Combining Clustering and Classification for Software Quality Evaluation

Diomidis Papas[1], Christos Tjortjis[2]

[1]Department of Computer Science & Engineering, University of Ioannina
P.O. Box 1186, GR 45110 - Ioannina, Greece
[2]School of Science & Technology, International Hellenic University
14th km Thessaloniki – Moudania, 57001 Thermi, Greece
E-mail: c.tjortjis@ihu.edu.gr

Abstract. Source code and metric mining have been used to successfully assist with software quality evaluation. This paper presents a data mining approach which incorporates clustering Java classes, as well as classifying extracted clusters, in order to assess internal software quality. We use Java classes as entities and static metrics as attributes for data mining. We identify outliers and apply K-means clustering in order to establish clusters of classes. Outliers indicate potentially fault prone classes, whilst clusters are examined so that we can establish common characteristics. Subsequently, we apply C4.5 to build classification trees for identifying metrics which determine cluster membership. We evaluate the proposed approach with two well known open source software systems, Jedit and Apache Geronimo. Results have consolidated key findings from previous work and indicated that combining clustering with classification produces better results than stand alone clustering.

1 Introduction

Quality evaluation is an important software engineering issue, addressed by various methods, in many cases involving metrics [1][2]. As the volumes of code produced increase so does the need for automating the process. Experience shows that collecting and analyzing fine grained test defects from large, complex software systems is difficult [1]. Data mining has been shown to facilitate quality evaluation when applied directly to source code as well as metrics extracted from code [3][4][5].

Static analysis using software quality metrics means that the code is analyzed without having to execute the program [2]. Static analysis tools for finding low-level programming errors are especially useful for evaluating maintainability, understandability, testability and expandability of the software [6]. Static analysis can also be applied early in the development process, to provide early fault detection, at a point where the code does not have to be fully functional, or even executable. Several tools of this nature already exist [7]. However size scaling challenges obstruct the evaluation of large data sets. Static analysis is unlikely to be adopted for improving software quality in the real world if it does not scale beyond small benchmarks.

In order to deal with this issue, we propose static analysis, using object oriented metrics combined with data mining techniques for analyzing large and real-world software systems. Using these metrics reflects a software system's source code attributes, such as volume, size, complexity, cohesion and coupling. Our approach is suitable for Java systems, which can easily be extended to cater for other object oriented languages. Data mining in static analysis allows for managing large volumes of data and is capable of producing unexpected results. This work does not only focus on software quality assessment, but also assesses the suitability of metrics for the evaluation, which is a useful and novel part of the evaluation process.

In order to do so, we identify outliers and employ K-means for clustering Java classes together according to their metric related similarity. Outliers are candidates for manual inspection, as they may be fault prone. Subsequently we use clusters as class labels and employ C4.5 decision tree classification algorithm for evaluating the selected metrics, in order to reflect their importance on defining clusters and evaluate software quality. Using C4.5 allows for establishing which metrics play an important role in the evaluation process and highlights metrics which do not affect tree building. We used source code from various large open source java systems in order to validate this approach. Experiments indicated that combining clustering with classification produces better results than stand alone clustering.

The rest of this paper is organized as follows: background and related work are discussed in section 2. Section 3 details the approach. Experimental results for validating the approach are described in section 4. Section 5 briefly discusses evaluation issues and threats to validity. The paper concludes with directions for future work in section 6.

2 Background and Related Work

Different quality metrics can be used to evaluate source code [8]. Chidamber and Kemerer in their seminal work [9] proposed the following metrics, now known as CK Suite: Weighted Methods per Class (WMC), Response For a Class (RFC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Object classes (CBO), Lack of Cohesion in Methods (LCOM). Several additions to the CK Suite have been made to cater for complexity [10]. Other metrics often used in quality assessment include Halstead's Maintainability Index (MI) [11], McCabe's Cyclomatic Complexity (CC) [12] and Lines Of Code (LOC). Various techniques have been proposed for analyzing source code and improving software development and maintenance.

Metrics help efficiently assessing software quality; however, they can be hard to manually calculate for vast amounts of code. Several approaches employ data mining to extract useful information from metrics for large software systems [3][13][14][15]. Details on object oriented metrics and rational for their selection can be found in [16]. Related work on object oriented source code can also be found in [17] where a subset of the CK object oriented metrics was used.

Mining large data volumes poses challenges, starting off with data preprocessing, metric and algorithm selection [18]. Clustering, and in particular K-means, has been used for mining source code and metrics, due to its simplicity and low-time complexity. It requires pre-defining the number of clusters K, but selecting a suitable value for K can be done by running a number of experiments, depicting the Sum of Squared Error (SSE) over K and identifying a “knee” in the diagram [19]. Identifying outliers can be also very useful, given that these represent “unusual” instances.

Classification, on the other hand, has not been extensively used for source code metrics, despite its simplicity and ability to extract rules that can be easily understood and interpreted by users. Decision trees classifiers produce descriptive models suitable for source code assessment.

Tribus et al. [5] focused on using classifiers for knowledge discovery and troubleshooting software written in C. Menzies et al. used data mining to predict errors and assist large project management [4]. They used metrics such as McCabe’s CC, Halstead’s MI as well as LOC for C code, and classifiers such as C4.5. Prasad et al. proposed an approach for source code evaluation by knowledge engineering [23]. It discovers weaknesses and errors in code using text mining, specifically using the frequency of words or symbols in C++ code.

Alternative approaches were formulated previously by Antonellis, et al. [17], where the criteria for code quality evaluation were associated with the ISO/ICE-9126 standard for software maintainability, which also includes functionality, efficiency and portability. Object-oriented metrics were used to measure class similarity, assess software maintainability and manage large systems [21][22]. We use this work that showed the correlation between source code analysis and classes’ similarity as a foundation for this paper, particularly for selecting object-oriented metrics and clustering algorithms, for analyzing, understanding, and controlling software.

3 Approach

This section presents the proposed approach which: (i) extracts metrics from Java code, (ii) finds outliers and potentially fault prone Java classes, and clusters classes based on their similarity according to selected metrics, and (iii) categorizes clusters using classification, in order to get insights into the clustering results and to produce a description model capable of assessing metric values in each cluster, with regards to their ability to evaluate software quality.

3.1 Preprocessing and Outlier Detection

Initially, the source code is parsed to extract data, such as variable names, functions, dependencies and calculate metrics for every class, such as cohesion, coherence, complexity and size. Data and metrics are stored in a database [20]. Each class is treated as a vector with as many dimensions as the number of attributes (metrics) used, that is seven.

At this point we can look for extreme points (outliers) in the 7-dimensional space. The existence of outliers can indicate individuals or groups that display behavior very different from most of the individuals of the dataset. Outlier detection has many applications, including data cleaning. In order to do so, we use the Inter-Quartile Range (IQR) technique, used for describing the spread of a distribution.

It is based on the graphical technique of constructing a box plot, which represents the median of all the observations and two hinges, or medians of each half of the data set. Most values are expected in the interquartile range (H) to be located between the two hinges. Values lying outside the ±1.5H range are termed as "mild outliers" and values outside the boundaries of ±3H as "extreme outliers". This method represents a practical alternative for manual outlier detection, in the case of examining each variable class individually, and proves to be very efficient in handling multidimensional values.

Frequently, outliers are removed to improve the accuracy of the estimators, but sometimes, deleting an outlier that has a certain meaning, means also deleting its certain explanation. In our case, despite of the fact that we find outliers, that does not block out the continuation of the process, because they are part of the software source code considered.

3.2 Clustering Methodology

The next step involves clustering Java classes and corresponding metrics using k-Means to establish similarity, without outlier removal. Keeping outlier classes is advisable as they are not an effect of noise or faulty information, but part of the software. For determining the most appropriate number of clusters K, we took into account that a small number of clusters is desirable in this domain, as it provides better software overview and essentially easier error or weakness prediction. A grouping into a large number of clusters would not facilitate error discovery; k-Means is fast for small values for K.

k-means is repeatedly executed providing information on data smoothness. The measure used here is the Sum of Squared Error (SSE). Selection of the right SSE requires several iterations of the algorithm, which increases considerably the duration of the process, but produces alternative sets of clusters and, in several cases, a "knee" in the diagram indicates an appropriate value for K.

It should be noted that, although the values for K we are interested in is small enough to facilitate software analysis and comprehension, there is no 'right' or 'wrong' cluster number. The "ideal" K is defined either using the SSE or by analyzing each set of clusters and the classes they contain. Collected results are stored in tables and clusters are analyzed for controlling the characteristic values for each cluster, as well as, the uniformity using the mean and variance.

3.3 Classification into Clusters Methodology

In the final step of the process, a decision tree is built, using C4.5 in order to get insights into any source code vulnerabilities. Internal nodes of the tree correspond to some metric and leaves represent clusters. C4.5's use of information gain serves not only for categorizing data, but also for highlighting which metric is higher in the tree hierarchy. Metrics that play more important role in node splitting are the most effective on clustering.

C4.5's post-pruning evaluates the estimated misclassification error at each node and propagates this error up the tree. At each node, C4.5 compares the weighted error of each child node with the misclassification error, if the children nodes were pruned assigned as the class label of the majority class. So, metrics that had no part in cluster separation are not present in the tree hierarchy and their ability to evaluate software quality is deemed low.

Metrics that are present in the tree hierarchy characterize the clusters. In other words, we can test the utility of every metric on a specific data set. Adding up to previous results, we conclude that it would take more clusters to be able to separate into groups, according to the different characteristics evaluated. When these metrics are missing, the software presents universality and regularity. However, as the clustering process divides the source code into clusters with related classes, there is no need for extended division of the existing clusters. Thus, through the descriptive model is created a subway map for software evaluation leading to clusters.

It is a handy and informative way to identify weaknesses in the software source code, and certainly a way to know which metric values need improvement, to migrate any class to a different cluster. It is also a useful guide, for the developers, when extending the software, since they know that the desired characteristics need to have classes created with the objective of smoothness and good software design. Certainly, it is not an illustrative tool that can operate standalone.

4. Experimental Results

This section describes experiments conducted to analyze and find weaknesses in source code of Java open source systems. We detail here two case studies: a code editor and an application server. First we describe the experimental setup.

1. Extract and store software metrics in a database using Java Analyzer [20]. For the remaining steps we used Weka [19].
2. Apply Outlier detection with IQR in order to detect classes with special or exceptional characteristics. This process identifies fault-prone areas of code that require further testing. A maintenance engineer is able to access these classes directly, as they are shown separated from the rest of the source code. We do not remove these classes for the rest of the process.
3. Apply k-Means Clustering to classes based on their metric values. We select a low SSE value, in order to ensure that the classes in the same cluster are similar as opposed to classes in different clusters.
4. As soon as the clustering process is finished, the maintenance engineer can look into each cluster separately, in order to judge for every cluster, but he/she can get an overview of the system without having to examine each class separately.
5. Finally we classify the clusters that are formed in step 3, in order to assess the ability of metrics to evaluate software quality. Building the decision tree enables analyzing the ability of each metric to support software quality assessment.

4.1 Case Study 1: JEdit

JEdit is an open source code editor for 211 programming languages, written in Java and well-known to java developers [24]. Its large size and public availability make it suitable for data mining and for result replication if needed. We used 4 recent versions of JEdit (4.3, 4.4.1, 4.5 and 5.0), in order to be able to compare results and track evolution traits. We applied preprocessing and looked for outliers in versions 4.3, 4.4.1, and 4.5; outlier classes are available in all editions. High complexity, outlier classes and corresponding WMC values are shown in Table 1.

Table 1. *JEdit* outlier classes

Class Name	WMC
Parser	351
GUI Utilities	71
ActionHandler	41

Table 2. Clustering 4 versions of *JEdit*

Version	Classes	Outliers	Clusters	SSE
4.3	808	172	14	6.1
4.4.1	935	182	4	7.7
4.5	953	-	5	9.7
5.0	1077	155	5	10.5

We conducted outlier qualitative analysis and observed classes which have sufficiently high RFC, i.e. classes with many local methods, thereby reducing code extensibility, and in many cases associated with high complexity and large LOC. As a result, controlling these classes is quite difficult, whilst scalability is limited. Regarding inheritance, we observed low values for DIT and NOC, which limit code reuse. Finally, there are outliers with very little consistency, resulting in reduced encapsulation and data encryption. Greater method consistency means better class implementation.

Next we applied clustering. Table 2 displays comparative experimental data from analyzing each of the 4 versions of JEdit. SSE shows how close to each other are the classes in clusters, as points in the 7- dimensional space. Avoiding outlier removal resulted in outliers getting grouped together into clusters with extreme values for some metrics. Such clusters are observed in all of JEdit's versions.

A typical example is the 2nd cluster in JEdit 4.4.1, which contains only four classes. Typical outlier class here is *Parser* with 366 LOC and 44 CBO. Also, a cluster found in JEdit 5.0 consisting of 42 classes, contains the same classes as the corresponding cluster of JEdit 4.4.1. An important observation is that in each version of JEdit there was one cluster containing the largest part of the code, except from classes that have high or low values on some metrics, such as outliers, classes associated with the Graphical User Interface (GUI) or interface classes.

The final step of the process involves cluster classification, and produced similar results for all the JEdit versions. This step attempts to assess the metrics' ability to evaluate software quality and determine how easy it is for each metric to partition the code. This provides useful information related to code disadvantages and aspects the design team has to focus on. Fig. 1 shows part of the pruned tree with 99.48 % accuracy created by C4.5 for JEdit 4.5.

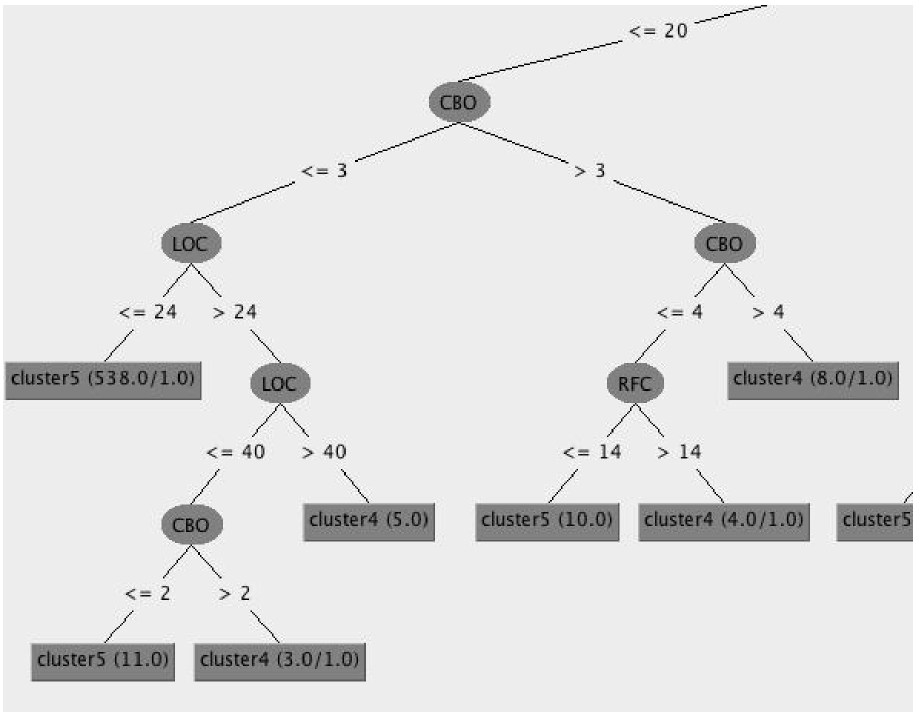

Fig. 1. Classification results for *JEdit*

All cases of classifying clusters in JEdit with pruning had a confidence factor of 0.25. However, increasing confidence factor did not improve accuracy. For instance, when confidence factor set to 1 and the tree is not pruned, results were the same. So, we found that omitting pruning did not improve accuracy, and, given that pruned trees are easier to understand one could opt for skipping pruning.

Table 3. *JEdit*: metric hierarchy

#	Metric	Metric Type
1st	DIT	Inheritance
2nd	RFC	Messaging
3rd	CBO	Coupling
4th	LOC	Volume
5th	WMC	Complexity

Table 4. Clustering *Geronimo*

Version	Classes	Clusters	SSE
2.1.8	2.523	5	51.1
3.0.0	3.100	7	3128

In the case of JEdit, LCOM and NOC are missing from the tree, meaning that in this case these metrics are used neither standalone, nor in conjunction with other metrics for clustering classes. We assume that DIT may have replaced NOC in clustering, as they are both related to inheritance. Interpreting the results for JEdit, we conclude that object oriented design metrics are more important than others. Furthermore, partitioning metrics according to their information gain value creates an inheritance and messaging measure that help us understand that software has scaling properties into these fields of design.

From a strategic project management perspective, all JEdit versions are well designed, but some classes have high complexity. Such clusters, containing high complexity classes, are the 4th cluster in version 5.0 containing 260 classes. That is a feature which considerably reduces software scalability. From the code reuse perspective, the inheritance relationship is quite low. Finally, the symbiotic relationship is quite high, resulting in fault proneness.

4.2 Case Study 2: Apache Geronimo

In order to reliably evaluate the proposed approach, we conducted a second experimental case study with a different type of open source software, the Apache Geronimo application server, written in Java [25]. It is widespread to server users, mainly for its functionality, such as supporting Eclipse servers for group writing Java code. The two versions we analyzed are 2.1.8 and 3.0.0. It is worth noting that the two versions have differences, including several new functions in 3.0.0.

This time we did not preprocess for finding outliers and extreme values. It is a large software system, and we wanted to analyze all the code and evaluate the characteristics of clustering in grouping outliers. Thus, groups of outliers were expected to be displayed after clustering [18]. First we clustered version 2.1.8. The iterative process did not result in finding a 'knee' in the curve of SSE, a fact that complicated the selection of K. The uniformity of the SSE and the number of clusters show that the code is consistently written. Nevertheless, 5 is a number of clusters where the slope of the SSE curve starts stagnating. So it is preferred to choose k=5 as this is also a convenient number of groups of classes, as explained earlier.

As far as version 3.0.0 is concerned, in a similar fashion we chose K=7. The figure below shows how the SSE decreases in relation to the number of clusters. As we see, the curve displays a knee between 3 and 4, but the slope of the SSE is large also between 3 and 10. This shows that a good K, is between 3 and 10 and will be chosen according to our needs. R should not be too large, because of the difficulty of further analysis results e.g. K = 50. For the continuity of the selected number of clusters, the SSE varies linearly with the number of clusters

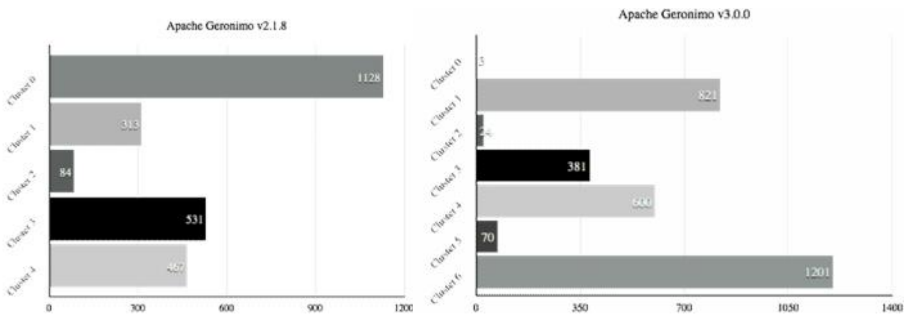

Fig. 2. Clustering results for *Apache Geronimo* v.2.1.8 & 3.0.0

Observing the distribution of clusters in the two versions, we see that it has changed enough, nevertheless, in both cases there are clusters with small numbers of classes, outliers, mainly due to complexity and coherence. Their percentage in the case of version 2.1.8 is 3% of the total code (Cluster 3) while in case of 3.0.0 constitutes 2.3% (Cluster 5). Whilst in the same direction as the previous version, the clusters of the next version contain the same classes. From a strategic perspective, the versions of Geronimo are well written and the most crowded classes have good metric values and require no special changes. The fact that some classes have larger size than others should not be of concern, as it is expected that some classes of the system to handle more data and functions than others.

Generally, the system is not badly designed; nevertheless one could reduce the consistency of complex classes breaking them into smaller, thus distributing functionality. Considering classes with many correlations, that are directly related, one could try reducing the connections. Cluster 0 of version 3.0.0 appears to have large children with several abstract classes, which could help future software expansion. Increasing the heredity in relation to the earlier version, indicates that the designers were concerned about future extensions, while moving in the same direction and reducing the complexity in relation to the number of classes.

In the next part of the experiment, we classify classes according to the previously resulted clusters, aiming at understanding important each metric is in affecting the clustering process. In other words, we try to explain and evaluate the ability of every metric used for clustering in the evaluation, to partition the software. In the case of Apache Geronimo, the classification process produced more complex results, as its size was bigger than JEdit. The size of the generated tree for 5 clusters in version 2.1.8 was 59 and the total number of leaves was 30. Again, as with JEdit, the highest information gain metric was DIT, followed by RFC. It is interesting that both versions of Geronimo produced the same metric hierarchy in the classification tree, as well as identical results with JEdit at the top and bottom of the tree. Table 5 displays the metric hierarchy produced by classification for Apache Geronimo.

In the case of classification accuracy, C4.5 misclassified seven instances (0.28% classification error) with confidence factor 0.25. We also examined other options in the classification process in order to find better results. The results without pruning were not better results in terms of accuracy but changed the metric hierarchy. More specifically, WMC was found second in the hierarchy along with RFC, in order to classify instances from clusters with outliers with big complexity.

Table 5. *Apache Geronimo* 2.1.8: metric hierarchy

#	Metric	Metric Type
1st	DIT	Inheritance
2nd	RFC	Messaging
3rd	LOC	Volume
4th	CBO	Coupling
5th	LCOM	Cohesion
6th	WMC	Complexity

We did not use the unpruned tree to evaluate the ability of the metrics because of the complexity of the tree consisting of 92 nodes, which appeared to merge metrics in order to find a good classification result. Using all of the metrics reduced accuracy from 99.72% to 99.33%. We continued trying out our options with 0.1 confidence factor, where accuracy was only reduced to 99.25% whilst reducing the tree size to 55 with 28 leaves.

5. Evaluation and Threats to Validity

The proposed evaluation was based on two criteria: firstly, it should be flexible, suitable and easy for analyzing systems and for assessing the ability of metrics to evaluate quality of systems with different functionality and nature. Secondly, it should be valid, reflecting the views and intuitions of the experts. Based on these criteria, we conducted two case studies on JEdit and Apache Geronimo, open source Java applications.

The aim of these case studies was to evaluate the selection of metrics, as well as the approach for assessing software quality. Starting with the evaluation and the selection of the metrics, we used a very popular suite, the CK metric suite [10] and a metric extraction application, which was made in order to extract and save the metric information from java source code [20]. The metric extraction rules we follow are described in section 2.

The proposed approach aims at finding weaknesses in source code, by identifying outliers and discovers classes that require attention by the software engineers. In section 4, we suggested several classes that need the attention of a supervisor software engineer. Search classes in modern software are oversized so their management becomes quite difficult; controlling such classes is quite laborious. The previous sub-process produces good results as the clustering of classes. In this part of the process we used a statistical overview. The IQR is a famous and useful path to find outliers in an unknown allocation. In our case, they were different across systems we tested, so validating the process was an important decision.

Thereafter, according to the proposed approach, we group similar classes through clustering. We display groups of classes with similar characteristics, which are not necessarily related to the class that implements them. Ranking the classes in clusters is determined by their distances, according to the values of their variables. This is suitable for their evaluation, regardless of their function, particularly in cases where the assessment has to be extended in the software. It is possible that someone outside the software development team can control the operation of the group.

The clustering process also has supervisory scope by the supervisor of the development team, and who can recognize the parts of code (clusters) which need better design. With regards to the validity of this process, we used SSE to guarantee that classes in the same cluster are correlated and classes in different clusters are not. With the continuous examination of the number of clusters, we tried to achieve the best possible result for their selection, so that a small number of them characterize the entire cluster population.

After the clustering process, we continue to the final and the most crucial part of the approach, where we use classification to access the ability of the metrics to evaluate the software. The aim of this process is to assess the ability of each metric to evaluate software quality depending on their ability of partitioning classes into clusters. We used the well known classification algorithm C4.5, which uses information gain as a splitting criterion. Also, by using pruning, we aim to display only the important metrics. Metrics that are not displayed in the tree do not play a crucial role in software evaluation.

Aiming to facilitate managerial decision making development, a decision tree is used as a classifier, with the objective knowledge about the behavior of groups that were created by clustering. Assistance is provided through understanding clusters, and the manager's decision to lead their team to draw lines using set point measures that lead to clusters with desired characteristics. Also, the decision tree can be used to understand the weaknesses of the source, as observed in the experimental procedure and the case of Apache Geronimo, where many classes were not reusing code without using data abstraction.

6. Conclusions and Further Work

We propose a source code quality evaluation approach, based on static analysis, using object oriented metrics. The use of these metrics reflects attributes, such as volume, size, complexity, cohesion and coupling. This work also focuses on assessing metric suitability for evaluation, which is to the best of our knowledge is novel. In order to do so, we employ C4.5 to evaluate the selected metrics and source code attributes in order to reflect their importance on evaluating software quality. By using this decision tree algorithm, we can separate the metrics which had a premium role in the process of evaluation and also make a decision related to the metrics participation in the process of quality evaluation.

Despite the positive results of this approach, there are certain limitations. The most important limitation is the lack of full automation, as decisions have to be made during the data mining process. This limitation may be overcome with further work on the availability of open source big projects and the ability to track to big companies' projects.

The proposed approach produces good results in the field of fault behavior detection with data preprocessing, grouping/organizing the classes with clustering techniques and finally categorizing the clusters using a description model with classification, so project managers have the ability to understand better the advantages and disadvantages of code. This approach facilitates maintenance engineers to identify classes which are fault prone and more difficult to understand and maintain, as well as, to assess the capability of expanding the system.

Data preprocessing and mining object oriented metrics extracted from source code, has the desirable result of testing driven by the metric characteristics instead of conducting directly changes on the source code. That reflects the most important aspects of a system concerning its quality and maintainability. The selected metric suite is the CK-suite [9], a well-known metric suite used for measuring the design of object oriented programs.

Starting with data preprocessing we chose the IQR statistical method because of its capability to respond to unknown data distributions, where the spread of the values is unknown, and differs across systems. In other words, with this type of preprocessing we can propose a small volume of likely fault prone classes, in a large class collection, so that the maintenance team, instead of checking all of the source code can immediately check the potentially fault prone proposed classes.

Secondly, we employ k-Means in order to manage large volumes of classes. k-Means offers a popular solution for clustering data, depending on their distance in space. Clustering utilizes the selected metrics for measuring distance. Every class is represented by a 7-dimension vector in space, so the algorithm works iteratively in order to group the classes. Classes with similar characteristics tend to be on the same cluster and non-associated classes are placed in other clusters. Then, we characterize each cluster as a single version of a category. Instead of just statistically analyzing the clustering results for evaluating metrics, we propose a classification process, using C4.5 in order to categorize clusters according to the features (metrics) used for node splits. This analysis facilitates decision making and assists categorization of clusters of similar classes using a decision tree.

In addition, we have experimented with several open source systems. Arguably, more can be done to expand the data sets to be used, both in terms of identifying systems of different kinds, and identifying important and relevant flaws and other software problematic behavior.

The results of our experiment show, that using clustering and classification algorithms on software metrics can facilitate fault prone classes detection and create system overviews, so further analysis can easily take place. In addition, maintainability and extendibility can be assessed by the "clustering followed by classification" technique we propose.

We plan to use other clustering techniques such as density based or hierarchical clustering in the future, given that the number of clusters is not known in advance. Using human experts will also consolidate validating the approach.

To sum up, the proposed approach provides a data mining model that can be used to discover useful information about software internal quality according to selected metrics. Different metric suites may assess software quality from another point of view. Data mining is a dynamic field and providing potential adjustments to software engineering offers notable results, especially as there is a growing need for discovering new methods to address contemporary software systems.

Acknowledgements

We would like to thank F. Vartziotis for providing his Java Source Code Analyzer.

References

[1] J. Tian, “Quality-Evaluation Models and Measurements”, *IEEE Software* Vol. 21, pp. 84-91, 2004.
[2] H.F. Li, W.K. Cheung, “An Experimental investigation of software metric and their relationship to software development effort”, *IEEE Transaction on software engineering*, Vol. 15, Issue: 5, pp. 649 - 653, May 1989.
[3] Y. Kanellopoulos, C. Makris and C. Tjortjis, “An Improved Methodology on Information Distillation by Mining Program Source Code”, *Data & Knowledge Engineering*, Elsevier, Vol. 61, No 2, pp. 359-383, May 2007.
[4] T. Menzies, J. Greenwald, A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, pp. 2-13, January 2007.
[5] H. Tribus, I. Morrigl, S. Axelsson, “Using Data Mining for Static Code Analysis of C”, *Proc. 8th Int’l Conf. Advanced Data Mining and Applications* (ADMA 2012), LNAI 7713, pp. 603-614, 2012.
[6] W.R. Bush, J.D. Pincus, D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software-Practice and Experience*, Vol. 20, pp. 775- 802, 2000.
[7] D. Spinnelis, *Code Quality the Open Source Perspective*, Addison Wesley, 2006.
[8] N.E. Fenton, *Software Metrics: A Rigorous Approach*, Cengage Learning EMEA, 1991.
[9] S. R. Chidamber, C. F. Kemerer, “Towards a Metrics Suite for Object Oriented Design”, *Proc. Conf. Object Oriented Programming Systems, Languages, and Applications* (OOPSLA’91), vol. 26, no. 11, pp. 197-211, 1991.
[10] S.R Chidamber , and C.F. Kemerer, “A metrics suite for object oriented design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493, June 1994.
[11] M. Halstead, *Elements of Software Science*, Elsevier, 1977.
[12] T.J. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, Vol. SE-2, Issue 4, pp. 308-320, December 1976.
[13] S. Dick, A. Meeks, M. Last, H. Bunke, A. Kandel, “Data mining in software metrics databases”, *Fuzzy Sets and Systems*, Vol. 145, Issue 1, pp. 81-100, July 2004.
[14] S. Zhong, T.M. Khoshgoftaar, N. Seliya, “Expert-Based Software Measurement Data Analysis with Clustering Techniques”, *IEEE Intelligent Systems*, Special Issue on Data and Information Cleaning and Preprocessing, pp. 22-30, 2004.
[15] N. Nagappan, T. Ball, A. Zeller, Mining Metrics to Predict Component Failures, *Proc. 28th Int’l Conf. Software Engineering* (ICSE 2006), pp. 452-461, 2006.

[16] Kanellopoulos Y., Antonellis P., Antoniou D., Makris C., Theodoridis E., Tjortjis C., Tsirakis N., "Code Quality Evaluation methodology using the ISO/IEC 9126 Standard", *Int'l Journal of Software Engineering & Applications*, Vol.1, No.3, pp. 17-36, July 2010.

[17] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, N. Tsirakis, "Employing Clustering for Assisting Source Code Maintainability Evaluation according to ISO/IEC-9126", *Proc. Artificial Intelligence Techniques in Software Engineering Workshop* (AISEW 2008) in ECAI08, 2008.

[18] M.H. Dunham, *Data Mining: Introductory and Advanced Topics*, Pearson Education, 2006.

[19] I.H. Witten, E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd Ed. Morgan Kaufmann, 2005.

[20] F. Vartziotis, Java Source Code Analyzer for Software Assessment, BSc Dissertation, Department of Computer Science & Engineering University of Ioannina, 2012

[21] Y. Kanellopoulos, I. Heitlager, C. Tjortjis, J. Visser, "Interpretation of Source Code Clusters in Terms of the ISO/IEC-9126 Maintainability Characteristics", *Proc. 12th European Conf. Software Maintenance and Reengineering* (*CSMR* 2008), IEEE Comp. Soc. Press, pp. 63-72, 2008.

[22] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, N. Tsirakis, "Clustering for Monitoring Software Systems Maintainability Evolution", *Electronic Notes in Theoretical Computer Science*, Elsevier, Vol. 233, pp. 43-57, March 2009.

[23] A.V.K. Prasad, S.R. Krishna, "Data Mining for Secure Software Engineering-Source Code Management Tool Case Study", *Int'l Journal of Engineering Science and Technology*, Vol. 2 (7), pp. 2667-2677, 2010.

[24] JEdit website: http://www.jedit.org Last accessed: 15/1/2014.

[25] Apache Geronimo website: http://geronimo.apache.org Last accessed: 15/1/2014.