Volume 1 No. 5, AUGUST 2011 ISSN 2222-9833
ARPN Journal of Systems and Software

http://www.scientific-journals.org

Mining Software Metrics from the Jazz Repository

Andy M. Connor
Software Engineering Research Lab, Auckland University of Technology, Auckland, New Zealand
andrew.connor@aut.ac.nz

ABSTRACT

This paper describes the extraction of source code metrics from the Jazz repository and the systematic application of data mining techniques to identify the most useful of those metrics for predicting the success or failure of an attempt to construct a working instance of the software product. Results are presented from a study using the J48 classification method used in conjunction with a number of attribute selection strategies applied to a set of source code metrics. These strategies involve the investigation of differing slices of code from the version control system and the cross-dataset classification of the various significant metrics in an attempt to work around the multicollinearity implicit in the available data. The results indicate that only a relatively small number of the available software metrics that have been considered have any significance for predicting the outcome of a build. These significant metrics are outlined and implication of the results discussed, particularly the relative difficulty of being able to predict failed build attempts.

Keywords: *Data Mining, Jazz, Software Metrics, Software Repositories*

I. INTRODUCTION

In a typical software development project there are a wide range of tools used during the actual production of the final software artefact, including integrated development environments, static analysis tools and version control systems to name but a few. Software repositories such as source control systems and bug tracking databases have become a focus for research because they are an additional source of information regarding the performance and management of software development projects. Mining software repositories [1] is emergent research field that attempts to gain a deeper understanding of the development process in order to build better prediction and recommendation systems.

Jazz is a technology platform developed by IBM for the collaborative development of software products. Jazz as an extensible framework that dynamically integrates and synchronises people, processes, and assets associated with software development projects. Jazz has been recognized as offering both opportunities and challenges in the area of mining software repositories [2]. Jazz integrates the software archive and bug database by linking bug reports and source code changes with each other through the concept of work items which provides much potential in gaining valuable insights into the development process of software projects.

This paper describes an extension of previous work [3] to continue to attempt the extraction of rich data from the Jazz dataset by utilizing source code metrics as a means of directly measuring the impact of code issues on build success. The next section provides a brief overview of related work. Section 3 discusses the nature of the Jazz data repository and metrics available for use in the data mining. Section 4 outlines the approach used for mining the software repository in Jazz, while results are presented in section 5. Finally, the paper concludes with a discussion of the limitations of the current work and a plan for addressing these issues in future work.

II. BACKGROUND & RELATED WORK

Jazz offers not only huge opportunities for software repository mining but also a number of challenges [2]. One of the appealing aspects of Jazz is that it provides a very detailed dataset in which all artefacts are linked to each other. Much of the work that utilizes Jazz as a repository has focused on the impact of team communication history, such as whether there is an association between team communication and build failure [4] or whether it is possible to identify relationships among requirements, people and software defects [5]. Other work [6] has focused purely on the collaborative nature of software development. To date, most of the work involving the Jazz dataset has focused on aspects other than analysis of the source code contained in the repository, with the exception of previously published work by the authors [3, 7].

Whilst not specifically related to Jazz, there has been a number of investigation into the prediction of defects from the analysis of source code metrics. Such research has generally shown that there is no single code or churn metric capable of predicting failures [8, 9, 10], though evidence suggests that a combination can be used effectively [11]. In previous work [3] source code analysis has been conducted on the Jazz project data to perform an in-depth analysis of the repository to gain insight into the usefulness of software product metrics in predicting software build failure. Whilst some successes have been achieved in determining the relationship between build outcomes and source code [3] there is still a pressing need to provide additional clarity to what is a complex problem domain. One of the challenges arises from the phenomenon of multicollinearity, which is apparent because individual metric values and the failure rates for a module all tend to be highly correlated with each other [12].

Buse and Zimmerman [13] suggest that whilst 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. In their paradigm of software analytics, Buse and Zimmerman suggest 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?). They 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. This is confirmed by Hassan [14] who argues that data from software repositories cannot be used to conclude causation instead it can only show correlation. There is a need to provide tools and approaches that extract meaning from data in software repositories to better inform the software development process.

The Jazz data has the potential to provide sufficiently rich information to support these goals. Previous work has involved the analysis of the software product metrics available through Jazz and shown not only that there is scope to classify a set of software changes by the source code metrics and predict the likely outcomes of the build immediately prior to compilation and testing [3]. It has also been shown that there is potential to transform the timing of a prediction event from the time the code is committed to the repository immediately prior to the build to an earlier and more useful time [7]. An early prediction event provides greater insight into the likely outcomes of a build and hence can be used in managing the risk inherent in project's dynamics and hence this research supports the goals of the software analytics paradigm.

III. THE JAZZ DATASET

A. Overview of Jazz

The nature of the Jazz framework has been detailed in the research literature a number of times [2, 3, 4, 7] however it is important to restate the concepts embedded in the underlying data model to explain the approach adopted in this paper. IBM Jazz is a fully integrated software development framework that automatically captures software development processes and artefacts. The Jazz repository contains real-time evidence that allows researchers to gain insights into team collaboration and development activities within software engineering projects [15]. Figure 1 illustrates that through the use of Jazz it is possible to visualize members, work items and project team areas.

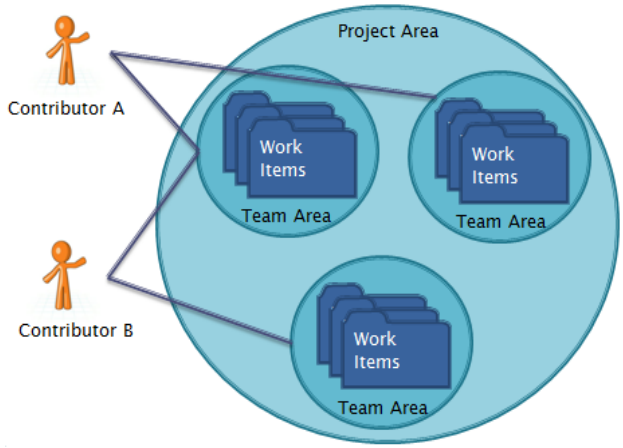

Figure 1: Jazz Repository Structure

Whilst Jazz provides the capability to extract social network data and relate such data to the software project outcomes, it is the ability to extract different baselines of the source code that is utilised in this current work. The source code available through the Jazz repository is related to work items, with work items being included in a given build of the software product.

The Jazz repository artefacts include work items, build items, change sets, source code files, authors and comments. A work item is a description of a unit of work, which is categorized as a task, enhancement or defect. A build item is compiled software to form a working unit. 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.

Whilst the Jazz repository has been opened up to facilitate investigative research [18] there are limitations for utilising it to its full capability. Firstly, the repository is highly complex and has huge storage requirements for tracking software artefacts. Another issue is that the repository is often missing data and often misleading elements which cannot be removed or identified easily. One of the primary causes of this is that 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 features of the framework have been introduced they have added new data at that point in time forward only.

There is a challenge in dealing with such inconsistency which may be circumnavigated by utilising an approach that delves further down the artifact chain than most previous work using Jazz. This work is based on the 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 project.

B. Software Metrics

One of the major advantages of Jazz is that the source code, version control, bug tracking and planning features are all integrated in a single repository. It has been suggested [2] that heuristics that rebuild artefact dependencies between disparate version control and bug tracking systems (as used by Zimmermann et al. [16]) should not be required when using Jazz. The ability to directly extract dependencies should raise the accuracy and detail level of extracted data sets [2]. With this in mind, software metrics have been calculated for source code extracted from the Jazz repository in an attempt to deal with the sparseness of the data without losing the level of detail in the data that should be available. Software metrics are used to measure the complexity, quality and effort of a software project [17].

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). Source code files were extracted for each available build within the repository. This was achieved by extracting all of the work items included in a given build and subsequently extracting all of the changesets associated with the work items. These changesets were filtered to remove non-source code files (e.g. XML files) that were part of the changeset.

Finally, software metrics were calculated by utilising the IBM Rational Software Analyzer tool. As a result the following basic, object orientated and Halstead software metrics were derived from the source code files for each build. These are shown in Table 1 along with the classification of the metric, either Basic (B), Object Oriented (OO) or Halstead (H). The metrics include the outcome of the build against which classification will be made. The build result is a nominal metric and a build is either failed or successful.

Table 1: Available Metrics

ID	Metric	Type
1	*Build result*	*Classification*
2	Abstractness	OO
3	Afferent coupling	OO
4	Average block depth	OO
5	Average lines of code per method	B
6	Average number of attributes per class	B
7	Average number of comments	B
8	Average number of constructors per class	B
9	Average number of methods	B
10	Average number of parameters	B
11	Comment/code ratio	B
12	Cyclomatic complexity	OO
13	Depth of inheritance	H
14	Difficulty level	H
15	Efferent coupling	OO
16	Effort to implement	H
17	Instability	OO
18	Lack of cohesion 1	OO
19	Lack of cohesion 2	OO
20	Lack of cohesion 3	OO
21	Lines of code	B
22	Maintainability index	OO
23	Normalized distance	OO
24	Number of attributes	B
25	Number of comments	B
26	Number of constructors	B
27	Number of delivered bugs	H
28	Number of import statements	B
29	Number of interfaces	B
30	Number of lines	B
31	Number of methods	B
32	Number of operands	H
33	Number of operators	H
34	Number of parameters	B
35	Number of types per package	B
36	Number of unique operands	H
37	Number of unique operators	H
38	Program length	H
39	Program level	H
40	Program vocabulary size	H
41	Program volume	H
42	Time to implement	H
43	Weighted methods per class	OO

In addition to software (source code) metrics a range of metrics that are unique to the Jazz environment are available, however at present this research only includes whether the build attempt is successful or whether it fails. A failed build is in essence one where the end product does not pass all of the test cases or does not behave as expected.

IV. EXPERIMENTAL METHOD

This work revolves around the use of classification methods for the analysis of software metrics, which differs from much of the work in this area that has focussed on clustering rather than classification [12]. For this purpose, the Weka [19] machine learning workbench was used. There are various challenges that arise when adopting data mining as a classification approach as available data is not always suitable for the mining process. Data gained from software development projects is often noisy, incomplete or even misleading data. This can give rise to negative impacts on the mining and learning process [20].

As has already been discussed, 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 had been completed. The implication of this is that gaps often appear at early stages of the project data set. Excluded from the data set were build instances that had no work items associated with a build, build warning results and builds that had missing values within the derived software metrics.

Software metrics from continuous builds were used to construct the 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 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, 129 builds were included, out of which there were 51 successful builds and 78 failed builds. This presents a situation where the number of features is fairly close to the number of instances available for analysis, which is not an ideal scenario, particularly given the multicollinearity that is apparent in the metric values.

It is therefore important to investigate various strategies for reducing the number of metrics used to classify the relatively small number of builds in the dataset. Previous work [3, 7] has shown that there is a lack of consistency in terms of identifying significant metrics. In this paper, hybrid strategies for identifying the most significant metrics that are based on attempting to classify and cross-classify across datasets derived from different snapshots (or "slices") of the available source code are investigated. This approach is adopted because previous work [7] has shown that using metrics calculated from the final commit of the source code prior to the build taking place can be used to predict the outcomes of the build when applied to the classification of source code prior to the development iteration commencing. The work presented in this paper is the first attempt to explore why such an approach leads to good results and attempt to further reduce the number of significant metrics down to the smallest possible number.

A. Dataset Representations

In the Jazz dataset a given build consists of a number of different work items. Each work item contains a *changeset* that indicates the actual source code files that are modified during the implementation of the work item. Each build has a corresponding *before* and *after* state. Initial work involving the extraction and mining of Jazz software metrics [3] used the *after* state to extract source code that included all changes in the build. The *after* state was utilised in order to ensure that the source code snapshot represented the actual software artefact that either failed or succeeded. Subsequent work has shown that it is possible to predict the outcome of a build on the basis of the *before* state source code [7]. The best classification arose by applying significant metrics identified using the metrics derived from the *after* state applied to the *before* state classification. The reason for this remains to be determined and in this work attempt to systematically explore the use of the metrics derived from *after* and *before* states as well as the difference between them and determine whether cross-dataset classification can lead to further insight. Cross-dataset classification is defined as the use of significant metrics determined from one dataset being used to classify the contents of a different dataset.

Source code metrics are calculated for each source code file in the changeset using the IBM Software Analyser tool. Previous work [3] has investigated different ways of characterising the changeset using a single metric value to represent all source code files in the changeset. This showed that the most reliable approach was to calculate the value for each metric for each source code file and then propagate the maximum determined value up to the build level. This approach is adopted in the current work.

B. Experiment Descriptions

The goal of the experimentation is to determine which software metrics give the best indicators of whether the build will be successful or will fail. The experiments systematically filter the available metrics using a variety of methods to simplify the problem space and determine the best classification outcomes. This is necessary as previous work [3] has determined that the ratio of metrics (42) to build instances (129) creates a complex classification scenario due to the multicollinearity of the metric values.

The methods used to filter the metrics used are shown in Table 2 and are limited to feature selection approaches available in Weka. Each involves selecting a relatively small number of the available software metrics and comparing them to the baseline classification where no filtering of the metrics is done.

Table 2: Metric Filtering Strategies

ID	Strategy
1	No filtering
2	Weka Feature Selection (CfsSubset)
3	Weka Feature Selection (Infogain)

In the first instance, each strategy is applied to the four distinct datasets that have been derived from the source code. The first two datasets correspond to metrics calculated from the *before* state and the *after* state of the builds. The third dataset is calculated by subtracting the values of the *before* state metrics from the value of the *after* state metrics. This dataset represents the degree of change to the source code that occurs during a development iteration leading to a build. The final dataset results from combining the *before* and *after* datasets into a larger dataset.

It has been noted in previous work [7] that applying the significant metrics determined from analysing one state to the classification of build outcome of another state can lead to improved classification. In particular, using significant metrics from the *after* state improves the ability to predict build outcomes on the *before* state source code. As a result, this work extends this Cross-dataset classification and applies significant metrics identified from each state to all other states.

V. RESULTS

For each of the experiments a metric filtering strategy is applied and then the J48 classification algorithm is used to attempt to discover common patterns amongst the selected metrics. Given the relatively small

size of the data, 10-fold cross validation is used in order to make the best use of the training data. Cross validation does result in a relatively optimistic outcome which is a limitation that will be addressed in future work when more data becomes available from the Jazz project.

A. Classification Results: Before State

The results of applying the filtering strategies from Table 2 to the metrics calculated from the *before* state source code are shown in Table 3. It can be seen that the Infogain method has identified a great number of significant metrics, which is to be expected as the CfsSubset method looks for inter-relationships between metrics to identify significant associations.

Table 3: Selected Metrics

ID	Selected Metrics
1	N/A
2	4,5,6,11,18,22,36,39
3	30,11,18,4,22,6,40,36,23,39,37,20,34,25,5,12,35,24

Table 4 shows the results of the classification for applying the selected metrics to the classification of the *before* state source code. The overall accuracy is given in each case along with the number of correctly (and incorrectly) classified builds. The bracketed values refer to the number falsely predicted to be either failures (in the case of the "Failed Builds" column) or successes (in the case of the "Successful Builds") column.

Table 4: Classification Results

ID	Accuracy	# Failed Builds Correct (Incorrect)	# Successful Builds Correct (Incorrect)
1	67.4419%	22 (29)	65 (13)
2	72.8682%	25 (26)	69 (9)
3	68.9922%	22 (29)	67 (11)

These results are a subset of those presented in previous work [7] and indicate that the prediction of failed builds is generally more challenging than the classification of successful builds. As with previous work the overall accuracy of the prediction is hovering around the 70% value, however using the significant metrics determined from the *before* state on the *before* state data tends to produce poorer classification of failures. Figure 2 illustrates the best classification tree achieved as a result of these experiments (ID: 2).

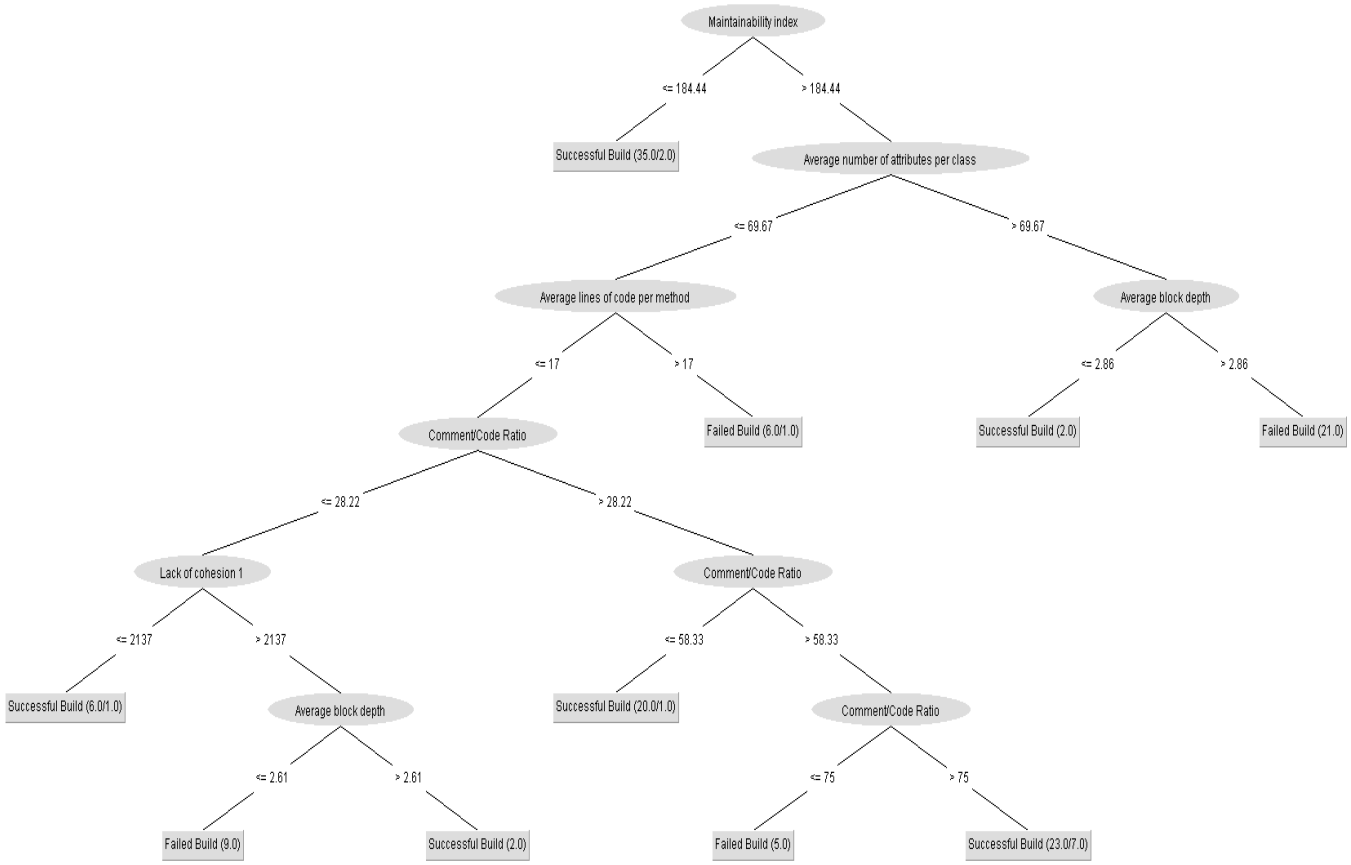

Figure 2: Classification Tree: Before State

B. Classification Results: After State

The results of applying the filtering strategies from Table 2 to the metrics calculated from the *after* state source code are shown in Table 5. As with the before state, it can be seen that the Infogain method has identified a great number of significant metrics.

Table 5: Selected Metrics

ID	Selected Metrics
1	N/A
2	4,6,11,18,19,25,28,35,37
3	11,6,4,28,37,36,40,25,19,18,24,34,35,20,13

Table 6 shows the accuracy of the classification for each dataset with the features selected using the each metric selection strategy. The overall accuracy is given in each case along with the number of correctly (and incorrectly) classified builds. The bracketed values refer to the number falsely predicted to be either failures (in the case of the "Failed Builds" column) or successes (in the case of the "Successful Builds") column.

Table 6: Classification Results

ID	Accuracy	# Failed Builds Correct (Incorrect)	# Successful Builds Correct (Incorrect)
1	75.1938%	36 (15)	61 (17)
2	75.9690%	27 (24)	71 (7)
3	77.5194%	33 (18)	67 (11)

These results are a subset of those presented in previous work [3], however differ from those previously published. This is due to a change in data extraction from the Jazz repository. The data extraction approach As with the results shown in Table 4 there is clearly more difficulty in identifying failed builds, though the outcome of applying the *after* state significant metrics to the *after* state data results in slightly higher overall accuracy and an improvement in identifying failed builds over the use of *before* state metrics to the *before* state data.

Figure 3 illustrates the best classification tree achieved as a result of these experiments (ID: 3) based on the overall accuracy.

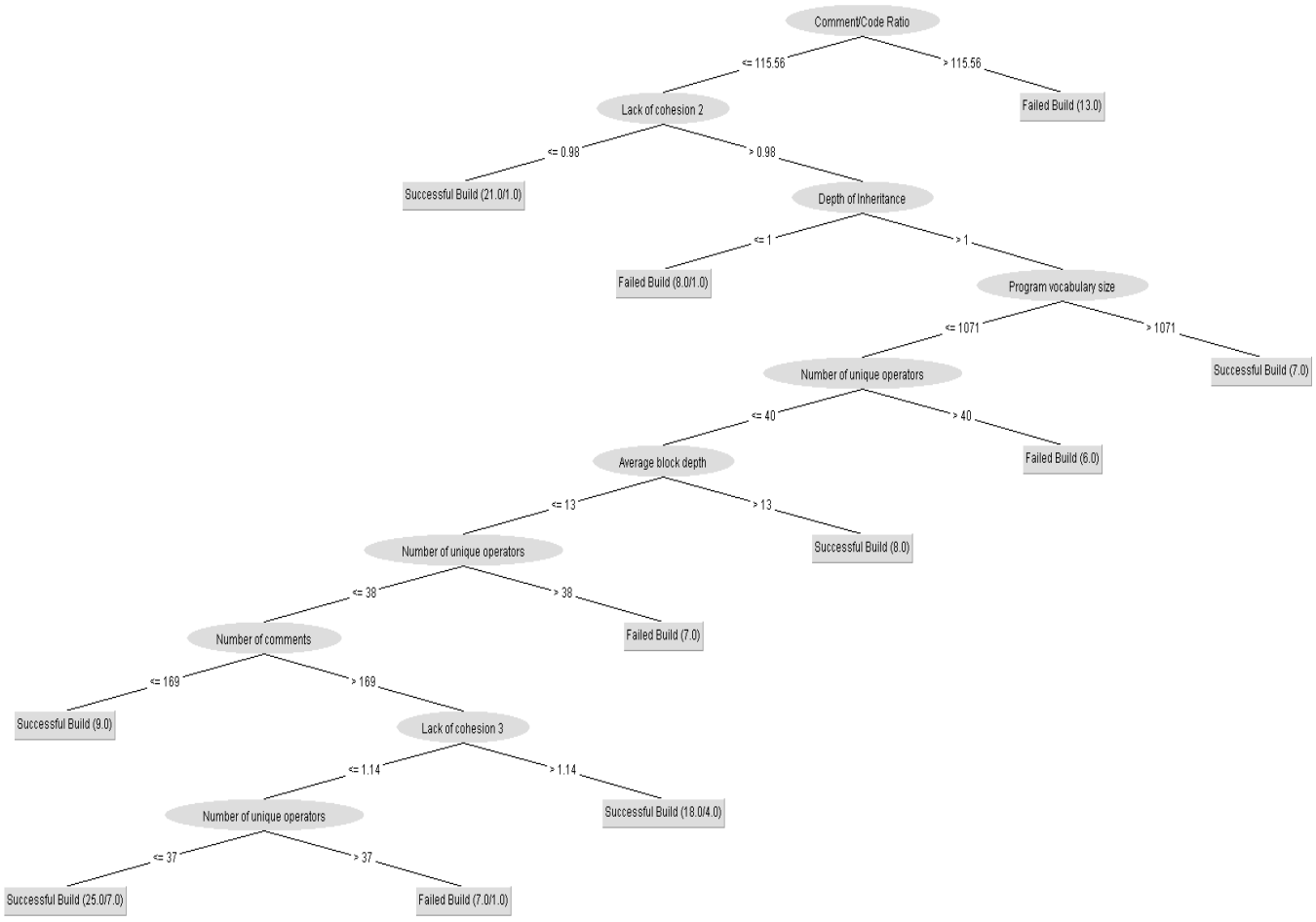

Figure 3: Classification Tree: After State

C. Classification Results: Difference

The results of applying the filtering strategies from Table 2 to the calculated difference in metric values between the *after* and *before* states are shown in Table 7. As with the before state, it can be seen that the Infogain method has identified a great number of significant metrics.

Table 7: Selected Metrics

ID	Selected Metrics
1	N/A
2	4,6,7,18,21,22,23,39
3	4,22,23,42,21,32,38,39,7,33,18,6,10

Table 8 shows the accuracy of the classification for each dataset with the features selected using the each metric selection strategy. The overall accuracy is given in each case along with the number of correctly (and incorrectly) classified builds. The bracketed values refer to the number falsely predicted to be either failures (in the case of the "Failed Builds" column) or successes (in the case of the "Successful Builds") column.

Table 8: Classification Results

ID	Accuracy	# Failed Builds Correct (Incorrect)	# Successful Builds Correct (Incorrect)
1	73.6434%	34 (17)	61 (17)
2	66.6667%	21 (30)	65 (13)
3	71.3178%	30 (21)	62 (16)

The best accuracy is obtained with no feature selection. This differs from the previous results in Tables 4 and 6. This is perhaps an indication that inspecting the change in metric values is reducing the extent to which significance can be identified. This is borne out to some extent by the classification tree show in Figure 4 which illustrates the best classification tree achieved as a result of these experiments (ID: 3). The classification tree is somewhat more complex than those shown in Figure 2 and 3 which implies that a clear classification is not possible for this data.

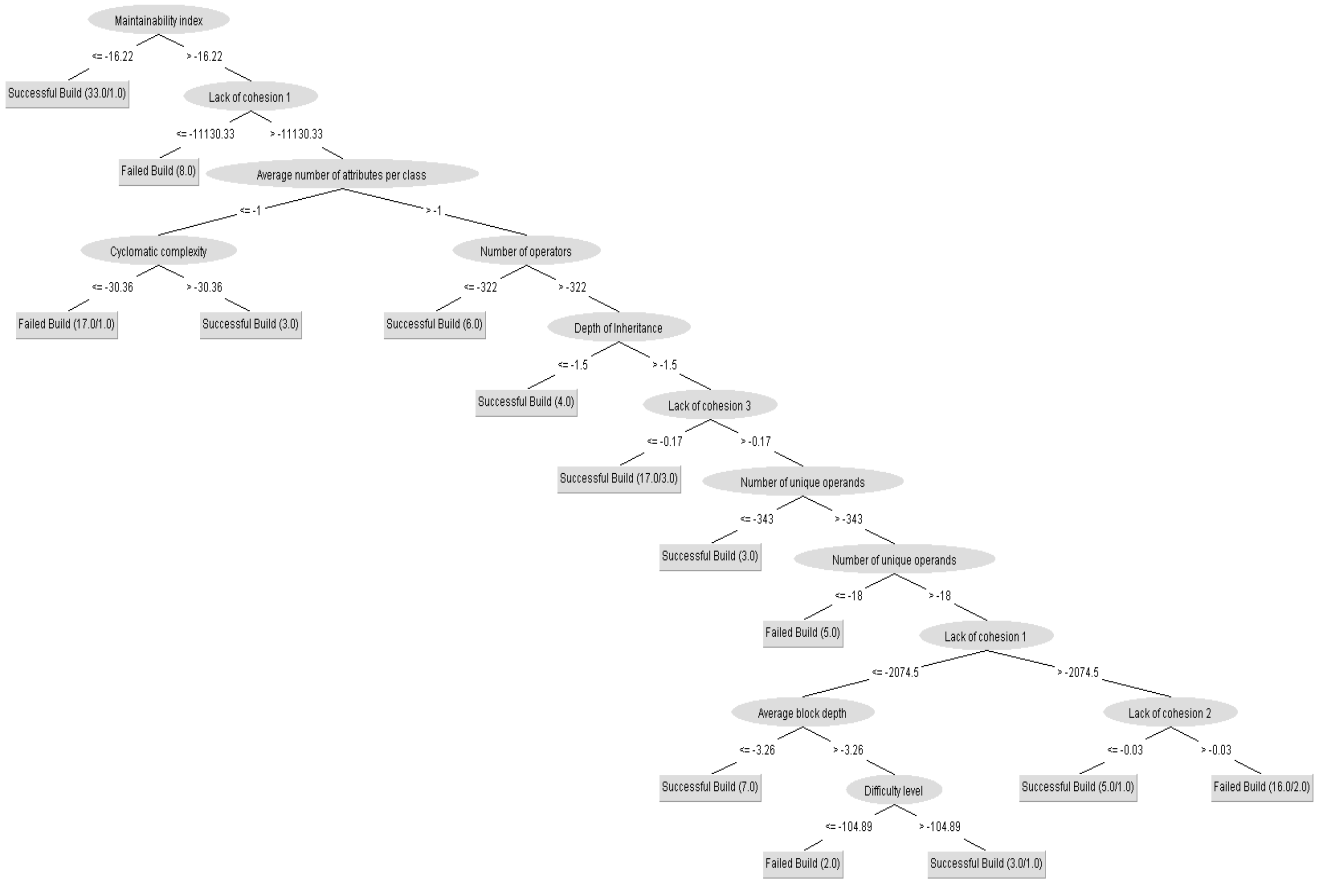

Figure 4: Classification Tree: Difference

D. Classification Results: Combined Dataset

The results of applying the filtering strategies from Table 2 to the dataset that combines the *before* and *after* state data. It can be seen that the Infogain method has identified a great number of significant metrics, which is to be expected as the CfsSubset method looks for inter-relationships between metrics to identify significant associations.

Table 4: Selected Metrics

ID	Selected Metrics
1	N/A
2	4,5,6,9,10,11,13,18,21,22,25,34,35,36,37
3	6,18,9,8,10,34,11,36,40,37,25,3,28,13,4,24,35,5,22, 23,20,21,7,19

Table 4 shows the results of the classification for applying the selected metrics to the classification of the *before* state source code. The overall accuracy is given in each case along with the number of correctly (and incorrectly) classified builds. The bracketed values refer to the number falsely predicted to be either failures (in the case of the "Failed Builds" column) or successes (in the case of the "Successful Builds") column.

Table 4: Classification Results

ID	Accuracy	# Failed Builds Correct (Incorrect)	# Successful Builds Correct (Incorrect)
1	79.0698%	71 (31)	133 (23)
2	82.1705%	74 (28)	138 (18)
3	79.0698%	69 (33)	135 (21)

Increasing the size of the dataset has slightly improved the overall accuracy and has improved the ability to identify failed builds.

Figure 5 illustrates the best classification tree achieved as a result of these experiments (ID: 2) based on the overall accuracy. It is interesting to note that a total of 53 failed builds are correctly classified on the basis of just three metrics, namely: Comment-Code Ratio, Average Number of Attributes per Class and Number of Comments. This differs from the previous classification outcomes where the upper nodes in the classification tree tend to classify successful rather than failed builds.

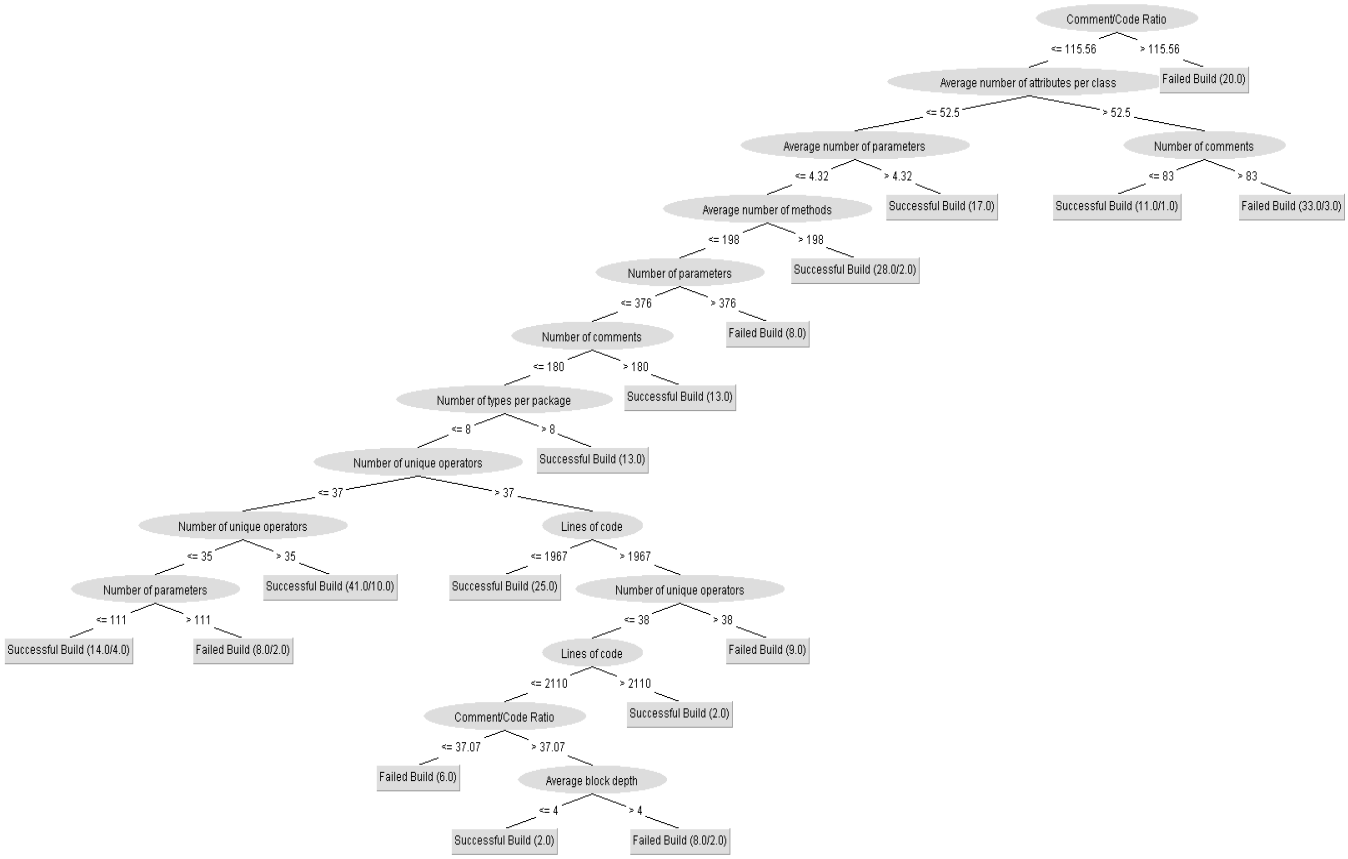

Figure 5: Classification Tree: Combined Data

E. Cross-Dataset Classification Results

It has been noted in previous work [7] that applying significant metrics from the *after* state to the *before* state data resulted in an improved classification of failure. Whilst it is possible that there is some data-interaction that predisposes this approach to over-fitting of the data, it is an interesting concept that needs further study.

In this paper the goal is conduct an initial investigation into the impact of cross-dataset classification. Hence the final results presented investigate the outcome of classifying the different datasets using metrics that are deemed significant for other datasets. The exception is that the difference between *before* and *after* states did not result in any significant metrics being identified.

Table 9: Experiment

ID	Cross-Dataset Classification
1	Applying best *after* state metrics to *before* state data
2	Applying best *after* state metrics to *difference* data
3	Applying best *after* state metrics to *combined* data
4	Applying best *before* state metrics to *after* state data
5	Applying best *before* state metrics to *difference* data
6	Applying best *before* state metrics to *combined* data
7	Applying best *combined* metrics to *after* state data
8	Applying best *combined* metrics to *before* data
9	Applying best *combined* metrics to *difference* data

Table 10 shows the results of the cross-dataset classification for each dataset with the features selected from other datasets. The overall accuracy is given in each case along with the number of correctly (and incorrectly) classified builds. The bracketed values refer to the number falsely predicted to be either failures (in the case of the "Failed Builds" column) or successes (in the case of the "Successful Builds") column.

Table 10: Cross-Dataset Classification Results

ID	Accuracy	# Failed Builds Correct (Incorrect)	# Successful Builds Correct (Incorrect)
1	79.8450%	38 (13)	65 (13)
2	77.5194%	34 (17)	66 (12)
3	79.8450%	72 (30)	134 (22)
4	79.8450%	27 (24)	76 (2)
5	76.7442%	31 (20)	68 (10)
6	75.1938%	50 (52)	144 (12)
7	71.3178%	29 (22)	63 (15)
8	80.6202%	34 (17)	70 (8)
9	75.9690%	34 (17)	64 (14)

The highest overall accuracy is just over 80%, however the greatest successful classification of failed builds is associated with experiment 1. This has a very small reduction in overall accuracy and as such would be considered the best results achieved.

It is interesting to note that again applying significant metrics from the *after* state data to the *before* state data has produced the best classification. Not only is experiment 1 the best classification in these cross-dataset classification but it also improves on the classification results presented in the previous sections.

Figure 6 illustrates the best classification tree achieved as a result of these experiments (ID: 1) based on the ability to correctly classify the highest number of failed builds. As with Figure 5, the classification tree has a high number of correctly classified failed builds in the upper nodes of the tree. Given the acknowledged difficulty in identifying failed builds this characteristic may be most desirable for the classification tree.

The overall classification has not yet met the accuracy and ability to classify failures determined in previous work [7]. However the results presented here have provided some insight into the potential value offered by cross-dataset classification.

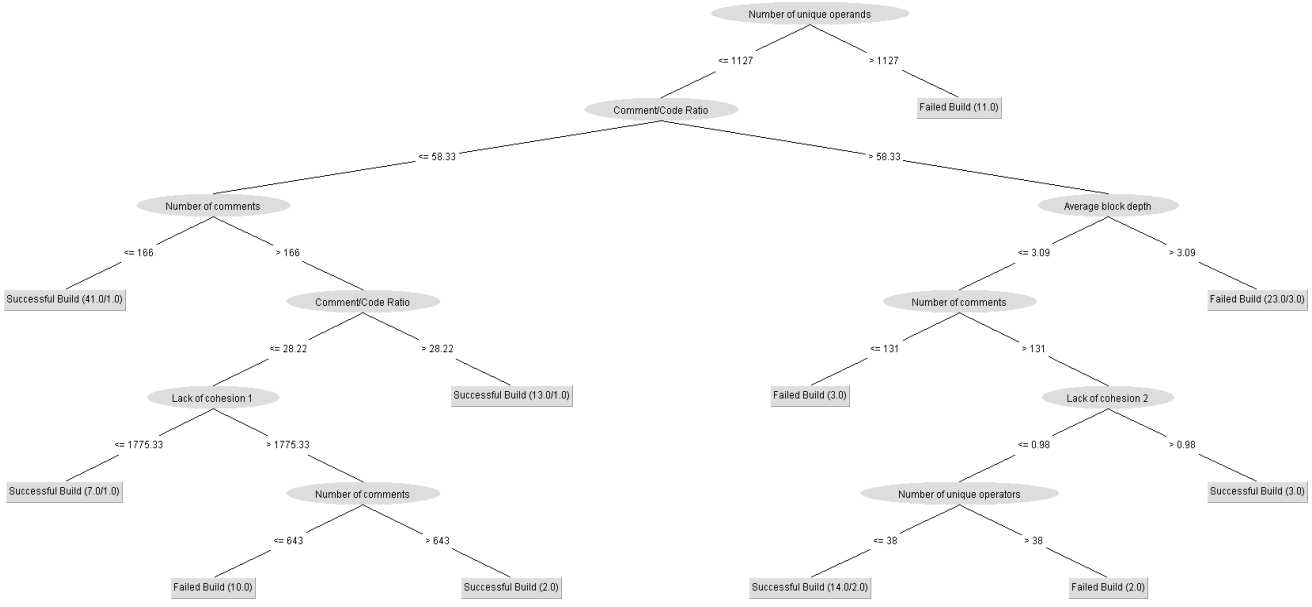

Figure 6: Classification Tree: Cross-Dataset Classification (After/Before)

VI. LIMITATIONS & FURTHER WORK

Most of the limitations in the current 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 (129) 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 dataset is very small. Even combining datasets from the *before* and *after* states to double the size of the data has not significantly impact the quality of the classification.

Whilst a new release of the Jazz repository is pending, in the meantime the main thrust of future work is to further expand the build data to improve the degree of granularity and potentially improve the quality of the classification.

Therefore another key aspect for further study is to investigate why using significant metrics calculated from source code at the end of a development cycle are better at predicting failure when applied to the code at the beginning of the build cycle. Some evidence exists in the literature that may explain this phenomenon. Kitcheham [21] has observed that "Code metrics extracted at a specific point in time are unlikely to predict fault rates well in evolving system" and also that "Code change metrics are likely to predict fault rates in an evolving system better than simple snap-shot based metrics". Examining the degree of change in metric values hasn't resulted in a significant improvement in classification accuracy in this paper, though this may be due to the relatively small size of the dataset.

It is possible that the use of *after* state metrics to predict the outcome of a build based on the *before* state source is a process of examining source code for the potential of failure. Therefore future work will be based around the idea of simulating the emergence of the existing data and whether the analysis of completed builds can be used to predict the outcome of the next build. By simulating the development process as a time series it may be possible to investigate whether there is the potential to learn from past erroneous builds to further improve early prediction of failure in future builds.

VII. CONCLUSIONS

This paper presents the outcomes of a study exploring the value of cross-dataset classification to predict build success and/or failure for a software product by utilizing source code metrics. Prediction accuracies of up to 82% have been achieved through the use of the J48 classification algorithm combined with 10-fold cross validation. The results presented confirm that there is value in using the metrics derived from different slices of source code in the early prediction of build outcome. The strategy of using metrics associated with the *after* state of the build to classify the *before* state source code may in some way be overfitting the data to the classification strategy and further work is needed to fully validate this approach.

REFERENCES

[1] Kagdi, H. H., Collard, M. L. and Maletic, J. I. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software and Maintenance*, 19(2):77-131.

[2] Herzig, K. and Zeller, A. 2009. Mining the Jazz repository: Challenges and opportunities, *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories,* (Vancouver, Canada, May 16-17, 2009). IEEE. 159-162. DOI: 0.1109/MSR.2009.5069495.

[3] Finlay, J., Connor, A.M. and Pears, R. 2011. Mining software metrics from Jazz. *Proceedings of the 9th ACIS Conference on Software Engineering Management, Research and Applications, (Baltimore, USA, Aug 10-12, 2011)*. IEEE Computer Society. 39-45.

[4] Wolf, T., Schroter, A., Damian, D. and Nguyen, T. 2009. Predicting build failures using social network analysis on developer communication, *Proceedings of the IEEE 31st International Conference on Software Engineering,* (Vancouver, Canada, May 16-24, 2009). IEEE. 1-11. DOI: 10.1109/ICSE.2009.5070503

[5] Park, S., Maurer, F., Eberlein, A. and Fung, T-S. 2010. Requirements attributes to predict requirements related defects, *Proceedings of the 20th Annual International Conference on Computer Science and Software Engineering*, (Markham, Canada, Nov 1-2, 2010), ACM, 42-56. DOI=10.1145/1923947.1923953

[6] Nguyen, T., Wolf, T. and Damian, D. 2008. Global Software Development and Delay: Does Distance Still Matter? *Proceedings of the IEEE International Conference on Global Software Engineering,* (Bangalore, India, August 17-20, 2008). IEEE. 45-54. DOI: 10.1109/ICGSE.2008.39

[7] Connor, A.M. and Finlay, J. 2011. Predicting software build failure using source code metrics. I*nternational Journal of Information and Communication Technology Research*. (To appear).

[8] Nagappan, N., Ball, T. and Zeller, A. 2006. Mining metrics to predict component failures. *Proceedings of the 28th International Conference on Software Engineering*, (Shanghai, China, May 20-28, 2006).

IEEE. 452–461.

[9] 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, 22(10), 751–761.

[10] Denaro, G., Morasca, S. and Pezz`e, M. 2002. Deriving models of software fault-proneness. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, (Ischia, Italy, July 15-19, 2002). ACM. 361–368.

[11] Mockus, A. and Weiss, D. M. 2000. Predicting risk of software changes. Bell Labs Technical Journal, 5(2):169–180.

[12] Dick, S., Meeks, A., Last, M., Bunke, H. and Kandel, A. 2004. Data mining in software metrics databases. *Fuzzy Sets and Systems*, 145(1):81-110. ISSN 0165-0114. DOI: 10.1016/j.fss.2003.10.006.

[13] Buse, R. P. L. and Zimmermann, T. 2010. Analytics for Software Development. *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research* (Santa Fe, New Mexico, USA, Nov 7-8, 2010). ACM, 77-80. DOI=10.1145/1882362.1882379.

[14] Hassan, A.E. 2008. The road ahead for Mining Software Repositories. *Frontiers of Software Maintenance at the 24th IEEE International Conference on Software Maintenance* (Beijing, China, Sept 28-Oct 4). 48-57. DOI: 10.1109/FOSM.2008.4659248

[15] Nguyen, T., Schröter, A., and Damian, D. 2008. Mining Jazz: An experience report. *Proceedings of the Infrastructure for Research in Collaborative Software Engineering Conference*. Retrieved 24/1/2011 from: http://home.segal.uvic.ca/~pubs/pdf/112/2008-iReCoSE.pdf

[16] Zimmerman, T., Premraj, R. & Zellar, A. 2007. Predicting defects for Eclipse. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (Minneapolis, Minnesota, USA, May 20-16). 9 -15. DOI=10.1109/PROMISE.2007.10

[17] 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. DOI: 10.1016/S0950-5849(02)00079-4

[18] Cheng, P., Chulani, S., Ding, Y. B., Delmonico, R., Dubinsky, Y., Ehrlich, K., Helander, M., et al. 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 2008 (*Atlanta, Georgia, USA, Nov 9, 2008).

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

[20] Chau, D., Pandit, S., and Faloutsos, C. (2006). Detecting Fraudulent Personalities in Networks of Online Auctioneers. Knowledge Discovery in Databases 4213, 103-114, DOI: 10.1007/11871637_14.

[21] Kitchenham, B. 2010. What's up with software metrics? A preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51