PREDICTION OF CODE REFACTORING USING CLASS AND FILE LEVEL SOFTWARE METRICS

by

Yasemin Köşker

B.S., Computer Engineering, Boğaziçi University, 2005

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2009

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Assistant Professor Ayşe Başar Bener for her patience, support and encouragement during this research. This thesis would not have been completed without her invaluable guidance and help.

I would also like to thank Professor Emin Anarım and Professor Fikret Gürgen for being in my thesis committee.

I would like to thank Software Engineering Research Laboratory members, especially Burak Turhan and Ayşe Tosun, for helping me in this research and sharing ideas with me.

My boyfriend Burak deserves special thanks for his unlimited patience and support. He encouraged me in all stages of my research.

Finally, I would like to thank Turkcell İletişim Hizmetleri A.Ş. for their contributions to my research. This research is supported in part by Boğaziçi University Research Fund under grant number 06HA104.

ABSTRACT

PREDICTION OF CODE REFACTORING USING CLASS AND FILE LEVEL SOFTWARE METRICS

Maintenance costs as a proportion of software development cost could be very high, especially in multi version real-time systems such as in telecommunications domain due to tight coupling of modules. The readability of the code becomes very hard with the development of complex classes, so the maintenance and the adaptation of the new developers to the project becomes a diffucult job. One way to overcome this problem is to predict what parts of the system are difficult to maintain and likely to change. Refactoring decisions are taken through a costly manual inspection of the code based on developer experience. It makes the system dependent to people rather than processes. Also, manual inspection increases the cost of the project. The managers are generally interested in projects which are completed on time and within budget rather than code quality. However, it would be preferable to make the project less costly and finish it before deadline by also preserving or enhancing the code quality and structure.

In this research we aim to detect the modules that need to be refactored by analyzing the code complexity of the projects with version history. We propose a machine learning based model that prioritizes attributes to predict modules to be refactored. Our prediction results revealed that assigning weights to certain attributes considerably improves the prediction performance of the model as high as 71% of probability of detection and as low as 18% of false alarm rates on the average in class-level. Further our proposed model provides on average as high as 81% efficiency in maintenance effort, over and above the manual code inspection.

ÖZET

SINIF VE DOSYALARIN YAZILIM ÖLÇÜTLERİ KULLANILARAK TEKRAR TASARIM DURUMLARININ İRDELENMESİ

Yazılım geliştirme maliyetindeki payı açısından bakım maliyetleri, özellikle de modüllerin sıkı eşleştirmesine bağlı olan telekomünikasyon alanındaki gibi çoklu versiyonlu gerçek-zamanlı sistemlerde çok yüksek olabilir. Karmaşık sınıfların geliştirilmesi ile birlikte kodun okunabilirliği çok zor bir hale gelebilir, bu yüzden yeni yazılım geliştiricilerin projeye katılma süreci zorlaşır ve projenin bakımı zor bir iş haline gelir. Bu problemin üstesinden gelmenin bir yolu; sistemin hangi kısımlarının bakımının zor olduğunu ya da hangi kısımların değişime eğilimli olduğunu tahmin etmektir. Tekrar tasarım kararları; yazılım geliştiricilerin deneyimini temel alan maliyeti yüksek, manuel kod incelenmesine dayalı olarak alınır. Bu durum sistemi süreçlerden çok insanlara bağımlı kılar. Aynı zamanda manuel inceleme, proje maliyetlerini de yükseltecektir. Yöneticiler genel olarak kodun kalitesinden çok projenin zamanında ve bütçe sınırları dahilinde tamamlanmasıyla ilgilenirler. Fakat, projeyi düsük maliyetli ve son teslim zamanından önce bitirmekle birlikte aynı zamanda kodun kalitesinin ve yapısının korunması ve hatta geliştirilmesi, daha tercih edilen bir durum olacaktır.

Bu araştırmada projeleri, versiyon geçmişiyle kod karmaşıklığını analiz ederek tekrar tasarlanması gereken sınıfları belirlemeyi hedeflemekteyiz. Biz, tekrar tasarlanması gereken sınıfları özelliklerine göre önceliklendiren makina öğrenme temelli bir model öneriyoruz. Bizim öngörü sonuçlarımız gösteriyor ki, belirli özelliklere ağırlıklar vermek: sınıf bazlı ortalamada %71 doğru tahmin ve %18 yanlış alarm oranlarında, performans modeli öngörüsünü oldukça geliştirmektedir. Ayrıca önerdiğimiz model, bakıma dayalı çalışmalarda; manuel kod incelemesine göre, ortalama olarak fazladan %81 verimlilik üstünlüğü sağlamaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
TABLE OF CONTENTS vi
LIST OF FIGURES viii
LIST OF TABLES ix
LIST OF SYMBOLS/ABBREVIATIONS xiii
1. INTRODUCTION 1
1.1. Motivation 1
1.2. Outline 3
2. BACKGROUND 4
2.1. AI in Software Engineering 4
2.2. Software Metrics 5
2.3. Code Refactoring 7
3. PROBLEM STATEMENT 10
3.1. Research Questions 11
4. PROPOSED MODEL 12
4.1. Inputs to the Model 13
4.2. Outputs of the Model 14
4.3. The Details of the Model 14
4.4. Assessing the Performance of the Model 22
5. DATA COLLECTION 24
5.1. Requirements for the Data 24
5.2. Data Extraction 25
5.3. An Overview of the Projects 29
6. EXPERIMENTS AND RESULTS 30
6.1. Experimental Design 30
6.2. Experiments 30
6.2.1. Experiment with McCabe's complexity metrics 31
6.2.2. Experiment with LOC metric 31

6.2.3. Experiment with Halstead metrics 32
6.2.4. Experiment with selected group of Halstead's metrics 32
6.2.5. Experiment without code churn metrics 32
6.2.6. Experiment with code churn metrics 32
6.3. Results 33
6.3.1. Experiments and Results with McCabe Metrics in class-level 33
6.3.2. Experiments and Results with LOC Metric in class-level 33
6.3.4. Experiments and Results with a selected subset of Halstead Metrics in class-level 33
6.3.5. Experiments and Results in file-level without code churn metrics in file-level 37
6.3.6. Experiments and Results in file-level with code churn metrics in file-level 37
6.3.7. Experiments and Results with McCabe Metrics in file-level 37
6.3.8. Experiments and Results with LOC Metric in file-level 39
6.3.9. Experiments and Results with Halstead Metrics in file-level 39
6.3.10. Experiments and Results with a selected subset of Halstead Metrics in file-level 39
6.4. Cost / Benefit Analysis 46
6.5. Discussion of the Results 48
6.6. Threats to Validity 50
7. CONCLUSION 52
7.1. Contribution 52
7.2. Future Work 53
8. APPENDIX A: SOFTWARE METRICS USED IN THIS THESIS 55
9. APPENDIX B: OBJECT-ORIENTED DESIGN METRICS 57
10. APPENDIX C: SIGNIFICANCE T-TESTS OF THE MODEL. 58
REFERENCES 60

LIST OF FIGURES

Figure 4. 1. The Pseudo Code for Constructing Data Set 19

Figure 4. 2. The Pseudo Code of the Model. 20

Figure 4. 3. The Schematic Description of the Model 21

Figure 4. 4. A typical ROC curve [44] 23

Figure 6. 1. Calculation of the Gained Effort 46

LIST OF TABLES

Table 4. 1. Metrics Collected From the projects in class-level 16

Table 4. 2. Metrics Collected From the projects in file-level 16

Table 4. 3. Confusion Matrix 23

Table 5. 1. Attribute and Class information of the GSM5 project [42] 25

Table 5. 2. Attribute and Class information of the GSM7 project [42] 25

Table 5. 3. Attribute and Class information of the GSM4 project [42] 26

Table 5. 4. Attribute and Class information of the GSM8 project [42] 26

Table 5. 5. Attribute and File information of the GSM5 project [42] 27

Table 5. 6. Attribute and File information of the GSM7 project [42] 27

Table 5. 7. Attribute and File information of the GSM4 project [42] 28

Table 5. 8. Attribute and File information of the GSM8 project [42] 28

Table 6. 1. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 1 34

Table 6. 2. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 2 35

Table 6. 3. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 3 35

Table 6. 4. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 4 36

Table 6. 5. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 5 (without code churn metrics) 37

Table 6. 6. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 6 (with code churn metrics) 38

Table 6. 7. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 1 38

Table 6. 8. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 2 39

Table 6. 9. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 3 40

Table 6. 10. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 4 40

Table 6. 11. All results for GSM5 project in class-level 41

Table 6. 12. All results for GSM7 project in class-level 42

Table 6. 13. All results for GSM4 project in class-level 42

Table 6. 14. All results for GSM8 project in class-level 43

Table 6. 15. All results for GSM5 project in file-level 43

Table 6. 16. All results for GSM7 project in file-level 44

Table 6. 17. All results for GSM4 project in file-level 44

Table 6. 18. All results for GSM8 project in file-level 45

Table 6. 19. Average results for GSM5, GSM7, GSM4 and GSM8 projects in class-level 45

Table 6. 20. Average results for GSM5, GSM7, GSM4 and GSM8 projects in file-level 45

Table 6. 21. Cost-benefit analysis of GSM projects based on assumption 1 in class-level (with McCabe Complexity Metrics) 47

Table 6. 22. Cost-benefit analysis of GSM projects based on assumption 2 in class-level (with LOC Metrics) 47

Table 6. 23. Cost-benefit analysis of GSM projects based on assumption 3 in class-level (with Halstead Metrics) 47

Table 6. 24. Cost-benefit analysis of GSM projects based on assumption 4 in class-level (with subsetted Halstead Metrics) 48

Table A. 1. Base Metrics 55

Table A. 2. Composite Metrics 56

Table B. 1. Object-Oriented Design Metrics 57

Table C. 1. T-tests for McCabe vs. LOC. 58

Table C. 2. T-tests for McCabe vs. Halstead 58

Table C. 3. T-tests for McCabe vs. subsetted Halstead 59

Table C. 4. T-tests for Class-Level vs. File-Level 59

Table C. 5. T-tests for with vs. without churn metrics 59

LIST OF SYMBOLS/ABBREVIATIONS

FN	False negative
FP	False positive
TN	True negative
TP	True positive
Pd	Probability of detection
Pf	Probability of false alarm
σ	Standard deviation
μ	Mean
CVS	Code Versioning System
LOC	Lines of Code
NB	Naïve Bayes
SCA	Static code attribute
WNB	Weighted Naïve Bayes
ROC	Receiver Operator Curve
IDE	Integrated Development Environment
AI	Artificial Intelligence
ATM	Automated Teller Machine

1. INTRODUCTION

1.1. Motivation

With the recent technological developments, software products became an indispensable part of our lives. Wherever we put our eyes on we can see these software in charge; for example the mobile devices we use, telecommunication systems, embedded systems such as white goods, ATM's, aeroplanes etc [1]. While the complexity, vitality of software development and the number of developers continue to increase, the methods to produce these software with less number of defects, on time and within the limits of our budget began to gain more and more importance.

Software product life cycle is a complex task to manage. The development costs are only the tip of the iceberg. Nearly 90% of the cost is maintenance due to error correction, adaptation and mainly enhancements. As Belady and Lehman [2] state that software will become increasingly unstructured as it is changed. Since there is not enough time to design the projects, the reusability of the modules, classes and even methods is very hard. When the new requirements, which are gathered from the end users, are implemented or some part of the code is re-implemented in order to correct the defects, the design of the software usually changes. The changes are generally done without carefully analyzing the code segments. So, the complexity of the code increases and the readability of the code become very hard. When the size of the project is small or it is developed by one or two developers it is not a big deal. However, it is very important in software projects to make it process dependent rather than people dependent. When new developers are included to the project, it is not easy for them to add new functionality or change something in the project. So, experienced developers are canalized to the complex parts of the system rather than junior ones, thus increase the cost of the project.

Refactoring approach is generally used in the software development and maintenance phases by developers in order to decrease development time. Also, it is used to improve the quality of the software product (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency) [3]. Refactoring is an approach which reduces the software

complexity by incrementally improving internal software quality. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [4]. At its heart there is a series of small behavior preserving transformations [4]. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it is less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. Refactoring neither fixes bugs nor adds new functionality, though it might precede either activity [5, 6]. Rather it improves the understandability of the code and changes its internal structure and design, and it removes dead code to make it easier to comprehend and more maintainable. Refactoring is usually motivated by the difficulty of adding new functionality to a program or fixing a bug in it. However, manual inspection of the code by experienced staff is required in order to decide which parts of the code to be refactored. Also, it is not easy or practical for developers to refactor the code in a software project without considering the cost and deadline of the project [6].

The question that needs to be answered is then how to find the parts which need to be refactored without manual inspection effort. This brings us to decision making under uncertainty. Basic models, algorithms and techniques for decision making under uncertainty in AI are commonly used in other disciplines for problem solving such as image processing, multi agent systems, game programming, cost and defect prediction and so on [1, 7].

The organizations may have different types of projects, different development and testing cycles, different domains and also different processes. However, with the usage of past project data or data gathered from different domains, prediction models can be designed. For example, with defect prediction models which parts are more defect-prone or defective can be predicted, so that the testing effort and cost be eliminated.

In this research our aim is to predict the code segments which are in need of refactoring without manual inspection effort. Therefore, our motivation is to model an oracle that would successfully predict code refactorings to solve current design problems of the software projects. We have selected a learning-based methodology for our

refactoring prediction model to predict modules which are in need of refactoring in software as much as possible by learning from past projects' data.

1.2. Outline

Concise information about AI in software engineering, its relation with software metrics is presented in the second chapter. Also, background of code refactoring and its relation with software version history in software systems are briefly described.

The third chapter includes the presentation of the problem statement and the research question.

In Chapter 4 the proposed model is explained. The proposed algorithm, the details of the proposed model; input and output of the model and the performance evaluation criteria are also argued in this section.

Chapter 5 explains the experimental setup and contains information about the data used in the research. The information of the data and the projects and their version history are discussed.

In Chapter 6, the experiment that is applied to the available data is explained. We give the results, analysis of the results and evaluate the threats to validity of our model in this chapter.

Chapter 7 contains the summary of the research and the results as well as contributions and future directions.

2. BACKGROUND

First two sections will demonstrate software metrics and prediction models used in the literature. The last section will focus on previous research on refactoring.

2.1. AI in Software Engineering

Data in the real world are rarely reliable; that is they are rarely complete, consistent, without error, and unmasked by extraneous data [8]. The methods and tools of software engineering are suited to problems that can be characterized as having reliable, static data for which the problem solution space is small. Problems with unreliable and dynamic data usually imply a much larger solution space, so that AI techniques have been developed to deal with this situation.

In software domain, effective predictors for software products have been generated using data mining methods including linear regression, discriminant analysis, decision trees, neural networks, Naive Bayes and so on [9, 10, 11, 12, 13]. The most promising work yielded by AI, with respect to practical applicability, are expert systems. These systems attempt to exhibit expert performance in difficult problem domains such as testing, planning and design [8].

In recent studies, prediction models such as cost, effort or defect, are included in some phases of the software development lifecycle and their effect as a tool in the processes are discussed [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. For a given set of requirements it is desirable to know how much it will cost to develop specific software which satisfies given requirements [21]. Thus, software practitioners recognized the importance of realistic estimates of effort and cost for the successful management of software projects. Also, with the help of defect prediction models the amount of time required for thorough testing is reduced by guiding the developers through defect-prone parts in the software. By using defect predictors, managers would reduce testing resources and preserve the quality of their software at the same time [19].

2.2. Software Metrics

Measurement provides a way for developers to assess the status of the program to determine if it has any weaknesses or in need of correction and process improvement [22]. A good measurement in the software project facilitates in early detection of problems and provides quantitative clarification of critical development issues. Thus, it must be integrated into the total software life cycle.

As in any machine learning problem, software prediction models require a set of features to characterize the problem and to give estimation on the system. These attributes are referred to as software metrics. A software metric can be defined as “a quantitative measure of the degree to which a system, component, or process possesses a given attribute” [23]. Software metrics are used to give programmers feedback about their program. Metrics are the attributes that represent software; they are the raw data for software domain. An effective management of any software development process requires monitoring and analysis of software metrics. Metrics can be used as guidelines for when to refactor [24]. They provide a way to measure the progress of code during development.

Static code attributes can be examined in three categories, namely quantitative metrics, Halstead metrics and McCabe metrics. Each category of metrics use a different perspective to estimate the complexity of the code.

The increased complexity of software applications also increases the difficulty of making the code reliable and maintainable. Also, the understandability and reusability decreases when the code becomes complex. Complex code segments are generally the indicator of bad smells, all the more worse is defects.

The first category of these metrics is simple quantitative attributes such as lines of code, blank lines of code and comment lines of code. Quantitative metrics are the simplest measures that can be extracted from source code.

McCabe metrics were introduced in 1976 by Thomas J. McCabe. McCabe designed cyclomatic complexity to indicate a program's testability and understandability [25]. It can

be used to indicate the test effort. McCabe metrics are listed as Cyclomatic Density, Decision Density, Essential Density, Cyclomatic Complexity, Essential Complexity.

Halstead Metrics were developed by Maurice Halstead in 1977. Halstead argued that a code that is harder to read would more likely to be defective [26]. Halstead metrics are based on operator and operand counts such as the total number of operator occurrences, number of unique operands and number of unique operators. Beside basic metrics (total operands, total operators, unique number of operands, unique number of operators) ,Halstead Program Difficulty, Halstead Program Length, Halstead Program Level, Halstead Programming Effort, Halstead Programming Time, Halstead Program Volume are the derived metrics from the basic metrics.

Apart from static code attributes, there are studies which collect Chidamber and Kemerer's object-oriented design metrics, after the object-oriented software development paradigm became popular [27]. The idea is again to capture the complexity level of software by inspecting within module and between module relations. Naturally, these metrics only apply to software developed with an object oriented language. Object-oriented design metrics are WMC (Weighted Methods per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children of a Class), CBO (Coupling between Object Classes), RFC (Response for a Class) and LCOM (Lack of Cohesion in Methods).

Code churn metrics are helpful for assessing the risk factor by measuring the degree of change in source code during the implementation phase [28]. Large scale software development typically involves the use of versioning systems such as Code Versioning System (CVS). In versioning systems, developers share a common repository of source code, so that they can work on different parts or on the same parts of the code simultaneously. Considering the difficulty of communication and coordination issues, an increased risk factor is introduced with the number of different developers working on the same code. Code churn metrics can be extracted with the help of versioning systems and they can be listed as: added / deleted / modified lines of code, status of the code (i.e. new, changed, unchanged), number of changes made on the code (i.e. commit count), number of distinct developers who worked on the code and whether a developer is working on that code for the first time.

Software metrics are used as an input for prediction models and they give feedback about the behavior of the software. However, according to the problem domain, the effect and contribution of an individual metric or a group of metrics provide different information. We also analyze the effect of different group of metrics in further sections in order to see the effects of them in our refactoring domain.

In our research, static code attributes are used with code churn metrics. Appendix A and Appendix B presents a complete list of the metrics under consideration, and how they are calculated.

2.3. Code Refactoring

Refactoring is an approach to improve the design of a software without changing its external behaviour. This means it always gives the same output with the same input after the change is applied [4]. In the context of software evolution, refactoring is used to improve the quality of the software product (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency) [3]. Refactoring neither fixes bugs nor adds new functionality, though it might precede either activity [5, 6]. Rather it improves the understandability of the code and changes its internal structure and design, and it removes dead code to make it easier to comprehend, to maintain and to change. Refactoring is usually motivated by the difficulty of adding new functionality to a program or fixing a bug in it.

When a system's source code is easily understandable, the system is much easier to maintain, leading to reduced costs and allowing precious development resources to be used elsewhere. At the same time, if the code is well structured, new requirements can be introduced more efficiently and with less problems. These two development tasks, maintenance and enhancement, often conflict since new features, especially those that do not fit smoothly within the original design, result in an increased maintenance effort. The refactoring process aims to reduce this conflict, by aiding non destructive changes to the structure of the source code, in order to increase code clarity and to enhance maintenance. .

However many developers and managers are hesitant to use refactoring. The most obvious reasons for this is the amount of effort required for even a minor change, and a fear of introducing bugs [3, 5]. Both of these problems can be solved by using an automated refactoring tool. An automated process support also decrease the time spent on maintenance and improve the effectiveness of maintenance stage.

Several researchers have explored the effects of software metrics, code complexity and software design as well as software maintenance on the decision of code refactoring. Welker and Oman [24] suggested measuring software's maintainability using a Maintainability Index (MI) which is a combination of multiple metrics, including Halstead metrics, McCabe's Cyclomatic complexity, lines of code, and number of comments. Hayes et al. [29] introduced and validated that the RDC ratio (the sum of requirement and design effort divided by code effort) is a good predictor for maintainability. Fowler et. al. [4] suggested using a set of bad smells such as long method to decide when and where to apply refactoring. Mens, Tourwé and Muñoz [5] implement a tool to detect bad smells and validate this tool by carrying out three experiments. In order to find out which parts of the source code need to be refactored, they suggest relying on the notion of bad smells. However, only selected bad smells are examined. Also, detection of some bad smells can be quite computation intensive and they restrict the number of subclasses examined. Zhao and Hayes [30] introduced a cost-benefit analysis to prioritize the classes that are identified with bad smells.

Stroggylos and Spinellis analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by this process, in order to evaluate whether refactoring is effectively used as a means to improve software quality within the open source community [31]. Weißgerber and Diehl [32] search for changes in the program code that correspond to refactorings. To evaluate how many of the actually performed refactorings their technique can detect, they manually inspected the log messages of two open-source projects, JEDIT and TOMCAT. To obtain refactorings that have been performed in a transaction they compare the syntactical entities of the versions in the transaction with the entities in the predecessors of these versions. However, they focus on some predetermined refactoring types and they did not cover others such as extract superclass or extract subclass. They also

indicated that to improve their technique they made experiments with other code-clone detection methods [32].

Our approach differs from the above approaches since we define the problem slightly differently and hence we propose a novel model to solve this problem.

3. PROBLEM STATEMENT

We can define the problem in a broad sense as a decision making under uncertainty: code quality, and maintenance. During the development of software projects, project managers have to complete the project on time and within budget. When new requirements are added or new developers are included to the team, extra time is needed. New requirements should be added to the system by reusing the methods, classes in the project to make the software more modular and reusable. However, in general companies do not allocate enough time to design the new component that needs to be added. The design function is not done by experienced architects, leading to very little or no attention is given to analyze and design the new requirements. So, adding the new requests and adapting them to the existing system usually requires more time. Another reason is that, it is very hard for developers especially new ones to understand large projects if the complexity is high. Since maintenance is a difficult task, managers assign their experienced staff to ensure product quality. As a result maintenance becomes one of the most expensive life cycle activities. Any change made in the design of the project will affect the design of new functionalities and the existing ones, thus the whole project development cycle. Unskilled staff is not trusted enough to make this kind of architectural changes, however it is costly to prefer experienced ones. So, it would also turn into a resource allocation problem. Therefore, effective strategies are needed to solve these kinds of problems. Manual inspection of the code, review of the comments and log files may provide some help to understand the parts which are in need of redesign [4, 5, 24, 30, 32], however it still requires a manual activity.

Recently, refactoring approach is used in the redesign phase and it helps developers to change the structure of the code with preserving its external behavior. After deciding which parts should be redesigned, the changes are done automatically with the help of IDE's refactoring functionality. However, deciding the parts which need refactoring still a manual and hard task. Thus, it is a typical problem of decision making under uncertainty. Experienced staff should examine each modules of the code and it is especially very hard for the projects which are composed of thousands even hundreds of modules. An

automated process would help developers or maintenance team, to pin point the parts of the software that needs refactoring; without inspecting each line of code manually.

Our aim in this research is to reduce the maintenance effort thus the cost of the software project by observing the architectural structure of the code as a refactoring prediction model.

3.1. Research Questions

Refactoring is a helpful approach for constructing the design of the software in order to make the software much easier to understand. It increases the maintainability and quality of the system by decreasing the overall complexity [4]. An effective refactoring strategy would shorten maintenance periods, decrease the development effort and improve software quality, which would lead to a decrease in the maintenance costs.

In this thesis we concentrate on the automatic prediction of refactoring candidates since in most of the software development settings a refactoring plan does not exists and the maintenance is seen as a second-class activity. Therefore, there is no incentive to spend money during development to reduce the costs of system change [33]. Moreover, the developers refactor the code based on their level of experiences through manual inspection without any documentation.

As we mention the phases of refactorings in section 1.1, the most important two parts are the identifying the code segments which need refactoring and analyzing the cost/ benefit affect of each refactoring. Therefore our research question is:

How can we predict which part of the code needs refactoring?

4. PROPOSED MODEL

Our motivation in this research is to help maintenance team to find out the modules in need of refactoring. We treat refactoring as a machine learning problem and try to predict the modules which are in need of refactoring in order to decrease the complexity, maintenance costs and defect-prone parts in the project. In this research we use both class level and file level information and define the problem as two way classification: refactored and not-refactored classes. We then try to estimate the modules that need refactoring.

Refactoring prediction is a recent study and there is not much work done in this subject. Some researchers try to look for a way to increase the maintainability of the software or find the possible refactorings in the source code with the help of software metrics, CVS log files, manual inspection or developer comments.

We inspired by the works done about prediction models in software engineering domain, especially defect prediction models. The results of defect prediction models and their applicability to the software products are very impressing. Numerous software metrics and statistical models have been developed in order to predict defects in software [9, 10, 12, 13, 15, 16, 34, 35, 36] so far. Instead of classical Naïve Bayes approach, which assumes that each dimension of the data has equal importance on the classification, "Weighted Naïve Bayes" is preferred. Weighted Naïve Bayes approach showed promising outcomes that can generate better results in defect prediction problems with the InfoGain and GainRatio weight assignment heuristics [10, 34], we try to make refactoring predictions by use of Weighted Naïve Bayes approach with the InfoGain weight assignment heuristic.

This research focuses on predicting the candidate refactorings in any given code prior to the maintenance phase. The experiment includes the extraction of metrics and refactor data from previous versions of the projects in order to build a model for predicting the refactoring candidates of new systems in the maintenance phase. Past project data is

used to adjust our model and we classify the modules of new projects as refactored or not-refactored based on the knowledge gathered from previous projects.

In the following sections, the inputs to the refactoring prediction model and the outputs of the model are discussed. The details of the refactoring prediction model and the performance assessments are given in the last sections.

4.1. Inputs to the Model

Static code attributes and code churn metrics are used as an input in order to predict candidate refactorings in pre-maintenance phase. In the equation 4.1 the software system *Sj* is illustrated as a collection of the modules *Mi*, and *Mi* is composed of static code attributes and code churn metrics. These attributes are represented with *ak*; k=1..n in the equation 4.1. Appendix A and Appendix B displays the full list of attributes.

$$M_i = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \tag{4.1}$$

Sj is a system with the modules Mi and the attributes of each module. *m* is the number of the modules and n is the number of the attributes.

$$Sj = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ a_{1n} & a_{2n} & . & . & . & a_{mn} \end{bmatrix} \tag{4.2}$$

The proposed model needs a refactor data to find out the relationship between the attributes and refactoring. Therefore in equation 4.3 the matrix *Dj* should contain the refactor information for the modules. The refactor information should be collected from the refactorings done in the modules which are recorded during the development and maintenance phases or in the product. Since we have no chance to collect this refactor data

during this research because the company does not log those changes, we make 6 assumptions to obtain this data which will be described in section 4.3.

$$D_j = \begin{bmatrix} d_1 \\ d_2 \\ . \\ . \\ . \\ d_m \end{bmatrix} \tag{4.3}$$

Where d_i is 1 for refactored modules, it is 0 for not-refactored modules.

4.2. Outputs of the Model

We categorize the outputs of the model in a binary classification. Hence the outputs of the model are grouped under two classes; refactored and not-refactored. Formally:

$$O = \{o_1, o_2\}, \text{ where } o_1 = \textit{refactored} \text{ and } o_2 = \textit{not-refactored} \tag{4.4}$$

4.3. The Details of the Model

In order to better understand the impact of refactorings to the system design and code, we used the observations that are stated by Stroulia and Kapoor [6]:

- Refactorings decrease the average LOC and the average number of statements of individual system classes.
- Refactorings decrease the average number of methods of individual system classes.
- Refactorings decrease the average number of collaborators of individual system classes.

Thus observations 1, 2, and 3 support that refactorings of the type such as “extract common superclass” or extract subclass” decrease the average complexity of the system design [6]. So, we mainly focus on complexity metrics since we believe that the main idea behind class refactoring is to reduce complexity. We have analyzed commonly used

refactoring types such as Extract Interface, Push Members Down and Extract Superclass to decrease the overall complexity of the classes while preserving their external behavior and increasing the readability and maintainability of the code and design [4].

It is very hard to understand or modify one part of a class before reading, modifying and testing it [30]. Also, new functionalities are added to the projects by using a method in a class, generally with the usage of some other methods in that class. Therefore, the overall complexity increases together with the maintenance costs of the system. In order to trace changes in the class structure, we propose a module (class or file) based approach to find out which modules are more likely to be refactored.

In this research, our aim is to implement and evaluate Weighted Naïve Bayes with InfoGain and show that it can be used for predicting the refactoring candidates. Naïve Bayes classifier is a simple yet powerful classification method based on the famous Bayes' Rule. Bayes' Rule uses prior probability and likelihood information of a sample for estimating posterior probability [37].

$$P(C_i \mid x) = \frac{P(x \mid C_i)P(C_i)}{P(x)} \tag{4.5}$$

To use it as a classifier, one should compute posterior probabilities P(C_i|x). For each class and choose the one with the maximum posterior as the classification result. Class posteriors in Naïve Bayes classification are calculated as follows:

$$P(C_i \mid x) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{x_j^t - m_{ij}}{s_j}\right)^2 + \log\left(\hat{P}(C_i)\right) \tag{4.6}$$

This simple implementation assumes that each dimension of the data has equal importance on the classification. However, this might not be the case in real life. For example, the Cyclomatic complexity of a class should be more important than the count of commented lines in a class. To cope with that problem, Weighted Naïve Bayes classifier is proposed and tested against Naïve Bayes [10, 38]. Class posterior computation is quite similar to Naïve Bayes only with the introduction of weights for each dimension. Formula for computing class posteriors in Weighted Naïve Bayes is as follows:

$$P(C_i \mid x) = -\frac{1}{2}\sum_{j=1}^{d} w_j \left(\frac{x_j^t - m_{ij}}{s_j} \right)^2 + \log\left(\hat{P}(C_i)\right) \tag{4.7}$$

Introduction of weights makes it possible to favor some dimensions over others but it also raises a new problem: determining the weights. In our case, dimensions consist of different attributes calculated from the source code (see Table 4.1 and Table 4.2) and we need some heuristics for determining the weights (or the importance) of the attributes.

Table 4. 1. Metrics Collected From the projects in class-level

Cyclomatic Density	Halstead Program Difficulty
Decision Density	Halstead Program Length
Essential Density	Halstead Program Level
Branch Count	Halstead Programming Effort
Condition Count	Halstead Programming Time
Cyclomatic Complexity	Halstead Program Volume
Decision Count	Maintenance Severity
Essential Complexity	Coupling Between Objects
LOC	Fan In
Total Operands	Number of Children
Total Operators	Percentage of Pub Data
Unique Operands Count	Response for Class
Unique Operators Count	Weighted Methods

Table 4. 2. Metrics Collected From the projects in file-level

Cyclomatic Density	Halstead Program Difficulty
Decision Density	Halstead Program Length
Essential Density	Halstead Program Level
Branch Count	Halstead Programming Effort
Condition Count	Halstead Programming Time
Cyclomatic Complexity	Halstead Program Volume
Decision Count	Maintenance Severity
Essential Complexity	Coupling Between Objects
LOC	Fan In
Total Operands	Number of Children
Total Operators	Percentage of Pub Data
Unique Operands Count	Response for Class
Unique Operators Count	Weighted Methods
Total Commits since last release	Lines Added since last release
Total Commiters since last release	Lines Removed since last release
Total Commits Per File	Gini Inequality Coefficient
Total Committer Percentage	

In this research we use InfoGain as the heuristic for weight assignment. InfoGain measures the minimum number of bits to encode the information obtained for prediction of a class (C). Concisely, the information gain is a measure of the reduction in entropy of the class variable after the value for the feature is observed [39].

$$InfoGain\ \ (x, A) = Entropy\ \ (x) - \sum_{a \in A} \frac{|\ x = A\ |}{|\ x\ |} Entropy\ \ (x = a) \tag{4.8}$$

In the equations "w" denotes the weight of attribute in data set which is calculated with below equation. In iterative InfoGain subsetting, predictors are learned using the i = 1; 2; . . . ; nth top-ranked attributes. Subsetting terminates when i + 1 attributes perform no better than i [9].

$$w_d = \frac{InfoGain\ \ (d) \times n}{\sum InfoGain\ \ (i)} \tag{4.9}$$

As we mentioned earlier, we use Weighted Naïve Bayes classifier in training our proposed model. In our model proposal the first step is to determine the parts to be considered as refactored and not-refactored. The collected data has an important effect in this research in order to build a useful prediction model and obtain meaningful results.

Since there is no data related with refactoring, we have used a heuristic [7, 40, 41] in order to estimate the classes that should be refactored during each version upgrade. Our assumption is that if the complexity metrics of a class decreases from the beginning of the project, then that class is assumed to be refactored. Thus, the refactor data is collected based on this assumption. We did not know which metrics directly affect the refactoring decision. Firstly we focus on complexity metrics since we think that the main idea behind refactoring is to reduce complexity [6]. However, we also make experiments with other groups of metrics: Halstead metrics and LOC. So, we collect four different refactor data based on our four assumptions:

Assumption 1: Collect refactor data depending on the changes of McCabe Complexity metrics.

Assumption 2: Collect refactor data depending on the changes of LOC metric.

Assumption 3: Collect refactor data depending on the changes of all Halstead metrics.

Assumption 4: Collect refactor data depending on the changes of a selected subset of Halstead metrics.

We also use the above four assumptions for construction of refactor data in file-level. In order to see the effect of code churn metrics in file level, we also make two new assumptions. The aim of these assumptions is that to compare the prediction performance of our model based on the effect of code churn metrics. So, our additional assumptions for the file-level metrics are:

Assumption 5: Collect refactor data depending on the changes of all set of metrics excluding the code churn metrics.

Assumption 6: Collect refactor data depending on the changes of all set of metrics including the code churn metrics.

We used four different projects from a local GSM operator company and their thirteen versions. These projects are implemented in Java programming language and they belong to middleware applications. We collected 26 static code attributes in mainly three classes such as Halstead metrics, McCabe's complexity metrics and lines of code from different versions of the projects. We also collect 7 code churn/ history metrics from the project versions in file level. We normalized the data in GSM datasets [42] since they are very complex projects at the application layer. They are refactored and changed frequently during the development phase of the project. Small samples from a skewed population are a problem, since the confidence intervals they produce are often off center and too narrow. If the data is very skewed, we might consider using the logarithmic transformation since it has the most impact on skewness [43]. In order to normalize the data in GSM datasets we have taken the logarithm of all values in them and compared the logarithmic values in

order not to be affected by small changes. The pseudo code of the construction of the data sets is given in Figure 4.1.

```
Change all zeros by 1 × 10^-6 in the data set
Take the logarithm of all values in the data set
For each two version pair
    For each class in the project
        if
            the value of selected groups' metrics of
            the current class in the current version
            greater than
            selected groups' metrics of the current
            class in the next version
        then
            the class is marked as refactored
        else
            the class is marked as not-refactored
        end if
    end for
end for
```

Figure 4. 1. The Pseudo Code for Constructing Data Set

After collecting refactored module data, we apply Weighted Naïve Bayes for automatic prediction of candidate modules. Then, we would be able to recommend the developers to refactor these modules first. The pseudo code of the model is given in Figure 4.2. The same algorithm is also run with the refactored file data for prediction of candidate files.

```
DATA = [GSM5, GSM7, GSM4, GSM8e]

LEARNER = [WNB (Weigted Naive Bayes)e]
 FILTER = [none loge]
 ATTRIBUTES = 26 or 33
 SHUFFLE = 10
 K = 10
 RUN =10
 FOR EACH data in DATA
          FOR EACH filter in FILTER

            data_1 = filter(data)
            rank data_1 attributes via InfoGain          // Equation 4.8
            FOR EACH attribute in ATTRIBUTE
              attribute_1 = the i th highest ranked attributes
              data_2 = select attribute_1 from data_1
              FOR EACH shuffle in SHUFFLE

                random order from data_2
                generate K folds from data_2
                FOR EACH run in RUN

                        TEST = fold in the current run
                        TRAIN= data_2 - TEST
                        WNB_PRED = Train WNB LEARNER with TRAIN
                        [wnb_pd, wnb_pfe] = WNB_PRED on TEST
              END
            END
          END
        END
 END
```

Figure 4. 2. The Pseudo Code of the Model

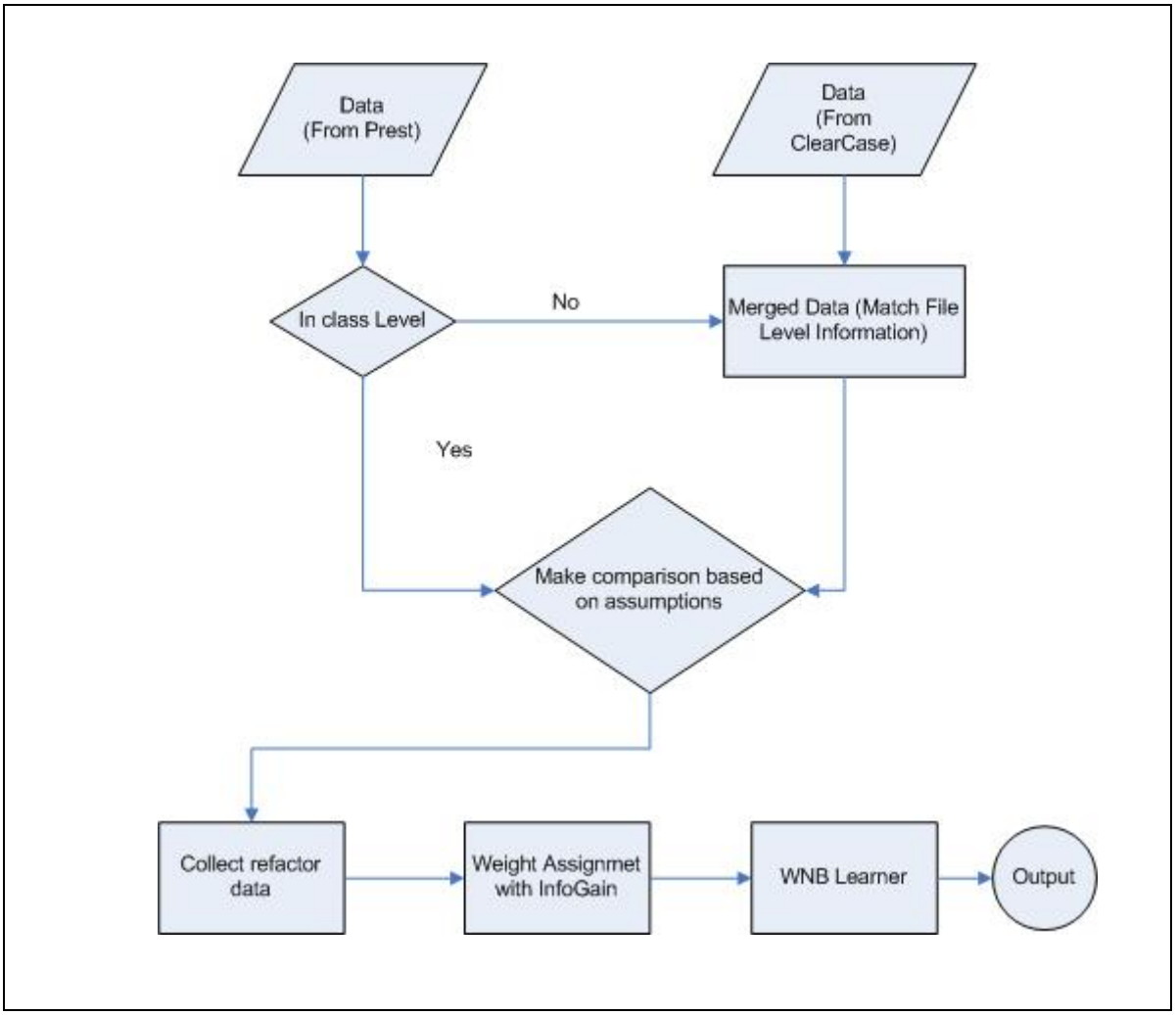

Figure 4. 3. The Schematic Description of the Model

The schematic description of the proposed model is given in Figure 4.3. We formed ten experiments for each dataset: four experiments based on class-level and six experiments in file-level. Assumptions 1, 2, 3 and 4 are used for collection of the refactor class data and refactored/ not-refactored label is added to the input data. Instead we have used only selected group of the metrics for collection of refactor data, we use all 26 metrics listed in Table 4.1 and the refactored/ not-refactored label as an input for our WNB model. For collection of refactor data in file level we use assumptions 1, 2, 3, 4, 5 and 6. Again, for the first 4 assumptions, only the selected group of metrics is used for labeling the refactored files and 33 file-level metrics (Table 4.2) and refactored/ not-refactored label are used as an input to the model. However, last two experiments are done in order to see the effect of code churn metrics; so refactor data is collected based on the changes of all 26

metrics in file level (assumption 5) and changes of all 33 metrics in file level (assumption 6) given in Table 4.2. Since we want to see the effect of churn metrics in our predictor's performance, for the last two experiments our inputs to the model are: 26 file-level attributes and refactor label for the fifth experiment and 33 file-level attributes and refactor label for the last one.

As a result we have a binary output set as classified modules namely refactored or not-refactored. Finally we confirmed how many refactored modules are correctly predicted and how many modules were marked as refactored despite being not-refactored.

4.4. Assessing the Performance of the Model

We need certain calculations to assess the prediction performance of our refactoring prediction model. So we prefer the evaluation criteria which are used to asses the prediction performance of defect prediction models. Generally, *receiver-operator* (ROC) curve from signal detection theory has been employed into defect prediction studies [44]. In signal detection theory, for the evaluation of the performance of different predictors, ROC curves are suggested. A typical ROC curve, as represented in Figure 4.4, shows the hit rates of actual signals and false alarms of the predictor [44]. Similarly, we adopt these measures to our model as; pd is the measure of detecting real refactored classes over all real refactored ones and pf is the measure of detecting classes as refactored that are not actually refactored over all not-refactored classes [9]. The line where (*pd, pf*) passes between (0,0) and (1,1) means that detection contains no information. Predictor never detects any refactored modules in the software or all its predictions are false alarms. In the perfect scenario the predictor should catch all of the refactored classes and never define a not-refactored class as refactored. Thus, higher pd values and lower pf values reflects the performance of the predictor. A perfect scenario should have $pd = 1$ and $pf = 0$.

To measure hit rates and false alarm rates of predictors, a confusion matrix is used, as shown in Table 4.3 [9]. If a module is actually refactored and the model is able to catch it, we increase the number of TP by 1. If our model wrongly predicts a not-refactored module as refactored, then we increase the number of FP by 1. After forming the confusion matrix,

we used equations 4.10 and 4.11 in order to calculate probability of false alarms (*pf*), probability of detection (*pd*) [9].

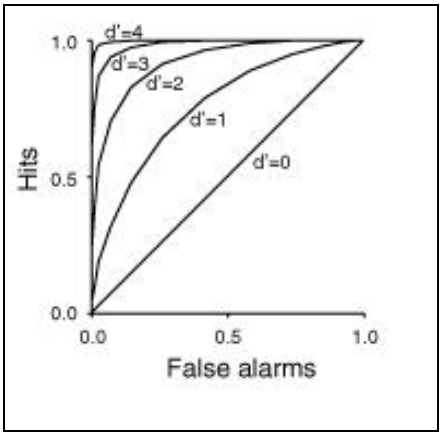

Figure 4. 4. A typical ROC curve [44]

Table 4. 3. Confusion Matrix

	Estimated	
Real	**Refactored**	**Not-Refactored**
Refactored	A	C
Not-Refactored	B	D

$$pd = \frac{A}{A + C} \tag{4.10}$$

$$pf = \frac{B}{B + D} \tag{4.11}$$

A: (TP) True positive, classified refactored module correctly.

C: (FP) False positive, classified refactored module incorrectly.

B: (FN) False negative, classified not-refactored module incorrectly.

D: (TN) True negative, classified not-refactored module correctly.

5. DATA COLLECTION

We have proposed a model which will be applied to all types of projects. The dataset that we used for this study is composed of four projects from a local GSM company. These projects include thirteen releases.

The following sections discuss the requirements for the data, the extraction of the data and an overview of the projects used in this research.

5.1. Requirements for the Data

The goal of a predictor is to find out the relation between attribute columns that predict values in a refactor column so that the developers can determine where to refactor first, if such relations exist.

In order to validate our approach we have built our refactor predictor from static code attributes and code churn metrics. The attributes can be collected in file, class or method level. However, we are interested in both class level refactorings since it is the smallest unit where most of the refactorings are done and file level refactorings in order to move to higher granularity [4, 6, 45]. The static code attributes are formulated in a table, where columns describe software attributes such as lines of code, McCabe and Halstead metrics. Each row in the data table corresponds to data from software modules. We also included another column that indicates the module is refactored or not. Although we are working in class level refactorings, we also include some of the code churn metrics which can be collected in file level. The extracted churn metrics can be listed as:

- Total commits count since last release
- Total commiters since last release
- Total commits per file since last release
- Top committer percentage
- Lines added since last release
- Lines removed since last release
- Gini inequality coefficient

5.2. Data Extraction

We used four different projects from a local GSM operator company namely GSM5, GSM7, GSM4 and GSM8 [42] and their thirteen versions (2.31, 2.32, 2.33, 2.34, 2.35, 2.36, 2.37, 2.38, 2.39, 2.40, 2.41, 2.43, 2.44).

Table 5. 1. Attribute and Class information of the GSM5 project [42]

Name	# Attributes	# Classes
GSM5 2.31	26	561
GSM5 2.32	26	561
GSM5 2.33	26	567
GSM5 2.34	26	566
GSM5 2.35	26	566
GSM5 2.36	26	566
GSM5 2.37	26	568
GSM5 2.38	26	568
GSM5 2.39	26	568
GSM5 2.40	26	580
GSM5 2.41	26	586
GSM5 2.43	26	588
GSM5 2.44	26	588

Table 5. 2. Attribute and Class information of the GSM7 project [42]

Name	# Attributes	# Classes
GSM7 2.31	26	2109
GSM7 2.32	26	2109
GSM7 2.33	26	2116
GSM7 2.34	26	2133
GSM7 2.35	26	2140
GSM7 2.36	26	2150
GSM7 2.37	26	2151
GSM7 2.38	26	2156
GSM7 2.39	26	2164
GSM7 2.40	26	2169
GSM7 2.41	26	2176
GSM7 2.43	26	2178
GSM7 2.44	26	2183

We collected 26 static code attributes such as LOC, Halstead and McCabe metrics from these projects with Prest [46]. Prest is a metric extraction tool which is developed in

SoftLab. Different than available commercial products, Prest is able to parse C, C++, Java, Jsp projects and extract static code attributes at package, file, class and method level. It can also form a dependency matrix based on caller-callee relations between modules to examine the complexity and design of software systems. Currently, we are extending this tool to collect PL/ SQL metrics as well. Table 5.1, 5.2, 5.3 and 5.4 show general class-level properties of all four projects used in this study.

Table 5. 3. Attribute and Class information of the GSM4 project [42]

Name	# Attributes	# Classes
GSM4 2.31	26	430
GSM4 2.32	26	436
GSM4 2.33	26	437
GSM4 2.34	26	437
GSM4 2.35	26	444
GSM4 2.36	26	467
GSM4 2.37	26	483
GSM4 2.38	26	483
GSM4 2.39	26	483
GSM4 2.40	26	486
GSM4 2.41	26	498
GSM4 2.43	26	496
GSM4 2.44	26	498

Table 5. 4. Attribute and Class information of the GSM8 project [42]

Name	# Attributes	# Classes
GSM8 2.31	26	424
GSM8 2.32	26	424
GSM8 2.33	26	424
GSM8 2.34	26	428
GSM8 2.35	26	428
GSM8 2.36	26	428
GSM8 2.37	26	428
GSM8 2.38	26	428
GSM8 2.39	26	428
GSM8 2.40	26	430
GSM8 2.41	26	431
GSM8 2.43	26	431
GSM8 2.44	26	432

Table 5. 5. Attribute and File information of the GSM5 project [42]

Name	# Attributes	# Classes
GSM5 2.31	33	3
GSM5 2.32	33	19
GSM5 2.33	33	32
GSM5 2.34	33	27
GSM5 2.35	33	75
GSM5 2.36	33	12
GSM5 2.37	33	25
GSM5 2.38	33	18
GSM5 2.39	33	38
GSM5 2.40	33	25
GSM5 2.41	33	17
GSM5 2.43	33	35
GSM5 2.44	33	16

Table 5. 6. Attribute and File information of the GSM7 project [42]

Name	# Attributes	# Classes
GSM7 2.31	33	2
GSM7 2.32	33	11
GSM7 2.33	33	14
GSM7 2.34	33	5
GSM7 2.35	33	3
GSM7 2.36	33	232
GSM7 2.37	33	4
GSM7 2.38	33	1
GSM7 2.39	33	8
GSM7 2.40	33	15
GSM7 2.41	33	10
GSM7 2.43	33	5
GSM7 2.44	33	6

Also, we have collected 7 file-level metrics (total commits count since last release, total committers since last release, total commits per file since last release, Top committer percentage, Lines added since last release, Lines removed since last release, Gini inequality coefficient) from the versioning system and match them with the file level metrics collected with Prest [46]. Table 5.5, 5.6, 5.7 and 5.8 show the file-level information of the projects. Only the file information on which the code churn metrics can

also be collected is given, since all the files are not expected to change during a version upgrade.

These data sets do not contain refactor data either; thus we used our six assumptions which we described in section 4.3, to construct the refactor data of previous versions.

Table 5. 7. Attribute and File information of the GSM4 project [42]

Name	# Attributes	# Classes
GSM4 2.31	33	3
GSM4 2.32	33	114
GSM4 2.33	33	33
GSM4 2.34	33	76
GSM4 2.35	33	38
GSM4 2.36	33	43
GSM4 2.37	33	32
GSM4 2.38	33	41
GSM4 2.39	33	72
GSM4 2.40	33	38
GSM4 2.41	33	51
GSM4 2.43	33	58
GSM4 2.44	33	26

Table 5. 8. Attribute and File information of the GSM8 project [42]

Name	# Attributes	# Classes
GSM8 2.31	33	1
GSM8 2.32	33	12
GSM8 2.33	33	9
GSM8 2.34	33	8
GSM8 2.35	33	6
GSM8 2.36	33	4
GSM8 2.37	33	3
GSM8 2.38	33	38
GSM8 2.39	33	57
GSM8 2.40	33	27
GSM8 2.41	33	18
GSM8 2.43	33	23
GSM8 2.44	33	35

5.3. An Overview of the Projects

The GSM company has grown very rapidly and successfully since its foundation in 1994. Their software systems have millions of lines of code that needs to be maintained. As the technology changes and the customers require new functionalities, they keep developing code faster than ever. Similar to other software development companies, testing is one of the most critical stages in their development cycle [19]. There are more than a hundred active developers and they are using Waterfall Model in their development process.

The company has the standard 3-tier architecture with presentation, application and data layers. However, the content in these layers can not be separated as distinct projects. A new version of the system becomes available in two week periods. However, this release can not be taken as one project, since all the applications are collected in a pool and there are nearly 25 applications exist in every release of the software. Any enhancement to the existing software by some means influences all or some of the layers/ applications at the same time, making it difficult to identify code ownership as well as to define distinct software projects. In this research, we mainly focus on the four projects namely GSM5, GSM7, GSM4 and GSM8 [42] due to the their complex natures.

6. EXPERIMENTS AND RESULTS

In this part, we present our experimental design and the results of our experiments to validate our hypothesis using our proposed model.

6.1. Experimental Design

We have designed the experiments to achieve high degree of internal validity by carefully studying the effect of independent variables on dependent variable in a controlled manner [47]. In order to carry on statistically valid experiments, datasets should be prepared carefully. For this reason, k-fold cross validation is used. In k-fold cross validation, data is divided into k equal portions and training process is repeated for k times with k-1 folds used as train data and one fold is used as test data. We chose k as 10 so at each run, we use 9 folds as train data and one fold as test data. Each holdout experiment is also repeated 10 times and in each repetition the datasets are randomized to overcome any ordering effect and to achieve reliable statistics. Moreover, since both the train and test data should have a good representation of the real data, the ratio among the refactored and not-refactored samples should be preserved. We have used stratified sampling so when dividing the data into 10 folds, we made sure that each fold preserves the refactored/ not-refactored samples ratio. So, this whole process is repeated 100 times with the shuffled data for each dataset.

6.2. Experiments

We used data from a large GSM company [19]. For each project and its versions, we collected 26 class-level software metrics and 33 file-level metrics as we mentioned in section 5.2. For each dataset, we construct a matrix where rows represent the classes and columns represent the attributes.

We normalized the data and we used the Weighted Naïve Bayes model for refactoring predictions [19, 43]. In order to make experiments with Weighted Naïve Bayes

model, we use Info Gain heuristic in order to assign weights to the attributes in accordance with their importance.

Since we have not any refactor data to train our predictor, we construct refactor data based on the four assumptions in class-level and six assumptions in file-level that we mentioned in section 4.3. While comparing the selected group of metrics value for each module, the comparison is based on the names of the modules. The Prest parser [46] assigns a different unique identifier to the modules during each parsing proceess. So, id's can be different between the versions, however names remain unchanged, so that it can be used for comparison. However, if a module is refactored and renamed during a version upgrade, then we could not catch this case. So, we remain this case as a future work.

Then the datasets are separated to 10 parts, as one of them is used as training and the others are used as test data while preserving the refactored/ not-refactored ratio same with the original dataset. Finally, the experiments are repeated 10 times with Weighted Naïve Bayes classifier with Info Gain heuristic.

We conducted four experiments for each Bayesian model with four dataset. The metrics are grouped in 4 parts, and the refactor data is constructed with each set of attributes independently. Since we do not know actually which metrics have effect on the refactoring decision, we try to predict the refactor data according to the change in the values of selected group of metrics. Here are the experiments for collecting the refactor data.

6.2.1. Experiment with McCabe's complexity metrics

We make experiment with McCabe complexity metrics namely; Cyclomatic Density, Decision Density, Essential Density, Cyclomatic Complexity, Essential Complexity, for the construction of refactor data. Assumption 1 is used with the algorithm given in Figure 4.1 both in class-level and file-level experiments.

6.2.2. Experiment with LOC metric

We also make experiment with lines of code metric based on assumption 2 both in class-level and file-level experiments.

6.2.3. Experiment with Halstead metrics

We make experiment with Halstead metrics namely; Halstead Program Difficulty, Halstead Program Length, Halstead Program Level, Halstead Programming Effort, Halstead Programming Time, Halstead Program Volume, for the construction of refactor data. Assumption 3 is used with the algorithm given in Figure 4.1 both in class-level and file-level experiments.

6.2.4. Experiment with selected group of Halstead's metrics

Since some of the Halstead metrics grows with each version due to their nature, we make a new experiment by selecting the metrics according to their tendency to increase/ decrease. Halstead Program Difficulty, Halstead Program Level, Halstead Programming Effort and Halstead Programming Time metrics are used for the construction of the data based on assumption 4 both in class-level and file-level experiments.

We also make two additional experiments in only file-level. Indeed we focus on the refactorings which are done in class-level, we also want to see our models performance in file level-refactorings with and without the newly collected code churn metrics.

6.2.5. Experiment without code churn metrics

The refactor data is collected based on the assumption 5. Only 26 attributes collected in file-level is used both in collection of refactor data and as an input to our prediction model.

6.2.6. Experiment with code churn metrics

The refactor data is collected based on the assumption 6. In additon to 26 attributes collected in file-level, we also add the seven code churn metrics in order to compare our models performance with the previous experiment.

6.3. Results

In this section we discuss the results of the experiments. To present the results of the experiments we use *probability of detection* (*pd*) and *probability of false alarm* (*pf*) values. The *pd* of a predictor measures the probability of the defection, in other words, how good it catches the candidate refactorings in the model. The *pf* of a predictor measures the false alarm rate. In a perfect scenario the higher *pd* and the lower *pf* values are preferable.

6.3.1. Experiments and Results with McCabe Metrics in class-level

As we mentioned in section 4.3 , we mainly focus on the complexity metrics. So, we firstly make experiments with McCabe complexity metrics namely; Cyclomatic Density, Decision Density, Essential Density, Cyclomatic Complexity, Essential Complexity, for the construction of refactor data. In Table 6.1 the results are listed.

6.3.2. Experiments and Results with LOC Metric in class-level

We also make experiments with LOC metric. According to researchers [3, 29, 32] lines of code metric doesn't give meaningful resuls. Since the same code can be written very short or long according to algorithms complexity or developers experience. However, in order to see the effect of lines of code on the refactor prediction, we also repeat our experiments with it. The results of the experiments is listed in Table 6.2.

6.3.3. Experiments and Results with Halstead Metrics in class-level

As we mentioned in section 4.3, we also make experiments with Halstead metrics namely; Halstead Program Difficulty, Halstead Program Length, Halstead Program Level, Halstead Programming Effort, Halstead Programming Time, Halstead Program Volume, for construction of refactor data. The results with all Halstead metrics is shown in Table 6.3.

6.3.4. Experiments and Results with a selected subset of Halstead Metrics in class-level

We finally repeat our experiments in class-level with a selected subset of Halstead metrics for the construction of refactor data. Since some of the Halstead metrics always grows with each version due to their characteristics and so computation, we make a new experiment by selecting the metrics according to their tendency to increase/decrease. For all projects, the tendency of Halstead metrics among each version upgrade, is examined and selected wth Info Gain subsetting heuristic. Then we see that Halstead Program Length, Halstead Program Volume cannot be used since they are generally increase with each version. After the subsetting process we see that Halstead Program Difficulty, Halstead Program Level, Halstead Programming Effort, Halstead Programming Time metrics are the meaningful metrics to collect refactor data. The results with the selected subset of Halstead metrics is shown in Table 6.4.

Table 6. 1. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 1

Project	**GSM5**		**GSM7**		**GSM4**		**GSM8**	
	InfoGain + WNB (%)		**InfoGain + WNB (%)**		**InfoGain + WNB (%)**		**InfoGain + WNB (%)**	
Version	**pd**	**pf**	**Pd**	**Pf**	**pd**	**pf**	**pd**	**pf**
2.31	55	20	61	19	63	16	54	24
2.32	63	19	68	18	72	16	58	24
2.33	61	18	73	18	75	15	62	23
2.34	69	16	77	17	77	15	67	22
2.35	75	15	79	16	79	13	73	21
2.36	78	15	81	15	81	12	78	21
2.37	82	16	84	15	80	13	78	20
2.38	85	14	86	14	83	12	79	18
2.39	88	14	89	14	85	11	80	17
2.40	88	13	87	15	85	11	83	17
2.41	89	13	88	14	86	12	84	16
2.43	89	13	89	14	87	11	84	16
2.44	90	13	89	14	87	11	85	15
Avg	***77.85***	***15.31***	***80.85***	15.62	***80***	***12.92***	***74.23077***	***19.54***

The results of our experiments show that the best result is obtained from the experiment which is based on assumption 1 in class-level. In this assumption, the historical refactor data is collected from the value changes in McCabe Complexity metrics between versons of each project. For the first experiment with IG+WNB model our predictor predicts the classes that need to be refactored on the average of 78%, 81%, 80% and 74% (Table 6.1). For the other experiments, the results are listed as 62%, 56%, 73% and 65%

with assumption 2 (Table 6.2), 53%, 80%, 71% and 65% (Table 6.3) and 65%, 79%, 76% and 74% (Table 6.4) with assumption 4. So, we can conclude that, complexity metrics have much influence on refactoring decision. Also, subsetting the Halstead metrics gives better results than the experiment based on full Halstead metrics.

Table 6. 2. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 2

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	Pd	Pf	Pd	pf	pd	pf
2.31	53	26	49	32	61	24	54	27
2.32	55	25	49	30	65	23	57	26
2.33	59	25	45	30	66	23	60	26
2.34	55	23	51	28	71	22	59	26
2.35	61	22	55	31	78	20	63	25
2.36	58	19	57	30	75	21	65	23
2.37	59	19	61	28	75	18	68	21
2.38	65	20	59	27	76	17	69	21
2.39	67	20	59	28	76	17	69	21
2.40	68	20	60	28	78	18	70	20
2.41	66	19	58	27	76	17	70	19
2.43	68	19	61	27	77	17	70	19
2.44	70	20	61	26	78	17	70	20
Avg	***61.85***	***21.31***	***55.77***	***28.62***	***73.23***	***19.54***	***64.92308***	***22.62***

Table 6. 3. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 3

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	Pd	Pf	Pd	Pf	pd	pf
2.31	45	30	63	19	59	19	53	24
2.32	48	28	68	18	63	19	51	26
2.33	48	28	72	17	69	18	55	23
2.34	49	26	77	18	67	17	58	23
2.35	53	25	77	17	71	14	61	24
2.36	50	26	83	15	72	14	68	23
2.37	51	26	83	14	73	14	65	23
2.38	55	25	85	15	73	13	70	22
2.39	57	27	86	14	75	14	72	21
2.40	59	25	87	14	73	14	73	21
2.41	59	25	87	14	75	14	75	21
2.43	58	25	87	15	75	14	75	21
2.44	60	24	87	14	75	13	75	19
Avg	***53.23***	***26.15***	***80.15***	***15.69***	***70.77***	***15.15***	***65.46154***	***22.38***

Table 6. 4. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 4

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	Pd	Pf	pd	Pf	pd	pf
2.31	51	25	61	19	60	19	59	21
2.32	53	25	68	19	65	19	61	19
2.33	55	23	72	18	69	18	63	18
2.34	59	21	75	18	69	18	65	16
2.35	61	20	77	17	73	17	69	13
2.36	62	19	81	16	75	17	73	13
2.37	65	19	80	15	79	15	77	13
2.38	68	17	83	15	83	15	79	12
2.39	73	17	85	14	85	14	81	12
2.40	72	17	87	14	83	15	82	13
2.41	73	17	87	15	83	14	82	12
2.43	74	16	86	14	85	13	84	12
2.44	74	16	86	14	85	13	84	12
Avg	***64.62***	***19.38***	***79.08***	***16***	***76.46***	***15.92***	***73.76923***	***14.31***

In the first versions of the software our proposed prediction model detects the classes that need to be refactored with 55%, 63%, 61% and 69% (Table 6.1) based on assumption 1 in class-level for GSM5 project. Up to the last versions of the projects, our predictor learning performance mostly increases. In the last versions of the GSM5 project the predictor's performance increases to 88%, 89%, 89% and 90% (Table 6.1). The same results are also obtained from the other experiments and also for other projects (Table 6.1, Table 6.2, Table 6.3 and Table 6.4). We can conclude that the learning performance improves as we move to more stable versions and learn more about the complexity of the code.

We also observe that as we move to later versions false alarm rates decrease (from pf:20 to pf:13 for GSM5, from pf:19 to pf:14 for GSM7, from pf:16 to pf:11 for GSM4 and from pf:24 to pf:15 for GSM8) with our proposed learner (Table 6.1). The same results are also obtained from the other experiments (Table 6.1, Table 6.2, Table 6.3 and Table 6.4). Low pf rates prevent software developers from manual analysis of classes which are not needed to be refactored. In the thirteen versions of a complex code such as GSM7 project we can predict on the average 74% of the refactored classes with 18% of manual inspection effort with assumption 1 in class-level.

6.3.5. Experiments and Results in file-level without code churn metrics in file-level

As we said in section 6.2, we repeat our experiments with file-level information. Our aim is to test our predictors permormance with the file based information. The same set of metrics are used both in class-level and file-level experiments. Our refactroring data is collected based on assumption 5. The results are listed in Table 6.5.

Table 6. 5. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 5 (without code churn metrics)

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	Pd	Pf	pd	Pf	pd	pf
2.31	45	28	45	30	46	30	44	31
2.32	48	28	46	29	48	30	47	31
2.33	49	26	45	29	45	33	46	33
2.34	53	26	47	31	53	35	48	31
2.35	53	25	48	31	51	33	49	31
2.36	56	25	49	30	57	30	51	31
2.37	59	25	50	28	60	28	53	30
2.38	63	24	52	27	62	25	55	30
2.39	64	25	53	26	63	23	56	29
2.40	65	24	53	25	63	25	57	28
2.41	66	24	55	25	64	24	57	28
2.43	66	23	55	25	64	24	58	29
2.44	67	23	57	24	64	24	59	28
Avg	***58.00***	***25.08***	***50.38***	***27.69***	***56.92***	***28.00***	***52.31***	***30.00***

6.3.6. Experiments and Results in file-level with code churn metrics in file-level

We repeat our file-level experiments with the addition of seven code churn metrics. Our aim is to compare our predictors file based performance with and without using churn metrics. Our refactroring data is collected based on assumption 6. The results are listed in Table 6.6.

6.3.7. Experiments and Results with McCabe Metrics in file-level

After making experiments with all metrics including churn metrics, we make additional experiments to see the effect of the group of metrics to our models performance.

So we repeat our class-level experiment with McCabe metrics in file-level. In Table 6.7 the results are listed.

Table 6. 6. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 6 (with code churn metrics)

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	Pd	pf	Pd	Pf	pd	Pf	pd	pf
2.31	52	23	48	26	51	24	45	29
2.32	53	23	49	25	55	24	45	29
2.33	53	22	51	24	53	23	48	28
2.34	55	21	53	22	57	22	48	27
2.35	59	21	57	23	59	22	50	28
2.36	61	20	55	23	60	22	52	26
2.37	54	20	59	22	64	21	54	25
2.38	66	21	61	22	66	22	56	24
2.39	68	20	62	22	67	19	57	23
2.40	69	19	64	21	69	20	58	23
2.41	70	18	66	21	69	19	59	22
2.43	72	18	69	20	71	19	60	22
2.44	73	17	70	21	72	19	61	22
Avg	***61.92***	***20.23***	***58.77***	***22.46***	***62.54***	***21.23***	***53.31***	***25.23***

Table 6. 7. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 1

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	Pd	pf	Pd	Pf	pd	Pf	pd	pf
2.31	55	24	51	25	52	24	50	26
2.32	57	23	52	25	55	24	53	26
2.33	58	22	53	23	53	22	53	25
2.34	61	21	55	22	56	22	55	25
2.35	62	21	57	22	59	22	57	24
2.36	64	20	58	23	61	21	59	22
2.37	66	19	58	21	64	21	59	22
2.38	67	18	60	21	67	21	61	21
2.39	69	18	62	20	68	20	62	21
2.40	73	16	63	20	70	19	63	20
2.41	73	16	65	19	70	19	63	19
2.43	74	15	68	19	71	19	65	18
2.44	73	15	70	19	73	18	65	18
Avg	***65.54***	***19.08***	***59.38***	***21.46***	***63.00***	***20.92***	***58.85***	***22.08***

6.3.8. Experiments and Results with LOC Metric in file-level

In order to see the effect of lines of code on the refactor prediction, we also repeat our experiments with it in file-level. The results of the experiments is listed in Table 6.8.

Table 6. 8. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 2

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	pd	Pf	pd	Pf	pd	pf
2.31	49	23	49	24	49	24	47	29
2.32	51	24	51	24	53	24	47	29
2.33	53	23	52	23	53	23	49	28
2.34	55	23	53	23	56	23	48	27
2.35	57	22	54	23	57	23	49	27
2.36	58	22	56	22	59	21	50	25
2.37	60	21	55	22	60	22	51	24
2.38	60	21	58	22	61	22	53	23
2.39	62	21	59	21	63	20	54	23
2.40	65	22	60	21	66	20	54	23
2.41	65	21	61	21	64	20	56	22
2.43	64	21	62	22	68	20	57	22
2.44	64	21	62	22	68	20	58	22
Avg	***58.69***	***21.92***	***56.31***	***22.31***	***59.77***	***21.69***	***51.77***	***24.92***

6.3.9. Experiments and Results with Halstead Metrics in file-level

We repeat our experiment mentioned in section 6.3.3 also in file-level. The results with all Halstead metrics is shown in Table 6.9.

6.3.10. Experiments and Results with a selected subset of Halstead Metrics in file-level

We finally repeat our experiments in file-level with a selected subset of Halstead metrics for the construction of refactor data. We use only the selected grouo of Halstead metris mentioned in section 6.3.3 and repeat that experiment with file-level imetrics. The results with the selected subset of Halstead metrics is shown in Table 6.10.

Table 6. 9. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 3

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	pd	Pf	Pd	Pf	pd	pf
2.31	50	23	50	23	50	23	48	23
2.32	52	24	52	23	54	24	49	24
2.33	52	24	53	23	54	23	51	23
2.34	54	23	55	22	55	23	53	23
2.35	55	23	56	23	57	23	53	23
2.36	56	22	58	22	56	22	53	22
2.37	58	23	57	22	61	23	54	23
2.38	58	21	59	21	61	22	56	22
2.39	60	26	60	21	63	21	59	21
2.40	61	24	61	21	67	22	59	22
2.41	61	22	63	20	67	21	61	21
2.43	63	22	63	20	69	21	63	21
2.44	63	22	63	20	70	20	63	20
Avg	***57.15***	***23.00***	***57.69***	***21.62***	***60.31***	***22.15***	***55.54***	***22.15***

Table 6. 10. Results for GSM5, GSM7, GSM4 and GSM8 projects based on assumption 4

Project	GSM5		GSM7		GSM4		GSM8	
	InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)		InfoGain + WNB (%)	
Version	pd	pf	pd	Pf	Pd	Pf	pd	pf
2.31	51	23	51	24	52	24	48	23
2.32	53	23	52	23	53	23	50	24
2.33	53	23	53	23	54	23	52	23
2.34	55	22	56	22	56	23	53	22
2.35	57	22	57	22	58	22	54	22
2.36	57	22	58	21	61	23	54	22
2.37	59	24	58	21	64	22	56	22
2.38	61	22	59	21	65	22	57	21
2.39	61	21	60	20	68	22	59	21
2.40	63	21	61	20	69	20	60	20
2.41	64	21	63	20	69	20	62	21
2.43	64	20	64	20	70	19	63	20
2.44	64	20	65	19	71	19	64	20
Avg	***58.62***	***21.85***	***58.23***	***21.23***	***62.31***	***21.69***	***56.31***	***21.62***

When we compare our class-level and file-level experiments, we see that the class-level results are much better. For IG+WNB model our proposed model predicts the classes that need to be refactored on the average 78.2%, whereas for prediction of files it is nearly

61.7% on the average according to the assumption1. The same case exist for other assumptions. It is due to the fact that the number of changed files is very limited according to changed classes so that our model learns less from the file-level information. Also, classes are the smallest units that the refactorings becomes more meaningful.

The addition of code churn metrics has significant effect on our models performance. According to the experiment with and without using code churn metrics, our models predict on the average 58.8% and 50.3% of the files that need in refactoring correspondingly for GSM7 project. Also, for other projects again addition of churn metrics has positive effect on the performance of our predictor.

The results for all ten experiments are given based on the version and projects, so that we can compare the results for projects. In order to compare the results, based on the metrics we organize the results in two group of tables: Table 6.11, 6.12, 6.13 and 6.14 in class-level and Table 6.15, 6.16, 6.17 and 6.18 in file level correspondingly.

Table 6. 11. All results for GSM5 project in class-level

	McCabe		LOC		Halstead		Subsetted Halstead	
Project	**Pd**	**pf**	**Pd**	**Pf**	**pd**	**Pf**	**pd**	**pf**
GSM5 2.31	55	20	53	26	45	30	51	25
GSM5 2.32	63	19	55	25	48	28	53	25
GSM5 2.33	61	18	59	25	48	28	55	23
GSM5 2.34	69	16	55	23	49	26	59	21
GSM5 2.35	75	15	61	22	53	25	61	20
GSM5 2.36	78	15	58	19	50	26	62	19
GSM5 2.37	82	16	59	19	51	26	65	19
GSM5 2.38	85	14	65	20	55	25	68	17
GSM5 2.39	88	14	67	20	57	27	73	17
GSM5 2.40	88	13	68	20	59	25	72	17
GSM5 2.41	89	13	66	19	59	25	73	17
GSM5 2.43	89	13	68	19	58	25	74	16
GSM5 2.44	90	13	70	20	60	24	74	16
Avg	***77.85***	***15.31***	***61.85***	***21.31***	***53.23***	***26.15***	***64.62***	***19.38***

Table 6. 12. All results for GSM7 project in class-level

	McCabe		LOC		Halstead		Subsetted Halstead	
Project	**Pd**	**pf**	**Pd**	**Pf**	**pd**	**pf**	**pd**	**Pf**
GSM7 2.31	61	19	49	32	63	19	61	19
GSM7 2.32	68	18	49	30	68	18	68	19
GSM7 2.33	73	18	45	30	72	17	72	18
GSM7 2.34	77	17	51	28	77	18	75	18
GSM7 2.35	79	16	55	31	77	17	77	17
GSM7 2.36	81	15	57	30	83	15	81	16
GSM7 2.37	84	15	61	28	83	14	80	15
GSM7 2.38	86	14	59	27	85	15	83	15
GSM7 2.39	89	14	59	28	86	14	85	14
GSM7 2.40	87	15	60	28	87	14	87	14
GSM7 2.41	88	14	58	27	87	14	87	15
GSM7 2.43	89	14	61	27	87	15	86	14
GSM7 2.44	89	14	61	26	87	14	86	14
Avg	***80.85***	**15.62**	***55.77***	***28.62***	***80.15***	***15.69***	***79.08***	***16***

Table 6. 13. All results for GSM4 project in class-level

	McCabe		LOC		Halstead		Subsetted Halstead	
Project	**Pd**	**pf**	**Pd**	**Pf**	**pd**	**pf**	**pd**	**Pf**
GSM4 2.31	63	16	61	24	59	19	60	19
GSM4 2.32	72	16	65	23	63	19	65	19
GSM4 2.33	75	15	66	23	69	18	69	18
GSM4 2.34	77	15	71	22	67	17	69	18
GSM4 2.35	79	13	78	20	71	14	73	17
GSM4 2.36	81	12	75	21	72	14	75	17
GSM4 2.37	80	13	75	18	73	14	79	15
GSM4 2.38	83	12	76	17	73	13	83	15
GSM4 2.39	85	11	76	17	75	14	85	14
GSM4 2.40	85	11	78	18	73	14	83	15
GSM4 2.41	86	12	76	17	75	14	83	14
GSM4 2.43	87	11	77	17	75	14	85	13
GSM4 2.44	87	11	78	17	75	13	85	13
Avg	***80***	***12.92***	***73.23***	***19.54***	***70.77***	***15.15***	***76.46***	***15.92***

Table 6. 14. All results for GSM8 project in class-level

	McCabe		LOC		Halstead		Subsetted Halstead	
Project	**Pd**	**pf**	**Pd**	**Pf**	**pd**	**pf**	**pd**	**Pf**
GSM8 2.31	54	24	54	27	53	24	59	21
GSM8 2.32	58	24	57	26	51	26	61	19
GSM8 2.33	62	23	60	26	55	23	63	18
GSM8 2.34	67	22	59	26	58	23	65	16
GSM8 2.35	73	21	63	25	61	24	69	13
GSM8 2.36	78	21	65	23	68	23	73	13
GSM8 2.37	78	20	68	21	65	23	77	13
GSM8 2.38	79	18	69	21	70	22	79	12
GSM8 2.39	80	17	69	21	72	21	81	12
GSM8 2.40	83	17	70	20	73	21	82	13
GSM8 2.41	84	16	70	19	75	21	82	12
GSM8 2.43	84	16	70	19	75	21	84	12
GSM8 2.44	85	15	70	20	75	19	84	12
Avg	***74.23***	***19.54***	***64.92***	***22.62***	***65.46***	***22.38***	***73.77***	***14.31***

Table 6. 15. All results for GSM5 project in file-level

	All Metrics		Without Code Churn		McCabe		LOC		Halstead		Subsetted Halstead	
Project	**pd**	**pf**	**pd**	**Pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**Pf**
GSM5 2.31	52	23	45	28	55	24	49	23	50	23	51	23
GSM5 2.32	53	23	48	28	57	23	51	24	52	24	53	23
GSM5 2.33	53	22	49	26	58	22	53	23	52	24	53	23
GSM5 2.34	55	21	53	26	61	21	55	23	54	23	55	22
GSM5 2.35	59	21	53	25	62	21	57	22	55	23	57	22
GSM5 2.36	61	20	56	25	64	20	58	22	56	22	57	22
GSM5 2.37	54	20	59	25	66	19	60	21	58	23	59	24
GSM5 2.38	66	21	63	24	67	18	60	21	58	21	61	22
GSM5 2.39	68	20	64	25	69	18	62	21	60	26	61	21
GSM5 2.40	69	19	65	24	73	16	65	22	61	24	63	21
GSM5 2.41	70	18	66	24	73	16	65	21	61	22	64	21
GSM5 2.43	72	18	66	23	74	15	64	21	63	22	64	20
GSM5 2.44	73	17	67	23	73	15	64	21	63	22	64	20
Avg	***61.92***	***20.23***	***58.00***	***25.08***	***65.54***	***19.08***	***58.69***	***21.92***	***57.15***	***23.00***	***58.62***	***21.85***

Table 6. 16. All results for GSM7 project in file-level

	All Metrics		Without Code Churn		McCabe		LOC		Halstead		Subset Halstead	
Project	**pd**	**pf**	**pd**	**Pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**Pf**
GSM7 2.31	48	26	45	30	51	25	49	24	50	23	51	24
GSM7 2.32	49	25	46	29	52	25	51	24	52	23	52	23
GSM7 2.33	51	24	45	29	53	23	52	23	53	23	53	23
GSM7 2.34	53	22	47	31	55	22	53	23	55	22	56	22
GSM7 2.35	57	23	48	31	57	22	54	23	56	23	57	22
GSM7 2.36	55	23	49	30	58	23	56	22	58	22	58	21
GSM7 2.37	59	22	50	28	58	21	55	22	57	22	58	21
GSM7 2.38	61	22	52	27	60	21	58	22	59	21	59	21
GSM7 2.39	62	22	53	26	62	20	59	21	60	21	60	20
GSM7 2.40	64	21	53	25	63	20	60	21	61	21	61	20
GSM7 2.41	66	21	55	25	65	19	61	21	63	20	63	20
GSM7 2.43	69	20	55	25	68	19	62	22	63	20	64	20
GSM7 2.44	70	21	57	24	70	19	62	22	63	20	65	19
Avg	***58.77***	***22.46***	***50.38***	***27.69***	***59.38***	***21.46***	***56.31***	***22.31***	***57.69***	***21.62***	***58.23***	***21.23***

Table 6. 17. All results for GSM4 project in file-level

	All Metrics		Without Code Churn		McCabe		LOC		Halstead		Subsetted Halstead	
Project	**pd**	**pf**	**pd**	**Pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**Pf**
GSM4 2.31	51	24	46	30	52	24	49	24	50	23	52	24
GSM4 2.32	55	24	48	30	55	24	53	24	54	24	53	23
GSM4 2.33	53	23	45	33	53	22	53	23	54	23	54	23
GSM4 2.34	57	22	53	35	56	22	56	23	55	23	56	23
GSM4 2.35	59	22	51	33	59	22	57	23	57	23	58	22
GSM4 2.36	60	22	57	30	61	21	59	21	56	22	61	23
GSM4 2.37	64	21	60	28	64	21	60	22	61	23	64	22
GSM4 2.38	66	22	62	25	67	21	61	22	61	22	65	22
GSM4 2.39	67	19	63	23	68	20	63	20	63	21	68	22
GSM4 2.40	69	20	63	25	70	19	66	20	67	22	69	20
GSM4 2.41	69	19	64	24	70	19	64	20	67	21	69	20
GSM4 2.43	71	19	64	24	71	19	68	20	69	21	70	19
GSM4 2.44	72	19	64	24	73	18	68	20	70	20	71	19
Avg	***62.54***	***21.23***	***56.92***	***28.00***	***63.00***	***20.92***	***59.77***	***21.69***	***60.31***	***22.15***	***62.31***	***21.69***

Table 6. 18. All results for GSM8 project in file-level

	All Metrics		Without Code Churn		McCabe		LOC		Halstead		Subsetted Halstead	
Project	**pd**	**Pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**Pf**
GSM8 2.31	45	29	44	31	50	26	47	29	48	23	48	23
GSM8 2.32	45	29	47	31	53	26	47	29	49	24	50	24
GSM8 2.33	48	28	46	33	53	25	49	28	51	23	52	23
GSM8 2.34	48	27	48	31	55	25	48	27	53	23	53	22
GSM8 2.35	50	28	49	31	57	24	49	27	53	23	54	22
GSM8 2.36	52	26	51	31	59	22	50	25	53	22	54	22
GSM8 2.37	54	25	53	30	59	22	51	24	54	23	56	22
GSM8 2.38	56	24	55	30	61	21	53	23	56	22	57	21
GSM8 2.39	57	23	56	29	62	21	54	23	59	21	59	21
GSM8 2.40	58	23	57	28	63	20	54	23	59	22	60	20
GSM8 2.41	59	22	57	28	63	19	56	22	61	21	62	21
GSM8 2.43	60	22	58	29	65	18	57	22	63	21	63	20
GSM8 2.44	61	22	59	28	65	18	58	22	63	20	64	20
Avg	***53.31***	***25.23***	***52.31***	***30.00***	***58.85***	***22.08***	***51.77***	***24.92***	***55.54***	***22.15***	***56.31***	***21.62***

Since the results are given based on each individual project, we also list the average values of 4 projects in both class and file-level in Table 6.19 and 6.20 correspondingly.

Table 6. 19. Average results for GSM5, GSM7, GSM4 and GSM8 projects in class-level

	McCabe		LOC		Halstead		Subsetted Halstead	
Project	**pd**	**pf**	**pd**	**Pf**	**pd**	**pf**	**pd**	**Pf**
GSM5	77.85	15.31	61.85	21.31	53.23	26.15	64.62	19.38
GSM7	80.85	15.62	55.77	28.62	80.15	15.69	79.08	16
GSM4	80	12.92	73.23	19.54	70.77	15.15	76.46	15.92
GSM8	74.23	19.54	64.92	22.62	65.46	22.38	73.77	14.31
Average	**78.25**	**16.00**	**64.00**	**22.75**	**67.25**	**19.75**	**73.50**	**16.25**

Table 6. 20. Average results for GSM5, GSM7, GSM4 and GSM8 projects in file-level

	All Metrics		Without Churn		McCabe		LOC		Halstead		SubsettedHalstead	
Project	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**pf**	**pd**	**Pf**
GSM5	61.92	20.23	58.00	25.08	65.54	19.08	58.69	21.92	57.15	23.00	58.62	21.85
GSM7	58.77	22.46	50.38	27.69	59.38	21.46	56.31	22.31	57.69	21.62	58.23	21.23
GSM4	62.54	21.23	56.92	28.00	63.00	20.92	59.77	21.69	60.31	22.15	62.31	21.69
GSM8	53.31	25.23	52.31	30.00	58.85	22.08	51.77	24.92	55.54	22.15	56.31	21.62
Average	**59.13**	**22.29**	**54.40**	**27.69**	**61.69**	**20.88**	**56.63**	**22.71**	**57.67**	**22.23**	**58.87**	**21.60**

6.4. Cost / Benefit Analysis

We would like to predict candidate refactorings in order to decrease the cost of the project. So, we also conducted a cost/ benefit analysis in order to analyze how much cost and time we can save by using our proposed learner (Table 6.21, 6.22, 6.23 and 6.24).

So, from a practical perspective we also wanted to answer the following questions which are stated by Arisholm and Briand [48] :

- How useful is such a prediction model when predicting future releases?
- What is the cost-benefit of using such a model to focus verification?

In order to make a cost/ benefit analysis, first we collect and mark the candidate refactored classes and not-refactored classes in each iteration. For example after the classification, our predictor says x of the classes are candidate refactorings and our pd value is y%. According to the random strategy that Arisholm and Briand [48] stated, we have to examine z (totalNumberOfClasses * y%) classes in order to predict y% of the classes as refactored. However, we found the same pd rate by only examining x classes. So, our gain is (z - x) * (100 / z). The formulas used for calculation of the gained effort are given in Figure 6.1.

NeededClasses = TotalClasses * pd / 100

inspectedClasses = classes that we marked as refactored

changeClasses% = inspectedClasses/totalClasess * 100

gainedEff% = (neededClasses-inspectedClasses) * 100 / neededClasses

Figure 6. 1. Calculation of the Gained Effort

Table 6. 21. Cost-benefit analysis of GSM projects based on assumption 1 in class-level (with McCabe Complexity Metrics)

	GSM5	GSM7	GSM4	GSM8	Average
Pd	78	81	80	74	78.25
Pf	15	16	13	20	16
totalClasses	541	2056	417	417	857.75
neededClasses	421.98	1665.36	333.6	308.58	682.38
inspectedClasses	71	278	50	58	114.25
changeClasses(%)	13.12	13.52	11.99	13.91	13.32
gainedEff(%)	**83.17**	**83.31**	**85.01**	**81.20**	**83.26**

Table 6. 22. Cost-benefit analysis of GSM projects based on assumption 2 in class-level (with LOC Metrics)

	GSM5	GSM7	GSM4	GSM8	Average
Pd	62	56	73	65	64
Pf	21	27	20	23	22.75
totalClasses	541	2056	417	417	857.75
neededClasses	335.42	1151.36	304.41	271.05	515.56
inspectedClasses	71	278	50	58	114.25
changeClasses(%)	13.12	13.52	11.99	13.91	13.32
gainedEff(%)	**78.83**	**75.85**	**83.57**	**78.60**	**77.84**

Table 6. 23. Cost-benefit analysis of GSM projects based on assumption 3 in class-level (with Halstead Metrics)

	GSM5	GSM7	GSM4	GSM8	Average
Pd	53	80	71	65	67.25
Pf	26	16	15	22	19.75
totalClasses	541	2056	417	417	857.75
neededClasses	286.73	1644.8	296.07	271.05	624.6625
inspectedClasses	71	278	50	58	114.25
changeClasses(%)	13.12	13.52	11.99	13.91	13.32
gainedEff(%)	**75.24**	**83.10**	**83.11**	**78.60**	**81.71**

For example, according to the experiment based on the first assumption in class-level, for GSM5 project, there are 541 classes (totalClasses) on average. Our proposed model predicts 78% of the refactored classes (pd: 78%) by only examining 71 (inspectedClasses) classes. According to the random strategy that Arisholm and Briand

[48] stated, we have to examine 421 (541*73/100) (neededClasses) classes in order to predict 78% of the classes as refactored. Thus, the percentage of changed classes is 13.1% (71/541*100) (changeClasses%) However, we can achieve the same pd rates by examining only 71 classes. By using our proposed model it is possible to gain 83% ((421-71)*100/421) (gainedEff%) efficiency in maintenance. The results for other GSM projects are also listed in Table 6.21. We can gain 83.3% efficiency on the average in four projects. Considering that maintenance is a very costly and time consuming phase for a software development organization, inspecting 278 classes instead of 2056 as in the case of GSM7 project for example, enables practitioners to allocate their resources more efficiently and to decrease their maintenance costs without compromising the desired level of quality. We also make cost/benefit analysis for the other assumptions where the results are listed in Table 6.22, 6.23 and 6.24. We can gain 77.9% (Table 6.22), 81.7% (Table 6.23) and 82.4% (Table 6.24) efficiency on the average in four projects.

Table 6. 24. Cost-benefit analysis of GSM projects based on assumption 4 in class-level (with subsetted Halstead Metrics)

	GSM5	GSM7	GSM4	GSM8	Average
Pd	65	79	76	74	73.5
Pf	19	16	16	14	16.25
totalClasses	541	2056	417	417	857.75
neededClasses	351.65	1624.24	316.92	308.58	650.3475
inspectedClasses	71	278	50	58	114.25
changeClasses(%)	13.12	13.52	11.99	13.91	13.32
gainedEff(%)	**79.81**	**82.88**	**84.22**	**81.20**	**82.43**

6.5. Discussion of the Results

We have set our experimental design in order to see the effect of predictors with the help of software metrics in software lifecycle. First, we have investigated a new learning mechanism that would guide research developers during the maintenance phase. Our motivation was to construct a refactoring prediction model for software projects that has high detection rates as well as low false alarms. Naive Bayes is one of most effective classification algorithms. Naïve Bayes manages to combine signals coming from multiple attributes. It simply uses attribute likelihoods derived from historical data to make

predictions [49]. Weighted Naive Bayes is an extension of Naive Bayes, in which attributes have different weights [34, 38, 50]. For this purpose, we used Weighted Naïve Bayes learner which is frequently used in data mining applications. Our predictor learns from previous versions of the project. We have observed that the pd and pf rates are appealing enough so that the refactoring prediction models as a tool, help developers in the maintenance and redesign phase. Therefore, it is worth using a learning mechanism that would help to learn from history and previous developments.

In defect prediction studies, the effect of working with real historical defect data is important. However, in refactoring studies it is seen that, accessing to real refactored data is almost impossible due to the lack of documentation. Researches collect the refactor data with manual code inspection, comments in the code or log reviews of the versioning systems. The data that we have collected also not includes any refactor information of the classes. Also, it is very costly to extract this data by manual process. So we made some assumptions to construct refactor data. Our prediction model have same design in our all experiments, the only difference is the previously refactored module information. According to our results we can state the usage of the McCabe complexity metrics during the refactored data construction phase have positive effect on our learner both in class-level and file-level.

We also include new code metrics to our model to see the effect of these metrics in learning stage, thus prediction performance of the model. The code churn metrics are extracted from the versioning system in file-level and the experiments are conducted in six different ways: four of them are with the selected set of metrics in the refactor data construction phase, the other two is with and without code churn metrics. The file-level results are not very exiting. The class-level results are much better than file-level ones in the common 4 experiments based on assumptions 1-4. It can be due to the fact that the file-level information is not as homogenous as class-level information, since it consists of too many classes which have different characteristics. However, we observe that addition of code churn metrics have positive effect on our predictor's performance.

Final analysis of our experimental results is related with the effect of our model to the software project. We analyze how much cost and time we can save by using our proposed

learner by making a cost/ benefit analysis. We are suggesting to developers the classes that need to be refactored with a good probability of detection, however false alarm rates still exist and we may mislead the developers to the code portions that are not in need of any structural change. According to our analysis we can state that our prediction model is worth enough to be used in software projects due to its appealing results. Therefore, the maintenance phase and the cost of the project can be reduced with the usage of prediction models.

6.6. Threats to Validity

Threats to validity is examined according to four aspects, such as internal, external construct and statistical conclusion validity. In this research, the challenges that we face are examined in two groups: external and construct validity. We eliminate internal and conclusion validity by repeating our experiments both in class and file level. Also, each set of selected metric groups are examined independently in order to not to be influenced by each other.

Our concern for external validity is the use of limited number of datasets. We used four complex projects and their nine versions. To overcome ordering effects we used 10-fold cross validation. We have applied t-test with $\alpha=0.05$ (the results are given in Appendix C) in order to determine the statistical significance of results. All implementations are done in MATLAB environment.

The threat for construct validity is that GSM projects do not have refactor data so that we could trace which metrics directly affect the refactoring decision. Generally refactoring is applied to a software object to decrease the complexity of the system. However, we do not know the effects of each metric on refactoring decision. So, in order to produce refactor data we made six assumptions (assumptions 1, 2, 3 and 4 are used in class-level and assumptions 1, 2, 3, 4, 5 and 6 are used in file-level refactor data collection) and use them to collect refactor data.

We compare the values of the selected group of metrics for each module among the versions of the projects in order to collect refactor data. The comparison is done, based on

the name of the modules which are collected with Prest tool [46]. During a version upgrade if a Rename type refactoring is applied to a module, then we could not compare the metric values because of the change in module names. Rather our proposed approach makes comparison on the basis of module names. Developers in GSM confirmed that the rename type refactoring is generally applied on the method or variable names not class or file level. We may explore the effects of rename type of refactoring on our oracles in the future.

7. CONCLUSION

We have proposed a prediction model based on Weighted Naive Bayes approach for refactoring prediction. This model provides additional information with the usage of static code attributes and code churn metrics to determine the possible refactoring of a software system during pre-maintenance phase.

In this chapter we discuss our contributions to the refactoring prediction literature. Finally we will express our possible future directions for improving our research.

7.1. Contribution

Software developers and the architects make refactoring decisions based on their years of experience. Therefore refactoring process becomes highly human dependent, subjective and costly. Process automation and tool support can help reduce this overhead cost as well as increase consistency, efficiency, and effectiveness of the code reviewing and refactoring decision process.

We addressed the problem as a machine learning problem and we used Weighted Naïve Bayes with InfoGain weighting heuristic for predicting the candidate refactorings of classes. We used 13 data sets from GSM5, GSM7, GSM4 and GSM8 projects and run our model on these data sets. We have seen that our algorithm works better in terms of higher pd rates and lower pf rates as it learns. We have seen that using oracles to predict which classes to refactor considerably decreases the manual effort for code inspection (note that avg pf for all projects is 16%, 23%, 20% and 16% based on for each assumption), identifies the complex and problematic pieces of the code and hence makes the maintenance less costly and trouble free process. After making a cost/ benefit analysis, we see that our model brings on the average 83%, 78%, 82% and 82% cost savings in maintenance effort compared to manual code inspection.

We also answered our research question that we asked at the beginning of this research and given in section 3. We implement a predictor which can guide developers to

refactor which code segments thus finds the candidate refactorings automatically rather than manual inspection. According to our cost/ benefit analysis, it makes this prediction by eliminating the cost of the project at the redesign phase.

Our research has both academic and practical contributions. In doing this research, we propose a novel combination of the software attributes (static code attributes and code churn metrics) and design of software to make predictions about refactoring content of software modules and present an empirical study of refactoring prediction. In order to collect refactor data we propose 6 assumptions and we suggest an algorithm based on these assumptions. We also propose a learning based model, which uses Weighted Naïve Bayes with InfoGain weighting heuristic for the prediction of the candidate modules for refactoring. Both class-level and file-level information is used and the effect of code churn metrics to our learner is examined.

There are practical contributions of this study that guide project managers and developers through process automation and resource allocation. Learning based refactoring prediction models can be used as a prediction tool for project managers to focus on the codes which should be redesigned first. It will reduce the time allocated to the redesign phase thus lead to a decrease in the projects costs. The number of experienced staff can be lowered with the help of the prediction model and they can be assigned to other parts of the software. The complex parts, which will increase the defect-proneness of the software, will be lowered or eliminated by such prediction models, thus will decrease the testing effort of the software. In a highly competitive market with tight deadlines and budget, those models guide developers to specific parts of the software so that they could effectively allocate their scarce resources.

7.2. Future Work

Our future direction would be to collect more refactor data and repeat our experiments. We think that our predictor is not limited with GSM projects; however we have to run on our experiments in other data sets to make our predictor more general.

To see the effect of code churn metrics on our predictor's performance that is based on class-level information, we consider extraction of these metrics in class-file in addition to file-level. Also, the metrics based on modified lines of code could not be extracted from the versioning system. So, we contemplate about a tool which can be used for this purpose.

In order to further lower the pf rates, it is better to make experiments with other heuristics such as Gain Ratio and Odds Ratio.

We will also expand our prediction model in order to handle the rename type refactoring. Since the refactor data is collected based on the comparison of id of the classes between the project versions, any rename type refactoring can not be caught. One way to handle this problem is to use real refactored data collected by the GSM company or it can be extracted from the comments or log files of the software, on the condition that the changes are logged.

Moreover, it is better to try our experiments with other implementation languages rather than Java.

8. APPENDIX A: SOFTWARE METRICS USED IN THIS THESIS

A complete list of static code attributes is presented in this appendix [25, 26]. These attributes can be categorized into two categories; base metrics and composite metrics.

Base metrics are directly extracted from the source code. These metrics are shown in Table A.1.

Composite metrics are calculated from the base metrics. These metrics are shown in Table A.2.

Table A. 1. Base Metrics

Base Metrics	Explanation
Branch Count	Number of branches in a given module.
Call Pairs	Number of calls to other functions in a module.
Condition Count	Number of conditionals in a given module.
Decision Count	Number of decision points in a given module.
Edge Count	Number of edges found in a given module.
Formal Parameter Count	Number of parameters to a given module.
Modified Condition Count	Every condition shown to independently affect a decision outcome.
Multiple Condition Count	Number of multiple conditions that exist within a module.
Node Count	Number of nodes found in a given module.
Operators	Total number of operators found in a module.
Operands	Total number of operands found in a module.
Unique Operators	Number of unique operators found in a module.
Unique Operands	Number of unique operands found in a module.
Executable of Lines of Code	Source lines of code that contain only code and white space.
Lines of Comment	Source lines of code that are purely comments
Lines of Code and Comment	Lines that contain both code and comment.
Blank Lines	Lines with only white space or no text content.
Lines of Code	As its name indicates, total number of lines in a module.

Table A. 2. Composite Metrics

Composite Metrics	Calculation
Halstead Vocabulary (n)	*n = number of unique operands + number of unique operators*
Halstead Length (N)	*N =operands + operators*
Halstead Volume (V)	*V = N*log(n)*
Halstead Difficulty (D)	*D = (unique operators / 2) * (operands / unique operands)*
Halstead Level (L)	*L = 1/D*
Halstead Programming Effort (E)	*E = V / L*
Halstead Error Estimate (B)	*B = V / S**
Halstead Programming Time (T)	*T = E / 18*
Cyclomatic Complexity - V(g)	*V(g) = edge count - node count + 2*num. unconnected parts in g*
Cyclomatic Density - Vd(g)	*V(g) / executable lines of code*
Decision Density - Dd(g)	*condition count / decision count*
Module Design Complexity - Iv(g)	*Iv(g) = call pairs*
Design Density - Id(g)	*Id(g) = Iv(g) / V(g)*
Normalized Cyc. Comp. - Norm V(g)	*Norm V(g) = V(g) / lines of code*

9. APPENDIX B: OBJECT-ORIENTED DESIGN METRICS

These metrics were proposed in [27]. Table B.1 contains the list of these metrics and their descriptions.

Table B. 1. Object-Oriented Design Metrics

Metric	Description
Weighted Methods per Class	Number of methods in a class
Depth of Inheritance Tree	The maximum length from the node denoting this class in the inheritance tree, to the root of the tree
Number of Children	Number of immediate subclasses of a class
Coupling between Objects	The count of the number of other classes to which a class is coupled
Response for a Class	Set of all methods called by a methods in a class
Lack of Cohesion in Methods	This is a count of method pairs whose similarity is 0

10. APPENDIX C: SIGNIFICANCE T-TESTS OF THE MODEL

Below, t-tests are presented for our experiments. The first three results represent the significance of *McCabe results* on *other group of metrics* for two performance measures, *pd and pf.* For each dataset, results of both class-level and file-level experiments are presented in Table C.1, C.2 and C.3. In these results if *McCabe* significantly dominates the others for the selected performance measure of the datasets, then it is indicated as 1, otherwise, it is indicated as 0. If none of the metric group is significant to each other, then it is marked as X. Class-level experiments are marked with CL, whereas file-level is marked with FL.

Table C. 1. T-tests for McCabe vs. LOC

		Pd	pf
GSM5	CL	1	1
	FL	1	1
GSM7	CL	1	1
	FL	1	X
GSM4	CL	1	1
	FL	X	X
GSM8	CL	1	X
	FL	1	1
TOTAL	wins	7	5
	ties	1	3
	losses	0	0

Table C. 2. T-tests for McCabe vs. Halstead

		Pd	pf
GSM5	CL	1	1
	FL	1	X
GSM7	CL	X	X
	FL	1	1
GSM4	CL	1	1
	FL	X	1
GSM8	CL	1	1
	FL	1	X
TOTAL	wins	6	5
	ties	2	3
	losses	0	0

Table C. 3. T-tests for McCabe vs. subsetted Halstead

		Pd	pf
GSM5	CL	1	1
	FL	1	1
GSM7	CL	X	X
	FL	1	X
GSM4	CL	1	1
	FL	1	1
GSM8	CL	X	1
	FL	X	1
TOTAL	wins	5	6
	ties	3	2
	losses	0	0

The next result represents the significance of *class-level results* on *file-level results* for two performance measures, *pd and pf.* For each dataset, results are presented in Table C.4.

Table C. 4. T-tests for Class-Level vs. File-Level

	pd	pf
GSM5	1	X
GSM7	1	1
GSM4	1	1
GSM8	X	1
Avg	1	1

The last result represents the significance of file-level *results with churn metrics* on *results without churn metrics* for *pd and pf.* The results are given in Table C.5.

Table C. 5. T-tests for with vs. without churn metrics

	pd	pf
GSM5	1	1
GSM7	1	1
GSM4	1	1
GSM8	X	1
Avg	1	1

REFERENCES

1. Turhan, B., "Improving the Performance of Software Defect Predictors with Local and Remote Information Resources", *PhD Thesis*, Computer Engineering, Bogazici University, 2008.

2. Belady, L. and M. Lehman, *Program Evolution Processes of Software Change*, Academic Press, 1985.

3. Mens, T. and T. Tourwé, "A Survey of Software Refactoring", *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126-139, 2004.

4. Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2001.

5. Mens, T., T. Tourwé,and F. Muñoz, "Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring", *Proceedings of the International Workshop on Principles of Software Evolution*, Vol. 10, pp. 39-44, 2003.

6. Stroulia, E. and R. Kapoor, "Metrics of Refactoring-based Development: An Experience Report", *Proceedings of the 7 th International Conference on Object-Oriented Information Systems*, Vol. 3, pp. 113-122, 2001.

7. Turhan, B., Y. Kosker and A. Bener, "An Expert System for Determining Candidate Software Classes for Refactoring*", Expert Systems with Applications Journal*, Vol. 36, No. 6, pp. 10000-10003, 2008.

8. Ford, L., "Artificial Intelligence and Software Engineering: A Tutorial Introduction to Their Relationship", *Artificial Intelligence Review*, Vol. 1, No. 4, pp. 255-273, 1987.

9. Menzies, T., J. Greenwald and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, pp. 2-13, 2007.

10. Turhan, B. and A. B. Bener, “Software Defect Prediction: Heuristics for Weighted Naive Bayes”, *ICSOFT 2007*, Vol. 2, pp. 244-249, 2007.

11. Turhan, B., and A. Bener, “A multivariate analysis of static code attributes for defect prediction”, *In Proceedings of the Seventh International Conference on Quality Software*, Vol.7, pp. 231-237, 2007.

12. Koru, A. G. and H. Liu, “Building effective defect-prediction models in practice Software”, *IEEE*, Vol. 22, No. 6, pp. 23-29, 2005.

13. Koru, A. G. and H. Liu, “An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures”, *Proceeding of PROMISE 2005*, Vol. 15, pp. 1-5, 2005.

14. Khoshgoftaar T. M. and N. Seliya, “Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques”, *Empirical Software Engineering,* Vol. 8, No. 3, pp. 255-283, 2003.

15. Menzies, T., J. Greenwald and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, Vol. 33, No.1, pp. 2-13, 2007.

16. Menzies, T., B. Turhan, A. Bener, and J. Distefano, “Cross- vs within-company defect prediction studies”, *Technical report, Computer Science*, West Virginia University, 2007.

17. Munson, J. and T. M. Khoshgoftaar, “Regression modeling of software quality: empirical investigation”, *Journal of Electronic Materials*, Vol. 19, No. 6, pp. 106-114, 1990.

18. Munson, J. and T. M. Khoshgoftaar, “The Detection of Fault-Prone Programs”, *IEEE Trans. on Software Eng.,* Vol. 18, No. 5, pp. 423-433, 1992.

19. Tosun, A., B. Turhan and A. Bener, “Ensemble of Software Defect Predictors: A Case Study”, *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08 Short Paper),* pp. 318-320, 2008.

20. Turhan, B., A. Bener, “Data Sampling for Cross Company Defect Predictors”, *Technical Report*, Computer Engineering, Boğazici University, 2008.

21. Giombetti, M., *Cost/Benefit-Aspects of Software Quality Assurance*, Master Seminar Software Quality, 2007.

22. DEPARTMENT OF THE AIR FORCE Software Technology Support Center, *Guidelines for Successful Acquisition and Management of Software-Intensive Systems,* Version 3.0, May 2000.

23. IEEE Standards Association, *IEEE Standard Glossary of Software Engineering Terminology,* http://standards.ieee.org/reading/ieee/std/se/610.12-1990.pdf, 2007.

24. Welker, K. and P. W. Oman, “Software maintainability metrics models in practice”, *Journal of Defense Software Engineering*, Vol. 8, No. 11, pp. 19-23, 1995.

25. McCabe, T., “A Complexity Measure”, *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, 1976.

26. Halstead, M.H., *Elements of Software Science*, Elsevier, New York, 1977.

27. Chidamber, S.R. and C.F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493, 1994.

28. Nagappan, N. and T. Ball, "Explaining Failures Using Software Dependencies and Churn Metrics", *In Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, 2008.

29. Zhao, L. and J. H. Hayes, "Predicting Classes in Need of Refactoring: An Application of Static Metrics", *In Proceedings of the Workshop on Predictive Models of Software Engineering (PROMISE), associated with ICSM 2006*, 2006.

30. Hayes, J. and L. Zhao, "Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems", *IEEE 21th International Conference on Software Maintenance*, pp. 601-604, 2005.

31. Stroggylos, K. and D. Spinellis, "Software Quality", *WoSQapos;07: ICSE Workshops 2007*, pp. 1-6, 2007.

32. Weißgerber, P. and S. Diehl, "Identifying Refactorings from Source-Code Changes", *21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pp. 231-240, 2006.

33. Sommerville, I., *Software Engineering (Eighth Edition)*, Addison-Wesley, 2006.

34. Turhan, B. and A. Bener, "Weighted Static Code Attributes for Software Defect Prediction", *In Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08)*, pp.143-148, 2008.

35. Fenton, N. E. and M. A. Neil, "A critique of software defect prediction models", *Software Engineering: IEEE Transactions*, Vol. 25, No. 5, pp. 675-689, 1999.

36. Fenton, N. E., "A Critique of Software Defect Prediction Models", IEEE Transaction on Software Engineering, Vol.25, No. 5, pp. 675-689, 1999.

37. Alpaydin E., *Introduction to Machine Learning*, MIT Press, 2004.

38. Ferreira, J. T. A. S., D. G. T. Denison and D. J. Hand, "Weighted naive Bayes modelling for data mining", *Technical Report*, Imperial College, 2001.

39. Giles, K., K. M. Bryson and Q. Weng, "Comparison of Two Families of Entropy-Based Classification Measures with and without Feature Selection", *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.

40. Kosker, Y., A.Bener and B. Turhan, "Refactoring Prediction Using Class Complexity Metrics", *ICSOFT 2008*, pp. 289-292, 2008.

41. Turhan, B., A. Bener and Y. Kosker, "Tekrar Tasarım Gerektiren Sınıfların Karmaşıklık Ölçütleri Kullanılarak Modellenmesi", *UYMK 2008*, pp. 107-114, 2008.

42. Boetticher, G., T. Menzies, and T. Ostrand, PROMISE Repository of empirical software engineering data http://promisedata.org/ repository, West Virginia University, Department of Computer Science, 2007.

43. Helsel, D. R. and S. J. Ryker, "Defining Surfaces for skewed, highly variable data: Environmetrics", Vol. 13, pp. 445-452, 2002.

44. Heeger, D., Signal Detection Theory, http://www.cns.nyu.edu/~david/handouts/sdt/sdt.html, 1998.

45. Simon, F.S. and C. Lewerentz, "Metrics based refactoring", *Proceedings of European Conference on Software Maintenance and Reengineering*, pp. 30-38, 2001.

46. PREST: Metric Collection Tool Developed at Softlab, Dept. of Computer Engineering, Boğaziçi University, Available from: http://softlab.boun.edu.tr/, 2009.

47. Mitchell, M. and J. Jolley, *Research Design Explained*, New York:Harcourt, 2001.

48. Arisholm, E. and L. Briand, “Predicting fault-prone components in a java legacy system”, *International Symposium on Empirical Software Engineering archive Proceedings of the 2006*, pp. 1-22, 2006.

49. Khoshgoftaar, T.M., E.B. Allen, J.P. Hudepohl and S.J. Aud, “Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System”, *IEEE Transactions on Neural Networks*, Vol. 8, No. 4, pp. 902-909, 1997.

50. Zhang, H. and S. Sheng, “Learning weighted naive Bayes with accurate ranking”, *In Proceedings of the 4th IEEE International Conference on Data Mining*, Vol. 1, No. 4, pp. 567- 570, 2004.