.818. There is some confusion displayed within the decision tree, where duplicate nodes occur referencing the average number of constructors per class, weighted methods per class and number of unique operators metrics. One branch of the tree successfully classified 32 failed build instances. This decision branch utilises simple average and count type software metrics with a single dependency metric (instability).

When compared to the best generated mining results in section 4.2.4, using the RSA before state data set, the application of SMOTE has slightly increased the precision and recall values. The second best performing models generated, while adopting the use of SMOTE, contained similar percentages for overall correctly classified instances (around 75%) and the number of correctly classified failed and successful builds slightly varied between experiments.

Table 25 Results for the Before State of RSA Data Set with 300% SMOTE using After State Subset Evaluation and j48 Classifier

# Successful builds correct (incorrect)		**# Failed builds correct (incorrect)**		**Correctly Classified Instances**	**Incorrectly Classified Instances**
31 (6)		18 (4)		83.1%	16.9%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.838	0.182	0.886	0.838	0.861
Failed Build	0.818	0.162	0.75	0.818	0.783
Weighted Average	0.831	0.174	0.835	0.831	0.832

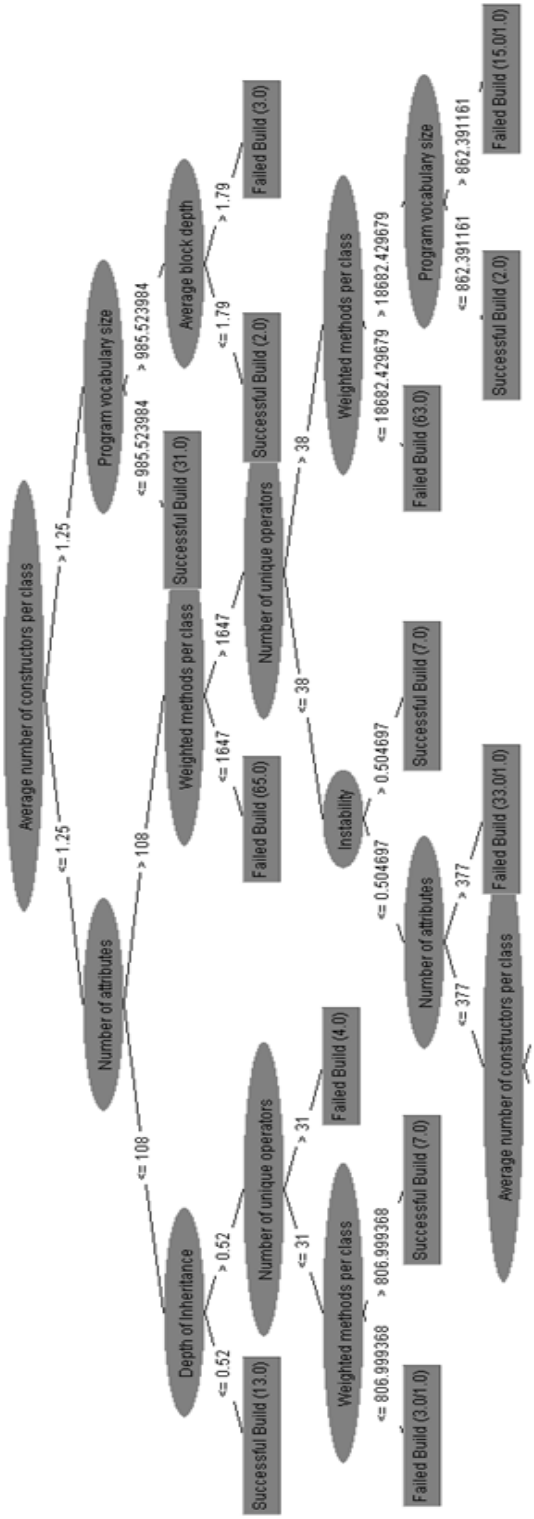

Figure 58 Classification Tree of Before State of RSA Data Set with 300% SMOTE using After State Subset Evaluation and j48 Classifier

Table 26 Results for the Before State of RSA Data Set with 100% SMOTE using After State Subset Evaluation and Bayesian Network Classifier

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
26 (11)		19 (3)		76.3%	23.7%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.703	0.136	0.897	0.703	0.788
Failed Build	0.864	0.297	0.633	0.864	0.731
Weighted Average	0.763	0.196	0.798	0.763	0.767

From all experiments conducted and presented in sections 4.1 and 4.4 there are common trends that have emerged from using combinations of various software metrics. In general terms of identifying which methods are best for feature selection and mining algorithms it is observed that the subset evaluator and the j48 classifier have generated the best results for a majority of mining experiments. While the max data set performed well when investigating frequency feature selection and upon applying after state features selected to before state metrics, the RSA data set has appeared to perform well, in terms of overall accuracy and classification of failed builds. For this reason the RSA data set is carried forward to the next stage of mining.

In the next section social network metrics and their relationships to predicting software build outcomes are explored. This is accomplished through using similar methods from sections 4.1 (in terms of feature selection and data mining methods).

4.9 Data Mining Results for Social Network Metrics

When considering the mining of social network metrics from the Jazz repository, without merging them with any software metric data sets, there are 191 build instances with social network metrics that could be extracted for analysis. Of these builds, there were 120 successful builds, 51 failed builds and 20 warning builds. The same data pre-processing method that was applied to the software metrics data sets is applied to the social network data. Warning builds were treated as failed build instances raising the total number of failed builds to 71. A software build can have one or more work items, therefore the communication captured will inherently overlap from build to build.

Unlike the software metrics there are no before and after states for communication, as the work items themselves are not stored as change sets. Communication metrics emerge over the duration of a software build and the final values calculated from all communication during a build are therefore naturally associated to the after state software metrics. All communication for a builds' work items is taken into account when extracting social network metrics. The social metrics represent the communication of work items, across all Jazz team areas, for each build instance. For this stage there is a total of 36 data mining experiments. This involves the use of 4 data sets of social network metrics time intervals, 3 feature selection methods (no feature selection, Subset Evaluation and Information Gain) and 3 mining algorithms (j48 classification tree, Naive Bayes and Bayesian Network). For all mining experiments, again, 10-cross fold validation is utilised.

Communication metrics are extracted from builds in relation to time, to generate four social metric data sets. The social metric data sets are constructed by selecting the first 25%, 50%, 75% and 100% of the communication that occurred since the previous build similar to the work presented by Wolf et al. (2009). Similar to this previous study it was found that no single communication metric is recognised a being suitable for prediction software builds, however a small combination of social metrics show potential in terms of prediction strength. Unlike the previous study, suitable levels of precision and recall were not obtained using the first 25% of the communication metrics interval. This is mostly likely due to the different rules that have been implemented to construct the network for this research that ensure that the social network metrics can be combined

with the software metric extractions. However, higher levels of precision and recall were obtained through using 100% of the communication metrics for a build. In this set of experiments precision values ranged from 52% to 72% and recall values ranged from 57% to 72%. Presented in Figure 59 are the overall classification accuracies achieved from the 36 mining experiments for communication metrics. In this case no feature selection is coined as No FS, Subset represents Subset Evaluation and Infogain is the Information Gain feature selection filter.

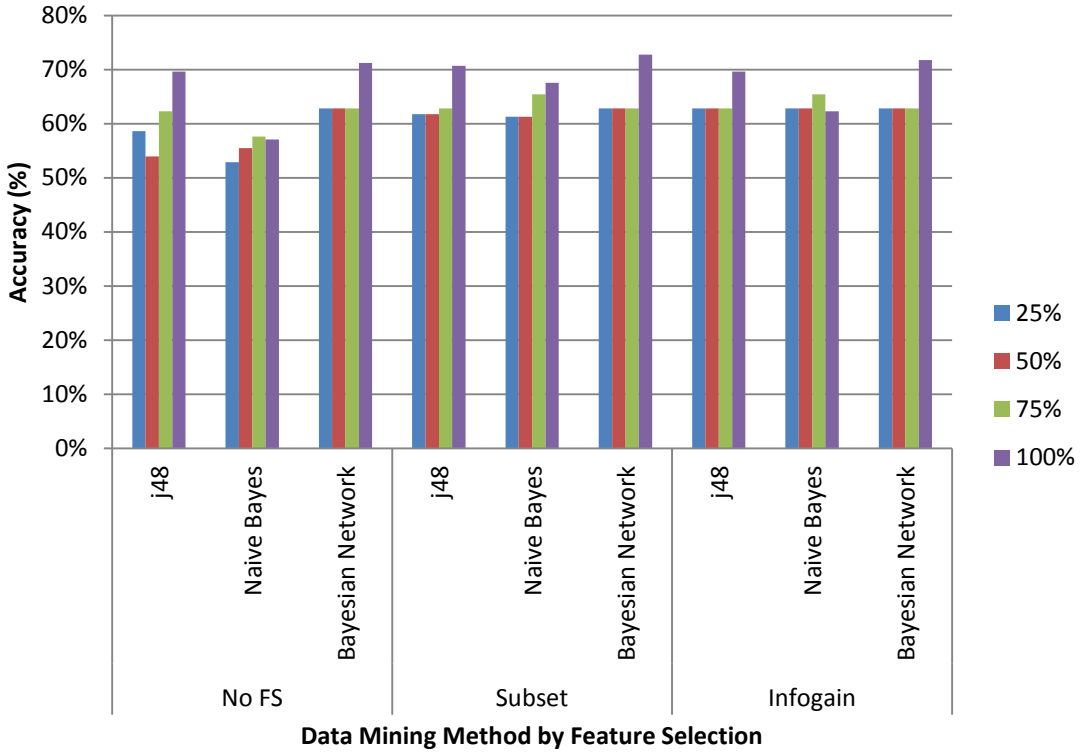

Figure 59 Communication Mining Results by Time Intervals

Similar to the initial software metric mining results it is again observed that failed build instances are more difficult to correctly classify than successful build instances. The best result for this stage was produced using the subset evaluator and the Bayesian Network classifier that is shown in Table 27. From this result approximately 86% of successful social collaborations were correctly classified with a TP rating of 0.858. However, in this case only 50% of failed builds were correctly classified, with TP rating of 0.507. The Bayesian Network classifier utilised the number of change sets metric to predict build outcomes. When the Information Gain filter is applied, in addition to the number of change sets metric the Group InOut-Degree Centrality metric is also selected as being a significant predictor.

Table 27 Results for the (100%) Social Network Data Set with Subset Evaluation Feature Selection and Bayesian Network classification

# Successful builds correct (incorrect)		# Failed builds correct (incorrect)		Correctly Classified Instances	Incorrectly Classified Instances
103 (17)		36 (35)		72.8%	27.2%
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Successful Build	0.858	0.493	0.746	0.858	0.798
Failed Build	0.507	0.142	0.679	0.507	0.581
Weighted Average	0.7277	0.3624	0.7214	0.727	0.7175

In an attempt to increase the accuracy for predicting failed builds using social network metrics SMOTE is again applied to the data set in increments of 100%. Unlike the software metrics it was found that this method did not improve classification results. The communication metrics data set aggregations were then merged with the RSA data set and the mining experiments were executed again using the 3 feature selection methods and the 3 mining methods. It was observed that no communication metrics were found to be significant factors and did not increase accuracy in terms of correctly classified instances or sensitivity measurements in this case when the software metrics and the social network metrics are combined into a single data set. These experiments were also applied to the Max software metric data set aggregation and again yielded similar results. It appears that it is best to keep software metrics and social network metric models separate for data mining.

Whilst the previous experiments have focused on utilising both *before* and *after* states of the source code, there is no reason why the approach could not be applied incrementally through the build cycle. This provides software teams an indication of whether exposure to risk is increasing or decreasing as a result of the changes to the source code that were being made. This is explored further in section 4.10 where the results for software metrics and communication metrics, presented as separate data streams, are presented. Since the j48 classifier has commonly produced the highest levels of classification accuracy using traditional mining methods the Hoeffding tree is utilised in a data stream mining context.

4.10 Data Stream Mining Results

In this section of results the outcomes of the data stream mining are presented by illustrating the trend of overall classification accuracy as well as accuracies specific to successful and failed builds predictions. The sensitivity ratings are also displayed over time, showing true positive, false positive, precision, recall and f-measure trends.

Due to the limited size of the data set the default value of grace period for the Hoeffding tree is lowered from the default 200 instances. At the beginning of this set of experiments various grace periods were trialled to see whether or not the beginning set of training instances had an effect on the final classification accuracy. Sampled training grace period values were trialled using 5, 20, 50 and 100 instances. The results indicated that if the grace period is set too high it will result in a loss of final accuracy, as the initial model built is over fitted to the data. In terms of results for sections 4.10.1 and 4.10.2 a grace period of 20 was found to generate the highest level of accuracy for 198 instances. The split confidence is 0.05 and the tie threshold option is set to 0.1. For the results presented in section 4.10.4 the grace period is set to 200 for 1990 instances (after the application of SMOTE), the split confidence is 0.5 and the tie threshold option is set to 0.05.

4.10.1 Software Metrics as Evolving Data Streams

Figure 60 presents the trend of overall classification accuracy for builds over time using the Hoeffding Tree method for the RSA software metrics data set. For this stage the after state software metrics are included. This is to ensure the outcomes of the software metric prediction comparable to the outcomes of social network metric predictions. The 179 instances are executed as time series of data streams. It is observed that after approximately 100 builds the prediction accuracy begins to stabilize and improve. This is to be expected because at the start of the training process insufficient instances exist, resulting in model under-fitting. The final overall prediction accuracy is 72.4%, with a weighted precision value of 0.748, which is a considerable improvement from the earlier prediction accuracies which start around 47%, with a weighted precision value of 0.476. The overall trend shows that, as more instances are trained, the classification accuracy steadily improves but appears to reach an asymptotic value.

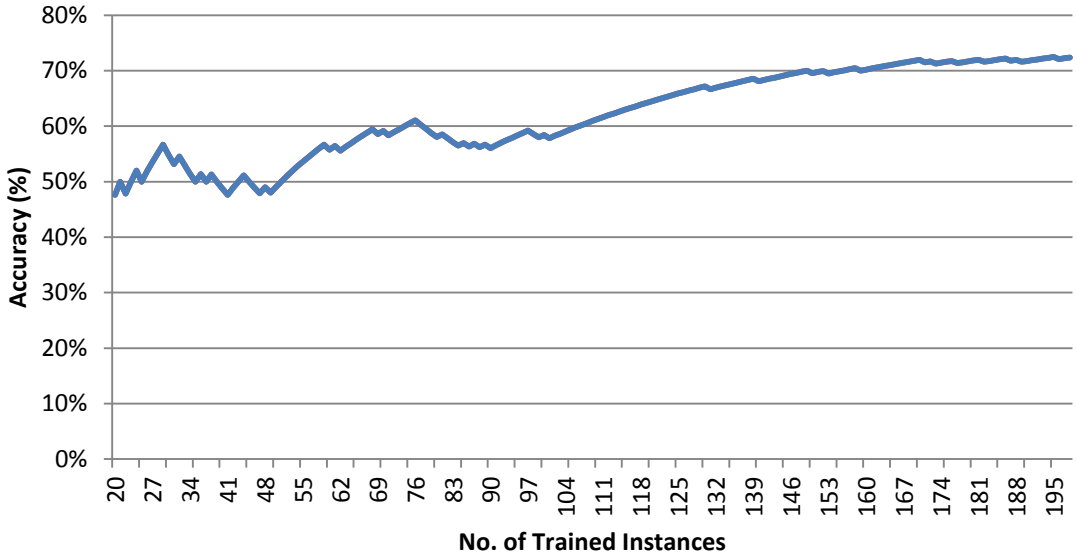

Figure 60 Hoeffding Tree Overall Classification Accuracy for RSA After State

To expand on the overall accuracy findings, Figure 61 presents the trend of classification accuracy for successful builds for the RSA after state. As the actual outcome of each build was known then it is possible to report the outcomes of the classification in more detail in terms of true positive and false positive measures. These measures are presented in Figure 62 for successful builds. For the case of successful builds, the true positive rate is the proportion of successful builds that have been correctly classed as successful builds. The false positive rate is proportion of failed builds that have been incorrectly misclassified as successful builds. At the beginning of the time series, using 20 training instances (11 successful and 9 failed builds), the true positive rate was 0.416. Towards the end of the time series successful builds were correctly predicted with 81.1% accuracy.

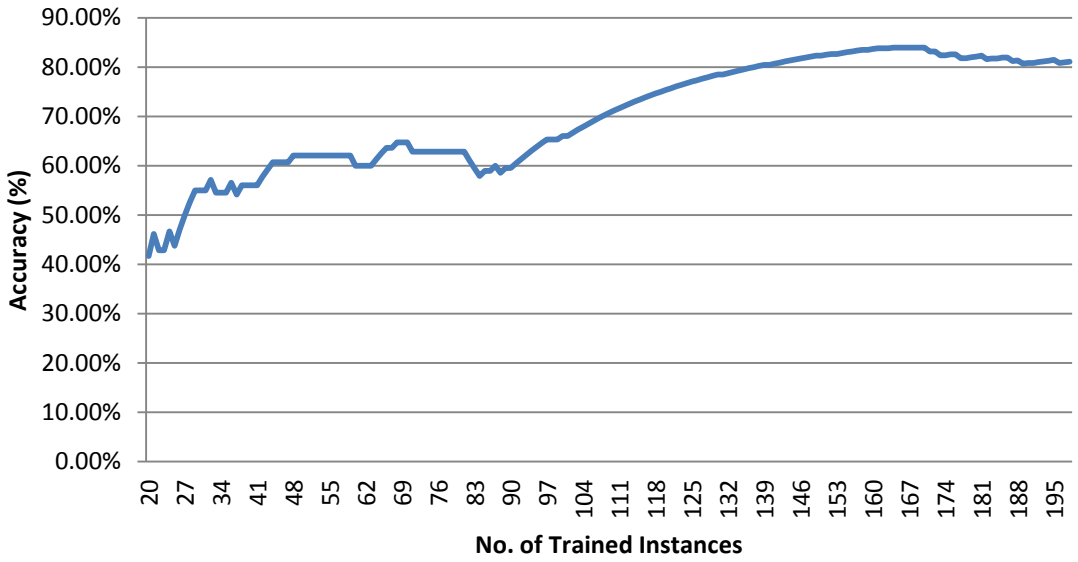

Figure 61 Hoeffding Tree Classification Accuracy for Successful Builds for RSA After State

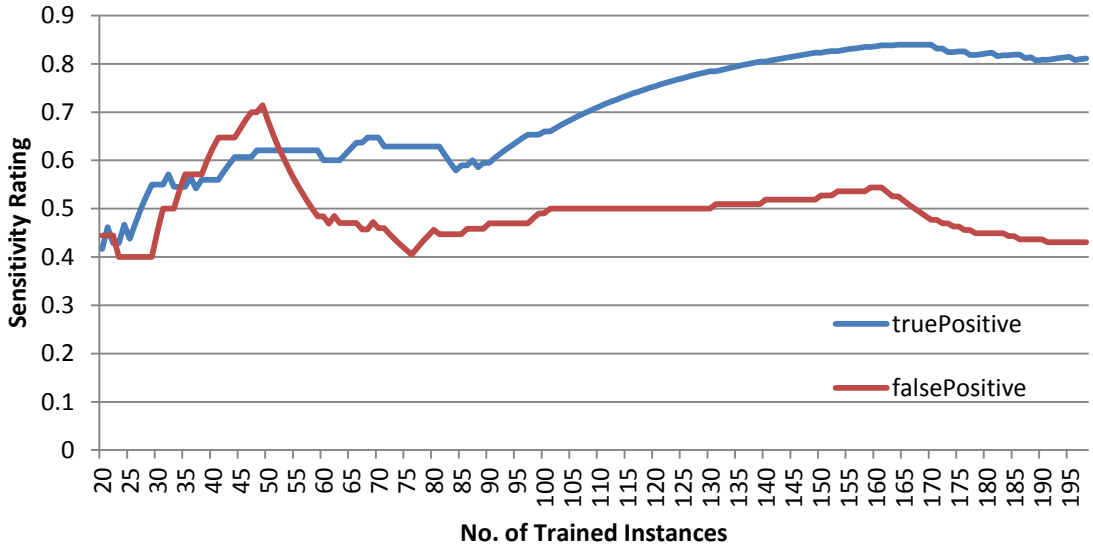

Figure 62 Hoeffding Tree Sensitivity Measurements for Successful Builds for RSA After State

The sensitivity measures indicate that there is a period of instability for correctly predicting success. As with the overall prediction accuracy, after approximately 100 builds there is a gradual increase in the ability to correctly classify successful builds. At approximately 170 builds there is a minor drop off and the true positive rate hovers at around 0.8. The false positive measure supports results from previous results that correctly classifying failed builds is harder to achieve. The percentage of failed builds

classified as successful builds hovers at around 50% up until 160 builds where a minor improvement is seen.

Presented in Figure 63 is the trend of accuracy for failed builds over time and the corresponding sensitivity measurements are shown in Figure 64. When comparing these results to previous mining experiments the same issue is encountered, where failure is more difficult to predict than success. At the beginning of the time series failed builds were predicted with an accuracy of 55.6%. At the 160 build mark there is an improvement in ability to correctly classify failed builds and after training the model on all 198 instances there appears to be a steady prediction accuracy of 57.0% for failed builds. Unlike the overall prediction accuracy and successful build accuracy, the trend for predicting failure does not appear to overly improve with more instances. It is difficult to predict what may occur as by definition, a data stream is unbounded in nature. Whilst it appears that the prediction accuracy has become asymptotic it may be that any improvement is periodic in nature and may require a sufficient number of builds to reinforce the statistical significance of the current window of data used to detect concept drifts.

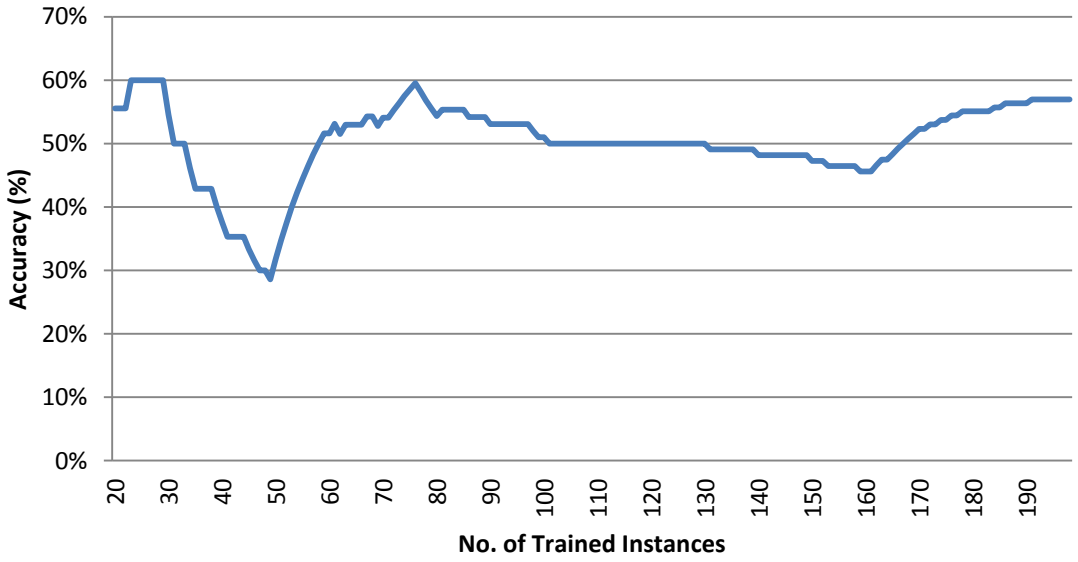

Figure 63 Hoeffding Tree Classification Accuracy for Failed Builds for RSA After State

In Figure 64 an interesting result is observed when classifying failed build instances. In this time series after about 100 training instances the trend for false positive test ratings

stabilize and start to decrease. In a training set with few failed instances, there are more instances that are predicted as failures incorrectly than there are instances of failed builds predicted correctly. However, as the stream progresses and more failed signatures are seen the false positive rate drops substantially.

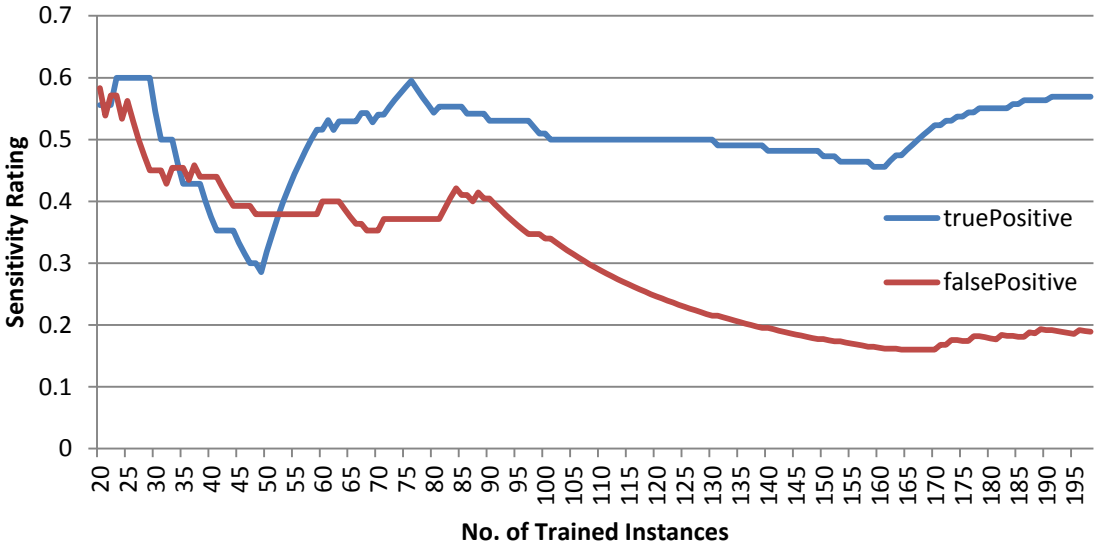

Figure 64 Hoeffding Tree Sensitivity Measurements for Failed Builds for RSA After State

Presented in Figure 65 is the final decision tree generated, from the data stream mining process, using the Hoeffding tree method. The leaves of the tree show the predicted outcome and the numeric values represent the votes used in the majority vote classifier. The value on left represents the weighted votes for failed builds and the value on the right represented the weighted votes for successful builds (i.e. failed builds | successful builds).

The tree size is small with a depth of just 2. Here it is observed that only 2 attributes out of the original 46 are used to classify instances. At the root of the tree the Average Number of Attributes per Class metric is presented, indicating that this metric has a high impact on the classification of results. The second significant metric is the Number of Interfaces. Failed builds are associated with a high number of attributes per class. This is intuitive because the higher the number of attributes the more complex a class may become. If the Average Number of Attributes metric value is low and the Number of Interfaces is high then this model also predicts a failed build. Too many interfaces have

the potential lead to a design defect known as the "Swiss Army Knife" anti-pattern (Moha, Guéhéneuc & Leduc, 2006), where a complex class uses a high number of interfaces to address many different requirements. This problem may be due to rapid changing, or misinterpretation of system requirements. Interestingly, the final decision tree shows that failures are predicted with a much greater degree of confidence on the basis of the Average Number of Attributes per class feature when compared to the Number of Interfaces feature.

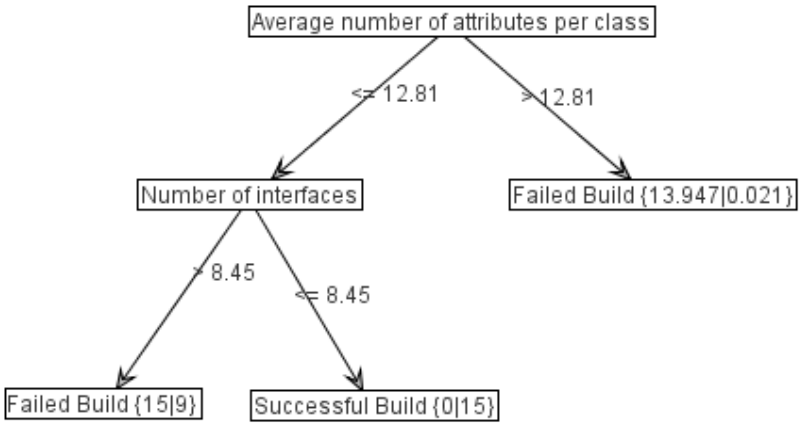

Figure 65 Final Hoeffding Tree for After State Software Metrics

From streaming data over time the data can become more dynamic as it changes and evolves (Baena-García, et al., 2006). To capture when such evolutions occur it is possible to apply concept drift detection during the data stream mining process using approaches such as ADWIN. It is observed that through learning via the 198 instances in the data stream that 50 concept drifts occur. Figure 66 and Figure 67 shows the concept changes that occur with respect to the two predictor features that the Hoeffding tree model uses. In both diagrams, the dotted line indicates when a concept drift is detected, irrespective of whether the change was triggered by a change in the particular metric. A step change in the cumulative drift detection indicates that a concept drift has been detected. As is evident from the figures, the concept changes are first fairly chaotic, with large metric values triggering changes as particular builds are added to the data stream. This response is not unexpected as the relatively small data set makes the emerging model sensitive to extreme changes in any given metric. Over time, the concept drift rate slows down, reflecting greater stability in the data streaming in, enabling overall classification accuracy to improve without the need for structural changes in the model.

The subtle nature of the concept changes would indeed make it difficult, if not impossible for a human designer to decide when such changes occur without the use of an automated change detector such as ADWIN. Apart from the need to update the decision tree model when such changes occur, the detection of such changes are useful in their own right to the human user as they represent changes in patterns, arising from changes in the software development environment that would be of use to developers and project managers.

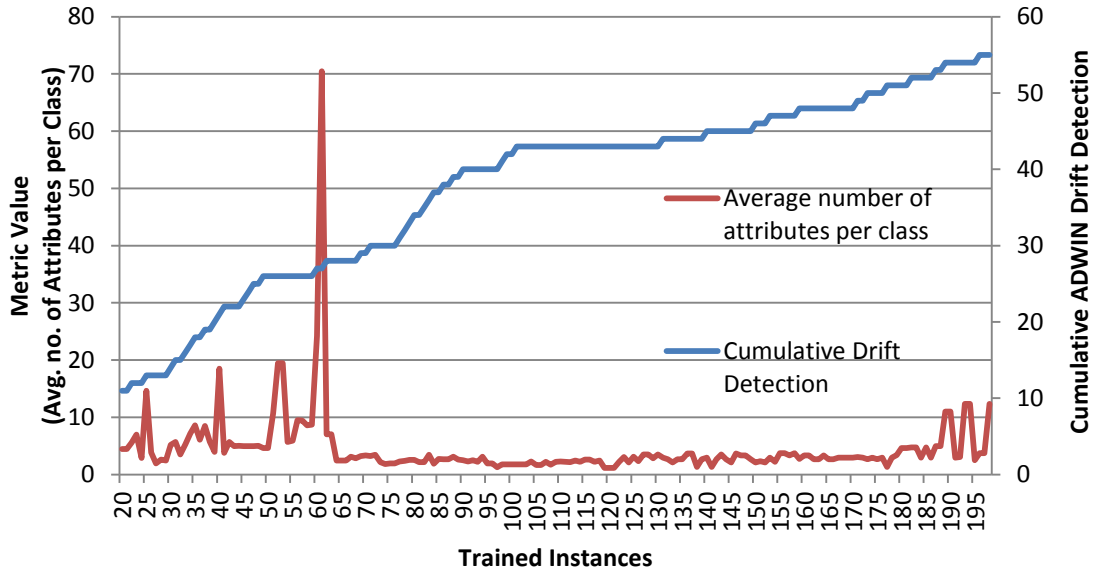

Figure 66 Trajectories of the Average Number of Attributes per Class feature and Cumulative Drift Count over time

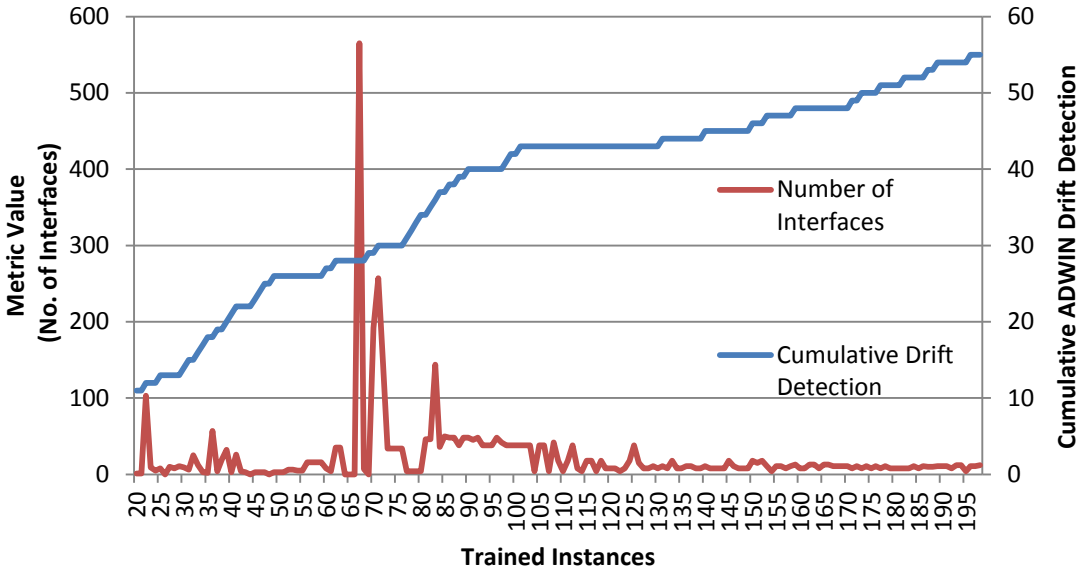

Figure 67 Trajectories of the Number of Interfaces feature and Cumulative Drift Count over time

Application of the Hoeffding tree analysis to the Jazz data stream results in the classification model shown in Figure 65. To fully understand the application of the Hoeffding tree approach it is important to analyse the emergence of this model, not just the final model itself. By examining Figure 60 it would seem reasonable to conclude that the minimum number of instances required to develop a classification tree that is reasonably stable would be around 100 instances. It is at this point that the prediction accuracy starts to stabilize and show a trend to improving asymptotically. However, an examination of the Hoeffding tree analysis at this point shows that no actual decision tree has been generated by the model at this point in time. In fact, the Hoeffding tree approach has not classified a single feature that has sufficient predictive power to use effectively. The approach is therefore attempting to classify a new build in the data stream against the majority taken over all instances and all attribute values. So, for example, if there were 60% successful builds, then all builds would be labelled success. This is an exceptionally degenerate case where severe model under-fitting is occurring due to lack of training examples resulting in no clear predictors. The Hoeffding tree approach identifies an actual decision tree only after 160 builds. This first decision tree identifies only a single attribute against which to classify a given build and the resulting decision tree is shown in Figure 68.

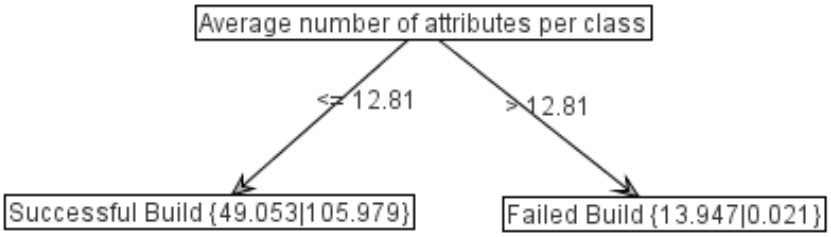

Figure 68 Initial Hoeffding Tree Model using Software Metrics

Finally in addition to the Hoeffding tree and concept drifting techniques a k-NN algorithm is trained using a similar process where the number of training instances are the same used for the Hoeffding tree. The final prediction accuracies of the Hoeffding tree and the k-NN are shown in Table 28, where $k = 5$. From this result the Hoeffding tree has performed better in terms of overall correctly classified instances for both successful and failed build outcomes when compared to the k-NN method.

Table 28 Final Prediction Accuracies of Hoeffding Tree and k-NN models For RSA After State

	Successful Builds		**Failed Builds**		**Overall**
	Correctly Classified Instances	**Incorrectly Classified Instances**	**Correctly Classified Instances**	**Incorrectly Classified Instances**	**Correctly Classified Instances**
Hoeffding Tree	103	24	41	31	72.6%
k-NN	93	23	31	32	69.3%

In this section the application of data stream mining was explored in terms of predicting build outcomes over time. The Hoeffding tree, ADWIN concept drifting and k-NN methods were used to explore how software metrics evolved over time. In the next section these techniques are applied to the communication network metrics.

4.10.2 Communication Metrics as Evolving Data Streams

Communication metrics were extracted for each build. The same data stream mining method that was used to generate the results in section 4.10.1 was applied. It was found at the end of the data stream mining process that 63% of instances were correctly classified. The communication metrics classified as being significant predictors of a

builds' outcome include the group InOut Degree Centrality and the Total number of change sets.

It is observed that communication metrics do not provide better levels of accuracy for predicting build outcomes when compared to software metric mining results. The trends across all levels of communication extractions are very similar, to summarise the findings of the results for 100% of metrics suffice. Presented in Figure 69 is the classification accuracy for the data stream mining results for 100% of the communication network metrics for builds. Similar to the stream mining results from software metrics the trends appear to begin to stabilise after approximately 90-100 build instances.

Figure 70 presents the corresponding classification accuracies over time for successful builds and the sensitivity ratings over time for successful builds is presented in Figure 71. For predicting successful builds using communication metrics the classification accuracy appears to steadily increase over time after training on approximately 90 instances.

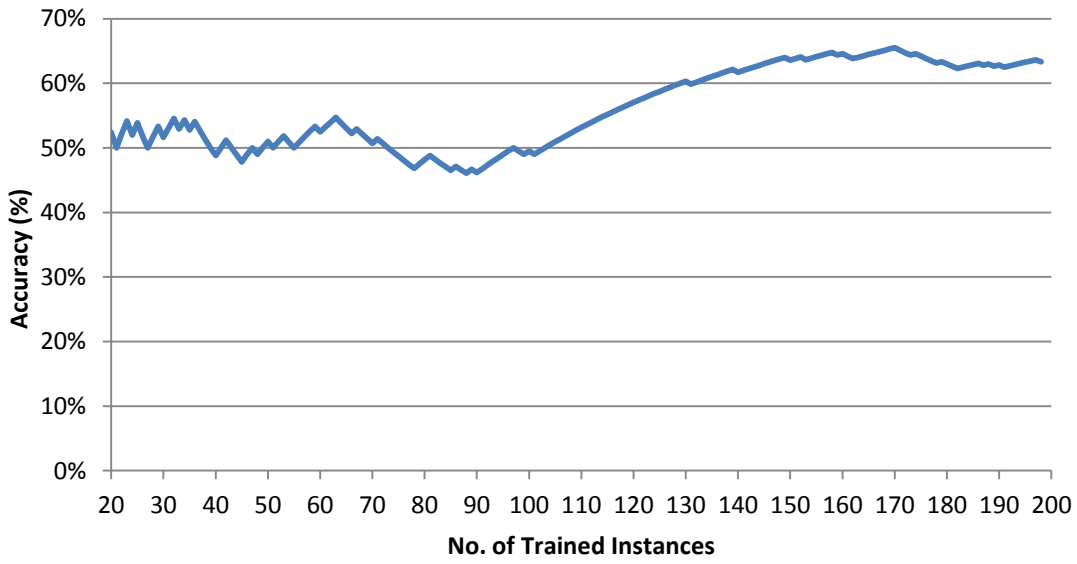

Figure 69 Hoeffding Tree Classification Accuracy with 100% of Communication Metrics of Builds

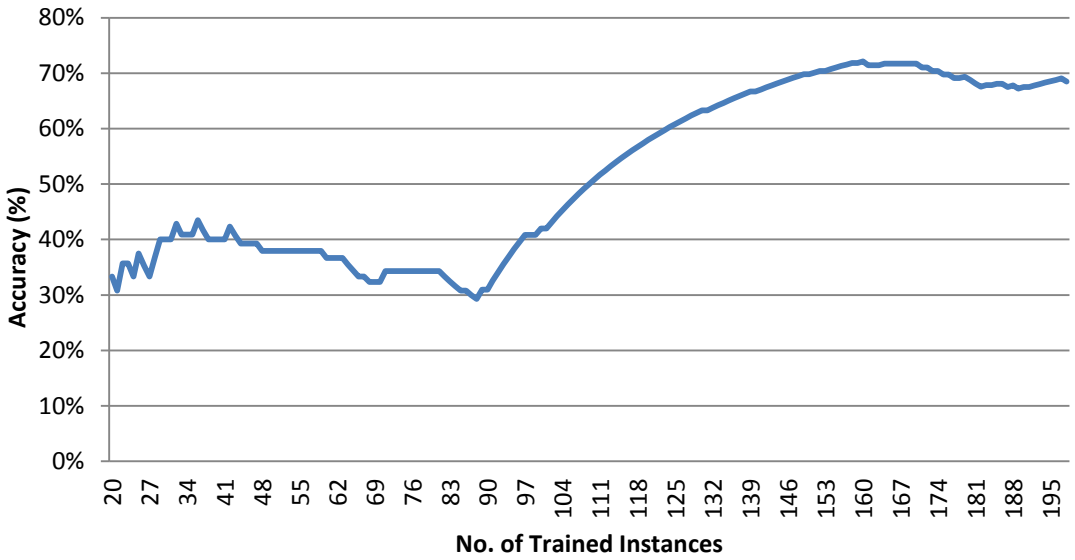

Figure 70 Hoeffding Tree Classification Accuracy for Successful Builds with 100% of Communication Metrics

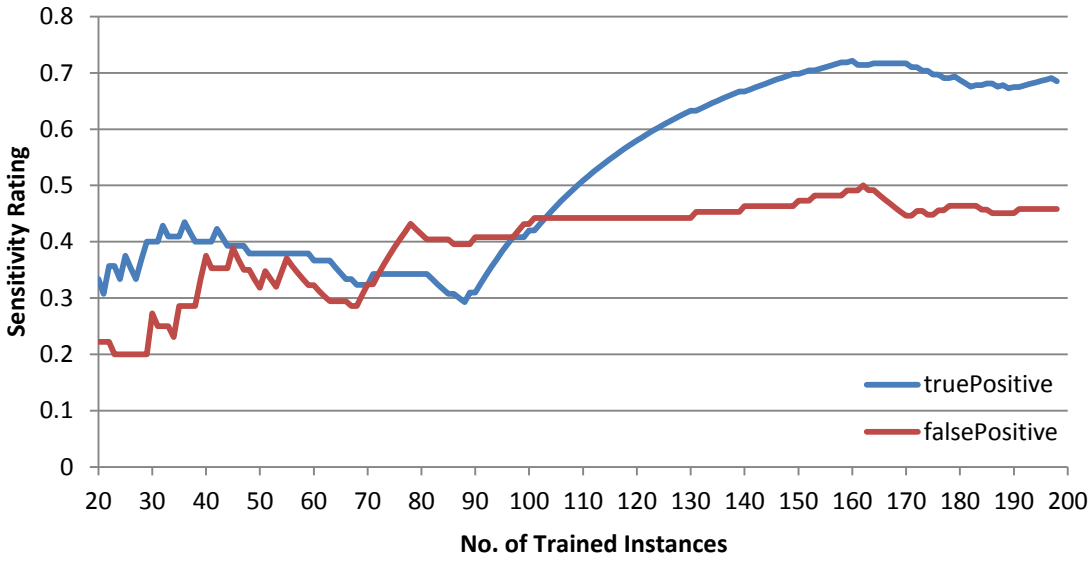

Figure 71 Hoeffding Tree Sensitivity Ratings for Successful Builds with 100% of Communication Metrics

Figure 72 presents the corresponding classification accuracies for failed builds and shown in Figure 73 are the sensitivity ratings for failed builds over time. Over the duration of the simulated data stream, the proportion of successful builds that are correctly classified by the decision tree steadily increases to around 70%. Over the same simulated period the proportion of failed builds that are correctly classified drops from a very high initial value to around 54%. The initial values are a result of the lack of data causing model under-fitting.

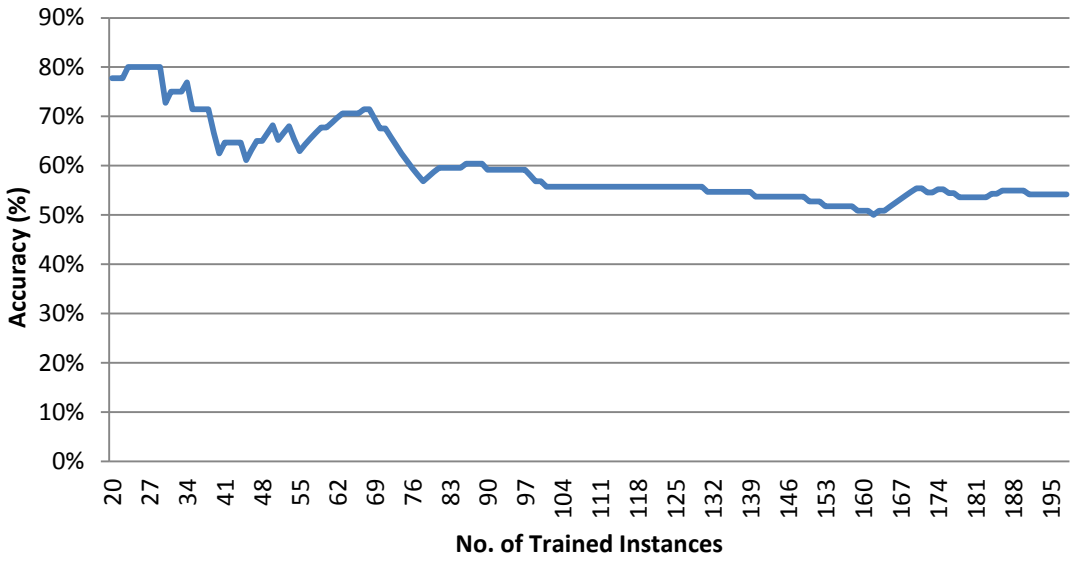

Figure 72 Hoeffding Tree Classification Accuracy for Failed Builds with 100% of Communication Metrics

The outcome of this data stream mining experiment supports observations made from using software metrics where it is significantly more challenging to identify a failed build than it is a successful one. In this case it appears that the developer communication metrics are equally as effective in this regard when compared to source code metrics. From the false positive data it is clear that there is a significant problem in failed builds being misclassified as successful builds.

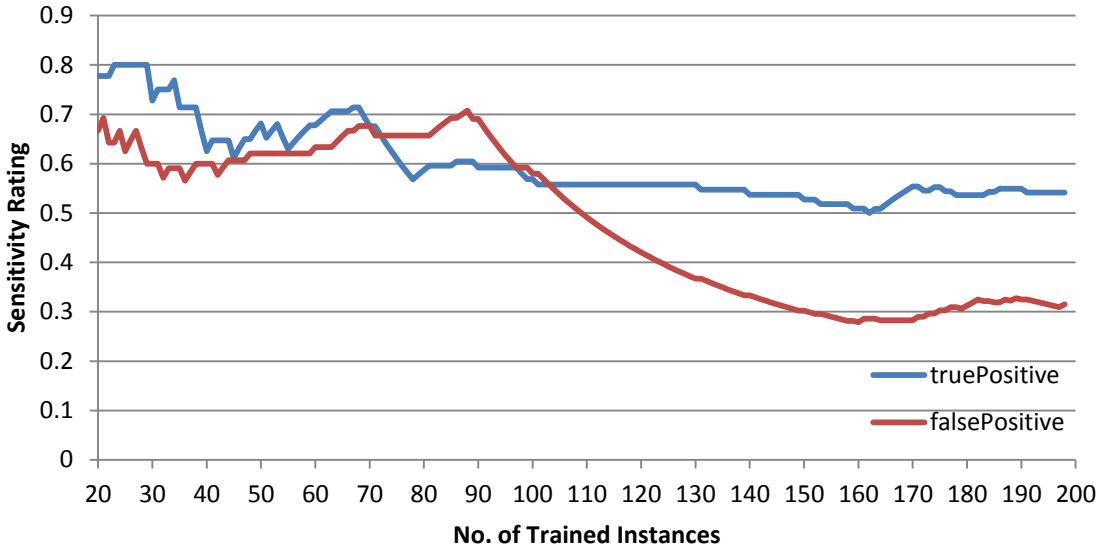

Figure 73 Hoeffding Tree Sensitivity Ratings for Failed Builds with 100% of Communication Metrics

Figure 74 presents the final decision tree using the Hoeffding Tree stream mining technique on social network metrics for build instances. It is observed that this is a small tree with the depth of 2. There is also no confusion or repeated metrics appearing within its branches. The tree is composed of two metrics, the group InOut degree centrality and edge group betweenness centrality. The classification tree appears to be intuitive in terms of the outcomes it presents. At the root of the tree the GroupInOutDegreeCentrality metric is found and if this value is higher than 0.36 the build result is predicted to be a failure. This indicates that if there are increases in the amounts of incoming and outgoing messages between nodes of a network then there may be a potential problem that needs to be resolved.

The addition of EdgeGroupBetweennessCentrality as a new metric has an interesting effect in that it refines the classification of successful builds only and does not improve the classification of failed builds at all. Of the 39 builds used in this incremental change, 15 of them are failed builds that are misclassified as successful builds by the addition of the new metric.

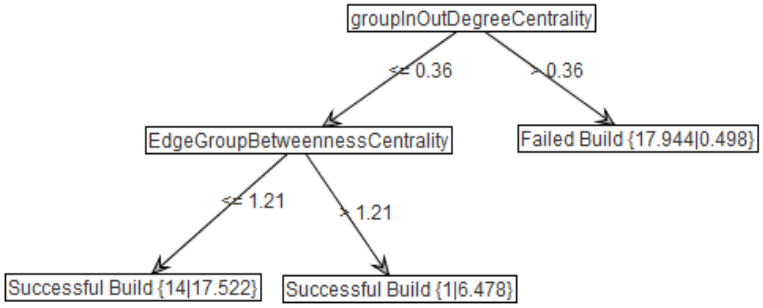

Figure 74 Final Hoeffding Tree for 100% of Social Network Metrics

It is observed that through learning process of 198 instances that 73 concept drifts occur. The starting variance value was 0.25 which remained stable ending with a value of 0.24. The starting estimation value was 0.476 and over time to 0.23. Similar to the software metrics, when comparing the changes of group InOut degree centrality and the edge betweenness centrality metrics to when a concept drift occurs there are conceptual drifts that occur when metrics fluctuate. This is presented in Figure 75 and Figure 76 respectively.

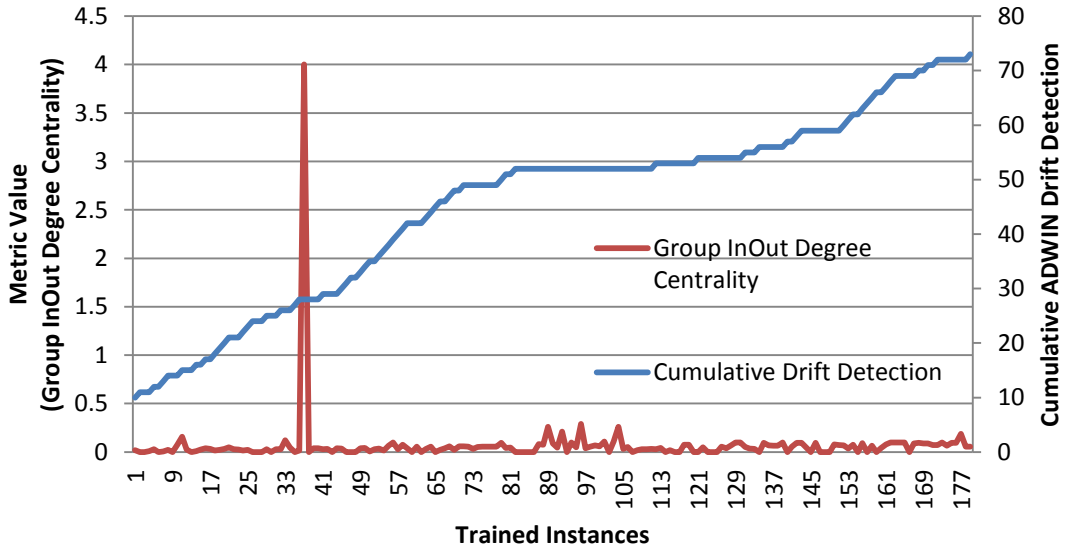

Figure 75 Trajectories of Group InOut Degree Centrality and Cumulative Drift Count over time

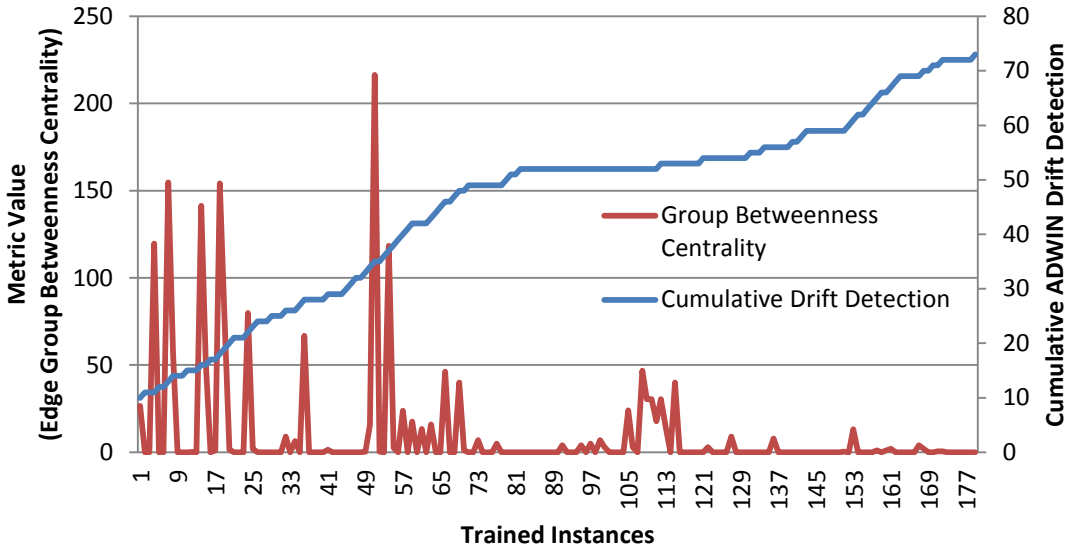

Figure 76 Trajectories of Edge Group Betweenness Centrality and Cumulative Drift Count over time

A k-NN algorithm is trained using a similar process where the number of training instances are the same that are used for the Hoeffding tree. The final prediction accuracies of the Hoeffding tree and the k-NN when applied to communication network metrics are shown in Table 29, where k = 5. Similar to the software metrics the Hoeffding tree has performed better in terms of overall correctly classified instances for both successful and failed build outcomes when compared to the k-NN method.

Table 29 Final Prediction Accuracies of Hoeffding Tree and k-NN models for 100% of Communication Metrics

	Successful Builds		Failed Builds		Overall
	Correctly Classified Instances	**Incorrectly Classified Instances**	**Correctly Classified Instances**	**Incorrectly Classified Instances**	**Correctly Classified Instances**
Hoeffding Tree	87	40	39	33	63.3%
k-NN	80	36	23	40	57.5%

This section has shown the mining results for both the RSA after state data set and communication metrics (at their various extractions times) in a data streaming context using the Hoeffding tree classifier, ADWIN and k-NN methods.

4.10.3 Software Metrics and Communication Metrics as Evolving Data Streams

Figure 77 presents the trend of overall classification accuracy for builds over time using the Hoeffding Tree method for the merged RSA After State software metrics and 100% of communication metrics data set. The time series is tested and trained with 179 instances. It is observed again after approximately 90 builds the prediction accuracy begins to stabilize and improve. This is as suspected because at the start of the training process there are not enough instances to substantially model the outcomes based on the data available and as a result over-fitting occurs. When using a combination of software and social metrics it has been found that identical classification accuracies were generated when compared to using software metrics. For instance the same final overall prediction accuracy is 74.84%, which was the same level of accuracy to that generated by using only the RSA After State metrics. The overall classification accuracy at the beginning of the data stream mining was also the same, starting at 47.6%.

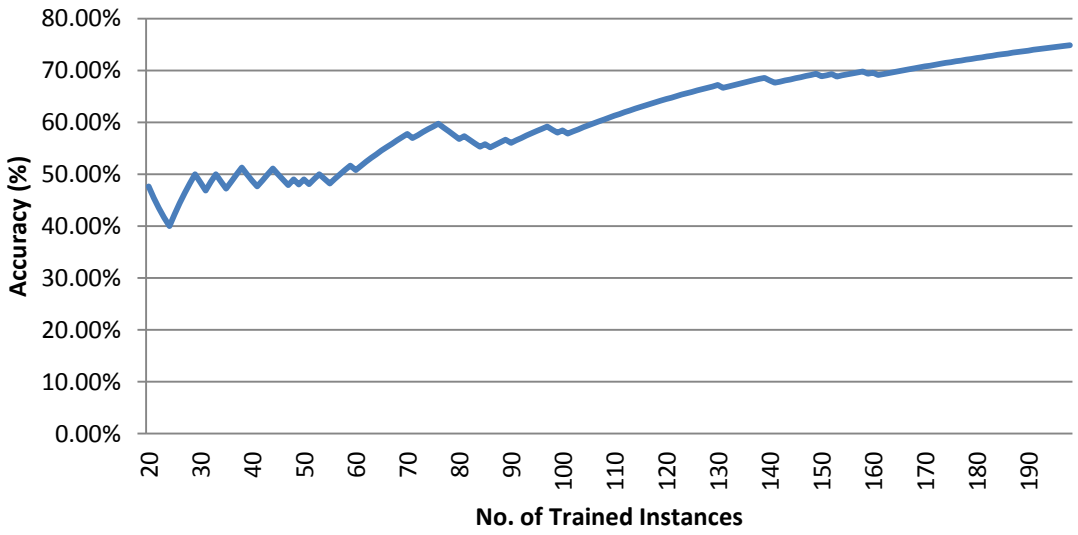

Figure 77 Hoeffding Tree Overall Classification Accuracy for RSA After State and 100% of the Social Network Metrics

To expand on the overall accuracy findings, Figure 78 presents the trend of classification accuracy for successful builds for the RSA after state and 100% of the

communication metrics. The sensitivity measurements (true positive and false positive values) for successful builds over time are also presented in Figure 79. Again it is observed that after approximately 90 builds the prediction accuracies appear to stabilise and steadily increase over time. At the beginning of the time series, using 20 training instances (11 successful and 8 failed builds), the prediction accuracy for successful builds is 25% (lower than the accuracy of only using the RSA software metrics). Towards the end of the time series successful builds were correctly predicted with a 85.0% accuracy (identical to the RSA software metrics outcome). The sensitivity ratings show the over-fitting occurring from training on only 100 instances, as the false positive dramatically increase from around 50 to 100 instances, similarly to the RSA software metrics results.

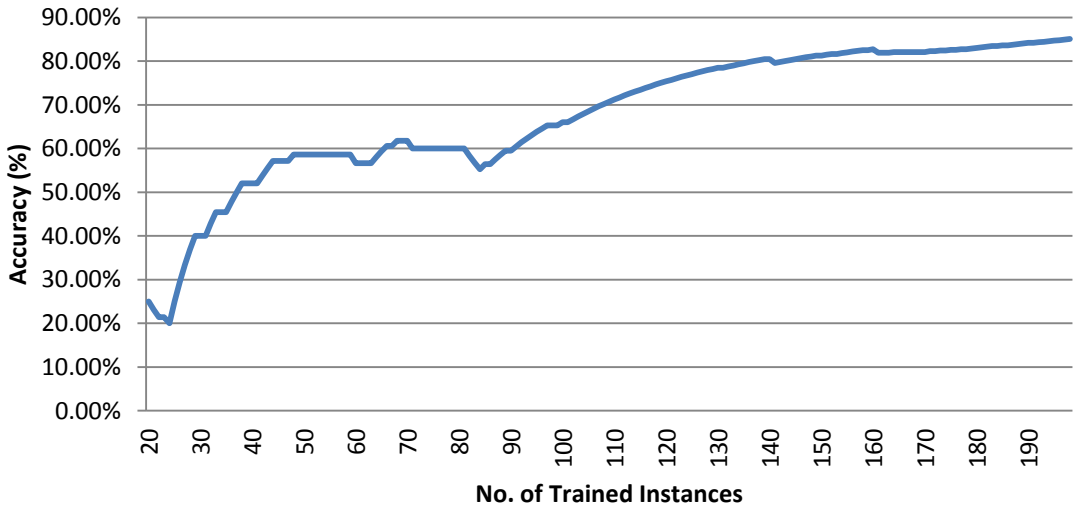

Figure 78 Hoeffding Tree Classification Accuracy for Successful Builds for RSA After State and 100% of the Social Network Metrics

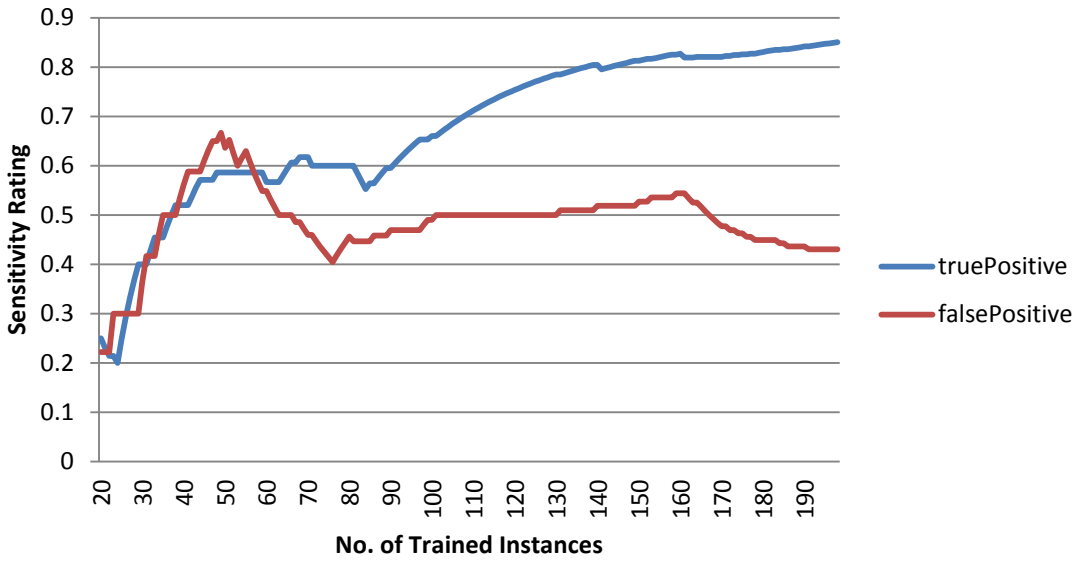

Figure 79 Hoeffding Tree Sensitivity Measurements for Successful Builds for RSA After State and 100% of the Social Network Metrics

Presented in Figure 80 is the trend of accuracy for failed builds over time and the corresponding sensitivity measurements are shown in Figure 81. Comparing these results to previous mining experiments, again, failure is more difficult to accurately predict than success. At the beginning of the time series failed builds were predicted with an accuracy of 77.8%. After training the model on 175 instances there appears to be a steady prediction accuracy of 57.0% for failed builds. The final accuracy for predicting failed builds using a combination of software and social network metrics is identical to the results obtained from using only software metrics. Again it is observed that the trend for predicting of failure does not appear to overly improve with more instances.

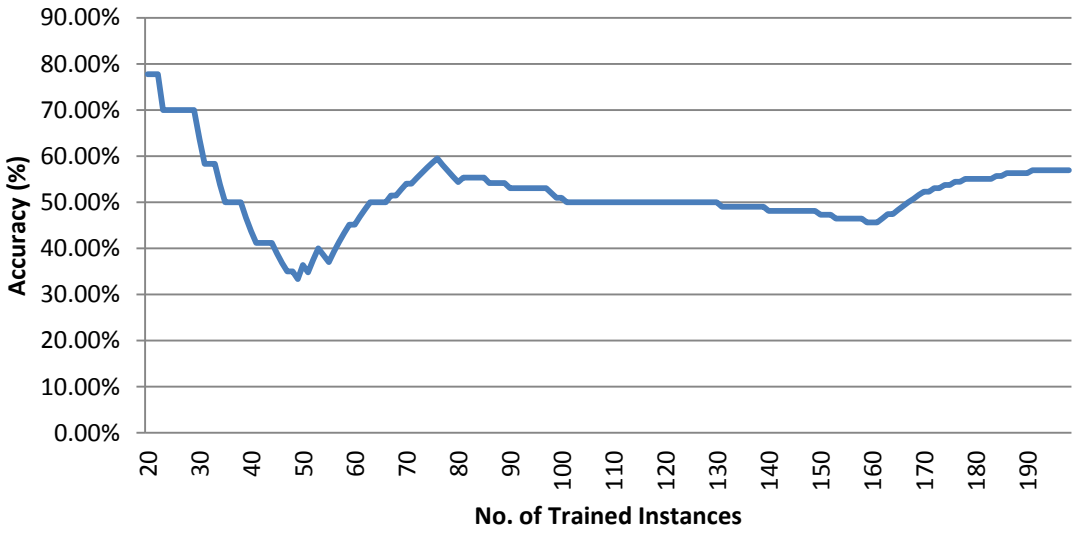

Figure 80 Hoeffding Tree Classification Accuracy for Failed Builds for RSA After State and 100% of the Social Network Metrics

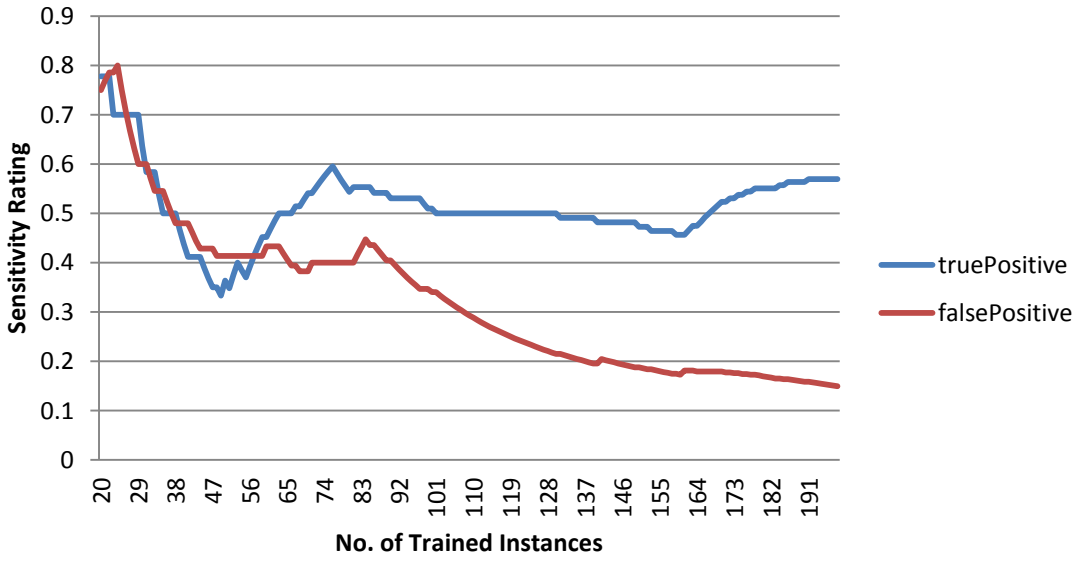

Figure 81 Hoeffding Tree Sensitivity Measurements for Failed Builds for RSA After State and 100% of the Social Network Metrics

Presented in Figure 82 is the final decision tree generated from the data stream mining process using the Hoeffding tree method. The tree size is small with a depth of 2. Here it is observed that only 2 attributes out of the original 58 (software metrics and social network metrics) are used to classify instances. At the root of the tree the Group inOut Degree Centrality metric is presented. The second significant metric again is the Maintainability index. Comparing this decision tree to the tree generated using only the

RSA software metrics data set, it appears that the average number of attributes per class metric has been merely switched for the group InOut degree centrality metric. In addition to this the Maintainability index threshold values for determining build success and failure (268.11) remains the same. When comparing this tree to the tree generated from only social network metrics, it appears that the edge betweenness centrality metric is exchanged for the Maintainability index metric. Again the threshold values for determining build failure (>0.36) using the group InOut Degree Centrality remain the same. Due to there being no increased instances of successfully predicting failed builds and similar levels of sensitivity have been generated, SMOTE is not applied on the merged data set.

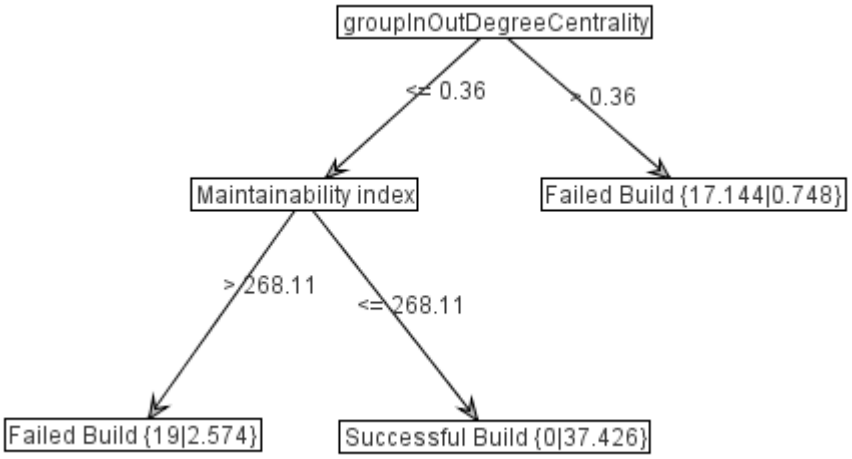

Figure 82 Final Hoeffding Tree for RSA After State and 100% of Social Network Metrics

4.10.4 Simulating Instances: Application of SMOTE

In this section SMOTE is applied to two data sets 1) the RSA after state and 2) 100% of communication metrics. In this set of experiments SMOTE is applied differently from the results presented in section 4.8 where various levels of SMOTE had been applied to increase the number of failed build instances. In this case SMOTE is applied twice to each data set to preserve the naturally occurring distribution ratio between classes. In the first application the number of failed build instances is increased and the second application increases the number successful build instances. In doing so each data set size is increased from 198 to 1990 instances. Within these instances 720 are failed builds and the remaining 1720 instances are successful builds. The goal of this set of experiments is to perform a "what-if" analysis to see what may occur to the data stream mining results if there is a larger data set available. This section of results is based on

the assumption that the new instances generated using SMOTE is a fair representation of real-world data, over an extended period of stability.

The results are presented in two sections: 1) the software metrics based on the RSA after state data set and 2) the results based on 100% of communication metrics. The Hoeffdings tree grace period parameter is set to 200 for this set of experiments.

Presented in Figure 83 is the classification accuracy for the data stream mining results for RSA after state metrics for builds. After the model is trained on 200 instances the classification accuracy at the start of the time series, is 65.2% and at the end of the stream the accuracy was 80.35%. The average overall accuracy for the entire time series was 70%. The general trend of accuracy, although higher than the result presented in section 4.10.1 (more specifically Figure 60), has great similarity. This indicates that the there is potential for the accuracy of prediction to improve as more real data emerges.

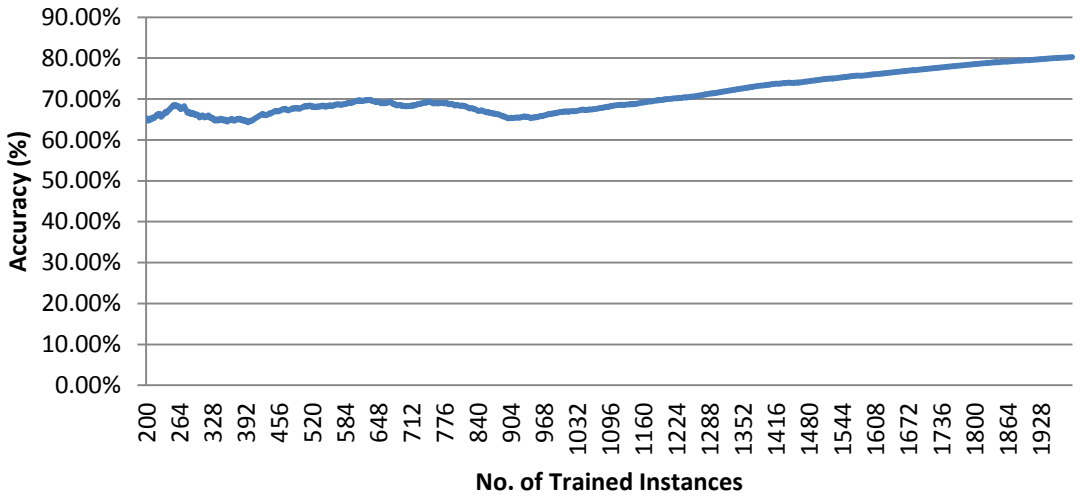

Figure 83 Hoeffding Tree Overall Classification Accuracy for RSA After State (with SMOTE applied twice)

Figure 84 presents the classification accuracies of successful builds and their corresponding sensitivity ratings over time is presented in Figure 85. For successful builds at the beginning of the data stream time series the accuracy was 66.38% and ended with 79.1% (with an average of 64%). It is observed that the general trend for classifying success starts to stabilise and gradually increase after approximately 900 instances. This trend was also observed in section 4.10.1, after the model was trained on

approximately 90 instances. This is also observed when comparing the sensitivity ratings for successful builds with the previous sections. Whilst the application of SMOTE is intended to demonstrate the potential for improved prediction with more data, it should be noted that it may simply have scaled the outcome of the real data by a factor of 10.

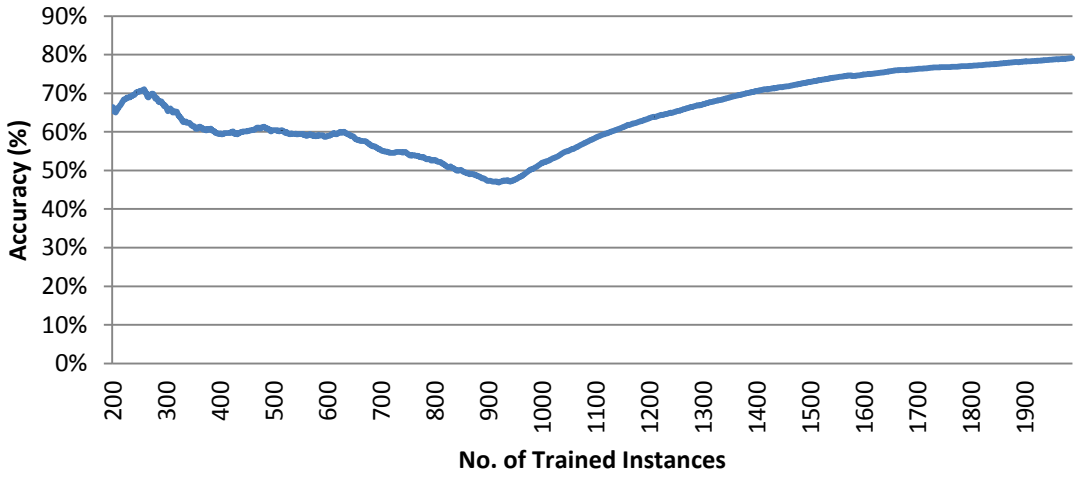

Figure 84 Hoeffding Tree Overall Classification Accuracy for Successful Builds for RSA After State (SMOTE applied twice)

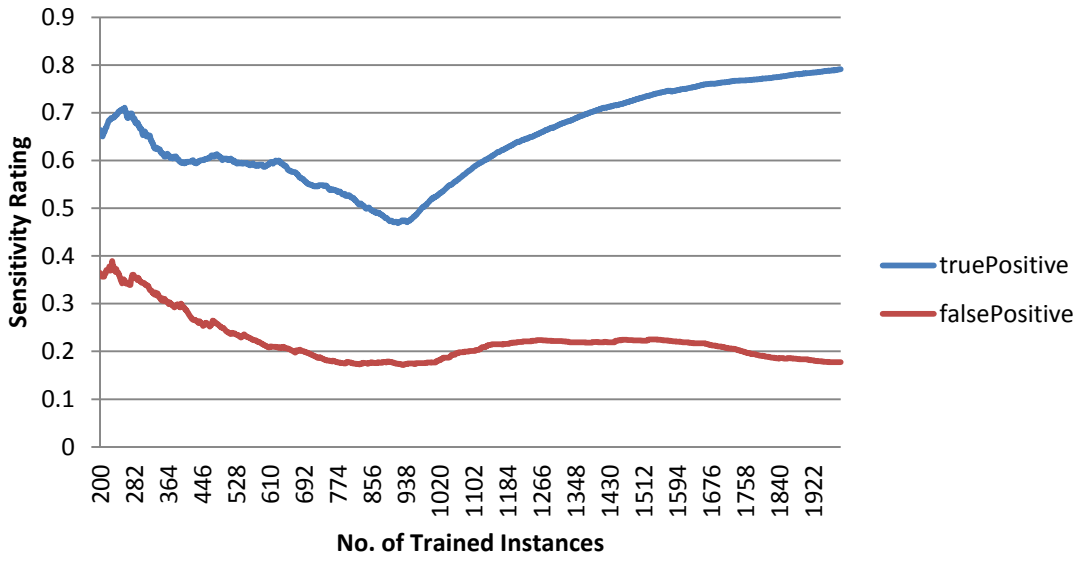

Figure 85 Hoeffding Tree Sensitivity Measurements for Successful Builds for RSA After State (SMOTE applied twice)

Presented in Figure 86 is the classification accuracies over time for failed builds and the corresponding sensitivity ratings for failed builds is presented in Figure 87. Classification accuracy for failed builds started at 63.5% and at the end of the time series was 82.2% (with an average of 78%). When comparing the accuracy of predicting failed builds to results found in section 4.10.1, the general trend appears to be similar after approximately 70 trained instances. The false positive values between 700 to 1000 trained instances appear to peak when classifying failed builds, due to over-fitting the model at earlier time segments. The false positive value then proceeds to decrease over time.

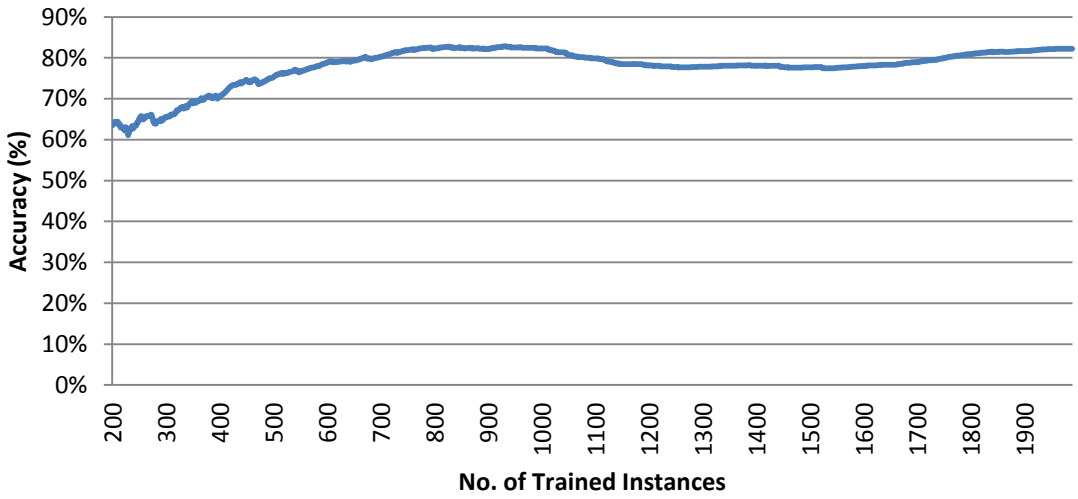

Figure 86 Hoeffding Tree Classification Accuracy for Failed Builds for RSA After State (with SMOTE applied twice)

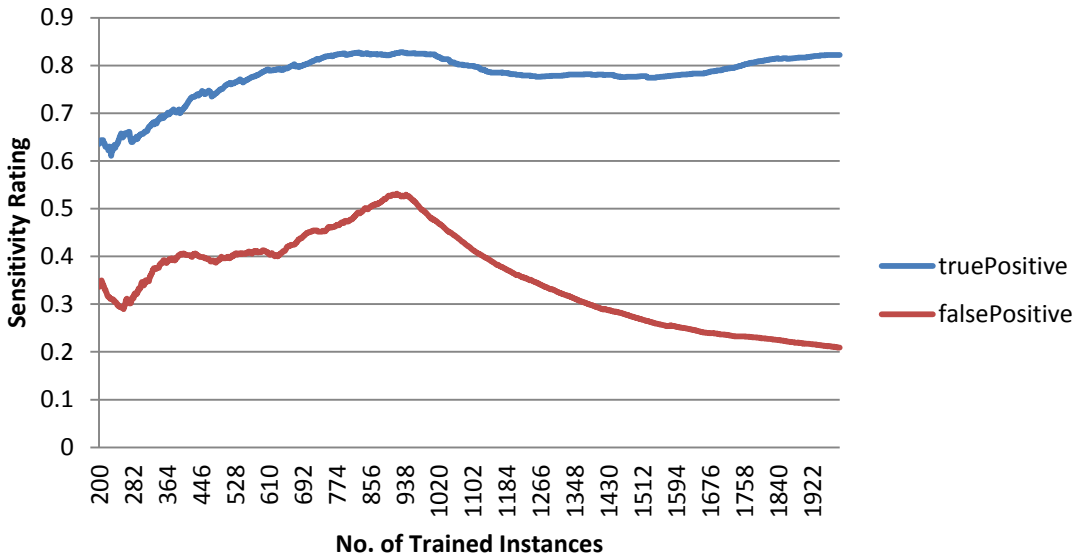

Figure 87 Hoeffding Tree Sensitivity Measurements for Failed Builds for RSA After State

Figure 88 illustrates the final decision tree using the Hoeffding Tree stream mining technique on the extended RSA after state software metrics data set. In this case the tree is larger than the previous software metric based Hoeffding tree, with a depth of 7. Upon inspecting the tree there are common sense classifications being made, for example a higher number of interfaces tends to be associated with failure. This is intuitive because if there are too many Java interfaces it can become tedious when debugging an error as the actual implementation of the error may be in an obscure location. Interfaces also add to the collection of files within the system and if an interface is "dead" (not used) and not removed it leads to a less elegant system design. The number of interfaces has a direct influence on dependency metrics, i.e. Abstractness.

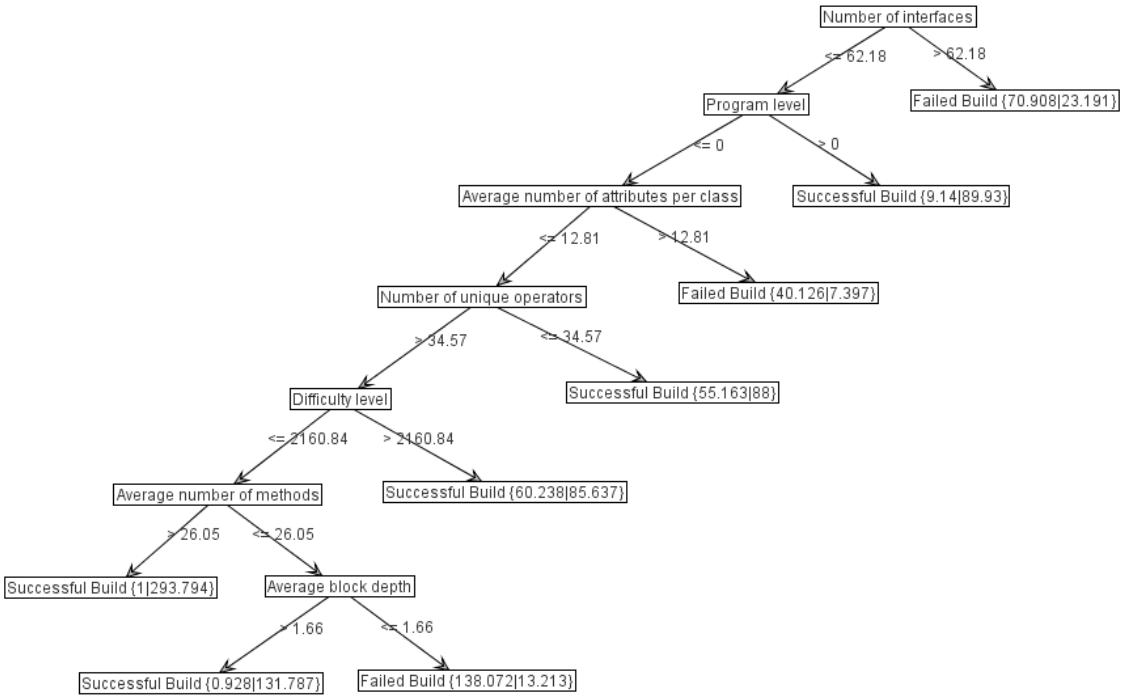

Figure 88 Final Hoeffding Tree for After State Software Metrics (with SMOTE applied twice)

Similar to the Hoeffding tree in the previous section (4.10.1), the average number of attributes metric also appears to be a strong indicator of build failure. Upon further inspection it is still observed that while an additional set of metrics are used for predicting failure in this tree, the result is mainly contingent on the number of unique operators. For example the following metrics are low or null i.e. the number of interfaces, program level, average number of attributes per class, difficulty level, average number of methods and average block depth and the number of unique operators is high there is a chance of a failed build. If there are a high number of operators, a failure outcome is more likely.

The output of ADWIN from the stream mining of the extended version software metrics data set shows a total of 55 concept drifts. This is not much of an increase when compared to the results from section 4.10.1, where there were 50 concept drifts using software metrics. Again, this may be that the application of SMOTE has simply scaled the outcomes present in the real data. The application of SMOTE in this way is intended to be indicative of potential rather than an absolute guarantee of increased predictive performance. The variance value after training on 200 instances was 0.122 and at the end of the streaming was 0.0536. The estimation value started at 0.1428 and ended at

0.0569. SMOTE interpolates rather than extrapolates values using existing data and as a result would not introduce new concepts to the data. Therefore the introduced complexity of the decision tree may be due to the fact that the model is synthesising the micro-patterns from the failed build instances.

In addition to the Hoeffding tree and concept drifting techniques a k-NN algorithm is trained using a similar process where the number of training instances is the same used for the Hoeffding tree. The final prediction accuracies of the Hoeffding tree and the k-NN when applied to communication network metrics are shown in Table 30, where k = 50. From this result it is observed that the Hoeffding tree and k-NN generate very similar levels of overall classification accuracies. The Hoeffding tree was able to correctly predict 16% more failed builds than the k-NN method.

Table 30 Final Prediction Accuracies of Hoeffding Tree and k-NN models for RSA After State (with SMOTE applied twice)

	Successful Builds		**Failed Builds**		**Overall**
	Correctly Classified Instances	**Incorrectly Classified Instances**	**Correctly Classified Instances**	**Incorrectly Classified Instances**	**Correctly Classified Instances**
Hoeffding Tree	1005	265	592	128	80.3%
k-NN	1024	131	418	217	80.6%

Figure 89 presents the overall classification accuracies for enlarged communication metrics data set using the Hoeffding tree. Similar to the software metrics overall accuracy shown in the previous section where the number of correctly classified builds appears to increase after approximately 1000 trained instances. The final classification accuracy for this model slightly increased when compared to streaming the original data set, where 70% of build instances were correctly classified.

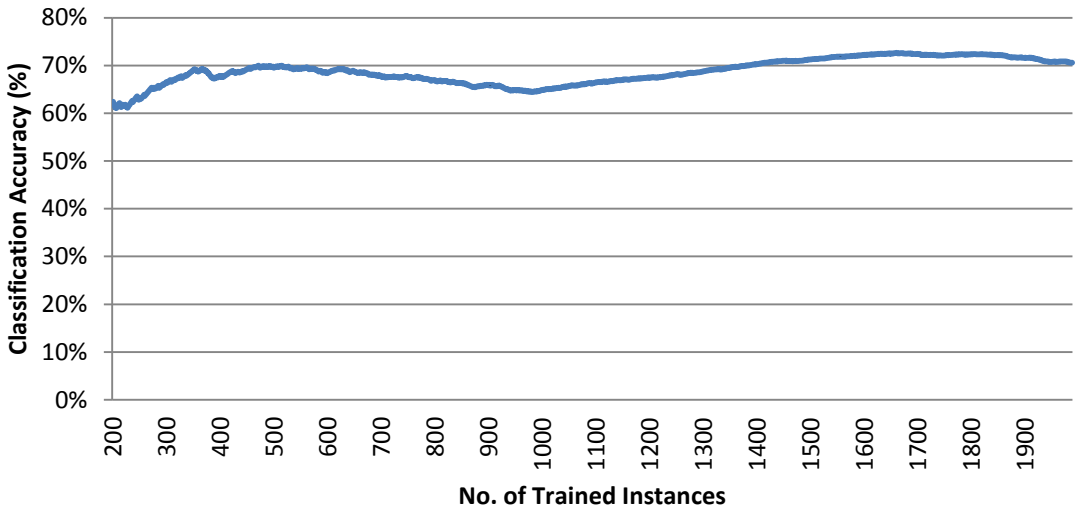

Figure 89 Hoeffding Tree Overall Classification Accuracy for 100% of Communication Metrics (with SMOTE applied twice)

To show accuracy of this result by class Figure 90 presents the classification accuracies of successful builds and their corresponding sensitivity ratings over time is presented in Figure 91. For successful builds at the beginning of the times series the accuracy was 50% and ended with 73.2% (with an average of 62.4%). It is observed that the general trend for classifying success starts to stabilise and gradually increase after approximately 900 instances. Similarly to the software metrics results and the trend of social metrics observed in section 4.10.2, after the model was trained on approximately 90 instances the accuracy gradually improved and stabilized. This is also observed when comparing the sensitivity ratings for successful builds with the previous sections.

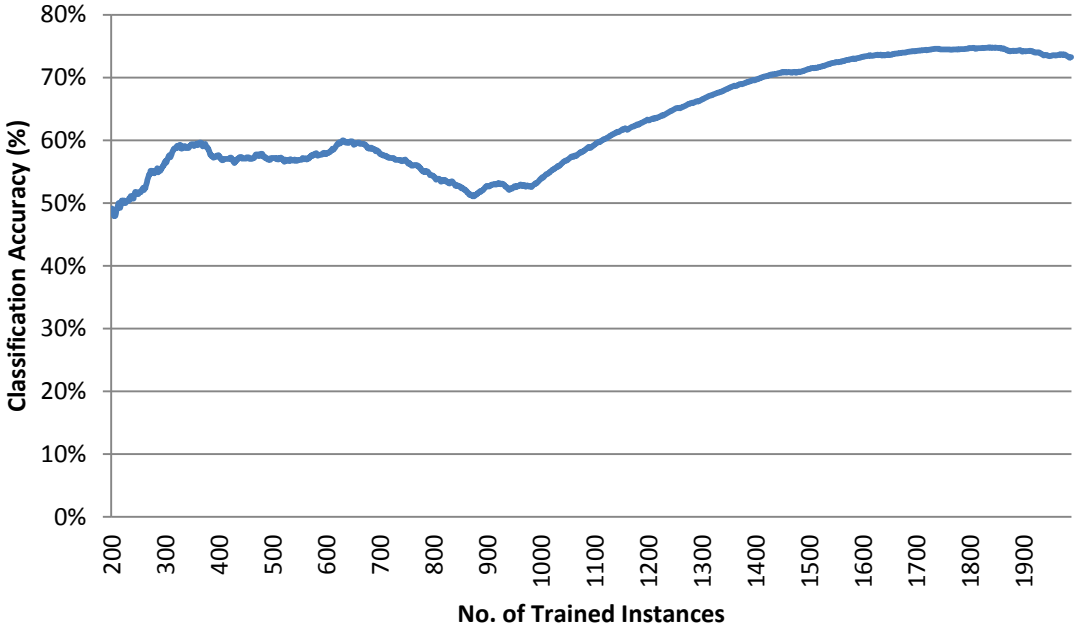

Figure 90 Hoeffding Tree Classification Accuracy for Successful Builds with 100% of Communication Metrics (with SMOTE applied twice)

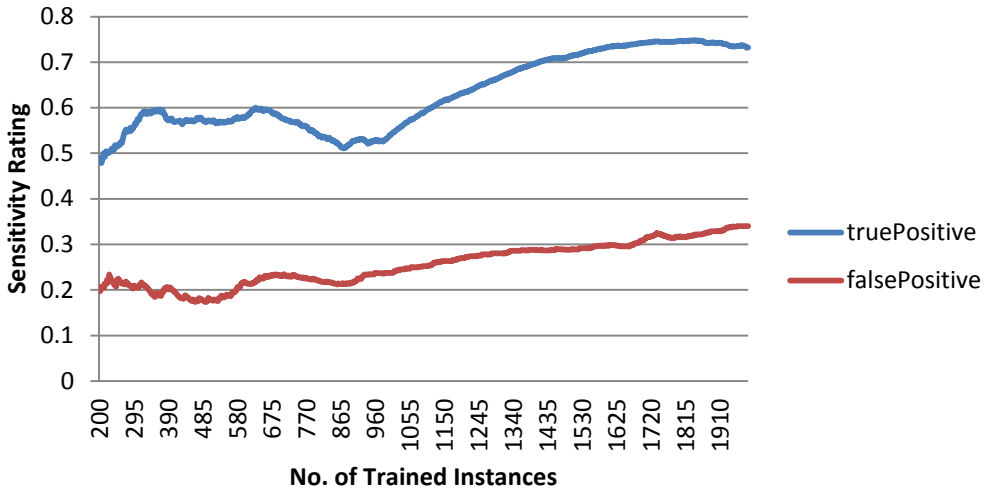

Figure 91 Hoeffding Tree Sensitivity Measurements for Successful Builds for 100% of Communication Metrics (with SMOTE applied twice)

Presented in Figure 92 is the classification accuracy over time for failed builds and the corresponding sensitivity ratings for failed builds are presented in Figure 93. Classification accuracy for failed builds started at 80% and at the end of the time series was 66.0% (with an average of 75%). Again when comparing the accuracy of predicting failed builds to results found in section 4.10.2 and to the trend of software

metrics in section 4.10.1, the trend appears to be similar after around 70 trained instances. It is also observed that in this experiment another false paradox phenomenon occurs for failed builds. True positive values are higher than false positives in this experiment after around 1350 trained instances. During this time period the general classification accuracy for correctly classifying failed builds decreases.

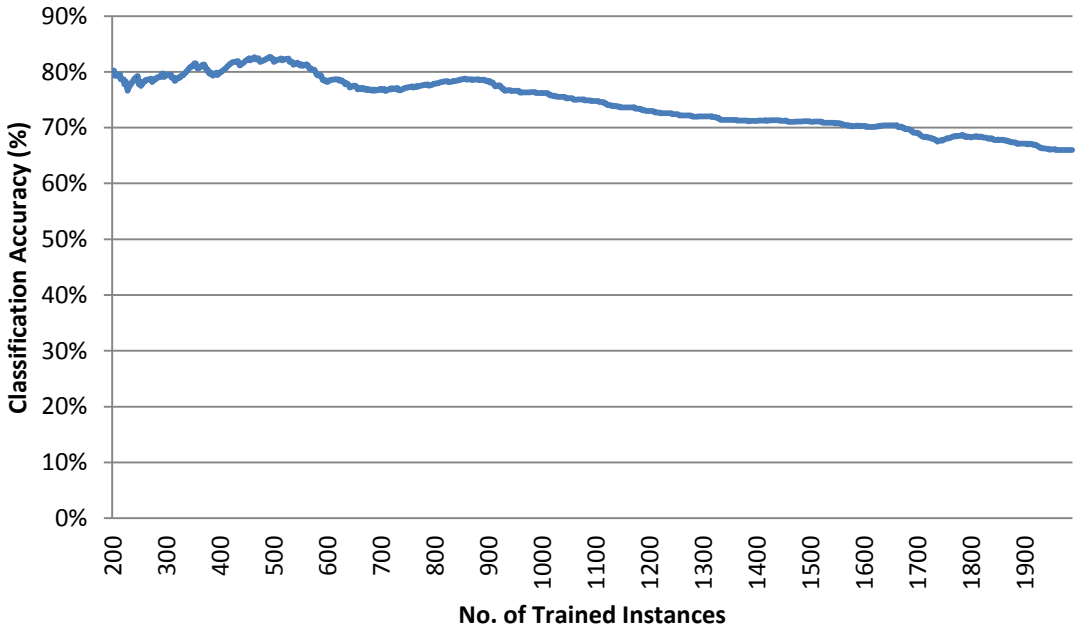

Figure 92 Hoeffding Tree Classification Accuracy for Failed Builds with 100% of Communication Metrics (with SMOTE applied twice)

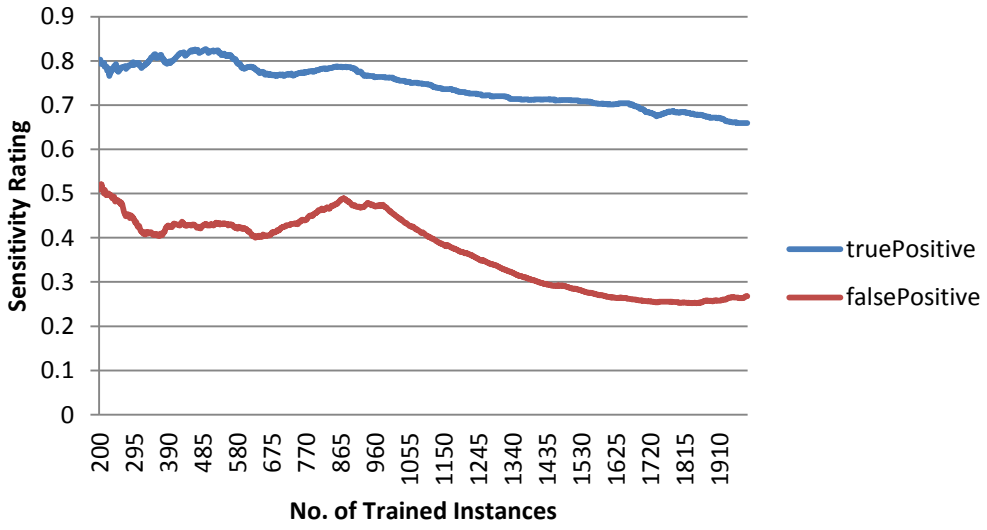

Figure 93 Hoeffding Tree Sensitivity Measurements for Failed Builds for 100% of Communication Metrics (with SMOTE applied twice)

Illustrated in Figure 94 is the final Hoeffding tree for the extended social network metrics data set. This tree has also grown in size when compared to the Hoeffding tree presented in section 4.10.2 and has a depth of 6. From this decision tree a majority of failed builds are predicted by low group out degree centrality values and a high edge (number of connections within the network) count. Contributors who display high out-degree centrality are said to be "influential" within the network. This decision diagram is intuitive, in this sense, because if there are a high number of connections within a network yet no one driving the discussion there is a higher chance of build failure.

There is some degree of confusion displayed within this tree. For example the node Markov centrality metric can essentially be removed, because if this value is low or high the predicted outcome is a successful build.

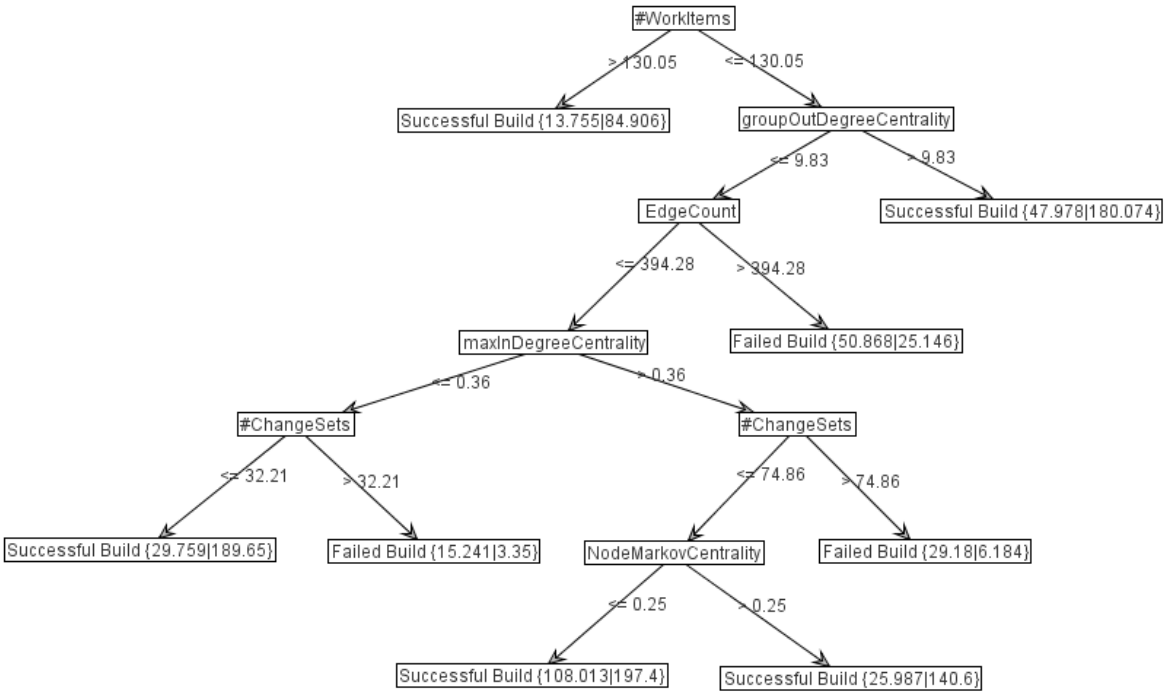

Figure 94 Final Hoeffding Tree for 100% of Social Network Metrics (with SMOTE applied twice)

Similar to the software metrics results in the previous section the total number of concept drifts did not increase when compared to the social network mining results presented in 4.10.2. The concept drifts increased from 74 to 76 in total. Again this was to be expected due to the nature of how SMOTE generates new instance values.

In addition to the Hoeffding tree and concept drifting techniques a k-NN algorithm is trained using a similar process where the numbers of training instances are the same used for the Hoeffding tree. The final prediction accuracies of the Hoeffding tree and the k-NN when applied to communication network metrics are shown in Table 31, where k = 50. From this result it is observed that the Hoeffding tree and k-NN generate very similar levels of overall classification accuracies. The Hoeffding tree was able to correctly predict 12% more failed builds than the k-NN method. The software metric results have again provided a better level of classification accuracy (with about a 10% improvement) when compared to using social network metrics.

Table 31 Final Prediction Accuracies of Hoeffding Tree and k-NN models for 100% of Communication Metrics (with SMOTE applied twice)

	Successful Builds		**Failed Builds**		**Overall**
	Correctly Classified Instances	**Incorrectly Classified Instances**	**Correctly Classified Instances**	**Incorrectly Classified Instances**	**Correctly Classified Instances**
Hoeffding Tree	930	340	475	245	70.6%
k-NN	889	256	341	294	69.3%

4.11 Chapter Summary

This chapter has presented the data mining results drawn from the extractions of software metrics and communication metrics. Feature selection methods explored included Information Gain, Subset Evaluation, principal component analysis, frequency feature selection and application of the after state features to the before state data sets. Traditional data mining methods explored included the j48 decision tree builder, Naive Bayes and Bayesian Networks. The results indicated that in terms of feature selection there was no single best method to use. However, in terms of the mining methods explored it was observed that the j48 decision tree typically provided better results in terms of correctly classifying instances. From mining different software metric data set aggregation it was found that the RSA data set typically provided better results for classifying builds. For software development teams looking to build predictive models of development endeavours, the metric values obtained from IBMs rational software analyzer tool and the maximum metric values have portrayed a valuable starting point for data mining activities.

Social network metrics were also extracted and data mined using traditional and data streaming methods. The results have shown prediction models built from software metrics provided better classification accuracies than the models built using social network metrics. In addition to this upon merging software and social network metric data sets, software metric predictors were found to be more significant in determining build success and failure as they were not selected using Information Gain or Subset Evaluation feature selection methods.

Finally data stream mining methods were explored where Hoeffding tree, ADWIN concept drifting and k-NN were used to see how software and social network metrics evolved over time. Similarly to the results presented that were based on traditional mining methods software metrics had higher prediction accuracies than social network metrics.

5 Discussion and Research Summary

5.1 Introduction

The methodology and results of the experimental studies are described in chapters 3.2 and 4 respectively. Whilst some analysis of the results was presented in the previous chapter, the main purpose of this chapter is to extend the analysis and interpret the outcomes. This chapter looks at the interpretations and philosophies that underpin the experimental research results in the context of this thesis and its scientific enquiries. In addition, this chapter also presents a simulated scenario of how a predictive model built on both the software and social networks could be deployed in a development team to successfully leverage the value of the predictive power of both elements. Finally, this chapter presents a summary of the research study.

The following section details the major findings of the study. Interpretations of the findings and their relationship to the original research question are discussed as well as their impacts on decision making for a software development project. There have been no alterations to the IBM Jazz repository before and after the extraction of software and social network metrics, to avoid false or misleading data and information.

5.2 Research Summary

This study has adopted the system development research methodology (Nunamaker, et al., 1991) and adhered to the design science guidelines to investigate, design, develop and implement predictive models. In conjunction with this a research via artifacts method (Nguyen, et al., 2009), similar to previous studies that have made use of the Jazz repository, is also adopted. Overall, this methodological approach is similar to other studies that have considered the creation of predictive models from historical data using data mining approaches (Khoshgoftaar, Allen, Jones & Hudepohl, 2001).

One of the challenges within this work was in getting a suitable level of granularity for predicting software artifact outcomes. This thesis has presented data mining results of software and communication metrics extracted from change sets of software build items and work items within the Jazz repository. Other levels of granularity were initially explored, for example software metrics were also derived at work item level change sets

and were classified by work item type. While there existed more (1157) instances of change sets at work item level it was found that metrics extracted at build level provide better classification outcomes for prediction of failure than those extracted at work item level predicting defects. However, despite the level of granularity between build item and work item software metrics, the same challenge was observed when predicting failure or defects, where there was much overlap in feature space.

Under the KDD process (Fayyad, et al., 1996) the first step is to construct the target data sets. For this multiple software metric state aggregations (RSA, Total, Mean, Median and Maximum) and social network metric (25%, 50%, 75% and 100% time segments) aggregations were derived from software builds. Such aggregations were necessary as they condensed vast amounts of information about source code into smaller representative numerical values that could be used to build predictive models. Secondly the data sets were pre-processed, where builds without associated change sets were removed. In total for each software metric data set there were 42 software metrics including traditional (size), complexity, dependency, cohesion, inheritance and Halstead metric categories. For each social metric data set there were 16 metrics including size metrics and centrality metrics. In addition to this simple count metrics were also included in the social network data sets to show the number of work items and change sets a particular discussion was connected to. All data sets presented in this thesis were classified by a build result outcome metric. There are a range of metrics that are unique to the Jazz environment (number of change sets, number of work items and the build outcome). It is important to note that the number of change sets metric is associated with a work item (included in the communication metric data set) is not the same as the number of change sets associated with a build item.

The results of the classifications were used to search for patterns of interest that are open for interpretation. While it remains very difficult to draw generalisations from the results of this work, in order to make comparisons to other software metric studies, understandings gained from this work are similar to the insights found within the literature surrounding software metrics for defect and quality prediction.

5.3 Analysis of Results

5.3.1 Data Mining Software Metrics

The aim of the first set of data mining experiments was not only to generate various prediction models, but to primarily find a suitable direction of inquiry to enhance classification accuracy and generate better understandings of the nature of various software metrics. More specifically the aim of the initial experiments was to find which software metric aggregation, feature selection method and data mining method produced the greatest promise in terms of classification accuracy and sensitivity. This initial phase was necessary in order to reduce the scale of the inquiry to a manageable size as a truly exhaustive search of all combinations would not be achievable in a reasonable timescale.

During the first stages of experiments, filtering methods (Subset Evaluation, Information Gain and PCA) were applied to various software metric aggregations (RSA, max, mean, median, total software metrics) to reduce the original number of predictors. Prediction models were generated using traditional data mining methods (j48, Naive Bayes and Bayesian network) and their performance (accuracy and sensitivity) were tested. This stage of experiments gave preliminary insights into whether software metrics could be used predict to software build outcomes on their own. The software metrics derived from the before and after state consisted of different values for each instance, however through data mining methods the patterns found from these states were very similar in terms of accuracy, sensitivity and metrics that were selected as being significant features for the prediction models.

The results were presented via three categories based on the data set, feature filtering and mining methods. During this stage it was found that the subset evaluator generated the best result when applied to the rational software analyser (RSA) data set. This was to be expected as the CfsSubset method looks for the inter-relationships between metrics to identify significant factors. Once the CfsSubset method was used to filter the metrics the j48 decision tree classifier was applied using 10-cross fold validation and produced the best results with an overall classification accuracy 80.4%. The precision and recall weighted average values were also high with a rating of 0.8.

However, it would not be correct to state that the subset evaluator performed the best for all experiments, therefore it is not the "single best" solution. This problem can also be seen within the mining software metrics literature as it remains unclear which feature selection method is "best" to use, as many studies contradict each other. It is more correct to state that the "best" feature selection method is highly contingent on the context of the software metric data set itself that is being investigated. In some cases selecting only 20% of the 42 software metrics the models built were not adversely affected. This result coincides with other studies (Fenton & Neil, 2000; Kehan Gao, Khoshgoftaar, Wang & Seliya, 2011).Yet in other cases, mining methods applied with no features filtered produced very similar or better results. Throughout these initial results and later results there was much variability in the models produced. This is most likely due to the relatively small size of the data set and the large number of predictors and perhaps the absence of robust relationships.

In terms of selecting a data mining method, it was observed that the j48 classifier frequently produced better classification accuracies in a majority of the experiments. However, upon inspecting the decision trees, there were degrees of confusion found within their branches and leaves, showing duplicate metric nodes or constructed rules which did not make sense. This was particularly true for classifying failed builds. Again, this may be a direct result of the ratio of predictors to data instances. In many cases it has been observed that there is not a major difference in the predictive power of different metrics which makes it difficult to select a significant set consistently. Across all experiments of the initial software metric based mining stage it was also observed failed builds were much more difficult to predict than successful builds. In particular, it has been observed that predicting failure is a different task than predicting non-success. The accuracy, precision and recall values for failed builds were considerably lower than successful builds across a majority of data mining experiments.

These initial findings support the claims of Buse and Zimmerman (2010) who suggest that while a range of metrics can be used to describe a project, other important aspects such as development process "health" can be unpredictable. The primary goal of their software analytics paradigm is to provide insight by identifying patterns based on multi-dimensional factors. In doing so development teams are provided with the opportunity

to proactively manage risk exposure through the development project. These results support other studies that have shown that there is no single code or churn metric that is capable of predicting failure (Basili, 1996; Denaro, et al., 2002; Nagappan, Ball & Zeller, 2006). In that context, the outcome of this phase of research is that a number of different methods have the potential to be applied in a real world development environment. Whilst there may be minor differences between the nature of the predictive models, they all could improve the outcomes of software development projects if applied in conjunction with human expert judgement.

Predicting failure is more challenging than predicting success and it is also observed that not predicting failure does not mean that success has been predicted. This is due to the fact that the build successes and failures overlap in feature space and "failure" signatures have a greater degree of fragmentation than their "success" counterparts. This is most apparent in the very different classification trees that have been discussed. Each shows a different set of software metrics that can be used to gain roughly the same overall prediction accuracy. This indicates that there is much uncertainty in the relationships between software build failures.

The degree of variability within each of the Jazz builds, each with various purposes, results in wide variations of metrics. This can make it difficult to predict which builds are likely to fail. Even if there were more failed build instances it may not lead to improved reliability of Jazz as a whole. Instead prediction of reliability should be viewed as complementary to defect or build failure density prediction.

5.3.2 Enhancing Performance of Experiments

The second phase of experiments aim was to increase the prediction accuracy and sensitivity values of failed builds. To do this phase two focused on filtering features based on their selected frequency during the prior experimental phase. Even though there was variability between the prediction models of the first phase, it was observed that certain metrics were commonly selected as being significant factors for predicting build outcomes. The variability suggests that the indicators may be weak themselves. Therefore by looking at how often they were used in the range of experiments was a way of investigating whether the strength of prediction could be reinforced.

As the number of selected metrics is reduced by applying a higher frequency threshold, the overall accuracy does not change significantly, yet there was a trend towards better classification of successful builds. This possibly indicates that some metrics are very strong indicators of success whereas others are weak indicators of failure. This again may be related to the fact that the build successes and failures overlap in feature space and “failure” signatures have a greater degree of fragmentation than their “success” counterparts. This insight reinforces the outcomes of this work that a predictive model cannot be applied blindly, but does have the potential to complement the expert opinions within the software development team.

It was observed that the before and after state of the software metrics produced fairly similar results in terms of overall classification accuracy and sensitivity. The j48 algorithm again generated the highest levels of accuracy across all experimental feature thresholds. It was observed that if using the j48 algorithm, despite the data set aggregation, that it is best to include metrics that were commonly selected from all experiments that were found to appear more than three times. This indicates that even if a certain metric is selected commonly from a range of data mining experiments, classification accuracy still depends on its relationship to other metrics.

In addition to exploring the frequency thresholds for selecting features, the after state selected features were applied to the before state metrics. This was to see whether or not classification improved when "past" metrics were selected from a filter provided from the "future". During the build cycle an iterative review of the risk that is exposed may be used to improve the accuracy prediction of failure and therefore manage the risk ahead of time. In this case the before state is used to determine whether build failure can be predicted prior to any changes made. This attempted to characterise source code that is about to be changed of its likelihood of being changed successfully or if there are patterns in existing code that can potentially lead to failure. Code that is likely to be changed in a build introduces potential defects or build failure (Khomh, Di Penta & Gueheneuc, 2009). The patterns that cause a build to fail may not be obvious until the change is made. Therefore identifying the causes of failure may determine significant predictors earlier during the build cycle.

It has been shown that features selected from the after state metrics to predict the outcomes of a build in the before state, can be used for process of examining code for potential failure. Some evidence exists in literature that may explain this phenomenon. For example, the results within this work is supported work by Kitchenham (2010) where it was found that software metrics reflecting change in code extracted at various points in time were more likely to predict fault rates in an evolving system better than final snap-shot based software metrics.

It appears that examining the degree of magnitude in change of a build offers the most clarity in terms of predicting the likelihood of a failed build. This approach also provides software development teams the opportunity to incrementally assess whether risk exposure is increasing or decreasing throughout a build cycle. However, it is important to appreciate that the metric values in the classification tree do not necessarily represent any absolute indication of change. This is particularly true for the results of the max data set, as the maximum metric may not be drawn from the same before and after state file.

The frequency feature selection threshold metrics from the after state were also used to filter metrics from the before state. When the feature frequency threshold was greater than three, from the after state and the features filtered the max before state data set the classification accuracy slightly improved (to 81.8%). From this experiment there was also an increase (approximately 10%) of precision and recall values for correctly classifying failed build instances. Despite this improvement there still remained much variability between models that were derived from the application of the after state metrics to the before state data and the frequency threshold selection experiments.

The goal of synthetically generating data was to explore what might happen if there was more data available for mining, more specifically to see if classification of builds improved with more data. While the use of SMOTE may not be a fair representation of new real world data, it does however interpolate values between existing instances to generate new instances. This provides insights into what may occur if there were no "new" anomalies encountered during the project. This may not be entirely realistic given that the causes of failure are not predictable and that new failure modes are likely to appear over time.

To do this the number of minority class (failed builds) instances were increased using SMOTE. It was found that classification accuracy improved when SMOTE was applied at 300% to the before state data and filtering metrics using the subset evaluator from the after state metrics. In this scenario the overall classification increased (to 83.1%). The accuracy correctly predicting failed builds also increased, correctly identifying 81% of failed instances. A slight increase in precision (0.75) and recall (0.82) values also occurred. It was observed that increases in precision and recall values were also obtained across experiments for classifying build failure.

From previous data mining experiments build failure metrics were very close to successful builds. This indicates that perhaps if more data is available accuracy for classifying builds will improve over time. The results obtained during this phase support other studies where software fault prediction increased in classification accuracy and stability when adopting the use of SMOTE (Kehan, Khoshgoftaar & Napolitano, 2011; Pelayo & Dick, 2007; Shatnawi, 2012) based on other project software metrics. In addition to the application of SMOTE this phase also showed that higher classification accuracy and sensitivity values can be achieved again by applying the future state metrics to before state data. This can be thought of in terms of the fact that the patterns that cause actual failure may already be present in code that has resulted in a successful build.

5.3.3 Data Mining Social Network Metrics

From the data mining experiments on only social network metrics it was again discovered that failure was more difficult to predict than success. This finding is also supported by previous work (Wolf, et al., 2009) where no single communication structure measurement (e.g. density, centrality or structural holes) could distinguish between success and failure build outcomes.

Additional predictors such as the number of work items and change sets were also included to see if an increased of size in complexity and number of changes related to a work item could be related to predicting build failures. Again the results are supported by Wolf et al. (2009), where metrics also did not significantly differentiate between success and failed builds. Results are also supported where higher classification accuracies were obtained from mining the first 25% of communication and all (100%),

rather than the first 50% and 75% of communication. However, unlike this study the precision and recall values of the prediction models generated were not as high. This is because communication metrics were extracted across all teams, rather than across teams that had more than 30 build results (consisting of at least 10 failed and 10 successful builds). For the work presented in this thesis communication metrics were taken across all work items of a build if the build itself consisted of software metrics derived from change sets. This resulted in fewer instances in the final data set. From the insights provided by SMOTE and work by Wolf et al. (2009), a larger data set might provide greater levels of accuracy and sensitivity. The purpose of initially mining the social network metrics separately from the software metrics was to gain insights and confidence from the social networks prior to merging them.

Unlike software metrics, the extracted social metrics are not derived from change sets and therefore have no before and after states. Hence there are fewer mining experiments performed on social metrics than there were on software metrics. Upon merging the software metrics data sets with the social network metrics aggregated states, there were no social network metrics that were detected as significant factors for constructing predictive models when using feature selection methods (Subset Evaluation and Information Gain). When comparing predictive models that were built from software metrics and built from social network metrics it was found that software metric models performed better in terms of prediction accuracy and sensitivity. While there are many studies that have explored the use of software metrics and social network metrics for predicting software quality, there are few studies that have explored using a combination of these metric types. It seems reasonable to suggest that software metrics have more significance in terms of predictive power but this is not to say that social network metrics have no value. An analysis of which builds in the Jazz repository are predicted correctly by either a software or a social network metric classification model shows that in some cases both models predict the same outcome, but this is not always the case. There are instances where software metrics do not correctly predict the outcome whereas the social network metrics do. The potential for a more robust mixed-metric prediction that leverages the advantages of both predictors is discussed later.

The models derived from traditional data mining methods are useable by the Jazz development teams to assess quality of their software and communication in relation to

the result of their future builds. However, predicting a build result has no value for a developer if they have no time left to react on the prediction, as they can simply run a build instead of using a derived decision model. Therefore a continual review of software and social network metrics is required to add value for decision making regarding build outcomes and defects. Traditional mining methods generate models for the end of a process and are often used for a "one off" analysis. Data stream mining involves extensive feedback and does not require all data to be stored for learning instances and therefore scales well when working with large data repositories.

5.3.4 Data Stream Mining Software and Social Metrics

This research has presented a potential solution for encoding software and social metrics as data streams. In this case the data streams were provided when a software build was executed. The real-time streams can be run against the model which has been generated from software build histories. From the real-time based predictions developers may delay a build to proactively make changes on a failed build prediction. One of the advantages of building predictive models using data stream mining methods is that they do not have large permanent storage requirements. One of the reasons why Jazz only keeps a limited number of build change sets is because of the huge storage requirements. This research has shown that data stream mining techniques hold potential as the Jazz environment, as the platform can continue to store the latest *n* builds without losing relevant information for a prediction model that has been built over an extended series of (older) software builds. As a tool, the predictive models can be encoded into the IDE and updated when builds are performed. This tool would provide contributors with real-time feedback during the development of their code in relation to the metrics extracted and predicted build outcome. It would also provide real-time insights into the way the team is communicating effectively for generating a successful build. While data stream mining has application in managing network, web searches traffic, systems, networks, ATM transactions and safety, few studies have investigated data stream mining using software and social network metrics. To the researchers' knowledge this is the first attempt ever made to use data stream mining techniques for predicting software build outcomes using software metrics and communication metrics and as a result this is the main contribution of this work.

From the data stream mining experiments it was observed that after approximately 90 instances classification accuracy began to stabilise and steadily improve. From a range of 42 metrics, in this case, only 2-5 will appear to be significant. This means that the classification trees will be easy to interpret and should not contain much confusion within its branches. This also holds true for social network metrics, where 2-3 communication measurements may be selected from a range of 16 metrics.

From the decision tree, built from software metrics, presented in section 5.3.4, the average number of attributes and the maintainability index were found to be significant for classifying builds. These are both examples of basic (sizing) and complexity type metrics and coincides with findings of previous studies (Khoshgoftaar & Munson, 1990). However, this again conflicts with other studies that size and complexity metrics are not sufficient for accurately predicting real-time software defects (Challagulla, Bastani, Yen & Paul, 2005). This result shows that again researchers who are utilising software metrics must be careful about what the metrics represent within the context of their project and to be careful with drawing generalisations. From this it would be fair to draw the same conclusion of Nagappan et al. (2006) where they found that there is no single set of complexity metrics that can acts as a "best" predictor of software defects. It is also clear that it is not possible to assume that just because a metric (or set of metrics) appears to have some ability to predict outcomes that this will always be the case and should be followed blindly. It is unlikely that a predictive model based on software or social network metrics will be useful when applied automatically. The value of such a system is to provide indications of where potential risks may arise to direct human intervention to inspect areas of concern and make expert decisions as to how to control and manage the risk. The potential for such a system has been realised by this research and how such a system could be deployed in the Jazz environment is discussed in the next section.

5.4 Real World Application

The models built may be used by Jazz teams to assess the quality of their current software metrics and communication metrics in relation to the result of an upcoming build. The process of applying the model starts at the end of the preceding build. On the completion of the preceding build once the outcome is known the Hoeffding tree method is used to rebuild the two prediction models based on software and social network metrics.

At the beginning of the build the developers will determine their best guess as to which code will need to be changed to implement new features and correct the bugs scheduled to be fixed in the build. This will allow an initial change set to be defined and software metrics to be extracted to classify the likely outcome of the build by applying the software metrics to the current classifier tree. If the predicted outcome is given as a failed build, this allows the software development team the opportunity to reconsider the scope of the build and potentially remove features that are intended for implementation in the build. The team can interactively remove features and the corresponding code from the change set and reapply the software metrics to the classification tree until the perceived risk in the build is at an acceptable level. Such modifications should not be done blindly on the basis of the classification outcome but should also include some aspects of expert judgement.

As the build cycle progresses a number of changes will occur. Firstly, it should be expected that developers will realise that the implementation of a particular set of features includes changes to source code files that were not expected at the outset of the build. This means that the actual change set will alter over the period of the build. Secondly, actual code changes will be made over that time as the features are implemented. Both of these will result in different software metric values being calculated for the change set. The final change that will occur is that developers will be discussing the build using the online discussion forums in Jazz which will allow social network metrics to be calculated.

During various time segments of a build (e.g. 25%, 50% and 75%) the software metrics and social network metrics are re-calculated and are used to re-predict build outcomes

based on the existing classification models. Again, this provides the software development team with the opportunity to consider the risk that is present in the current build. If the initial prediction at the beginning of the build was positive (i.e. a successful build was predicted) but interim predictions start to indicate that the build will fail then this provides the development team the opportunity to proactively manage the risk that has emerged. Again, this could involve reconsidering the scope of the build or alternatively it could highlight where more effort needs to be applied or where greater communication about the complexity of the build is required.

Once a build is (100%) complete the outcomes of the build can be used to update the predictive models that can be used in the same way as a decision support tool for the next build cycle. It has been shown in chapter 4 that over time the usefulness of the predictive model will increase but the application of the model will always be in terms of directing the development team to the presence of risk rather than blindly controlling the development process.

While a team is working towards a build the model provides feedback about the current significant software metrics and communication structures from work items that are related to the build. This knowledge enables the identification of potential build outcomes and supports intervention in cases where build failure predictions are observed. With the adaptation of data stream mining patterns are classified from emergent states without the addition of huge storage requirements.

To demonstrate how the above process may be used in practice a real example of a small series of Jazz builds is presented. Table 32 shows how a combination of models based on software metrics and social network metrics can be used to predict a build outcome. Due to storage constraints the top section of the table presents 4 builds, their emergence of significant software metrics and predicted outcomes for the before and after state. The lower section of the table presents the same 4 builds, the emergence of significant social network metrics and predicted outcomes for 25%, 50%, 75% and 100% of the communication intervals during the build. The “Pred.” field is the predicted outcome generated from using the Hoeffding tree classifier for each modelled instance.

In this example for the first build (I20070628-0026) the classifier has not been trained on enough instances for the predicted outcome to be reliable, it simply provides a starting point for the learning process. The next visible build (C20080619-1123) has been trained on 159 instances, where prediction based on the before and after state metrics correctly determine the outcome of a successful build. This also remains true in this case for all social network metric time segments. It is observed that the before and after state software metrics, while having slightly different values share similar models for predicting a builds outcomes.

The following build (C20080619-1236), while being predicted as primarily an "OK" build, when the social network metrics were extracted at the 75% interval an indication of a failure is observed. If this pattern persists there may be a risk of build failure in the future. The last build presented (C20080619-1345) details a great risk of failure as both before and after state metrics generate an "ERROR" status. In addition to this 75% and 100% of social network metric intervals also predict potential failure.

Table 32 Example of Real World Application

Software Metrics																				
			Before State			After State														
Build ID	Actual Result	Average number of attributes per class	Maintainability index	Pred.	Average number of attributes per class	Maintainability index	Pred.	Social Network Metrics												
								25% Social Metrics				50% Social Metrics			75% Social Metrics			100% Social Metrics		
Build ID	Actual Result	Node Markov Centrality	No. Of WorkItems	Pred.	Node Markov Centrality	No. Of WorkItems	Pred.	Node Markov Centrality	No. Of WorkItems	Pred.	No. Of WorkItems	Effective Size	Pred.	Group InOut Degree Centrality	Number of Change Sets	Pred.
I20070628-0026	OK	5.82	280.9	OK	3.34	281.55	OK	0	0	ERROR	0	0		0.010476	132	OK
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
C20080619-1123	OK	3.35	253.59	OK	3.35	266.06	OK	0.479184099	1	OK	1	36	OK	0.095585	19	OK
C20080619-1236	OK	3.35	253.59	OK	3.35	266.06	OK	0.479184099	1	OK	3	84	ERROR	0.095585	19	OK
C20080619-1345	ERROR	2.66	270.08	ERROR	2.69	269.94	ERROR	0.268470141	3	OK	0	0	ERROR	0.049681	193	ERROR

5.5 Guidelines for Data Stream Mining

Depending on the software environment or repository used the following presents the guidelines for adopting data stream mining techniques for software and communication metrics:

1. Ensure that the development infrastructure can collect relevant data
    a. This study has shown that a certain volume of data is required to build a meaningful model. Even if there is no immediate intention to build a predictive model, it is important to ensure that sufficient data will be available at some time in the future to inform decisions regarding what are unit of analysis and what outcomes are measureable.
2. Evaluate what outcomes can be classified
    a. For this study each instance was classified by software build outcome. Other potential outcomes could be unit test results, whether or not a module is delivered on time (for effort and cost prediction), or current module states.
3. Determine the level of granularity for analysis
    a. For this study, the unit of analysis was a software build. This was determined by considering different units of greater or less granularity. It is expected that a single bug fix is likely to be too granular, whereas a product level of analysis would be too coarse. The goal is to find a suitable mid-point for analysis that has useful outcomes but for which there is sufficient data points to build a model.
4. Create the target data instance or instances
    a. Find significant and useful features within the data depending on the goal of the mining task.
    b. Extract software and communication metrics that have a relationship to the outcome to be classified.
    c. Ensure that the metrics are extracted at a suitable level of granularity for classification purposes.
    d. Encode each instance as a data stream.

5. Automate the data cleaning process
    a. Depending on the intervals between data streams (how often target data instances are extracted), automation of the cleaning process may be vital to ensure the models integrity.
6. Perform data stream mining
    a. Searches for patterns of interest.
    b. Identifies significant metrics.
    c. Identifies conceptual evolutions within the stream.
    d. Generates a representation from the classification process (e.g. a decision tree if using the Hoeffding tree method).
7. Interpret and utilise the mined patterns to aid proactive decision making during the software development life cycle.

5.6 Chapter Summary

This chapter has presented the implications of the experimental results in the context of the objectives of this thesis and briefly presents a summary of this research. The outcomes of this work require further investigation in order to draw any sorts of generalisations for other software development projects. In terms of the research question originally presented in chapter 1, the results for each and evidence supporting these results are presented in Table 33. The final chapter presents potential future research and draws conclusions based on the evidence and contributions made by this thesis.

Table 33 Evidence to Support Experimental Research Goals

To what extent can a combination of software and social network metrics extracted from IBMs Jazz repository be used to generate predictive models to determine software build success and failure more effectively than either of the individual models?	
Metrics Types	**Evidence (sections)**
Software Metrics	4.1 - 4.8, 4.10.1and 4.10.3
Social Network Metrics	4.9 and 4.10.2
Combinations of Software and Social Network Metrics	4.9 and 4.10.3

Experimental work has been done to address the primary research question. It has been found that the software metrics, derived from IBM's Jazz project repository, can be used to generate predictive models to determine software build success and failure. It has also been found that a range of social network metrics, derived from IBM's Jazz Project repository, can be used to generate predictive models for determining software build outcome. When the software metrics and social network metrics are combined, the social network metrics did not provide any significant predictors of build outcomes when using the j48 decision tree, Naive Bayes and Bayesian Network mining methods. When using the Hoeffding tree, both software and social network metrics were found to be significant predictors of build outcome, however there was no substantial improvement observed in terms of overall classification accuracy.

6 Conclusions

This thesis has presented the outcomes of a thorough and systematic investigation into the development of affective prediction models for use in software development. This research made use of the IBM Jazz repository to predict the outcome of a software build through software metrics with social network metrics analysis. Although this type of analysis for quality prediction has been extensively studied in the past few decades using a range of data mining methods, the study of combining these metrics has remained relatively unexplored. In addition to this, data mining and encoding software metrics and social metrics as data streams for software build outcome prediction has also remained unexplored. Both of these areas have been considered in this work and the implications of combining software metrics and social network metrics for building prediction models in the context of IBMs' Jazz project has been discussed in detail.

This thesis makes a number of key contributions:

- Data Mining Software Metrics
    - This is the first study that has extracted software metrics from Jazz builds to generate prediction models for software build outcomes. From this stage prediction models that have been built using the software metrics derived from IBMs' Rational Analyzer tool provided the highest levels of classification accuracy out of the various metric aggregations investigated.
- Combining Software and Social Network Metrics
    - This work also presents the investigation of merging software metrics and communication metric types. From this it was observed that while software metrics and social network metrics can be used to construct models for predicting build outcomes separately, it was found that it was best not to merge the two metric types.
- Software Metrics as Data Streams
    - This is the first study to apply data stream mining methods (Hoeffding tree) to software metrics to predict software build outcomes.

- Social Network Metrics as Data Streams
    - This is the first study to apply data stream mining methods (Hoeffding tree) to social network metrics to predict software build outcomes.
- Application of SMOTE
    - Contributes to the body of knowledge of the application of SMOTE to software metrics, social network metrics and a combination of the two used in conjunction with traditional data mining and data stream mining methods.
- Proactive Management of Development Activities
    - The main outcome of this thesis is the combination of the above contributions into a systematic method for proactive management of development activities on the basis of perceived risk determined by software and social network metric analysis.

The outcomes of the initial experiments are a systematic attempt to predict build success and failure for a software project by utilizing source code metrics. The goal of these experiments is not to build a single predictive model, but to explore which combinations of data mining algorithms and data set offer potential for further study and application in software development environments. Prediction accuracies of 70-80% have been achieved through the use of the j48 classification algorithm using 10-fold cross validation. Despite this overall accuracy, there is greater difficulty in predicting failure than success and at present the classification trees' content display some uncertainty and confusion. However, the results show promise in terms of informing software development activities in order to minimize the chance of failure. While overall prediction accuracies slightly improved the prediction of failed builds remained problematic. This remained true during the frequency feature selection and application of the after state features to the before state experimental phases.

SMOTE was then introduced to increase the number of instances of the minority class (failed builds). In doing so while the classification of failed builds improved, when using software metrics, there is no single set of metrics that could be used to determine failure.

From exploring the communication metrics, again a similar theme emerged where prediction of failure remained a challenge. However, when applying SMOTE to the communication data little improvements were found in classifying failure. It was also found that, while using traditional data mining methods, there were no combinations of social and software metrics that could be used to determine build failure. However, data stream mining found combinations of both software and social network metrics to be significant for predicting build outcomes.

6.1 Limitations and Threats to Validity

There are limitations for incorporating the Jazz repository into research. To reiterate, the repository is highly complex and has huge storage requirements for tracking software artifacts. Another issue is that the repository contains holes and misleading elements which cannot be removed or identified easily. This is because the Jazz environment has been used within the development of itself. Therefore many features provided by Jazz were not implemented at early stages of the project. There is a great challenge in dealing with such inconsistency and the methodology has adopted an approach that delves further down the artifact chain than most previous work using Jazz. It is a premise that the early software releases were functional, so whilst the project "meta-data" may be missing details (such as developer comments) the source code should represent a stable system that can be analyzed to gain insight regarding the development project. Some of the holes in the repository are as a result of the Jazz team purposefully removing core libraries from the data released for analysis. Again, what is not clear is whether failure in one software module is actually caused by a code change in a dependent module. Looking at the root cause of a failure and where the failure is actually realised is an interesting area of future work.

The results presented within this thesis are only applicable to the Jazz project and may not be generalisable to any other software project. The prediction models built and results generated can only be directly compared to other studies that have made use of IBMs' Jazz Repository. Even when comparing to other Jazz studies there are concerns over validity that arise from the possibility of different extraction techniques being applied. However, the approach for creating a predictive model by mining data streams that relate to software and social network data should be able to be applied to other repositories and as such is a generalisable process. Similarly, the process of using the predictive model to identify build outcome risk and proactively manage the build scope

and activities is equally applicable to other projects. The actual prediction models and significant models are likely to be very different for other projects, but the techniques for developing them are entirely generic.

Other limitations from this study are products of the relatively small sample size of build data from the Jazz project combined with the sparseness of the data itself. For example, the ratio of metrics (42) to builds (199) is such that it is difficult to truly identify significant metrics. Whilst various strategies for reducing the number of metrics used in the classification have been investigated, this does not address the fundamental problem that the data set is rather small. Even though a sampling technique (SMOTE) is applied to increase the number of instances, there is no way of telling whether or not the generated instances accurately reflect real-world data. Cross-fold validation was also utilised to provide estimates for how well the generated models would perform in real-world scenarios.

For generating communication metrics similar assumptions about the construction of the network are made to the study presented by Wolf et al. (2009). Communication metrics were extracted from only work items that were included in software builds. While work items contain much discussion between contributors other channels that may have been used for communication were not included. For example other communication channels used by contributors include emails, face-to-face meetings, web-based forums and teleconferences. These channels were not captured as they were either not easily available or would be too invasive to capture. However, the risk to validity in this particular research is low. Whilst such communication events inevitably occur, one of the keystone decisions for the Jazz team in terms of managing their global virtual team is to specifically focus intra-team discussion through the Jazz collaboration environment itself. In the construction of the network itself, it is also assumed that every contributor commenting on or subscribing to a work item reads all comments made on that work item. However, upon manually reading the discussion of work items it is clear that contributors make reference to older comments within the discussion.

Despite these difficulties, the results show that there is much potential for predicting build success or failure on the basis of an analysis of source code that will be changed during a build, even when the degree of change is not known. Due to the relatively

small data set, this potential has not yet been fully realised. However, more clarity in the prediction is gained when the degree of change during a build is analysed. This provides the opportunity for development teams to incrementally examine their exposure to risk during the build cycle. Unfortunately access was not available to the live development which means that the models could not be applied to the live development process. Therefore the approach may not be well received or even used despite its potential to be useful to mitigate risks of build failure.

6.2 Future Work

This thesis has presented the results for data mining experiments of software and social network metric aggregations using a range of traditional and stream mining methods. While this research has gone some way to addressing the challenges associated with data mining software repositories, there is still much potential for future work in understanding evolving success and failure patterns found within the SDLC.

6.2.1 Exploring other Software Repositories

The methodologies used for this research are able to be applied to other software repositories provided they contain change sets for software builds and social network artifacts. Although studies of other repositories may not be directly comparable to this research, they may provide more insights into aspects of project success and failure. Within this study build success and failure were used to classify build instances. In other software repositories it would be interesting to investigate the levels of granularity used for classifying success and failure.

Unfortunately at the time of this work, IBM was not able to provide a subsequent snapshot, due to architectural technicalities in making the data anonymous and in removing sensitive source code. The removal of source code from the current repository should also be considered a significant threat to the validity of this work.

6.2.2 Merging Software and Social Metrics

When mining a combination of software and social metrics there were no social network metrics that were picked as being significant factors when using Subset Evaluation and Information Gain feature selection filters. Through visualising output of simple k-means clustering it is observed that there is still an overlap in feature space for both software and social metrics. There is a potential of future work to investigate whether social network metrics can provide additional insights into the prediction of

build failure when used with software metrics. This work would involve a hybrid of data modelling processes utilising both software and social metrics to predict an outcome. To do this more build instances are needed than what has been available during this research.

In the context of this work and providing there were more real instances of data available, a voting system could be utilised. This could be achieved through using either traditional mining methods or data stream mining methods. To begin, this process requires two data sets where "data set X" is the data set of software metrics and "data set Y" is the data set consisting of social metrics. Both data set X and Y have a build result (actual outcome) parameter that is used for classifying instances. Each instance from data set X and Y corresponds so that they both reference the same software build. As a result the number of instances in data set X will be the same number of instances in data set Y. Both data sets X and Y are split into a training set, a test set and a holdout test set. For instance the training set will consist of 70% of the original instances and will reflect the naturally occurring distribution of successful and failed build outcomes. The holdout data set will be necessary for testing the impact of the final merging of software and social metric data sets together. However, due to the limited number of instances this was not feasible to obtain. A potential data mining process, for future work, is illustrated in Figure 95.

Two models are built using the decision tree classifier. The first model ("model A") is generated from using data set X. This produces the same results to previous software metrics mining models, without using feature selection methods. The second model ("model B") is based on data set Y. Once both models are implemented, the instances from the test set of data set X are executed against model A and the instances from the test set of the Y data set are executed against model B. This produces two sets of "predicted build outcomes" for each build instance, one that is based entirely from software metrics and the other from social network metrics.

In addition to the predictions from models A and B a k-NN approach using Euclidean distance is also utilised. In this case the classifier works at instance level rather than group level where each instance will have its own unique blend of software metric bias and communication metric bias. This k-NN approach is not available using the GUI

(Graphical User Interface) from Weka, however Weka's API (Application Programming Interface) offers support for this method. A Java application is implemented which utilises this for processing instances. Using the data sets X and Y models A' and B' are built from the k-NN approach, where X is the set of features used by model A' and Y is the set of features used by model B'. Training instances (form data sets X and Y) are used to search for neighbouring instances for each (X and Y) test set instances. A class label is then assigned based from the model that has the higher probability. For example if p1 > p2 for an instance, then the class label from model A is assigned. Classification of an instance can also be achieved by a probability threshold. For example if a prediction probability is greater than a threshold set by the user, then the class label is assigned based on the predicted outcome from that model.

From the j48 classification predictions, if the two predicted build outcomes for a single build instance are different it indicates that there is a conflict between the use of software and social metrics. From this a new data set (data set Z) is formed based on these conflicts and in this case an instance is composed of a combination of software and social network metrics. From these conflicted instances a new model (model C) is generated. Model C is then implemented and tested against the holdout test set. The k-NN approach is also applied to the holdout test set. For this the nearest neighbours are searched using the training data set and the conflicted data set. Both sets are composed of software and social network metrics.

This model has been implemented and some initial experimentation conducted. However, the results could not be presented within this work as the need for test sets, trainings sets and a hold out sets resulted in very little data to train from. Training sets were very small compared to the 100 instances required from the data stream results for stabilised classification accuracies. Nevertheless, from using this method, it was found that 94% of instances were correctly classified when the software metric and social network models did not conflict (145/153 non conflicted instances correctly classified, using the j48 classifier, with no filtered features). There were 46 instances that had conflicting predictions, of which 35 instances were failed builds.

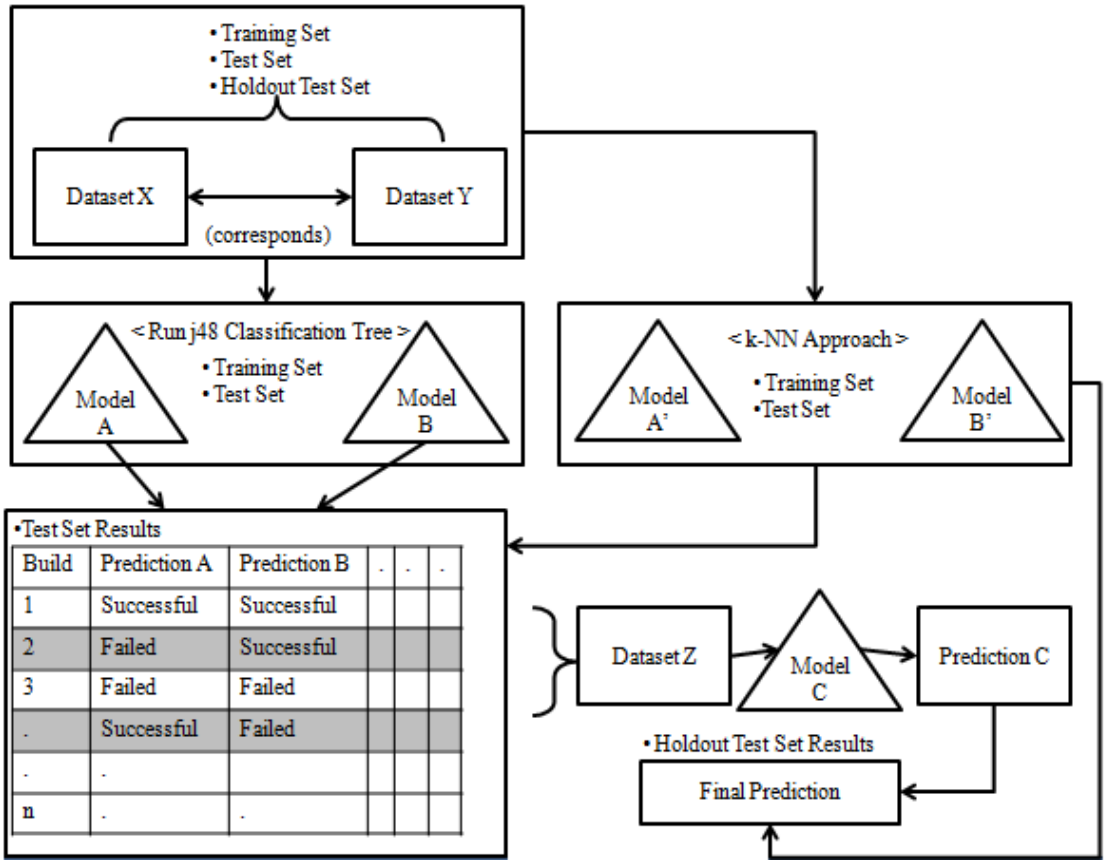

Figure 95 Future Work: Predicting Build Outcome from Software and Social Metrics Using Data Mining and Voting Logic

Predicted build outcome instances that are the same from using both models A and B indicates a higher probability of correctly classifying an instance. However, for conflicted instances the results from Model C are used. This provides an additional vote for conflicted instances, where two out of the three predictions determines the class label. For the k-NN predictions this vote approach is also applied and the results from the k-NN and j48 models are compared for accuracy. Metrics can also be changed from the voting stage to see if other metrics are better indicators for build outcomes.

It is difficult to extract real knowledge from the experiments on the existing data using this approach due to the lack of data. However, some statistical analysis of the differences between builds that result in conflict when compared to builds where the predicted outcome was the same from the social and software metric models are quite insightful and indicate that there is potential to understand how failure is represented through metric values.

Table 34 shows a very simple statistical analysis of the average values of certain key metrics. This is broken down into values for the complete data set and then the outcomes of the first stage of the voting logic described above. This makes it possible to compare the instances where conflict occurs to both the non-conflict set and the full set of instances.

Table 34 Average Metric Values for Full, Conflicted and Non-Conflicted Data Sets

Software Metric	**Full (Average)**	**Conflict Set (Average)**	**Non-Conflict Set (Average)**
Average block depth	1.724	1.656	1.818
Weighted methods per class	4574.953	3292	2524.4
Maintainability index	260.155	266.922	262.252
Instability	0.467	0.418	0.526
Number of unique operators	36.813	38.2	38.4
Program vocabulary size	797.973	780.6	876.8
Depth of Inheritance	0.541	0.768	0.528
Number of attributes	508.693	386.8	168.2
Average number of constructors per class	1.047	4.76	3.46
Average number of methods	23.792	16.95	26.64

It can be seen from this data that there is a statistical difference between the metrics associated with conflict between software and social metric predictions and those where there is no conflict in the prediction. This is a very simple analysis and should not be

over-emphasised, however it does indicate that there is a potential to further refine the understanding of how failure can be indicated through a more targeted model.

This conflict model may also be applied using the Hoeffding tree mining of data-streams and applied by the real development team. Assuming that sufficient builds exist to overcome the over-fitting that has been observed in the results chapter, the conflict approach can be used to develop four models based on four data-streams. Upon completion of a build the final set of software and social network metrics are applied to the classifiers and the final predicted outcome determined. If the software and social network classifiers produce the same predicted outcome then the "normal" data-streams are updated irrespective of actual build outcome. However, if the social and software metric based prediction differs then the build data is added to the conflict data-streams instead. During use, the normal data-streams are used to classify the outcome of a build and only when the two outcomes differ the conflict models applied. This results in a two stage risk assessment process during a build. Again, the only concern is the need for at least 90 conflict builds to have some confidence in the predictive power of the conflict model.

There is also possible future work that could investigate the nature of the cause of failure, in particular the concept that changes in the source code in one module can actually cause a failure in another module. Understanding the way that source code propagates failure through a software structure would provide great insight in the actual cause of failure. However, this work would not use data mining and is therefore out of scope of this thesis.

6.3 Chapter Summary

There are no absolute truths in software engineering, the goal of this research was to improve on existing software development practices and potentially produce technology that adds value to the evolutionary IT world. This thesis presents the outcomes of a systematic attempt to predict build success or failure for a software product by utilizing source code and social network metrics. Prediction accuracies of 70-80% have been achieved through the use of decision trees, Naive Bayes, Bayesian Networks and k-NN methods. Despite this overall accuracy, there is greater difficulty in predicting failure than success and at present the classification trees contained some uncertainty and confusion. However, the results show potential in terms of informing software development activities in order to minimize the chance of failure. The difficulty in classifying failed builds can be due to a number of reasons: 1) There is too much overlap within the feature space for successful and failed builds, 2) The metrics explored simply do not capture failure entirely, 3) A combination of the metrics explored do not capture failure entirely, 4) more instances of failure are required and 5) there is a need for more data of both classes.

The early phases of this work involved the building of static classification models based on the entire set of historical data in the Jazz repository. This has been extended to show that it is possible to build models over time using approaches that consider the emergence of software and social network metrics as data-streams. These models offer much value in terms of guiding software development teams towards understanding the risk in their development activities, however it is clear that there is a minimum volume of data required to have some confidence in the models. In the case of the Jazz project it appears that confidence in the model develops after around 90 builds have been included.

The real-time streams can be run against the model which has been generated from software build and work item histories. From the real-time based predictions developers may delay a build to proactively make changes on a failed build prediction. It would also provide real-time insights into the way the team is communicating effectively for generating a successful build. One of the advantages of building predictive models using data stream mining methods is that they do not have large permanent storage constraints. The main reason why Jazz only stores a limited number of build change sets is because of the huge storage requirements.

This chapter has briefly presented a summary of this research study as well as the implications of the experimental results in the context of the objectives of this thesis. The frameworks and results of this work have much potential for future investigations. This final chapter outlines potential future work and presents conclusions based on the contributions made by this thesis.

Appendices

Appendix A: Software Metrics

There are many different types of software metrics that can be extracted from source code. The software metric categories commonly found within the literature include ranges of basic software metrics (Table 35 and Table 36), dependency metrics (Table 37), complexity metrics (Table 38) cohesion metrics (Table 39), Halstead metrics (Table 40) and the inheritance metric (Table 41). Basic software metrics provide simple counts of various source code elements providing indications towards a software systems' size. Basic software metrics are also used towards calculating other metric categories. Basic average metrics (Table 36) provide similar information at a package, class or method level and make use of basic software metrics. Both categories of basic metrics are useful in generating a general understanding of the size and design of the software system artifacts. The following tables reflect metrics that are extracted from Java source code and are explored within this research.

Table 35 Basic Software Metrics

Metric Name	Formula	Description
Number of Types per Package	Total number of classes/objects within the project	Derived from overall project or for each project package.
Number of Comments (NoC)	Total number of comments	Derived from overall project or for each project package or class. Including Java doc and "in code" comments e.g. // and /** notations
Lines of Code (LoC)	Total lines of executable code	Derived from overall project or for each project package or class. In code comments and empty lines are omitted in this count
Comment/Code Ratio	$\frac{NoC}{LoC} \times 100$	Derived from overall project or for each project package, class or method.

Number of Import Statements	The total number of imports within all classes	Derived from overall project or for each project package or class.
Number of Interfaces	Total number of interface classes/objects	Derived from overall project or for each project package or class.
Number of Methods (NoM)	Total number of methods	Derived from overall project or for each project package or class. Constructors are omitted from this count. This count includes "getter" and "setter" property functions.
Number of Parameters (NoP)	Total number of parameters	Derived from overall project or for each project package or class. This metric includes constructor parameters.
Number of Lines (NoL)	Total number of lines	Derived from overall project or for each project package or class. These includes lines that have no code and comments. The number of lines count stops after the final } is found within a class.

Table 36 Basic Average Metrics

Average Number of Attributes Per Class	$\frac{Number\ of\ attributes}{Number\ of\ Types/Objects}$	Derived from overall project or for each project package
Average Number of Constructors Per Class	$\frac{Number\ of\ Constructors}{Number\ of\ Types/Objects}$	Derived from overall project or for each project package
Average Number of Comments	$\frac{Number\ of\ Comments}{Number\ of\ Packages}$	Derived from overall project or for each project package or at class (type) level. At class level metric is the total number of comments.
Average Lines of Code Per Method	$\frac{(LoC/NoM)}{10}$	Derived from overall project or for each project package or at class (type) level.

Average Number of Methods	$\frac{NoM}{Number\ of\ Packages}$	Derived from overall project or for each project package.
Average Number of Parameters	$\frac{Number\ of\ Parameters}{NoM}$	Derived from overall project or for each project package.

The dependency metrics that are presented in Table 37, provide insights into relationships between software modules by assessing the systems' abstractness and stability. If a software system has good abstraction, a change in one module will not require multiple changes in other modules which make use of it. As a result one module does not depend on other modules in order to function (the lower the coupling, the better) (Martin, 1994) (Harman, 2007). Dependency metrics are used towards understanding the quality of a systems' design, reusability and maintainability by making use of class, interface and references metrics.

Table 37 Dependency Metrics

Abstractness (A)	$\frac{Number\ of\ Abstract\ Types}{Number\ of\ Types}$	Derived from overall project or for each project package
Afferent Coupling (Ca)	$NoICR + NoECR$ Where *NoICR* is the Number of Internal Class References and *NoECR* is the Number of External Class References.	Derived from overall project or for each project package or at class (type) level. An internal class reference is where a class references another class within its own package. An external class reference is where a class references another class outside of its own package.
Efferent Coupling (Ce)	*Total number of times a class makes references outside its own package*	Derived from overall project or for each project package or at class (type) level.
Instability (I)	$\frac{Ce}{(Ca + Ce)}$ Where *Ce* is Efferent coupling and *Ca* is Afferent Coupling.	Derived from overall project or for each project package or at class (type) level.
Normalized Distance	$(I + A) - 1$	Derived from overall project or for each project package or at class (type) level. This metric provides us data about if packages are balanced and has a good mix between abstraction and instability.

Complexity metrics (Table 38) indicate the complexity of the software and can be used towards design effort and software maintenance estimation. Developed by McCabe in 1976, cyclomatic complexity measures the number of possible independent pathways through source code. In order to do this a control flow graph of the program is generated, where nodes within the graph represent a source code command, or sequence of commands and an edge that connects nodes represents the possible order of execution. For example if the source code consists of an "IF" statement of a single condition there are two possible pathways the code could be executed. One pathway is taken if the resulting value of the "IF" statement is TRUE and the other pathway if the

value is FALSE. Whereas if the code consists of no loops or "IF" statements, there is only one possible pathway the code could be executed, therefore there would only be one pathway for execution.

Table 38 Complexity Metrics

Average Block Depth	$\frac{(NoM + NoConstr)}{NoNC}$ Where *NoM* is the number of methods, *NoConstr* is the number of constructors and *NoNC* is the number of nested code	Derived from overall project or for each project package, class or method. Try/catch are included in the nested count. Value is 1 if the code is not nested. High average block depth values are considered detrimental.
Average Cyclomatic Complexity	$\frac{CC}{(NoM + NoConstr)}$ Where *CC* is the Cyclomatic Complexity. Cyclomatic complexity is calculated by the number of pathways that can be taken within the code. $CC = NoE - NoN + 2$ Where *NoE* is the number of edges and *NoN* is the number of nodes. *NoConstr* is the number of constructors	Derived from overall project or for each project package, class or method. Cyclomatic complexity per method is the total number of pathways that the code within that method can take.

Cohesion metrics provide insights into the responsibilities and design of software modules. High cohesion values indicate the reusability and readability of source code.

A cohesive software component is analysed by looking at how often methods share access to fields (Chae, Kwon & Bae, 2000). A cohesive software system will have good class subdivision and a cohesive class will have a high degree of encapsulation. Lack of cohesion metrics (LCOM) with high measured values indicate a lower level of cohesion.

There are three different types of LCOM each with their advantages and disadvantages. For instance LCOM1 can be used to identify classes that are consisting of many different objectives and are likely to have less predictable behaviours. However, one of the disadvantages of using LCOM1 is that it can provide a value of zero for classes that are very different and it is not well suited for classes that make use of their own properties (i.e. getter and setters methods have high LCOM1 values). It is easy to misinterpret the meaning of LCOM1 and it may not necessarily be the best method for measuring the cohesiveness of classes that fall within the object-orientated software paradigm.

In order to overcome the limitations of LCOM1, LCOM2 and LCOM3 were created. Both LCOM2 and LCOM3 are similar metrics and a low value again indicates high cohesion. However, the limitations of using any LCOM metrics is that they consider variables that may not be in use. Therefore it is important to remove any "dead" variables from the source code in order to eliminate wrong interpretations of results.

Table 39 Cohesion Metrics

Lack of Cohesion 1 (LCOM1)	The total number of pairs of methods that do not share instance variables	Derived from overall project or for each project package or at class (type) level. A higher LCOM indicates lower cohesion. LCOM1 indicates if a class needs to be split into more classes as variables exist within disjointed sets. High values in LCOM1 are found to be fault prone. A cohesive class will have a value of 0.

Lack of Cohesion 2 (LCOM2)	$LCOM2 = P - Q \; if \; P > Q$ $Else \; LCOM2 = 0$ For each pair of methods in a class if they access different sets of instance variables *P* is increased by 1. For every variable they share *Q* is increased by 1.	Derived from overall project or for each project package or at class (type) level. If the number of methods or variables is 0, then LCOM2 is 0. LCOM2. If the metric value is greater than or equal to 1 then the class if considered to be problematic in terms of cohesion.
Lack of Cohesion 3 (LCOM3)	For LCOM3 consider an undirected graph where the vertices are methods of a class and an edge is the corresponding methods that share at least one instance variable. $LCOM3 = \left(\frac{NoP - sum(mA)}{a}\right)(m - 1)$ Where *NoP* is the number of procedures, *mA* is the total number of methods that access a variable (attribute) and *a* is the number of attributes	Derived from overall project or for each project package or at class (type) level. LCOM3 has a range of 0-1 and has no single threshold value.

Halstead metrics (Table 40) provide software project estimate values by measuring the complexity of software through operands and operators from code. Empirical studies have shown that Halstead metrics can be used to estimate the level of difficulty required to understand, implement, maintain and debug software modules.

Table 40 Halstead Metrics

Number of Operands	Total Number of Operands These include: Identifiers: All identifiers that are not reserved words Programming Types: Reserved words that specify type. Eg: bool, char, float, double, int, long, short and void etc. Programming Constants: Character, numeric or string constants	Derived from overall project or for each project package, class or method.
Number of Operators	Total Number of Operators Eg: "+", "/", "-", "*" etc	Derived from overall project or for each project package, class or method. Note that comments are not counted when calculating operands or operators
Number of Unique Operands	*Number of operands that are unique*	Derived from overall project or for each project package, class or method.
Number of Unique Operators	*Number of operands that are unique*	Derived from overall project or for each project package, class or method.
Program Volume	$Program\ Volume = Program\ Length \times \log\{Program\ Vocabulary\}$	

Difficulty Level	$$D = \frac{n1}{2} \times \frac{N2}{n2}$$ Where *D* is the Difficulty level, *n1* is the number of unique operators, *n2* is the number of unique operands and *N2* is the total number of operands	Derived from overall project or for each project package, class or method.
Effort to Implement (E)	$$D * V$$ Where *D* is the Difficulty Level and *V* is the Program Volume	Derived from overall project or for each project package, class or method.
Number of Delivered Bugs (NoDB)	$$E^{(\frac{2}{3}))/3000}$$ Where *NoDB* is the number of delivered bugs and *E* is the Effort to Implement metric	Derived from overall project or for each project package, class or method.
Time to Implement (T)	$T = E / 18$ Where *E* is the Effort to Implement	Derived from overall project or for each project package, class or method. The time to implement or understand a program (T) is proportional to the effort. Halstead has found that dividing the effort by 18 give an approximation for the time in seconds
Program Length	$$Program\ Length = Number\ of\ Operands + Number\ of\ Operators$$	Derived from overall project or for each project package, class or method.

Program Level	L = 1 / D Where *D* is the Difficulty Level metric	Derived from overall project or for each project package, class or method. The program level (L) is the inverse of the error proneness of the program. I.e. a low level program is more prone to errors than a high level program.
Program Vocabulary Size	*Number of unique operands + number of unique operators*	Derived from overall project or for each project package, class or method.

The depth of inheritance metric is presented in Table 41. A high level of nested inheritance can indicate highly complex and coupled software components. For example if one class inherits from another its depth is 2. If a class does not inherit from other classes its depth is one.

Table 41 Inheritance Metric

Depth of Inheritance (DoI)	$\frac{LoLP}{AHR}$ Where LoLP is the length of the longest path from a given module and AHR is the aspect hierarchy root.	Derived from overall project or for each project package or at class (object) level.

Appendix B: Social Network Metrics

- Density
    - Measurement of the connectivity between team members
    - How knowledge is distributed
    - Is calculated as a percentage of the existing connections to all possible connections within the social network. A fully connected network has the density of 1.
- Centrality and Team members positions (a.k.a. the ego network)
    - Measures the importance of actors within the social network
    - Out-degree (number outgoing connections), In-degree (number of incoming connections) and In-Out-Degree (the sum of In-Degree and Out-degree)
    - A probability index for each node is generated that assumes that communication takes the shortest path from one node to another
    - Measures the extent of which a person is in between two other people. That is, the “actors in the middle” and “interpersonal influence”
- Structural holes
    - Measures any missing links between nodes and the redundancies within the social network.
    - The effective size of a node is the number of direct neighbours minus the average degree of those nodes' ego network (not including the number of connections).

Appendix C: Before State Software Metrics Results

[Contents on CD]

Appendix D: After State Software Metrics Results

[Contents on CD]

Appendix E: Frequency Feature Selection Thresholds

[Contents on CD]

Features Selected for the Before State:

The first iteration features (where features appeared at least once were included) are: Average block depth, Weighted methods per class, Maintainability index, Cyclomatic complexity, Abstractness, Afferent coupling, Efferent coupling, Instability, Normalized Distance, Number of operands, Number of operators, Number of unique operands, Number of unique operators, Number of delivered bugs, Difficulty level, Effort to implement, Time to implement, Program length, Program level, Program vocabulary size, Program volume, Lack of cohesion 2, Lack of cohesion 3, Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average number of comments, Average lines of code per method, Average number of methods, Average number of parameters, Number of types per package, Comment/Code Ratio, Number of constructors, Number of import statements, Lines of code, Number of comments, Number of methods, Number of parameters, Number of lines

For the second iteration where features that appeared twice or more were included are: Average block depth, Weighted methods per class, Maintainability index, Afferent coupling, Efferent coupling, Instability, Normalized Distance, Number of operands, Number of unique operands, Number of unique operators, Number of delivered bugs, Difficulty level, Time to implement, Program length, Program level, Program vocabulary size, Lack of cohesion 2, Lack of cohesion 3, Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average number of comments, Average lines of code per method, Average number of methods, Average number of parameters, Number of types per package,

Comment/Code Ratio, Number of constructors, Number of import statements, Number of comments, Number of methods, Number of parameters, Number of lines

For the third iteration features that appeared 3 times or more were included. These features are: Average block depth, Maintainability index, Afferent coupling, Normalized Distance, Number of operands, Number of unique operators, Number of delivered bugs, Difficulty level, Program length, Program level, Program vocabulary size, Lack of cohesion 2, Depth of Inheritance, Number of attributes, Average number of constructors per class, Average number of comments, Average lines of code per method, Average number of methods, Number of types per package, Comment/Code Ratio, Number of constructors, Number of comments, Number of lines.

For the fourth iteration features that appeared 4 or more times were included. These features are: Maintainability index, Afferent coupling, Number of operands, Number of unique operators, Number of delivered bugs, Difficulty level, Program vocabulary size, Lack of cohesion 2, Number of attributes, Average number of constructors per class, Average number of methods, Number of types per package, Comment/Code Ratio, Number of constructors, Number of comments, Number of lines.

The fifth iteration features that appeared 5 or more times were included. These features are: Maintainability index, Number of unique operators, Difficulty level, Number of attributes, Average number of constructors per class, Average number of methods, Number of types per package

For the final iteration features that appeared 6 or more times were included. These features are: Maintainability index, Number of unique operators, Number of attributes, Average number of constructors per class, Average number of methods, Number of types per package

Features Selected for the After State:

For each feature frequency iteration the features selected are as follows:

The first iteration features who appeared at least once were included. These features are: Average block depth, Weighted methods per class, Maintainability index, Cyclomatic complexity, Abstractness, Afferent coupling, Efferent coupling, Instability, Normalized Distance, Number of operands, Number of operators, Number of unique operands, Number of unique operators, Number of delivered bugs, Difficulty level, Time to implement, Program length, Program level, Program vocabulary size, Program volume, Lack of cohesion 2, Lack of cohesion 3, Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average number of comments, Average lines of code per method, Average number of methods, Average number of parameters, Number of types per package, Comment/Code Ratio, Number of constructors, Number of import statements, Number of comments, Number of lines

For the second iteration features that appeared twice or more were included. These features are: Average block depth, Weighted methods per class, Maintainability index, , Abstractness, Afferent coupling, Efferent coupling, Instability, Normalized Distance, Number of operands, Number of unique operands, Number of unique operators, Number of delivered bugs, Difficulty level, Time to implement, Program level, Program vocabulary size, Lack of cohesion 2, Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average lines of code per method, Average number of methods, Average number of parameters, Number of types per package, Comment/Code Ratio, Number of constructors, Number of import statements, Number of comments, Number of lines.

For the third iteration features that appeared 3 times or more were included. These features are: Average block depth, Weighted methods per class, Maintainability index, Abstractness, Afferent coupling, Instability, Number of unique operators, Difficulty level, Lack of cohesion 2, Depth of Inheritance, Number of attributes, Average number of attributes per class, Average number of constructors per class, Average number of methods, Number of types per package, Comment/Code Ratio, Number of import statements, Number of comments, Number of lines.

For the fourth iteration features that appeared 4 or more times were included. These features are: Weighted methods per class, Maintainability index, Abstractness, Depth of Inheritance, Number of attributes, Average number of constructors per class, Average number of methods, Comment/Code Ratio, Number of comments, Number of lines

The fifth iteration features that appeared 5 or more times were included. These features are: Abstractness, Depth of Inheritance, Number of attributes, Average number of constructors per class, Average number of methods, Number of comments

For the final iteration features that appeared 6 or more times were included. These features are: Abstractness, Number of Attributes, Average Number of Constructors

References

Agrawal, M. and Chari, K. (2007). Software Effort, Quality and Cycle Time: A Study of CMM Level 5 Projects. *IEEE Transactions on Software Engineering, 33*(3), 145 - 156.

Ahsan, S., Afzal, M., Zaman, S., Gütel, C. & Wotawa F. (2010). Mining Effort Data from the OSS Repository of Developer's Bug Fix Activity. *Journal of IT in Asia, 3*, 67-80.

Al-Fayoumi, M., Banerjee, S. and Mahanti P.K. (2009). Analysis of Social Network Using Clever Ant Colony Metaphor. *World Academy of Science, Engineering and Technology, 53*, 970 - 974.

Amor, J., Robles, G. and Gonzalez-Barahona, J. (2006). Effort Estimation by Characterizing Developer Activity. *International Conference on Software Engineering, Proceedings of the 2006 International workshop on Economics Driven Software Engineering Research*, 3 - 6.

Babcock, B., Babu, S., Datar, M., Motwani, R. & Widom, J. (2002). *Models and issues in data stream systems*. Paper presented at the Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, Madison, Wisconsin.

Babcock, B., Datar, M. & Motwani, R. (2002). *Sampling from a moving window over streaming data*. Paper presented at the Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, San Francisco, California.

Babu, S. & Widom, J. (2001). Continuous queries over data streams. *SIGMOD Rec., 30*(3), 109-120. doi: 10.1145/603867.603884.

Baena-García, M., del Campo-Ávila, J., Fidalgo, R., Bifet, A., Gavaldà, R. & Morales-Bueno, R. (2006). Early Drift Detection Method. *ECML PKDD 2006 Workshop on Knowledge Discovery from Data Streams*.

Barretoa, A., de O. Barrosb, M. & Wernera, C. M. L. (2008). Staffing a software project: A constraint satisfaction and optimization-based approach. *Computers & Operations Research, Brazil, 35*(10), 3073-3089.

Basili, V. R., Briand, L. C. and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 10*(22), 751–761.

Begel, A. & DeLine, R. (2009, 16-24 May 2009). *Codebook: Social networking over code.* Paper presented at the Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference.

Belady, L. A. & Lehman, M. M. (1976). A model of large program development. *IBM Systems Journal, 15*(3), 225-252. doi: 10.1147/sj.153.0225.

Bibi, S., Tsoumakas, G., Stamelos, I. & Vlahavas, I. (2008). Regression via Classification applied on software defect estimation. *Expert Systems with Applications, 34*(3), 2091 - 2101.

Bifet, A. (2009). Adaptive learning and mining for data streams and frequent patterns. *SIGKDD Explor. Newsl., 11*(1), 55-56. doi: 10.1145/1656274.1656287.

Bifet, A. & Frank, E. (2010). Sentiment Knowledge Discovery in Twitter Streaming Data. Discovery Science. Springer Berlin/Heidelberg.

Bifet, A., Holmes, G., Pfahringer, B., Read, J., Kranen, P., Kremer, H., . . . Seidl, T. (2011). MOA: A Real-Time Analytics Open Source Framework. *Machine Learning and Knowledge Discovery in Databases*, 617-620.

Boehm, B. W. (1984). Software Engineering Economics. *Software Engineering, IEEE Transactions on, SE-10*(1), 4-21. doi: 10.1109/tse.1984.5010193.

Bolstad, C. A. & Endsley, M. R. (2003). Tools for Supporting Team Collaboration. *Human Factors and Ergonomics Society Annual Meeting Proceedings, 47*, 374-378.

Braga, P., Oliveira, A., Ribeiro, G. & Meira, S. (2007). Bagging Predictors For Estimation of Software Project Effort. *Proceedings of International Joint Conference on Neural Networks*, 1595-1601.

Breiman, L., J. Friedman, R. Olshen and C. Stone. (1984). Classification and Regression Trees. *Belmont, CA: Wadsworth International Group.*

Brooks, G. (2008, 4-8 Aug. 2008). *Team Pace Keeping Build Times Down.* Paper presented at the Agile, 2008. AGILE '08. Conference.

Browning, T. (2001). Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. *IEEE Transactions on Engineering Management, 48*(3), 292-306.

Buffenbarger, J. (2005). A large-scale fault-tolerant distributed software-build process. *British Computer Society Configuration Management Specialist Group Conference.*

Buse, R. P. L. & Zimmermann, T. (2010). *Analytics for software development*. Paper presented at the Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA.

Calefato, F., Gendarmi, D. & Lanubile, F. (2009). Adding social awareness to Jazz for reducing socio-cultural distance between distributed teams. *Eclipse-IT 2009, 4th Italian Workshop on Eclipse Technologies, Bergamo, Italy*, 19-28.

Cannon-Bowers, J. A., Salas, E. & Converse, S. . (1993). Shared mental models in experts team decision making. *Journal of applied psychology, 85*(2), 221-246.

Carmel, E. & Agarwal, R. (2001). Tactical approaches for alleviating distance in global software development. *Software, IEEE, 18*(2), 22-29. doi: 10.1109/52.914734

Cerpa, N. & Verner, J. M. (2009). Why did your project fail? *Commun. ACM, 52*(12), 130-134. doi: 10.1145/1610252.1610286

Chae, H., Kwon, Y. & Bae, D. (2000). A cohesion measure for object-oriented classes. *Software - Practice and Experience, 30*, 1405–1431.

Challagulla, V. U. B., Bastani, F. B., Yen, I. L. & Paul, R. A. (2005, 2-4 Feb. 2005). *Empirical assessment of machine learning based software defect prediction techniques.* Paper presented at the Object-Oriented Real-Time Dependable Systems. 10th IEEE International Workshop.

Chau, D., Pandit, S. & Faloutsos, C. (2006). Detecting Fraudulent Personalities in Networks of Online Auctioneers. *Knowledge Discovery in Databases 4213*, 103-114. doi: 10.1007/11871637_14

Chawla, N. (2010). Data mining for imbalanced data sets: An overview. *Data Mining and Knowledge Discovery Handbook*, 875 - 886.

Chawla, N. (2010). Data Mining for Imbalanced Data sets: An Overview Data Mining and Knowledge Discovery Handbook. In O. Maimon & L. Rokach (Eds.), (pp. 875-886): Springer US.

Chawla, N., Bowyer, K., Hall, L. & Kegelmeyer, W. (2002). SMOTE: Synthetic Minority Over-sampling TEchnique. *Journal of Articial Intelligence Research, 16*, 341-378.

Chawla, N., Cieslak, D., Hall, L. & Joshi, A. (2008). Automatically countering imbalance and its empirical relationship to cost. *Data Mining and Knowledge Discovery, 17*(2), 225-252. doi: 10.1007/s10618-008-0087-0

Chen, A., Chou, E., Wong, J., Yao, A. Y., Qing, Z., Shao, Z. & Michail, A. (2001, 2001). *CVSSearch: searching through source code using CVS comments.* Paper

presented at the Software Maintenance. Proceedings. IEEE International Conference.

Cheng, P., Chulani, S., Ding, Y. B., Delmonico, R., Dubinsky, Y., Ehrlich, K., . . . Ying, A. (2008). Jazz as a research platform: experience from the Software Development Governance Group at IBM Research. *First International Workshop on Infrastructure for Research in Collaborative Software Engineering (IRCoSE) at FSE*.

Chicano, F. & Alba, E. (2005). Software Project Management with GAs. *Paper presented at the 6th Meta-heuristics International Conference, Vienna, Austria, 177*(11), 2380-2401

Christensen, C. & Albert, R. (2007). Using Graph Concepts to Understand the Organization of Complex Systems. *International Journal of Bifurcation and Chaos, 17*(0), 2201. doi: 10.1142/S021812740701835X

Coleman, D., Ash, D., Lowther, B. & Oman, P. (1994). Using metrics to evaluate software system maintainability *Computer, 27*(8), 44 - 49 doi: 10.1109/2.303623

Conchúir, E., Ågerfalk Pär, Olsson, H. & Fitzgerald, B. (2009). Global software development: where are the benefits? *Commun. ACM, 52*(8), 127-131. doi: 10.1145/1536616.1536648

Curtis, B., Sheppard, S. B. & Milliman, P. (1979). *Third time charm: Stronger prediction of programmer performance by software complexity metrics*. Paper presented at the Proceedings of the 4th international conference on Software engineering, Munich, Germany.

Denaro, G., Morasca, S. & Pezz, M. (2002). *Deriving models of software fault-proneness*. Paper presented at the Proceedings of the 14th international conference on Software engineering and knowledge engineering, Ischia, Italy.

Denning, P. J. & Riehle, R. D. (2009). The profession of IT: Is software engineering engineering? *Commun. ACM, 52*(3), 24-26. doi: 10.1145/1467247.1467257

Dick, S., Meeks, A., Last, M., Bunke, H. & Kandel, A. (2004). Data mining in software metrics databases. *Fuzzy Sets and Systems, 145*(1), 81-110. doi: DOI: 10.1016/j.fss.2003.10.006

Domingos, P. & Hulten, G. (2000). *Mining high-speed data streams*. Paper presented at the Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, Massachusetts, United States.

Drown, D. J., Khoshgoftaar, T.M. Seliya, N. (2009). Evolutionary Sampling and Software Quality Modeling of High-Assurance Systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions 39*(5), 1097-1107. doi: 10.1109/TSMCA.2009.2020804

Duvall, P., Matyas, S. & Glover, A. (2007). *Continuous integration: improving software quality and reducing risk*: Addison-Wesley Professional.

Ebert, C. & De Neve, P. (2001). Surviving global software development. *Software, IEEE, 18*(2), 62-69. doi: 10.1109/52.914748

Fayyad, U., Piatetsky-Shapiro, G. & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine, 7*(3), 37-54.

Fenton, N. & Neil, M. (2000). Software Metrics: Roadmap. *ACM, Future of Software Engineering, Limerick, Ireland*, 357-369.

Fenton, N. E. & Neil, M. (1999). A critique of software defect prediction models. *Software Engineering, IEEE Transactions on, 25*(5), 675-689. doi: 10.1109/32.815326

Forman, G. (2003). An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res., 3*, 1289-1305.

G. Hulten, L. S. and P. Domingos. (2001). Mining time-changing data streams. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining KDD 01*, 97–106. doi: 10.1145/502512.502529

Gaber, M. M., Zaslavsky, A. & Krishnaswamy, S. (2005). Mining data streams: a review. *SIGMOD Rec., 34*(2), 18-26. doi: 10.1145/1083784.1083789

Gao, K., Khoshgoftaar, T. M. & Napolitano, A. (2011, November). Impact of data sampling on stability of feature selection for software measurement data. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on* (pp. 1004-1011). IEEE.

Gao, K., Khoshgoftaar, T. M., Wang, H. & Seliya, N. (2011). Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience, 41*(5), 579-606. doi: 10.1002/spe.1043

Gilb, T. (1977). *Software Metrics*: Winthrop Publishers (Cambridge, Mass.)

Golab, L., DeHaan, D., Demaine, E. D., Lopez-Ortiz, A. & Munro, J. I. (2003). *Identifying frequent items in sliding windows over on-line packet streams*. Paper

presented at the Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, Miami Beach, FL, USA.

Gray, A. R. & MacDonell, S. G. (1997). A comparison of techniques for developing predictive models of software metrics. *Information and software technology, 39*(6), 425-437. doi: 10.1016/s0950-5849(96)00006-7

Gray, D., Bowes, D., Davey, N., Sun, Y. & Christianson, B. (2009). Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics. *Engineering Applications of Neural Networks*, 223-234

Engineering Applications of Neural Networks. In D. Palmer-Brown, C. Draganova, E. Pimenidis & H. Mouratidis (Eds.), (Vol. 43, pp. 223-234): Springer Berlin Heidelberg.

Grimstad, S., Jørgensen, M. & Moløkken-Østvold, K. (2006). Software effort estimation terminology: The tower of Babel. *Information and Software Technology, 48*(4), 302-310.

Guimerà, R., Uzzi, B., Spiro, J. & Amaral, L. (2005). Team Assembly Mechanisms Determine Collaboration Network Structure and Team Performance. *Science, 308*(5722), 697-702. doi: 873529361

Haibo, H. & Garcia, E. A. (2009). Learning from Imbalanced Data. *Knowledge and Data Engineering, IEEE Transactions on, 21*(9), 1263-1284. doi: 10.1109/tkde.2008.239

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. (2009). The WEKA Data Mining Software: An Update; SIGKDD Explorations. *11, 1*, 10-18. doi: 10.1145/1656274.1656278

Hall, M., Frank, E., Holmes, G. & Pfahringer B. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations, 11*(1).

Hanneman, R. A. & Riddle, M. (2005). Introduction to social network methods Retrieved from http://wiki.gonzaga.edu/dpls707/images/6/6e/Introduction_to_Social_Network_Methods.pdf

Harman, M. (2007). The Current State and Future of Search Based Software Engineering. *Future of Software Engineering (FOSE '07).*

Harman, M. & Tratt, L. (2007). Pareto Optimal Search-based Refactoring at the Design Level. *Proceedings of the 9th annual conference on Genetic and evolutionary computation.* 1106 - 1113.

Hassan, A. E. (2006, 24-27 Sept. 2006). *Mining Software Repositories to Assist Developers and Support Managers.* Paper presented at the Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference.

Hassan, A. E. (2008). *The road ahead for Mining Software Repositories.* Paper presented at the Frontiers of Software Maintenance, 2008. FoSM 2008.

Herbsleb, J., Cataldo, M., Damian, D., Devenbu, P., Easterbrook, S. & Mockus, A. (2008). *Socio-technical congruence (STC 2008)*. Paper presented at the Companion of the 30th international conference on Software engineering, Leipzig, Germany.

Herbsleb, J., Mockus, A., Finholt, T. & Grinter, R. (2001). *An empirical study of global software development: distance and speed*. Paper presented at the Proceedings of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada.

Herbsleb, J. & Moitra, D. (2001). Global software development. *Software, IEEE, 18*(2), 16-20. doi: 10.1109/52.914732

Hernández, M. A. & Stolfo, S. J. (1998). Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Mining and Knowledge Discovery, 2*(1), 9-37. doi: 10.1023/a:1009761603038

Herzig, K. & Zeller, A. (2009). Mining the Jazz Repository: Challenges and Opportunities. *Mining Software Repositories MSR '09. 6th IEEE International Working Conference*, 159-162.

Hevner, A., March, S., Park, J. & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*(1), 75-105.

Horovitz, O., Krishnaswamy, S. & Gaber, M. M. (2007). A fuzzy approach for interpretation of ubiquitous data stream clustering and its application in road safety. *Intelligent Data Analysis, 11*(1), 89-108.

Howison, J. & Crowston, K. (2004). The perils and pitfalls of mining SourceForge. *Proceedings of the Mining Software Repositories Workshop at the International Conference on Software Engineering (ICSE)*.

Huan, L. & Lei, Y. (2005). Toward integrating feature selection algorithms for classification and clustering. *Knowledge and Data Engineering, IEEE Transactions on, 17*(4), 491-502.

IBM. (2007). About the Repository. *Rational Software Information Center* Retrieved 15th October, 2011, from

http://publib.boulder.ibm.com/infocenter/rtc/v1r0m1/topic/com.ibm.team.scm.doc/topics/c_repo.html

IBM Rational Jazz Project. (2009) Retrieved November, 2009, from http://Jazz.net/

Ibrahim, W. M., Bettenburg, N., Shihab, E., Adams, B. & Hassan, A. E. (2010, 2-3 May 2010). *Should I contribute to this discussion?* Paper presented at the Mining Software Repositories (MSR), 2010 7th IEEE Working Conference.

Jazz Source Control Overview. (2009) Retrieved March, 2010, from https://Jazz.net/learn/LearnItem.jsp?href=content/docs/platform-overview/scm-overview.html

Jazz Work Item Overview. (2009) Retrieved November, 2009, from http://Jazz.net/library/LearnItem.jsp?href=content/docs/platform-overview/work-item-overview.html

Jeatrakul, P., Kok Wai, W., Chun Che, F. & Takama, Y. (2010, 19-23 Sept. 2010). *Misclassification analysis for the class imbalance problem.* Paper presented at the World Automation Congress (WAC), 2010.

Jensen, D. & Neville, J. (2002). Data Mining in Social Networks. *In National Academy of Sciences Symposium on Dynamic Social Network Modeling and Analysis*.

Jiang, N. & Gruenwald, L. (2006). Research issues in data stream association rule mining. *SIGMOD Rec., 35*(1), 14-19. doi: 10.1145/1121995.1121998

Jiang, Y., Li, M. & Zhou, Z. H. (2011). Software Defect Detection with R ocus. *Journal of Computer Science and Technology*, *26*(2), 328-342.

Jiawei, H. & Chang, K. C. C. (2002). Data mining for Web intelligence. *Computer*, *35*(11), 64-70. doi: 10.1109/mc.2002.1046977

Jørgensen, M. (2007). Forecasting of software development work effort: Evidence on expert judgement and formal models. *International Journal of Forecasting, 23*(3), 449 - 462.

Jørgensen, M., Faugli, B. & Gruschke, T. (2007). Characteristics of software engineers with optimistic predictions. *The Journal of Systems and Software, 80*(9), 1472 - 1482.

Jørgensen, M. & Shepperd, M. (2007). A Systematic Review of Software Development Cost Estimation Studies. *IEEE transactions on Software Engineering, 33*(1), 33-53.

Kagdi, H., Collard, M. & Maletic, J. (2007a). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice, 19*, 77 - 131.

Kagdi, H., Collard, M. & Maletic, J. (2007b). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE, 19*(2), 77-131. doi: 10.1002/smr.344

Kagdi, H., Collard, M. L. & Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE, 19*(2), 77-131. doi: 10.1002/smr.344

Khomh, F., Di Penta, M. & Guéhéneuc, Y. G. (2009, October). An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on* (pp. 75-84). IEEE.

Khoshgoftaar, T., Allen, E., Jones, W. & Hudepohl, J. (2001). Data Mining of Software Development Databases. *Software Quality Journal, 9*, 161–176.

Khoshgoftaar, T. M. & Munson, J. C. (1990). Predicting software development errors using software complexity metrics. *Selected Areas in Communications, IEEE Journal on, 8*(2), 253-261. doi: 10.1109/49.46879

Khoshgoftaar, T. M. & Seliya, N. (2002). Software quality classification modeling using the SPRINT decision tree algorithm. In *Tools with Artificial Intelligence, 2002.(ICTAI 2002). Proceedings. 14th IEEE International Conference on* (pp. 365-374). IEEE.

Khoshgoftaar, T. M. & Seliya, N. (2004). Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. *Empirical Software Engineering, 9*(3), 229-257. doi: 10.1023/B:EMSE.0000027781.18360.9b

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K. & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on, 28*(8), 721-734.

Kwan, I., Schröter, A. & Damian, D. (2009). Does Congruence Have An Effect? A Study of Coordination and Builds in a Software Project. *IEEE Transactions on Software Engineering, 99*(10), 1-23.

Lanubile, F., Ebert, C., Prikladnicki, R. & Vizcaino, A. (2010). Collaboration Tools for Global Software Engineering. *Software, IEEE, 27*(2), 52-55. doi: 10.1109/ms.2010.39

Lee, W. & Xiang, D. (2001). Information-theoretic measures for anomaly detection. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (pp. 130-143). IEEE.

Lehman, M. (1996). Laws of software evolution revisited. Software Process Technology. In C. Montangero (Ed.), (Vol. 1149, pp. 108-124): Springer Berlin / Heidelberg.

Yu, L. & Liu, H. (2003, August). Feature selection for high-dimensional data: A fast correlation-based filter solution. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE*- (Vol. 20, No. 2, p. 856).

Leung, H. & Fan, Z. (2002). Software cost estimation. *Handbook of Software Engineering, Hong Kong Polytechnic University*.

Liu, H., Lin, Y. & Han, J. (2011). Methods for mining frequent items in data streams: an overview. *Knowledge and Information Systems, 26*(1), 1-30. doi: 10.1007/s10115-009-0267-2

Lokan, C. (2005). What Should You Optimize When Building an Estimation Model? *11th IEEE International Software Metrics Symposium (METRICS'05)*.

MacDonell, S. & Shepperd, M. (2003). Combining techniques to optimize effort predictions in software project management. *The Journal of systems and software, 66*(2), 91 - 98.

Madden, S. & Franklin, M. J. (2002, 2002). *Fjording the stream: an architecture for queries over streaming sensor data.* Paper presented at the Data Engineering, 2002. Proceedings. 18th International Conference on.

Manduchi, G. and Taliercio, C. (2002). Measuring software evolution at a nuclear fusion experiment site: a test case for the applicability of OO and reuse metrics in software characterization. *Information and Software Technology, 44*(10), 593 - 600.

Marczak, S., Damian, D., Stege, U. & Schroter, A. (2008). Information Brokers in Requirement-Dependency Social Networks. *In the proceedings of International Conference on Requirements Engineering*, 53 – 62.

Martin, R. (1994). OO design quality metrics - An Analysis of Dependencies. *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*.

Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B. & Jiang, Y. (2008). *Implications of ceiling effects in defect predictors*. Paper presented at the Proceedings of the 4th international workshop on Predictor models in software engineering, Leipzig, Germany.

Miller, A. (2008). *A Hundred Days of Continuous Integration.* Paper presented at the Agile, 2008. AGILE '08. Conference.

Mockus, A. & Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal, 5*(2), 169-180. doi: 10.1002/bltj.2229

Moha, N., Guéhéneuc, Y. G. & Leduc, P. (2006). Automatic generation of detection algorithms for design defects. *21st IEEE/ACM International Conference. Automated Software Engineering. ASE'06.*

Nagappan, N., Ball, T. & Zeller, A. (2006a). Mining metrics to predict component failures. *IEEE Proceedings of the 28th International Conference on Software Engineering*, 452–461.

Nagappan, N., Ball, T. & Zeller, A. (2006b). *Mining metrics to predict component failures*. Paper presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China.

Nguyen, T., Schröter, A. & Damian, D. (2009). Mining Jazz: An Experience Report. *Infrastructure for Research in Collaborative Software*, 1 - 4.

Nunamaker, J., Chen, J. & Purdin, T. (1991). Systems development in information systems research. *Journal of Management Information Systems, 7*(3), 89-106.

Nunamaker Jr, Jay F. and Minder Chen. "Systems development in information systems research." *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*. Vol. 3. IEEE, 1990.

Olatunji, S., Idrees, S., Al-Ghamdi, Y. & Al-Ghamdi, J. (2010). Mining Software Repositories – A Comparative Analysis. *International Journal of Computer Science and Network Security, 10*(8), 161-174.

Patroumpas, K. & Sellis, T. (2006). Window Specification over Data Streams Current Trends in Database Technology – EDBT 2006. In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler,

M. Spiliopoulou & J. Wijsen (Eds.), (Vol. 4254, pp. 445-464): Springer Berlin / Heidelberg.

Pelayo, L. & Dick, S. (2007, June). Applying novel resampling strategies to software defect prediction. In *Fuzzy Information Processing Society, 2007. NAFIPS'07. Annual Meeting of the North American* (pp. 69-72). IEEE.

Pelayo, L. D., S. (2007). Applying Novel Resampling Strategies To Software Defect Prediction. *Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American* 69-72. doi: 10.1109/NAFIPS.2007.383813

Pendharkar, P., Subramanian, G. & Rodger, J. (2005). A probabilistic model for predicting software development effort. *EEE Transactions on Software Engineering, 31*(7), 615–624.

Perry, D. E., Porter, A. A. & Votta, L. G. (2000). *Empirical studies of software engineering: a roadmap*. Paper presented at the Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland.

Pfleeger, S. L. (1999). Albert Einstein and empirical software engineering. *Computer, 32*(10), 32-38. doi: 10.1109/2.796106

Poncin, W., Serebrenik, A. & van den Brand, M. (2011, March). Process mining software repositories. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on* (pp. 5-14). IEEE.

Rahm, E. & Do, H. (2000). Data Cleaning: Problems and Current Approaches. *IEEE Computer Society*, *23*(4), 3-13.

Ramler, R. & Wolfmaier, K. (2008). *Issues and effort in integrating data from heterogeneous software repositories and corporate databases*. Paper presented at the Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, Kaiserslautern, Germany.

Robbes, R. (2007). Mining a Change-Based Software Repository. *Proceedings of the Fourth International Workshop on Mining Software Repositories*, 15 - 23.

Rus, I. & Lindvall, M. (2002). Knowledge management in software engineering. *Software, IEEE, 19*(3), 26-38. doi: 10.1109/ms.2002.1003450

Sack, W., Détienne, N., Burkhard, J., Mahendran D. & Barcellini, F. . (2006). A Methodological Framework for Socio-Cognitive Analyses of Collaborative Design of Open Source Software. *Computer Supported Cooperative Work, 15*, 229–250. doi: 10.1007/s10606-006-9020-5

Scott, J. (1988). Social Network Analysis *Sociology, 22*(1), 109-127. doi: 10.1177/0038038588022001007

Sebastian, F. (2002). Machine Learning in Automated Text Categorization. *ACM Computing Surveys, 34*(1), 1-47.

Seiffert, C., Khoshgoftaar, T. M. & Hulse, J. V. (2009). Improving software-quality predictions with data sampling and boosting. *Trans. Sys. Man Cyber. Part A, 39*(6), 1283-1294. doi: 10.1109/tsmca.2009.2027131

Seliya, N., Khoshgoftaar, T. M. & Hulse, J. V. (2010). *Predicting Faults in High Assurance Software*. Paper presented at the Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering.

Sentas, P., Angelis, L., Stamelos, I. and Bleris G. (2005). Software productivity and effort prediction with ordinal regression. *Information and software technology, 47*(1), 17–29.

Serce, F., Alpaslan, F., Swigger, K., Brazile, R., Dafoulas, G., Lopez, V. & Schumacker, R. (2009). Exploring Collaboration Patterns among Global Software Development Teams. *2009 Fourth IEEE International Conference on Global Software Engineering*, 61 - 71.

Settimi, R., Cleland-Huang, J., Ben Khadra, O., Mody, J., Lukasik, W. & DePalma, C. (2004). Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of* (pp. 49-54). IEEE.

Shatnawi, R. (2012, March). Improving software fault-prediction for imbalanced data. In *Innovations in Information Technology (IIT), 2012 International Conference on* (pp. 54-59). IEEE.

Shepperd, M. (2011). *Data quality: cinderella at the software metrics ball?* Paper presented at the Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, Waikiki, Honolulu, HI, USA.

Simons, D. & Chabris, C. (1999). Gorillas in our midst: Sustained inattentional blindness for dynamic events. . *Perception, 28*, 1059-1074. doi: 10.1.1.65.8130

Sparrowe, R., Liden, R., Wayne, S. & Kraimer, M. (2001). Social Networks and the Performance of Individuals and Groups. *The Academy of Management Journal, 44*(2), 316-325.

Sparrowe, R., Liden, R., Wayne S. & M., K. (2001). Social Networks and the Performance of Individuals and Groups *The Academy of Management Journal, 44*(2), 316 - 325.

Spinellis, D. (2006). Bug busters. *Software, IEEE, 23*(2), 92-93. doi: 10.1109/ms.2006.40

Stamelos, I. (2009). Software project management anti-patterns. *The Journal of Systems and Software, 48*(1), 52-59. doi: 10.1016/j.jss.2009.09.016

Stamelos, I., Angelis, L., Dimou, P. & Sakellaris, E. (2003). On the use of Bayesian belief networks for the prediction of software productivity. *Information and software technology, 45*(1), 51 - 60.

Stonebraker, M., Çetintemel, U. & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec., 34*(4), 42-47. doi: 10.1145/1107499.1107504

Subramaniam, V. & Hunt, A. (2006). *Practices of an agile developer*. Pragmatic Bookshelf.

Subramanian, G. & Corbin, W. (2001). An empirical study of certain object-oriented software metrics. *The Journal of Systems and Software, 59*(1), 57 - 63.

Subramanyam, R. & Krishnan, M. S. (2003). Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on, 29*(4), 297-310. doi: 10.1109/tse.2003.1191795

Kim, S., Zhang, H., Wu, R. & Gong, L. (2011, May). Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on* (pp. 481-490). IEEE.

Tao, Y. (2011). Mining Time-Changing Data Streams. *Published doctoral dissertation, University of Waterloo, Ontario, Canada*.

Thwin, M. M. T. & Quah, T.-S. (2005). Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software, 76*(2), 147-156. doi: 10.1016/j.jss.2004.05.001

Tichy, W. F. (2000). Hints for Reviewing Empirical Work in Software Engineering. *Empirical Software Engineering, 5*(4), 309-312. doi: 10.1023/a:1009844119158

Trendowicz, A., Ochs, M., Wickenkamp, A., Münch, J., Ishigai, Y. & Kawaguchi, T. (2008). Integrating Human Judgment and Data Analysis to Identify Factors Influencing Software Development Productivity. *e-Informatica Software Engineering Journal, 2*(1), 47-69.

Vandecruys, O., Martens, D., Baesens, B., Mues, C., Backer, M. & Haesen, R. (2008). Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software, 81*, 823–839.

Venkatasubramanian, V., Rengaswamy, R., Kavuri, S. N. & Yin, K. (2003). A review of process fault detection and diagnosis: Part III: Process history based methods. *Computers & chemical engineering*, *27*(3), 327-346.

Warner, N., Letsky, M. & Cowen, M. (2005). Cognitive Model of Team Collaboration: Macro-Cognitive Focus. *Human Factors and Ergonomics Society Annual Meeting Proceedings, 49*(5), 269-273.

Weiss, C., Premraj, R., Zimmermann, T. & Zeller, A. (2007). *How Long Will It Take to Fix This Bug?* Paper presented at the Proceedings of the Fourth International Workshop on Mining Software Repositories.

White, S. & Smyth, P. (2003). *Algorithms for estimating relative importance in networks*. Paper presented at the Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, Washington, D.C.

Williams, C. C. & Hollingsworth, J. K. (2005). Recovering system specific rules from software repositories. *SIGSOFT Softw. Eng. Notes, 30*(4), 1-5. doi: 10.1145/1082983.1083144

Wolf, T., Schroeter, A., Damian, D. & Nguyen, T. (2009). Predicting build failures using social network analysis on developer communication. *Proceedings of the IEEE International Conference on Software Engineering (ICSE), Vancouver*.

Yasutaka, K. (2007). *The Effects of Over and Under Sampling on Fault-prone Module Detection*.

Ying, A. T. T., Murphy, G. C., Ng, R. & Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on, 30*(9), 574-586. doi: 10.1109/tse.2004.52

Zimmermann, T. (2007, May). Mining workspace updates in CVS. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on* (pp. 11-11). IEEE.

Zimmermann, T., Premraj, R. & Zeller, A. (2007, May). Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on* (pp. 9-9). IEEE.

Zimmermann, T. & WeiBgerber, P. (2004). Preprocessing CVS data for fine-grained analysis. *IEE Seminar Digests, 2004*(917), 2-6.

Zimmermann, T., Weibgerber, P., Diehl, S. & Zeller, A. (2004, May). Mining version histories to guide software changes. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on* (pp. 563-572). IEEE.