Interactive scalable condensation of reverse engineered UML class diagrams for software comprehension

Osman, M.H.B.

Chapter 7

Condensing Reverse Engineered Class Diagram using Object-Oriented Design Metrics

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 chapter 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 (RE-CD for the same system). Based on this training set, the method learns to select the key classes for inclusion in the high-level class diagrams. In this chapter, we studied a set of nine classification algorithms from the machine learning community and evaluated which algorithms perform best for predicting the key classes in an application.

7.1 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

This chapter is adapted from a publication entitled **"An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams"**, In Proceedings of the International Conference on Software Maintenance (ICSM 2013)

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 [118]. 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 system's components and their relationships and create design representations of the system at a higher level of abstraction [41]. Reverse engineering also refers to methods aimed at recovering knowledge about a software system in support of execution some of software engineering tasks [170]. 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 [22]. However, current reverse engineering techniques do not yet solve this problem adequately. In particular, RE-CDs are typically a detailed representation of the underlying source code. This makes it difficult for software engineers understand what the key elements in the software structure are [125].

This study is partially motivated by a scenario when new programmers want to join the 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 recognize which classes in the system play important roles or can be considered as key classes in the system.

Fernández-Sáez et al. [59] 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 complex class diagrams or help to predict the severity of a defect in a software system.

This study specifically aims at providing suitable classification algorithms to decide which classes should be included in a high-level class diagram. We seek an automated approach to classifying the key classes in an application. We require algorithms that are 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 (in particular RE-CD).

In this chapter, we focus on the use of 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 of practical use to software developers. Also, we analyse the predictive power of the predictors to know how influential each of these predictors are, with respect to the performance of the classifier.

We explore several classification algorithms for predicting key classes that should be included in a class diagram. As the input for this study, we use sets of source codes from the 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 RE-CDs. 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 (k-NN), 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% were 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 substantial 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 chapter is structured as follows: Section 7.2 discusses related research and Section 7.3 describes the research questions. Section 7.4 explains the approach on how we conducted the evaluation. Section 7.5 presents the results and Section 7.6 discusses our findings. This is followed by conclusions and future work in Section 7.7.

7.2 Related Work

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

Zaidman and Demeyer [184] proposed a method for identifying key classes by using Hyperlink-Induced Topic Search (HITS) web mining technique. They used dynamic (runtime) analysis of source code as the input 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. [136] proposed ranking software artifacts using the PageRank algorithm. They used the Pharo Smalltalk system and Moose re-engineering 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. [77] 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 to the system. They found that 15% of the classes in the case studies were changed from six design changes.

Steidl et al. [154] presented an approach to retrieve important classes of a system by using network analysis on the dependency graphs. They performed an empirical study to find the best combination of centrality measurement and dependency graphs. Classes recommended by their test project developers were used as the baselines. They found that the centrality indices perform best when using the undirected dependency graph that include information about inheritance, parameter and return dependency.

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. The aforementioned works validate their approach to identify the key classes in class diagrams by comparing the prediction result with the information derived from software documentation, repository and developer(s) recommendation. In this study, we validate our approach by comparing selected classes against those actually found in the forward design.

7.3 Research Questions

This section describes the research questions of this study that will be answered in Section 7.5.

RQ1: Which design metrics are influential predictors in classifying key classes?
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 algorithms is influenced by partitioning the predictor-variables in different groups with different characteristics.

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

7.4 Approach

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

7.4.1 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.

Examined Predictors

We used a set of software metrics that are typically used to characterize design characteristic of classes in class diagrams as input to our classification algorithms. The SDMetrics [180] tool is capable of computing 32 types of class diagram metrics. These metrics are divided into five categories, namely Size, Coupling, Inheritance, Complexity and Diagram. We select 11 class diagram metrics from the Size and Coupling category. These categories of metrics were selected for the following reasons: i) our survey in Chapter 5 and 6 demonstrated that developers use Size and Coupling as predictors of key classes, ii) experts in the area of software metrics (Briand et al. [36] and Genero et al. [29]) stated that Coupling is an important structural dimension in object-oriented design, iii) the work in [188] and [73] 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 7.1.

Tools

The tools used in this study are the following:

- *Reverse Engineering Tool*: MagicDraw[9] 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 [178]. It contains tools for data pre-processing, classification, clustering, and visualization. WEKA version 3.6.6 was used for this study.

Table 7.1: *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 [40] and Number of Methods (NM) in [99].
NumPubOps	Size	The number of public operations in a class. Also known as Number of Public Methods (NPM) in [99].
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 as attributes type. This is a version of OAEC +AAEC in [34].
IC_Attr	Coupling (import)	The number of attributes in the class have another class or interface as their type. This is a version of OAIC+AAIC in [34].
EC_Par	Coupling (export)	The number of times the class is externally used as a parameter type. This is a version of OMEC+AMEC in [34].
IC_Par	Coupling (import)	The number of parameters in the class have another class or interface as their type. This is a version of OMIC+AMIC in [34].

7.4.2 Case Studies

We used the following criteria for selecting case studies:

- The software should be an open source software project that provides both the implementation source code and forward design class diagram.
- The number of classes in the implementation (source code) ≥ 50 classes.

Based on these criteria, nine open source software/systems were selected. In these projects, we selected a forward UML class diagram from the documentation and then selected a matching version of the source code. The number of classes in these case studies ranges from 59 to 903 (see Table 7.2). The ratio between the number of classes included in the UML class diagram and the number of classes in the implementation

Table 7.2: *List of Case Study*

No.	Project	Total Classes in Source Code (S)	Total Classes in Design (D)	D:S ratio as %
1	ArgoUML	903	44	4.9
2	Mars	840	29	3.5
3	JavaClient	214	57	26.6
4	JGAP	171	18	10.5
5	Neuroph 2.3	161	24	14.9
6	JPMC	121	24	19.8
7	Wro4J	87	11	12.6
8	xUML	84	37	44.1
9	Maze	59	28	47.5

(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 [6] for future research and for validation of this study. Detailed information about these case studies can be found in Section 3.3.

7.4.3 Process

This subsection describes the steps performed for this study. The inputs for this process are forward designs and RE-CDs (constructed from the 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 7.1.

Data Preparation

For data preparation, class diagram metrics were extracted from RE-CD (obtained through reverse engineering process). 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 7.3. In this table, Step 1 to 3, performed the extraction of all required data. Then, the data is cleaned up by removing the external library and runtime classes since we only focused on the application and domain related classes (as suggested in Chapter 5). Step 5 and 6 perform the merging of data and Step 7 select the software metrics that are useful for prediction.

We expect that there is some noise in the predictors. For instance, the getters and

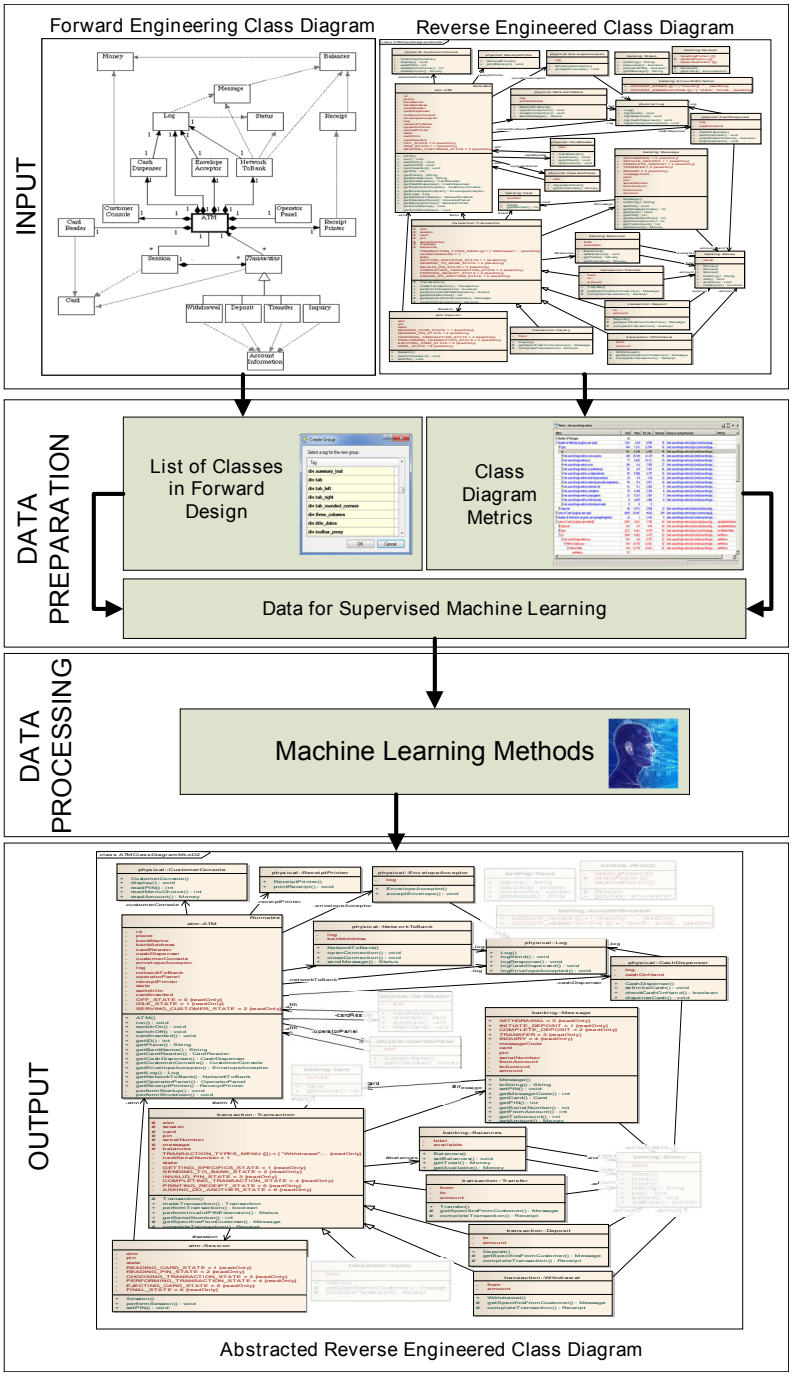


Figure 7.1: *Design Abstraction Process*

Table 7.3: *Data Preparation Steps*

No.	Preparation Step	Description
1.	List all the classes that appear in the UML design-class diagram	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 of the RE-CD 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

setters predictor completely relies on the conformance of source code to naming conventions (e.g. 'get','has'). Not all case studies have this kind of naming convention. In RQ2, we desired to explore the performance of the classification algorithms across different group of predictors (based on predictors' characteristics). Therefore, we experimented with different groups 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: a set of predictors that only uses Coupling metrics (predictor set C). The details of all predictor sets are shown in Table 7.4.

Data Processing

For every run of the classification algorithm, we randomly split the dataset (for every case study) into 50% for the training set and the other 50% for the test set. To further

Table 7.4: *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

improve reliability, we ran each experiment 10 times using different randomization. The main reason for doing this is that the data are typically imbalanced 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 (refer [62] for more detail discussion). For example, let say we have an imbalanced dataset with 900 examples with only 1% (9) of the examples are positive. If we used 10-fold cross validation, there is a possibility of the positive example is not included in the test set. Thus, the True Positive Rate (TPR) calculation is not reliable in this situation. We avoided detailed fine-tuning because we assumed our end users have no knowledge of data mining. Algorithms ran with default WEKA configuration. We used WEKA as the tool and algorithms ran with WEKA default parameter setting; Except for k-Nearest Neighbor, for which we used two different neighborhood size settings (1 and 5 neighbors). A different number of k in k-NN may present a substantial difference in classification performance. Therefore, this experiment investigates two sets of k-NN: (a) k=1 (extreme lowest value of k, or plain nearest neighbor); and (b) k=5 (which we believe it represents a more average k value for the dataset).

Evaluation

In this study, the analyses are conducted using two evaluation measures: i) the univariate analysis, and ii) the analysis of classification performance. These measures are explained as follows:

Univariate Analysis: To measure the predictive power of the predictors, we use the information gain with respect to the class [76]. Univariate predictive power means measuring how influential a single predictor is in predicting performance. The results of this algorithm are normally used to select the most suitable predictor. Nevertheless, in our study, we did not use it for predictor selection, but for an exploratory analysis of the usefulness of various predictors (in this case: class diagram metrics).

For Univariate analysis, the predictors were evaluated by using the Information Gain (InfoGain) Attribute Evaluator in WEKA. 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).

Analysis of classification performance: As discussed in Chapter 2, classification performance is analyzed by using the Area Under ROC Curves (AUC). The evaluation of machine learning classification algorithms started with generating a confusion matrix (as shown in Table 2.3), based on applying a classification algorithm using WEKA.

WEKA provides AUC calculations 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.

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

7.5 Evaluation of Results

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

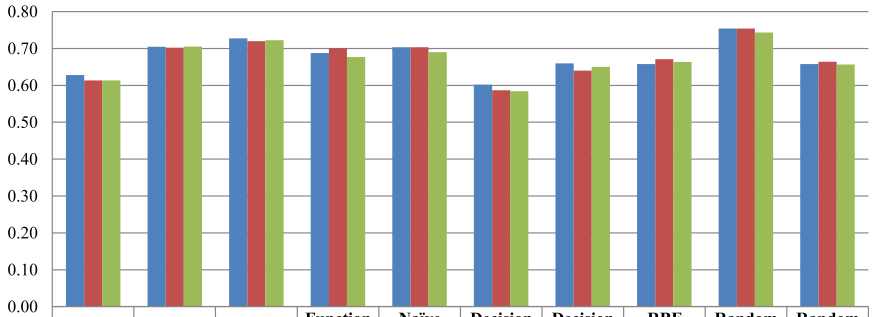
7.5.1 Predictor Evaluation

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. Table 7.5 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 hand, ArgoUML and JPMC have only one influential class diagram metric. Table 7.5 also shows that the Coupling category metrics are influential for every case study. In all cases, at least one of the Coupling metrics is

Table 7.5: *Univariate Predictor Performance (Information Gain)*

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
No. of InfoGain > 0	**3**	**5**	**4**	**3**	**3**	**4**	**5**	**3**	**3**	**6**	**2**
Average	**0.037**	**0.050**	**0.045**	**0.044**	**0.025**	**0.058**	**0.067**	**0.036**	**0.036**	**0.068**	**0.081**



	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**	**0.69**	**0.58**	**0.65**	**0.66**	**0.74**	**0.66**

Figure 7.2: *Average AUC Score for Every Dataset.*

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 found that only five case studies listed 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 (based on the predictor sets defined in Table 7.4). Figure 7.2 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 7.2 the difference in average AUC is only ± 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

Table 7.6: *Results for Predictor set C.*

No.	Project	J48	k-NN(1)	k-NN (5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
1	ArgoUML	0.50	0.69	0.69	0.54	0.56	0.50	0.55	0.50	0.64	0.60
		0.00	0.04	0.05	0.05	0.06	0.00	0.07	0.07	0.04	0.03
2	Mars	0.53	0.69	0.75	0.61	0.62	0.52	0.70	0.58	0.73	0.61
		0.06	0.03	0.05	0.05	0.13	0.05	0.07	0.16	0.09	0.08
3	JavaClient	0.76	0.83	0.86	0.81	0.79	0.78	0.75	0.80	0.86	0.81
		0.09	0.03	0.04	0.05	0.04	0.07	0.06	0.04	0.04	0.05
4	JGAP	0.54	0.60	0.62	0.67	0.66	0.51	0.59	0.65	0.72	0.60
		0.07	0.05	0.07	0.12	0.09	0.02	0.04	0.10	0.10	0.17
5	Neuroph	0.61	0.79	0.82	0.71	0.87	0.56	0.63	0.72	0.78	0.68
		0.14	0.06	0.06	0.10	0.04	0.09	0.08	0.16	0.09	0.08
6	JPMC	0.54	0.66	0.67	0.69	0.57	0.50	0.58	0.61	0.69	0.59
		0.08	0.06	0.06	0.08	0.08	0.01	0.03	0.06	0.09	0.08
7	Wro4j	0.63	0.70	0.68	0.77	0.77	0.62	0.70	0.69	0.74	0.68
		0.18	0.09	0.19	0.16	0.15	0.12	0.13	0.21	0.14	0.14
8	xUML	0.74	0.78	0.77	0.69	0.73	0.69	0.72	0.82	0.83	0.75
		0.06	0.07	0.06	0.12	0.06	0.10	0.06	0.04	0.06	0.06
9	Maze	0.67	0.61	0.64	0.60	0.68	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
	No. of InfoGain ≥ 0.60	**5**	**8**	**9**	**7**	**7**	**3**	**6**	**6**	**9**	**5**
	Average	**0.61**	**0.71**	**0.72**	**0.68**	**0.69**	**0.58**	**0.65**	**0.66**	**0.74**	**0.66**
		0.09	**0.08**	**0.08**	**0.08**	**0.10**	**0.10**	**0.07**	**0.10**	**0.07**	**0.08**

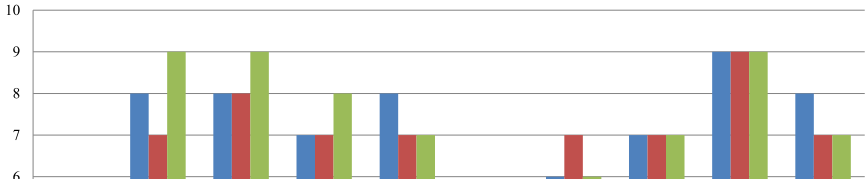
Note : The first row for each predictor set is the average AUC, the second row lists the standard deviation. Cells with AUC < 0.60 are highlighted.

and B, the set of predictors is still reliable for prediction purposes. This shows that the Coupling category (Predictor Set C) strongly influences the prediction performance.

7.5.2 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 7.6 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. For our study, we consider 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 performing the experiments, we found that the Random Forests and K-Nearest Neighbor (k-NN(5)) algorithms perform the best in classifying the key classes in class diagrams in terms of overall AUC, as well as robustness over various predictor sets. Figure 7.3 shows the prediction performance of all selected classification algorithm. This figure illustrates the number of case studies (for each predictor set) in which the



	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	7	3	6	7	9	7

Figure 7.3: *AUC Score >= 0.60*

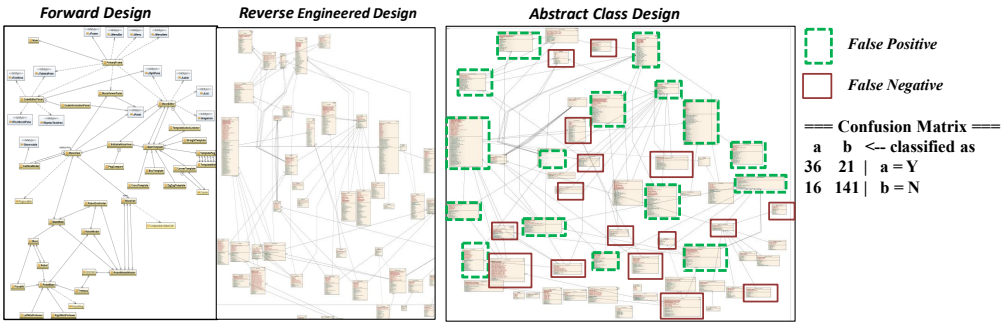


Figure 7.4: *Application of Random Forests Classification Algorithm.*

classification algorithm produces an AUC score greater than 0.60. Random Forests and k-NN(5) perform the best prediction where both classification algorithms produced AUC scores above 0.60 for at least 8 case studies of all datasets. Meanwhile, Naive Bayes, Random Tree, Function Logistic, RBF Network and Decision Stump performed less robust prediction across all predictor set. These classification algorithms performed reasonably well. They produced an AUC above the threshold for 6 to 8 case studies. J48 and Decision Table appear not to be suitable to be used in these case studies, given the low number of results with AUC $\geq$ 0.60 (between 3 to 5). The average AUC score of more than 0.72 for Random Forests and k-Nearest Neighbor (k-NN(5)) shows their suitability for all predictor set. Figure 7.4 illustrates the application of our method. In particular, it applies the Random Forests classification algorithm to the JavaClient case

study. As a 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 diagram. 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).

7.6 Discussion

With this result, we can conclude that the class diagram metrics from the Coupling and Size category can be good predictors for classifying 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 finding is consistent with the findings in Chapter 5 and Chapter 6 where the Number of Operation and Relationship (related to coupling) are the elements that are most software developers looked at in order to find the important classes in a class diagram.

The results show that k-NN(5) and Random Forests perform best and 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 the algorithm k-NN(5) are predicted False Negative in the Random Forests and vice versa. We compared all the result manually from those two algorithms applied to several case studies and some of the true and false results are different. The possibility to enhance this predictive power is by combining those classification algorithms to achieve the best result. Given the imbalanced data, all selected algorithms were not able to produce high AUC scores.

This study was aimed at discovering suitable classification algorithms that 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 [52].

7.6.1 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 compared to the version of the source code

used. Feedback from the system developer may enhance the accuracy of these key classes from forward design. However, collecting such feedback requires more effort.

The input of this study is dependent on the RE-CD constructed by the MagicDraw CASE tools. As mentioned in Chapter 4, there are several weaknesses of CASE tools' reverse engineering features. This weakness may influence the accuracy of the class diagram metrics calculation. There is a higher risk for large system that the CASE tool may leave out several information of some classes.

We only cover nine open source case studies. Based on the amount of classes, we can consider that the case studies represent small to medium size projects. The result may differ if we include large systems in our case studies.

7.7 Conclusion and Future Work

In this study, we proposed an approach for condensing RE-CD by selecting the key classes in it. We studied how well machine learning techniques perform in selecting the key classes in a class diagram by using supervised learning methods. The machine learning algorithms were trained on a set of open source projects. These projects contain a forward design class diagram which was used as a reference ('ground truth') for validating the quality of the condensation. Given the imbalanced nature of the data, Area under ROC curve was used as a performance evaluator for these algorithms.

This study evaluated (1) the influential predictors in classifying key classes and, (2) compared 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 that the Export Coupling Parameter, Dependency In and Number of Operation are the most influential predictors for classiying key classes in a class diagram. On these predictor sets, Random Forests and k-Nearest Neighbor provided the best results. For all listed case studies, the Random Forests 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 RE-CDs 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 [157].

The results of this research may be improved by finding complementary explanatory variable. We also expect better results by taking the meaning of classes into account (see Chapter 8). 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.

7.7.1 Future Work

For future work, there is 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 types 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 the identification of 'features' as a unit of inclusion or exclusion in the UML class diagrams. 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 the 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 [77]. It is also interesting to compare our approach with other works that apply different algorithms such as HITS web mining (used in [184]), network analysis on dependency graphs (used in [154]) and PageRank [136], and provide guidelines in which cases that approach would be preferred, or to create hybrid approaches.

We extend this research by validating the result of our proposed technique for condensing UML class diagrams in Chapter 10. The result of this study also allows us to create an automated tool to condense class diagrams. The automated tool is presented in Chapter 9 .