2013 IEEE International Conference on Software Maintenance

An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams

Hafeez Osman*, Michel R.V. Chaudron*§ and Peter van der Putten*
*Leiden Institute of Advanced Computer Science
Leiden University, Leiden, the Netherlands
{hosman,putten}@liacs.nl
§Joint Department of Computer Science and Engineering
Chalmers University of Technology and Goteborg University
Gothenburg, Sweden
chaudron@chalmers.se

***Abstract*—There is a range of techniques available to reverse engineer software designs from source code. However, these approaches generate highly detailed representations. The condensing of reverse engineered representations into more high-level design information would enhance the understandability of reverse engineered diagrams. This paper describes an automated approach for condensing reverse engineered diagrams into diagrams that look as if they are constructed as forward designed UML models. To this end, we propose a machine learning approach. The training set of this approach consists of a set of forward designed UML class diagrams and reverse engineered class diagrams (for the same system). Based on this training set, the method 'learns' to select the key classes for inclusion in the class diagrams. In this paper, we study a set of nine classification algorithms from the machine learning community and evaluate which algorithms perform best for predicting the key classes in a class diagram.**

***Keywords*-Software Engineering; UML; Reverse Engineering; Machine Learning; Program Comprehension;**

I. Introduction

Up-to-date design documentation is important, not just for the initial design but also in later stages of development and in the maintenance phase. UML models created during the design phase of a software project are often poorly kept up to date during development and maintenance. As the implementation evolves, correspondence between design and implementation degrades [1]. For legacy software, faithful designs are often no longer available, while these are considered valuable for maintaining such systems.

A popular method to recover an up-to-date design of a system is reverse engineering. Reverse engineering is the process of analyzing the source code of a system to identify the systems components and their relationships and create design representations of the system at a higher level of abstraction [2]. Reverse engineering also refers to methods aimed at recovering knowledge about a software system in support of execution some of software engineering task [3]. Tool support during maintenance, re-engineering or re-architecting activities has become important to decrease the time that software personnel spends on manual source code analysis and helps to focus attention on important program understanding issues [4]. However, current reverse engineering techniques do not yet solve this problem adequately. In particular, reverse engineered class diagrams are typically a detailed representation of the underlying source code. This makes it hard for software engineers to understand what the key elements in the software structure are [5]. Although several Computer Aided Software Engineering (CASE) tools have options to leave out several properties in a class diagram they are unable to automatically identify classes that are less important.

This paper is partially motivated by a scenario where new programmers want to join a development team. They need a starting point in order to understand the whole system before they are able to modify it. Provided with the software design, the programmer will normally browse the class design and try to synchronize the design with the source code. There is a need for programmers to know which classes in the system play important roles or can be considered as key classes in the system.

Fernández-Sáez et al. [6] found that developers experience more difficulties in finding information in reverse engineered diagrams than in forward designed diagrams and also find the level of detail in forward designed diagrams more appropriate than in reverse engineered diagrams. In order to achieve better reverse engineered representations we need to learn which information from the implemented system to include and which information to leave out. A method to assist software engineers to focus on the key classes and aspects of the design is needed. The identification of key classes can also be used to simplify a complex class diagrams or help to predict the severity of a defect in a software system.

Research Problem. This paper specifically aims at providing suitable classification algorithms to decide which classes should be included in a class diagram. We seek an automated approach to classify the key classes in a class diagram. The

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.25

algorithms need to be able to produce a score, not just a classification, so that a user potentially has the option to choose a particular level of abstraction for representing a reverse engineered design.

This paper focusses on using design metrics as predictors (input variables used by the classification algorithm). The advantage of using design metrics is that these can be obtained very efficiently with little effort. This fits our objective of creating a method that will be of practical use to software developers. Also, we analyse the predictive power of the predictors to know how influential each of these predictors is in the performance of the classifier.

Contribution. We explore several classification algorithms for predicting key classes that should be included in a class diagram. As input for this study, we use sets of source codes from open source projects with corresponding UML models that contain forward designed class diagrams. We use these class diagrams as 'ground truth' to validate the quality of the prediction algorithms. The methods we study will learn from the forward designed class diagrams which classes should be selected from the reverse engineered class diagrams.

In total, nine algorithms were selected for this comparison study. These algorithms will be evaluated in terms of accuracy and robustness with respect to the information that they recommend to keep in and leave out of the class diagram. The candidate set of algorithms includes: J48 Decision Tree, k-Nearest Neighbor, Logistic Regression, Naive Bayes, Decision Tables, Decision Stumps, Radial Basis Function Networks, Random Forests and Random Trees.

We have collected a diverse collection of data sets consisting of nine pairs of UML design class diagrams and associated Java source code derived from open source software projects. The number of classes in the source code of these projects ranges from 59 to 903. Out of these classes 3% to 47% was found to be included in the forward UML class diagram. The open source projects were chosen for a number of reasons. We wanted the data to be representative for the diversity and complexity of real world projects. The quality of documentation for open source projects varies widely and there is also a large variation in the ratio of classes in the forward design versus classes in the source code. In open source projects software design is not a mandatory requirement, and these projects rely on volunteers working together in a distributed fashion. Also, by using open source projects we make it easier for other researchers to reproduce or compare against our results, and develop new methods on the same data.

The paper is structured as follows: Section II discusses related research and Section III describes the research questions. Section IV explains the basics of machine learning. Section V explains the approach on how we conducted the evaluation. Section VI presents the result and section VII discusses our findings and future work. This is followed with conclusion in Section IX.

II. Related Work

The following studies are related to our research from the perspective of identifying key classes from software artefacts.

Zaidman and Demeyer [7] proposed a method for identifying key classes using web mining techinques. They used dynamic analysis of the source code as basis for the identification of key classes. For validating their method, they manually identified key classes from the software documentation. Recall and precision were used to evaluate the approach and they found that their approach was able to recall 90% of the key classes and the precision was slightly under 50%. However, dynamic analysis approaches need significant effort for collecting run-time traces.

Perin et al. [8] proposed ranking software artefacts using the PageRank algorithm. They used the Pharo Smalltalk system and Moose reengineering environment as case studies. For the Pharo Smalltalk system, they reported that their approach was able to detect 42% of the important classes (prediction based on classes) and to detect 52% of the important classes when prediction was based on methods. However, no precision result was presented for the Moose system.

Hammad et al. [9] proposed an approach to identify the critical software classes in the context of design evolution. Version (commit) history and source code were used as the input for this study. They assumed that the classes that were frequently changed in the software evolution are the classes that are critical for the system and as the result, they found only about 15% of the classes in the case studies were impacted from six design changes.

Steidl et al. [30] presented an approach to retrieve important classes of a system by using network analysis on dependency graphs. They performed an empirical study to find the best combination of centrality measurement of dependency graph. They used classes recommended by their tests projects developers as the baselines.

Zimmermann et al. [10] presented a prototype tool that applies data mining to predict and suggest further changes to the developer when a system is modified. Version management repositories of open source projects were used as input.

Singer et al. [11] developed a tool to recommend files that are potentially relevant to the file that a developer is currently viewing. This study aimed at easing navigation in source code. The tool ranks the files that are (potentially) related.

Our work also aims at identifying key classes but we explore diverse classification algorithms based on supervised machine learning. In contrast, static analysis is used for our data collection as it is easy to obtain from open source projects. Our validation consists of comparing selected classes against those actually found in the forward design.

III. Research Questions

This section describes the research questions of this study that will be answered in section VI.

RQ1: Which individual predictors are influential for the classification? For each case study, we explore the predictive power of individual predictors.

RQ2: How robust is the classification to the inclusion of categories of predictors? We explore how the performance of the classification algorithm is influenced by partitioning the predictor-variables in different sets.

RQ3: What are suitable classification algorithms in classifying key classes? The candidate classification algorithms are evaluated to find the suitable algorithm(s) in classifying the key classes in a class diagram.

IV. Basics of Machine Learning

This section describes the basics of machine learning terms and algorithms used in this paper.

A. Univariate Analysis

To measure predictive power of predictors we used the information gain with respect to the class [33]. Univariate predictive power means measuring how influential a single predictor is in prediction performance. The results of this algorithm are normally used to select the most suitable predictor for prediction purposes. Nevertheless, in our study we did not use it for predictor selection, but for an exploratory analysis of the usefulness of various predictors (in our case: class diagram metrics). In our approach, class diagram metrics were used as predictors for the prediction of key classes.

B. Machine Learning Classification Algorithm

Prior to making a selection of the classification algorithms, we ran exploratory experiments on a wider range of algorithms. We do not expect that there will be a single silver bullet algorithm that will outperform all others across all sets of problems. Also, we are not just interested in a single algorithm that scores a top result on a given problem, but are looking for sets of classifiers (i.e. classification algorithms) that produce robust results across domains. In this way, algorithms become more portable across problems with very different rates of inclusion of classes in designs. We also aimed for a mix of classifiers in terms of expected bias (what relationships can be captured) and variance (does the prediction change when trained on different random samples) [15]. As discussed, we wanted to use a diverse set of algorithms representative for different approaches. For example decision trees, stumps, tables and random trees or forests all divide the input space up in disjoint smaller sub spaces and make a prediction based on occurrence of positive classes in those sub spaces. K-NN and Radial Basis Functions are similar local approaches, but the sub spaces here are overlapping. In contrast, logistic regression and Naive Bayes model parameters are estimated based on potentially large number of instances and can thus be seen as more global models. The nine classification algorithms are the following (see [16] for more explanation):

1) Decision Table: A Decision Table consists of rows and columns that associate a set of conditions or test with a set of action. The Waikato Environment for Knowledge Analysis (WEKA) [33] used a simple Decision Table Majority (DTM) classifier.

2) Decision Stumps: Decision Stumps are decision trees consisting of just a single level and split [15]. A decision stump makes a prediction based on the value of just a single input feature, and is a good baseline classifier to compare against decision trees and other classifiers, to determine what results can already be achieved with a very basic model.

3) J48 Decision Tree (J48): J48 is a WEKA implementation of the C.45 decision tree algorithm. This algorithm generates a classification-decision tree for the given data set by recursive partitioning of data.

4) k-Nearest Neighbour (k-NN): k-NN classification finds a group of k objects in the training set that are most similar to the test object and bases its classification on the predominance of a particular class in this neighborhood [18].

5) Logistic Regression(LR): uses a linear input combination of input variables to provide and output score, which is then mapped to a probability by applying a logistic function [19].

6) Naive Bayes: Naive Bayes is a classification algorithm based on the Bayes rule of conditional probability. It assumes independence across the predictors.

7) Radial Basis Function (RBF) Networks: RBF Networks are a type of neural network. We used simple normalized Gaussian functions that each cover a part of the input space, and the output is then fed into a basic feed forward neural network [20].

8) Random Forest: Random Forest is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [21].

9) Random Tree: Random Tree algorithm builds a classification tree considering K randomly chosen predictors at each node.

C. Evaluation Method

For Univariate analysis, the predictors were evaluated by using the InfoGain Attribute Evaluator (InfoGain). This method produces a value which indicates the influence of a predictor in prediction performance based on the case studies. A higher value of InfoGain denotes a stronger influence of the predictor (i.e. closer to 1 is better). The evaluation of machine learning classification algorithms started with generating a confusion matrix based on applying a classification algorithm in WEKA. Table I shows an example of a confusion matrix. In this table, for the case of the

Table I
CONFUSION MATRIX

Prediction Result		Actual Result
Y	**N**	
TP	*FN*	**Y**
FP	*TN*	**N**
Example :		
Y	**N**	
11	33	**Y**
10	849	**N**

actual data is positive (Y), *TP* represents the number of correct predictions (true positive) and *FN* represents the number of incorrect predictions (false negative) done by the classification algorithms. In the case of the actual data is negative (N), *FP* represents the incorrect predictions (false positive) while *TN* represents correct predictions (true negative). The example in Table I shows that the *TP* is 11 classes while the *TN* is 849. The *FN* is 33 and the *FP* is 10. The total number of classes in this case study is 903.

The classification algorithms were evaluated using Area Under ROC (ROC means Receiver Operating Characteristics) curve [22]. The AUC shows the ability of the classification algorithms to correctly rank classes as included in the class diagram or not. The larger the ROC area, the better the classification algorithms in term of classifying classes [23]. Based on this, WEKA provides AUC calculations that ease our evaluation task. This tool presents the result as a number between 0 and 1. A value closer to 1 means a better classification result while a value close to 0.50 means the classification performs almost randomly. The algorithms need to provide reliable estimates across the score range, thus we evaluate using the AUC value rather than for instance accuracy (e.g. percentage of correct or incorrect prediction). For highly unbalanced data this also avoids the issue of favouring models that just predict the majority outcome class.

In general, this measure is better than instance accuracy such as percentage correct because for projects with very low ratios of classes in the UML class diagram compared to the source code, a classification algorithm that would exclude all classes would score high on accuracy. For instance, referring to the example in Table I, the percentage of correct prediction is 95.24% and incorrect prediction is 4.76%. It seems that the algorithm performs an excellent prediction. However, the 95.24% correct prediction is the result for overall prediction result by taking the sum of correct prediction divided by number of classes. The percentage of correct prediction for *TN* is 98.80% while the percentage of correct prediction for *TP* is 25.00%. The resulting prediction performance for *TP* is very low even though overall correct prediction is very high. The AUC value measures performance of a model over the entire range of models scores, i.e. how well it separates by changing the score threshold of a class over the entire score range. Therefore, AUC is preferred over accuracy as a measure.

Based on early observation on our case studies, we decided the threshold for the AUC value = 0.60. This means, if the AUC value >= 0.60, the classification algorithm is considered to be usable for prediction for our specific problem.

V. APPROACH

This section describes the Examined Predictors and Tools, Case studies and Process.

A. Examined Predictors and Tools

In this section, we describe i) the metrics that we used as inputs to the prediction algorithms, and ii) the tools used for this research.

1) Examined Predictors: We used a set of software metrics that are typically used to characterize classes in class diagrams as input to our classification algorithms. SDMetrics[1] is capable of computing 32 types of class diagram metrics. These metrics are divided into five categories namely Size, Coupling, Inheritance, Complexity and Diagram. These classes of metrics were selected for the following reasons: i) our earlier survey [29] [31] showed that developers use Size and Coupling as predictors for key classes, ii) experts in the area of software metrics (Briand et al. [27] and Genero et al. [24]) stated that Coupling is an important structural dimension in object-oriented design, iii) the work in [14] and [12] showed that WMC (a metric in the Size category) and CBO (a metric in the Coupling category) are influential for defect prediction. The specific set of 11 metrics used is shown in Table II.

2) Tools: The tools used in this study are the following:

- Reverse Engineering Tool : MagicDraw[2] is a system modeling tool that provides reverse engineering facilities. MagicDraw version 17.0 (academic evaluation license) was used for this study.
- Software Metrics Tool : SDMetrics is a tool that computes a large set of metrics for UML models. SDMetrics version 2.2 (academic license) was used for this study.
- Data Mining Tool : Waikato Environment for Knowledge Analysis (WEKA) is a collection of machine learning algorithms for data mining tasks [16]. It contains tools for data pre-processing, classification, clustering, and visualization. WEKA version 3.6.6 was used for this study.

B. Case Studies

We used the following criteria for selecting case studies:

- Open source software/system project that provides implementation source code and forward design class diagram.

[1]www.SDMetrics.com
[2]www.nomagic.com

Table II
LIST OF CLASS DIAGRAM METRICS

Metrics	Category	Description
NumAttr	Size	The number of attributes in the class.
NumOps	Size	The number of operations in the class. Also known as WMC in [13] and Number of Methods (NM) in [25].
NumPubOps	Size	The number of public operations in a class. Also known as Number of Public Methods (NPM) in [25].
Setters	Size	The number of operations with a name starting with 'set'.
Getters	Size	The number of operations with a name starting with 'get', 'is', or 'has'.
Dep_Out	Coupling (import)	The number of dependencies where the class is the client.
Dep_In	Coupling (export)	The number of dependencies where the class is the supplier.
EC_Attr	Coupling (export)	The number of times the class is externally used at attributes type. This is a version of OAEC +AAEC in [26].
IC_Attr	Coupling (import)	The number of attributes in the class having another class or interface as their type. This is a version of OAIC+AAIC in [26].
EC_Par	Coupling (export)	The number of times the class is externally used as parameter type. This is a version of OMEC+AMEC in [26].
IC_Par	Coupling (import)	The number of parameters in the class having another class or interface as their type. This is a version of OMIC+AMIC in [26].

- The amounts of classes in the implementation (source code) should be at least 50 classes.

Based on these criteria, nine open source software/systems were selected. In this projects we selected a forward UML class diagram from the documentation and then selected a matching version of the source code. The amount of classes in these case studies ranges from 59 to 903 (see Table III). The ratio between the number of classes included in the UML class diagram and the number of classes in the implementation (source code) spreads across a wide range: from 3 to 47%. This large range in characteristics of the input may be a difficulty for building a reliable classifier for our domain. For this reason, we focus on algorithms that will produce a score for a class concordant with the likelihood that it would be included in the UML diagram. This will allow a developer to vary the amount of classes included, i.e. the level of abstraction, by changing the threshold on the score.

We make the case studies used for this research available at [17] for future research and for validation of this study.

[3]http://argouml.tigris.org/
[4]http://mars-sim.sourceforge.net/
[5]http://java-player.sourceforge.net/
[6]http://sourceforge.net/projects/jgap/
[7]http://neuroph.sourceforge.net/
[8]http://jpmc.sourceforge.net/
[9]http://code.google.com/p/wro4j/
[10]http://code.google.com/p/xuml-compiler/
[11]http://code.google.com/p/maze-solver/

Table III
LIST OF CASE STUDY

No	Project	Total Classes in Source Code (S)	Total Classes in Design (D)	D:S ratio as %
1.	ArgoUML[3]	903	44	4.87
2.	Mars[4]	840	29	3.45
3.	JavaClient[5]	214	57	26.64
4.	JGAP[6]	171	18	10.52
5.	Neuroph 2.3[7]	161	24	14.90
6.	JPMC[8]	121	24	19.83
7.	Wro4J[9]	87	11	12.64
8.	xUML[10]	84	37	44.05
9.	Maze[11]	59	28	47.45

C. Process

This subsection describes the steps performed for this study. The inputs for this process are forward designs and reverse engineered class diagrams (constructed from source code of the case studies). The output of the machine learning phase is the list of key classes that can be used to condense a class diagram. The approach is illustrated in Figure 1.

1) Data Preparation: For data preparation, class diagram metrics were extracted from the reverse engineered class design (obtained using reverse engineering with the Magic-Draw CASE tool). Then, the information about the presence of a class in the forward design was entered in a table.

The data preparation steps are described in Table IV. We expect that there is some noise in the predictors. For instance, the getters and setters predictor completely rely on conformance of source code to naming conventions (e.g 'get','has'). Not all case studies have this kind of naming convention. There is also the possibility that operations that are shown at a child class are actually inherited. In this case, the operation is counted twice. To study whether this noise has significant influence, we experimented with different sets of predictors: experiment A: the full set of predictors (predictor set A), experiment B: all metrics, but excluding metrics related to getters, setters and Number of Public Operation (predictor set B), and experiment C: set of predictors that only uses Coupling metrics (predictor set C). The details of all predictor sets are shown in Table V.

2) Data Processing and Analysis of Results: For every training and test activity for the classification algorithm, we randomly split every single predictor set (for every case study) into 50% for the training set and the other 50% for the test set. The main reason for doing this is that the data are highly unbalanced where the number of classes in design (the 'positives') is very low compared to the number of classes in the source code. If we would have used 10 fold cross validation, it means that we use only 10% of the data for testing and 90% for training. Thus, the possibility of any positives to be included in the test data was very low, and test set error measurements would not have been reliable. To further improve reliability, we ran each experiment 10 times using different randomization.

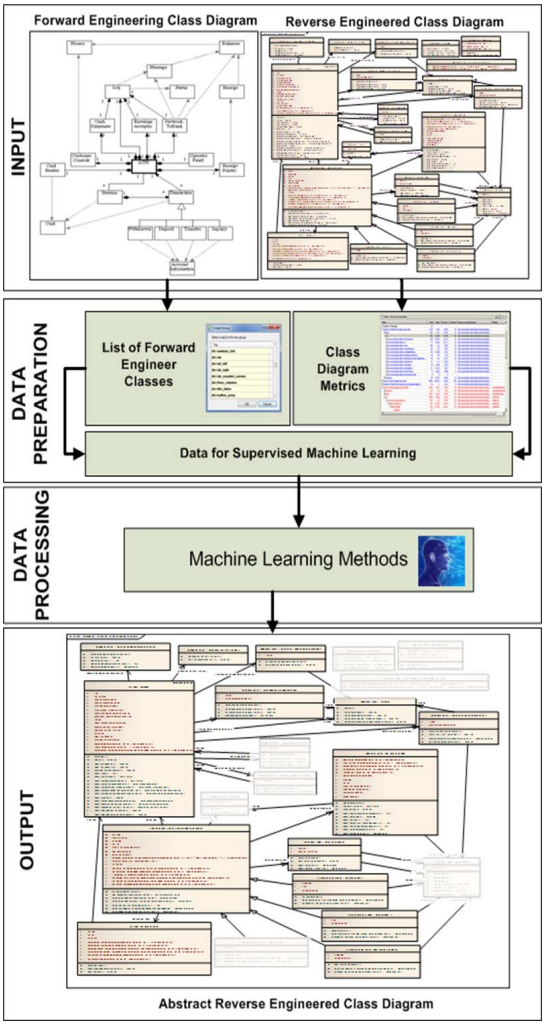


Figure 1. Design Abstraction Process

We assumed a fully automated method and that our end users have no knowledge of data mining. We also wanted to test for robustness under a selected threshold. Algorithms ran with default WEKA configuration, except for k-Nearest Neighbor, for which we used two different neighborhood size settings (1 and 5 neighbors).

VI. Evaluation of Results

This section presents our evaluation on i) predictive power of predictors and ii) overview of benchmark AUC results.

A. Predictor Evaluation

This subsection presents our univariate analysis results that measure the predictive performance of each predictor using information gain.

RQ1. The results show the influence of a predictor for the classification algorithm. A class diagram metric is considered to be influential for prediction when the value of the InfoGain Attribute Evaluator is greater than 0. Figure 2 shows that out of eleven class diagram metrics used in this study, nine of them influenced the prediction in the JavaClient project while xUML and Mars have eight and seven influential class diagram metrics. On the other

Table IV
Data Preparation Steps

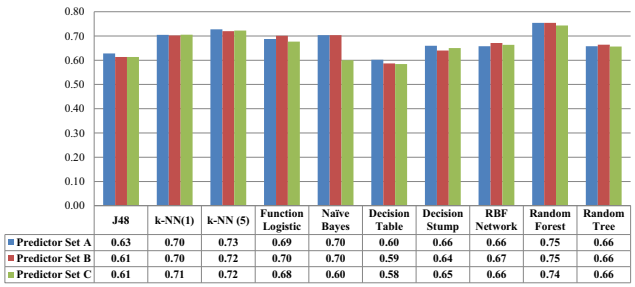
No	Preparation Step	Description
1.	List all the classes that appear in the UML design-class diagram	Input to get the class in design vs. implementation ratio
2.	Reverse engineer the source code into a class diagram using MagicDraw. Save the class diagram in XML Metadata Interchange (XMI) file format	To get the design from the source code prepared for the metrics tool input
3.	Calculate the software metrics on the reverse engineered class diagram using SDMetrics and save in CSV format	Class diagram metrics calculation and data mining input preparation
4.	Manually remove external library classes and runtime classes from the list	To extract only developed classes in the source code
5.	Merge the software metrics information from the source code and the classes in the forward design	To map between classes in design and classes from software metrics obtained from the source code
6.	Amend the CSV file by adding the information of "In Design" properties (N for not presented in Design Document, Y for presented in Design Document)	Add the information about the class in design in the software metrics information
7.	Remove software metrics properties that show no significant information (data cleanup) present an overall data summary in plot graph	To extract only the selected independent variable (class diagram metrics) and present the summary of data

Table V
Predictor Sets

No	Predictor	Predictor set A	Predictor set B	Predictor set C
1	NumAttr	Yes	Yes	No
2	NumOps	Yes	Yes	No
3	NumPubOps	Yes	No	No
4	Setters	Yes	No	No
5	Getters	Yes	No	No
6	Dep_out	Yes	Yes	Yes
7	Dep_In	Yes	Yes	Yes
8	EC_Attr	Yes	Yes	Yes
9	IC_Attr	Yes	Yes	Yes
10	EC_Par	Yes	Yes	Yes
11	IC_Par	Yes	Yes	Yes

Project	NumAttr	NumOps	NumPubOps	Setters	Getters	Dep_out	Dep_In	EC_Attr	IC_Attr	EC_Par	IC_Par
ArgoUML	0.000	0.000	0.000	0.000	0.000	0.024	0.000	0.000	0.000	0.000	0.000
Mars	0.000	0.013	0.017	0.011	0.025	0.000	0.047	0.037	0.000	0.031	0.000
JavaClient	0.093	0.048	0.044	0.000	0.050	0.215	0.093	0.000	0.183	0.092	0.225
JGAP	0.073	0.056	0.000	0.078	0.000	0.047	0.000	0.000	0.000	0.058	0.000
Neuroph	0.000	0.054	0.062	0.000	0.000	0.000	0.084	0.000	0.000	0.106	0.000
JPMC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.059	0.000	0.000
Wro4J	0.000	0.000	0.000	0.000	0.000	0.000	0.212	0.111	0.000	0.196	0.000
xUML	0.168	0.281	0.281	0.306	0.147	0.240	0.000	0.000	0.085	0.000	0.506
Maze	0.000	0.000	0.000	0.000	0.000	0.000	0.171	0.178	0.000	0.125	0.000

Figure 2. Univariate Predictor Performance (Information gain)



	J48	k-NN(1)	k-NN (5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
Predictor Set A	0.63	0.70	0.73	0.69	0.70	0.60	0.66	0.66	0.75	0.66
Predictor Set B	0.61	0.70	0.72	0.70	0.70	0.59	0.64	0.67	0.75	0.66
Predictor Set C	0.61	0.71	0.72	0.68	0.60	0.58	0.65	0.66	0.74	0.66

Figure 3. Average AUC Score for Every Data set

hand, ArgoUML and JPMC have only one influential class diagram metric. Figure 2 also shows that the Coupling category metrics are influential for every case study. For all cases, at least one of the Coupling metrics is listed as influential for prediction. This means that class diagram metrics categorized in Coupling (i.e. IC_Par, EC_Par, IC_Attr, EC_Attr, Dep_In and Dep_out) have a strong influence on prediction ability. If we compare Coupling metrics with Size metrics (i.e. NumAttr, NumOps, NumPubOps, Getters, Setters) we find that only five case studies list at least one of the Size-metrics as influential predictor. EC_Par is the most influential class diagram metrics because it is listed as influential in prediction for six out of nine case studies.

RQ2. We have studied the predictors through three different experiments (as defined in Table V). Figure 3 shows the average AUCs of classification algorithms for all experiments. We expected to see a large difference in prediction performance among the three experiments. However, there is not much difference in prediction performance as we can see in Figure 3 the difference in average AUC is only $\pm$ 0.02. From this figure, we found out that the performances slightly degrade for experiment C but the amount of degradation is not very significant. This means, even though the number of predictors in experiment C is smaller than in experiments A and B, the set of predictors is still reliable for prediction purposes. This shows that the Coupling category strongly influences the prediction performance.

B. Benchmark Scoring Results

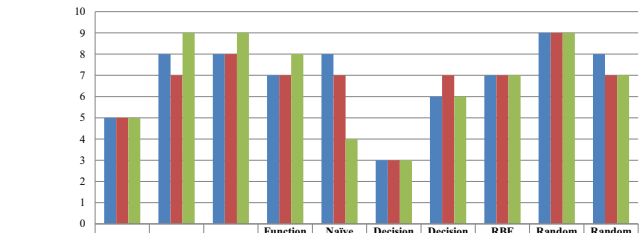
RQ3. The classification algorithms were evaluated by measuring the average and standard deviation of the AUC over ten runs for each predictor set. Table VI shows an example of results for experiment C. We have highlighted those cells that contain very weak classification results, i.e. AUC<0.60. Note that an AUC of 0.50 means that the classifier produces completely random result. We decided a value of AUC of 0.60 or higher indicates a useful algorithm. This means, the classification algorithms that are able to produce this score for almost all case studies for all experiments are considered suitable for classifying key classes.

After running all the experiments, we found that the Random Forest and K-Nearest Neighbor (k-NN(5)) algorithms perform the best in classifying the key classes in a class diagram in terms of overall AUC as well as robustness over various predictor sets.

Table VI
RESULTS FOR PREDICTOR SET C

Project	J48	k-NN(1)	k-NN (5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
ArgoUML	0.50	0.69	0.69	0.54	0.50	0.50	0.55	0.50	0.64	0.60
	0.00	0.04	0.05	0.05	0.00	0.00	0.07	0.07	0.04	0.03
Mars	0.53	0.69	0.75	0.61	0.73	0.52	0.70	0.58	0.73	0.61
	0.06	0.03	0.05	0.05	0.09	0.05	0.07	0.16	0.09	0.08
JavaClient	0.76	0.83	0.86	0.81	0.82	0.78	0.75	0.80	0.86	0.81
	0.09	0.03	0.04	0.05	0.02	0.07	0.06	0.04	0.04	0.05
JGAP	0.54	0.60	0.62	0.67	0.51	0.51	0.59	0.65	0.72	0.60
	0.07	0.05	0.07	0.12	0.02	0.02	0.04	0.10	0.10	0.17
Neuroph	0.61	0.79	0.82	0.71	0.53	0.56	0.63	0.72	0.78	0.68
	0.14	0.06	0.06	0.10	0.07	0.09	0.08	0.16	0.09	0.08
JPMC	0.54	0.66	0.67	0.69	0.50	0.50	0.58	0.61	0.69	0.59
	0.08	0.06	0.06	0.08	0.00	0.01	0.03	0.06	0.09	0.08
Wro4j	0.63	0.70	0.68	0.77	0.62	0.62	0.70	0.69	0.74	0.68
	0.18	0.09	0.19	0.16	0.13	0.12	0.13	0.21	0.14	0.14
xUML	0.74	0.78	0.77	0.69	0.62	0.69	0.72	0.82	0.83	0.75
	0.06	0.07	0.06	0.12	0.11	0.10	0.06	0.04	0.06	0.06
Maze	0.67	0.61	0.64	0.60	0.57	0.58	0.63	0.60	0.70	0.59
	0.07	0.10	0.14	0.11	0.08	0.07	0.06	0.10	0.10	0.08
Average	**0.61**	**0.71**	**0.72**	**0.68**	**0.60**	**0.58**	**0.65**	**0.66**	**0.74**	**0.66**
	0.09	**0.08**	**0.08**	**0.08**	**0.11**	**0.10**	**0.07**	**0.10**	**0.07**	**0.08**

Note : The first row for each predictor set is the average AUC, the second row lists the standard deviation. Cells with AUC <0.60 are highlighted.



	J48	k-NN(1)	k-NN(5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
Prediction Set A	5	8	8	7	8	3	6	7	9	8
Prediction Set B	5	7	8	7	7	3	7	7	9	7
Prediciton Set C	5	9	9	8	4	3	6	7	9	7

Figure 4. AUC Score >= 0.60

Figure 4 shows the prediction performance of all selected classification algorithm. This figure illustrates the number of case studies (for each predictor set) in which the classification algorithm produces an AUC score greater than 0.60. Random Forest and k-NN(5) perform the best prediction where both classification algorithm produced AUC score above 0.60 for at least 8 case studies for all datasets. Meanwhile, Random Tree, Function Logistic, RBF Network and Decision Stump perform less robust across all predictor sets. These classification algorithms performed reasonably well. They produced an AUC above the threshold for 6

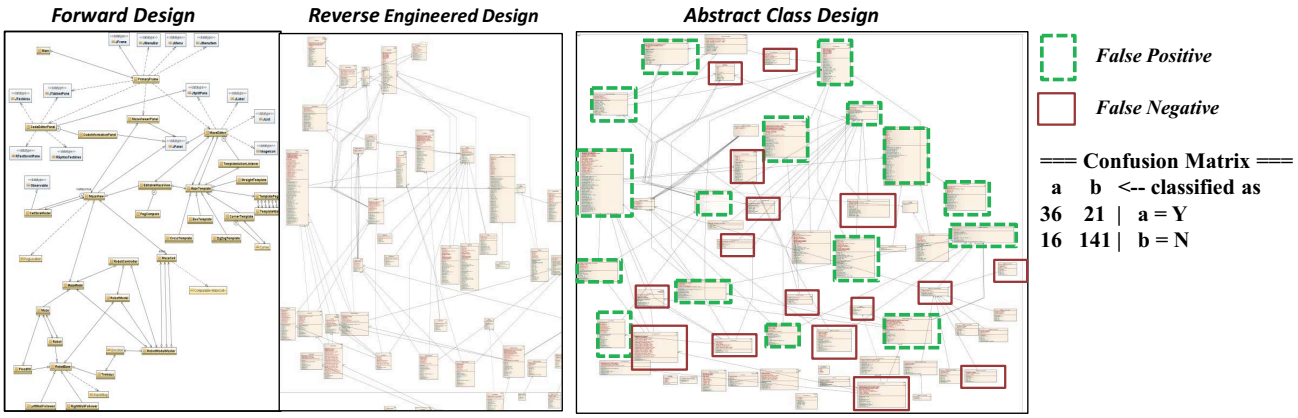


Figure 5. Application of Random Forest Classification Algorithm

to 8 case studies. Naive Bayes performed well for case studies using predictor set A and predictor set B but not using predictor set C. J48 and Decision Table appear not to be suitable to be used in these case studies, given the low number of results with AUC >= 0.60 (between 3 to 5). In addition, Figure 3 shows that the performance for Naive Bayes degrades tremendously when predictor set C is used for training even though the average AUC score for predictor set A and predictor set B are equal 0.70. The average AUC score of more than 0.72 for Random Forest and k-Nearest Neighbor (k-NN(5)) shows their suitability for all predictor sets.

Figure 5 illustrates the application of our method. In particular, it applies the Random Forest classification algorithm to the JavaClient case study. As result, a confusion matrix was generated. It shows that the total number of classes is 214 with 57 of the classes in the forward design. The generated confusion matrix shows that 36 out of 57 classes are correctly predicted as should be present in the class diagrams. Also 141 out of 157 classes are correctly predicted as should be omitted from the abstract class diagram. On the other hand, there are 21 false negatives (predicted as leave out, but should be included) and 16 classes that are false positives (predicted as 'include', but should not be included).

VII. Discussion and Future Work

With this result, we can conclude that class diagram metrics from the Coupling and Size category can be good predictors for key classes in class diagrams. In summary, there are three class diagram metrics that should be considered as influential predictors: Export Coupling Parameter (EC_Par), Dependency In (Dep_In) and Number of Operation (NumOps). This means, the higher value of these metrics in a class may indicate that this class can be a candidate of an important class. Thus, newcomers in the project may use this information to manually predict the key classes when joining project. This finding is consistent with [29] and [31] where the Number of Operation and Relationship (related to coupling) are the elements that are most software developer looked at in order to find the important classes in a class diagram.

The results show that k-NN(5) and Random Forest are suitable classification algorithms in this study. We took a step forward by exploring this classification algorithm by applying the algorithm individually to several case studies. As a result, some of the predicted True Positive in algorithm k-NN(5) are predicted False Positive in Random Forest and vice versa. We compared all the result manually from those two algorithms applied to several case studies and some of true and false results are different. The possibility to enhance this prediction power is by combining those classification algorithms to achieve the best result. Given the unbalanced data, the algorithms were not able to produce high AUC scores.

Basically, this study is expected to discover suitable classification algorithms which could provide a rank score concordant with the likelihood for classes to be included in the UML class diagram. Based on this result, we are able to produce an approach for ranking classes for importance. This will allow the software engineer to generate a UML diagram at different levels of detail. To construct the abstraction of the class diagrams, the software engineer may apply the abstraction of relationship in class diagrams as presented by Egyed [28].

This study was an early experimental benchmark, and we see a number of ways to extend this work. Alternative input parameters for predicting the key classes in a class diagram could be investigated. This could include the use of other type of design metrics for example based on (semantics of) the names of classes, methods and predictors. There are also possibilities to use source code metrics such as Line of Code (LOC) and Lines of Comments as additional predictors for the classification algorithms. Moreover, we could look at identification of features as unit of inclusion or exclusion in

the UML class diagram. Also, more extensive benchmarking should take place, for instance by learning models on one problem and testing it on another, or testing out an ensemble approach that combines classification algorithms. Specific approaches exist to better transfer knowledge across different problems, such as transfer learning.

Another approach to deal with limited availability of 'ground truth'-data for validation is to use a semi supervised or interactive approach, where a user first selects some limited top level classes, then the system learns and recommends further classes to be included, and the user responds by confirming or rejecting recommendations. Building an interactive application may also help to guide future research.

In terms of predictive performance, it could be interesting to compare the result of this study with other approaches. This study uses the classes in the forward design as the ground truth. In version history mining the classes that are frequently changing are seen as candidates for key classes [9]. It is also interesting to compare our approach with other works that apply different algorithm such as HITS web mining (used in [7]), network analysis on dependency graphs (used in [30]) and PageRank [8], and provide guidelines in which cases which approach would be preferred, or to create hybrid approaches.

VIII. Threats to Validity

This study assumed that all the classes that existed in the forward designs were the important classes. There is a possibility that some of these classes were not important or not the key classes of the system. Also, there is a possibility that the forward design used is too 'old' or in other words obsolete. Feedbacks from the system developer may enhance the accuracy of these key classes from forward design. However, collecting the feedback requires more effort.

The input of this study is dependent on the reverse engineered class diagram constructed by the MagicDraw CASE tools. As mentioned by Osman [5], there are several weakness of CASE tools' reverse engineering features. This weakness may influence the accuracy of the class diagram metrics calculation. There is higher risk for large system that the CASE tool may leave out several information of some classes.

We only cover nine opensource case studies. Based on the amount of classes we can consider that the case studies represent of small to medium size project. The result may differ if we include large systems in our case studies. However, to get large open source systems that have forward design is really difficult.

IX. Conclusion

In this paper, we propose an approach for condensing reverse engineered class diagram by selecting the key classes in it. We study how well machine learning techniques perform in selecting the key classes in a class diagram by using supervised learning methods. The machine learning algorithms are trained on a set of open source projects. These projects contain a forward design class diagram which is used as a reference for validating the quality of the condensation. Given the unbalanced nature of the data, Area under ROC curve is recommended as a performance evaluator for these algorithms.

This paper evaluates the influential predictors in classifying key classes and also compares various machine learning classification algorithms on nine case studies derived from open source software projects, to identify candidate algorithms with the most accurate as well as robust behavior across predictor sets. We discovered Export Coupling Parameter, Dependency In and Number of Operation are the most influential predictors for predicting key classes in a class diagram. On these predictor sets, Random Forest and k-Nearest Neighbor provided the best results. For all listed case studies, the Random Forest method scores an AUC above 0.64 and the average AUCs for every prediction set is 0.74. These algorithms are able to produce a predictive score that can be used to rank important classes by relative importance. Based on this class-ranking information, a tool can be developed that provides views of reverse engineered class diagrams at different levels of abstraction. In this was, developers may generate multiple levels of class diagram abstractions, ranging from highly detailed class diagram (equal to source code) to abstract class diagram (satisfying architect's preference for high level views). In a broader perspective, this approach supports both the Bottom-Up and also the Top-Down approach for understanding of programs [32].

Indeed, based on the selected case studies, this approach could not produce 100% correct prediction of key classes in a class diagram. Hence, further research is needed to find complementary explanatory variables. We expect better results by taking the meaning of classes into account.

Finding the set of projects suitable for this study was a very time consuming task. This set can now be used by the scientific community as a benchmark for further studies.

Acknowledgment

This work is partly supported by Public Service Department of Malaysia. We would like to thank the CASE tools and software metrics providers for the licenses.

References

[1] A. Nugroho, M. R. V. Chaudron, "A Survey of the Practice of Design - Code Correspondence amongst Professional Software Engineers," Proc. of the 1st Intl. Symp. on Empirical Softw. Eng. and Measurement(ESEM 2007), Sep. 2007, pp.467-469.

[2] E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Softw., vol. 294, Jan. 1990, pp. 13-17.

[3] P. Tonella, M. Torchiano, B.D. Bois and T. Systä. “Empirical Studies in Reverse Engineering : State of the Art and Future Trends”, Empirical Softw. Eng.,vol. 12,Oct. 2007,pp.551-571.

[4] B. Bellay and H. Gall, “A Comparison of Four Reverse Engineering Tools,” Proc. of the 4th Working Conf. on Reverse Eng. (WCRE), Oct. 1997, pp. 2-11.

[5] H. Osman, M.R.V. Chaudron, “Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Model,” GSTF Journal on Computing, Vol. 2,Apr 2012, pp. 193-201.

[6] A.M. Fernández-Sáez, M.R.V. Chaudron, M. Genero, and I. Ramos, “Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?,” Proc. of the 17th Intl. Conf. on Evaluation and Assessment in Softw. Eng.(EASE ’13), 2013, pp. 60-71.

[7] A. Zaidman, S. Demeyer, “Automatic Identification of Key Classes in a Software System using Webmining Techniques ,” J. Softw. Maint. Evol.: Res. Pract., John Wiley & Sons, vol. 20, pp. 387-417.

[8] F. Perin, L. Renggli, and J. Ressia. “Ranking software artifacts.” 4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010), 2010.

[9] M. Hammad, M.L. Collard and J.I. Maletic, “Measuring Class Importance in the Context of Design Evolution,” IEEE 18th Intl. Conf. Prog. Comprehension (ICPC), July 2010, pp. 148-151.

[10] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, Andreas,“Mining Version Histories to Guide Software Changes,” Proc. of the 26th Intl. Conf. on Softw. Eng.(ICSE ’04), 2004, pp. 563-572.

[11] J. Singer, “NavTracks: Supporting Navigation in Software,” Proc. of the 21st IEEE Intl. Conf. on Softw. Maint.(ICSM‘05), 2005, pp. 325-334.

[12] T. Gyimothy, R. Ferenc and I. Siket, “Empirical Validation of Object-Oriented metrics on Open Source Software for Fault Prediction,” IEEE Trans. Softw. Eng., vol.31, Oct. 2005.

[13] S.R. Chidamber and C.F. Kemerer, “A metrics suite for object-oriented design,” IEEE Trans. on Softw. Eng., vol. 20, Jun 1994, pp. 476493.

[14] Z. Yuming and H. Leung, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults,” IEEE Trans. on Softw. Eng., vol.32, Oct. 2006, pp. 771-798.

[15] P.v.d. Putten and M. van Someren, “A Bias-Variance Analysis of a Real World Learning Problem: The CoIL Challenge 2000,” Kluwer Academic Publishers, Oct. 2004, vol. 57, pp. 177-195.

[16] I.H. Witten, E.Frank and M.A. Hall, “Data Mining: Practical Machine Learning Tools and Techniques (Third Edition),” Morgan Kaufmann, Jan. 2011

[17] Datasets, http://www.liacs.nl/~hosman/DataSets.rar

[18] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G.J. McLachlan, A.F.M. Ng, B. Liu, P.S. Yu, Z.-H. Zhou, M. Steinbach, D.J. Hand, and D. Steinberg, “Top 10 Algorithms in Data Mining,” Knowl. Inf. Syst., vol. 14, Dec. 2007, pp. 1-37.

[19] A. Field, Discovering Statistics Using SPSS Third edition, SAGE Pub., 2009

[20] M. Orr, “Introduction to radial basis function networks. Tech. report, Institute for Adaptive and Neural Computation,” Edinburgh Univ., 1996, http://www.anc.ed.ac.uk/#mjo/rbf.html.

[21] L. Breiman, “Random Forests,” Mach. Learn., Kluwer Academic Publishers, vol. 45, Oct. 2001, pp. 532.

[22] J.A. Hanley and B.J McNeil. The Meaning and use of Area Under a Receiver Operating Characteristic(ROC) Curve, Diagnostic Radiology, vol. 143, Apr. 1982

[23] A. Dallal and L.C. Briand, “A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes,”ACM Trans. Softw. Eng. Methodol.,vol.21,Mar.2012.

[24] M. Genero, M.Piattini and C. Calero, “A Survey of Metrics for UML Class Diagram, Journal of Object Technology,” vol. 4, Nov. 2005, pp. 59-92.

[25] A. Lake, and C.R. Cook, “Use of Factor Analysis to Develop OOP Software Complexity Metrics,” Tech. Report, Oregon State Univ., 1994.

[26] L. Briand, W. Melo and P. Devanbu, “An Investigation into Coupling Measures for C++,” Proc. of the 19th Intl. Conf. on Softw. Eng.(ICSE ’97), ACM, 1997, pp. 412-421.

[27] L.C. Briand, J. Wüst, S.V. Ikonomovski and H. Lounis, “Investigating Quality Factors in Object-oriented Designs: an Industrial Case Study,”Proc. of the 21st Intl. Conf. on Softw. Eng.(ICSE ’99), ACM, 1999, pp. 345-354.

[28] A. Egyed, “Automated Abstraction of Class Diagrams,” ACM Trans. on Softw. Eng. and Methodol., vol. 11, Oct. 2002, pp. 449- 491.

[29] H. Osman, D.R. Stikkolorum, A.v. Zadelhoff and M.R.V. Chaudron, “UML Simplification : What is in the developers mind?,” Proc. of the 2nd Workshop on Experiences and Empirical Studies in Softw. Modelling(EESSMOD ’12),ACM,2012.

[30] D. Steidl, B. Hummel and E. Juergens, “Using Network Analysis for Recommendation of Central Software Classes,” Proc. of the 19th Working Conf. on Reverse Eng. (WCRE), IEEE, Oct. 2012, pp. 93-102.

[31] H. Osman, A.v. Zadelhoff and M.R.V. Chaudron, “UML Class Diagram Simplification : A Survey for Improving Reverse Engineered Class Diagram Comprehension,” Proc. of the 1st Intl. Conf. on Model-Driven Eng. and Softw. Dev. (MODELSWARD 2013), Feb 2013, pp. 291-296.

[32] M.-A.D. Storey, F.D. Fracchia, and H.A. Mueller, “Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization,” Proc. of the 5th Intl. Workshop on Prog. Comprehension (WPC ’97). IEEE, 1997

[33] Waikato Environment for Knowledge Analysis (WEKA), http://www.cs.waikato.ac.nz/ml/weka/