An Effort Prediction Framework for Software Defect Correction

by

Alaa Hassouna

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Developers apply changes and updates to software systems to adapt to emerging environments and address new requirements. In turn, these changes introduce additional software defects, usually caused by our inability to comprehend the full scope of the modified code. As a result, software practitioners have developed tools to aid in the detection and prediction of imminent software defects, in addition to the effort required to correct them. Although software development effort prediction has been in use for many years, research into defect-correction effort prediction is relatively new. The increasing complexity, integration and ubiquitous nature of current software systems has sparked renewed interest in this field. Effort prediction now plays a critical role in the planning activities of managers. Accurate predictions help corporations budget, plan and distribute available resources effectively and efficiently. In particular, early defect-correction effort predictions could be used by testers to set schedules, and by managers to plan costs and provide earlier feedback to customers about future releases.

In this work, we address the problem of predicting the effort needed to resolve a software defect. More specifically, our study is concerned with defects or issues that are reported on an Issue Tracking System or any other defect repository. Current approaches use one prediction method or technique to produce effort predictions. This approach usually suffers from the weaknesses of the chosen prediction method, and consequently the accuracy of the predictions are affected. To address this problem, we present a composite prediction framework. Rather than using one prediction approach for all defects, we propose the use of multiple integrated methods which complement the weaknesses of one another. Our framework is divided into two sub-categories, *Similarity-Score Dependent* and *Similarity-Score Independent*. The *Similarity-Score Dependent* method utilizes the power of Case-Based Reasoning, also known as Instance-Based Reasoning, to compute predictions. It relies on matching target issues to similar historical cases, then combines their known effort for an informed estimate. On the other hand, the *Similarity-Score Independent* method makes use of other defect-related information with some statistical manipulation to produce the required estimate. To measure similarity between defects, some method of distance calculation must be used. In some cases, this method might produce misleading results due to observed inconsistencies in history, and the fact that current similarity-scoring techniques cannot account for all the variability in the data. In this case, the *Similarity-Score Independent* method can be used to estimate the effort, where the effect of such inconsistencies can be reduced.

We have performed a number of experimental studies on the proposed framework to assess the effectiveness of the presented techniques. We extracted the data sets from an operational Issue Tracking System in order to test the validity of the model on real project data. These studies involved the development of multiple tools in both the Java programming language and PHP, each for a certain stage of data analysis and manipulation. The results show that our proposed approach produces significant improvements when compared to current methods.

Acknowledgements

It is first on to Allah I must acknowledge the patience and perseverance bestowed on me in pursuit of my Master's studies.

I would like to confer my sincerest gratitude to Professor Ladan Tahvildari for all her support and guidance. I would further like to express my utmost appreciation to Professor Amir Khandani for sharing his invaluable wisdom and direction.

I would also like to thank Professor Otman Basir and Professor Fakhri Karray for accepting to be members of my dissertation committee. I must thank them for taking the time out of their busy schedules to review my thesis. I also thank Professor Kostas Kontogiannis for attending my seminar and for his insightful comments and suggestions.

I am also grateful to the members of the Software Technologies Applied Research group for their support and cooperation.

Last but not least, I would like to thank my family for their patience, understanding and encouragement. Which provided me with the motivation to pursue my graduate studies.

Dedication

This is dedicated to the one I love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past few decades, the use of software has become widespread throughout systems ranging from simple mobile devices to multimillion-dollar defence systems. It has become an integral part of our economy, military and government operations. This called for studies into a better understanding of software through developing Software Process Models [83]. Software Process Models, also known as Software Development Process Models, are used to describe the structure of the development process in addition to that of the software itself. These models, accompanying some statistical analysis, have been used to predict the effort required to develop and maintain software systems. Many models such as COCOMO [10] and Neural Networks [9, 29, 47, 73] have become widely spread in the software community as effective prediction tools. Consequently, effort prediction has become an important tool in corporate management's arsenal, especially for today's multinational projects. Accurate predictions allow management to plan, budget and notify customers of the expected delivery dates.

In software development, the maintenance phase constitutes 60% to 70% of the software development life cycle [12]. It involves making changes to various software modules, documentation and sometimes even hardware to support the systems operational effectiveness. Some of these changes are required to improve a systems performance, correct problems, enhance security, or address user requirements. Therefore, to ensure such modifications do not disrupt operations or the integrity of the system, organizations employ appropriate change management procedures and standards. Once again, effort prediction proves to be one of the important tools which can help in planning and executing these procedures effectively.

In this work, we address the problem of predicting effort for entries in Issue Tracking Systems early in their lifetime. These systems are used to manage the different issues and defects that arise during the maintenance phase (an issue could either be a *bug*, a *feature request* or a *task*). Such early estimates could be used by testers to set schedules, and by managers to plan costs and provide earlier feedback to customers about future releases. Predicting defect-correction effort (the effort

needed to fix a reported issue or bug) is a more challenging task than predicting software development effort. While software development is a construction process, defect-correction is mainly a search process possibly involving all of the program's code [82]. Furthermore, testers cannot trust the original developers' assumptions and conditions [82]. This may require them to explore areas of the code that they are not familiar with; adding to the complexity of the process.

The heart of our approach to predicting defect-correction effort, is an Instance-Based Reasoning [67] method called the *Nearest Neighbour* approach [31, 67]; inspired by the method proposed by Weiss *et al.* in [82]. It leverages experience from resolved issues to predict correction-effort for similar emergent issues early in their lifetime. Our proposed framework implements four key enhancements to the *Nearest Neighbour Base Approach*: *Data Enrichment*, *Majority Voting*, *Adaptive Threshold* and *Binary Clustering*. In addition to the information used to form the text-similarity query in the *Base Approach*, *Data Enrichment* injects additional issue information collected from the Issue Tracking System. This aims to increase the accuracy of the similarity scores. The mean prediction method used in the *Base Approach* is replaced by *Majority Voting*. Since effort values are usually taken from a distinct set; *Majority Voting* capitalizes on the fact that certain values often appear more frequently in similar historical matches, which are close to the actual effort. *Adaptive Threshold* allows the model to compute estimates by considering higher scoring matches first, which should yield better results. If no matches are found at a higher threshold, it is systematically decreased until the required number of matches is reached. In some cases, similarity scores could be considered too low, at which point it might be misleading to use them. *Binary Clustering* alleviates this problem by using common properties of issues to form clusters (independent of the similarity scores), which are then used to produce the predictions.

Most of the existing work addressing defect-correction effort prediction use a single approach to generate predictions. For example, to the best of our knowledge, no work has used both the *Nearest Neighbour Approach* with *Clustering*, or *Clustering* with *Regression*. We can see many works comparing the different approaches, but not using them in a composite approach. We propose using complimentary approaches that help address the weaknesses of one another. In addition to employing the *Nearest Neighbour Approach*, we use *Binary Clustering* to address cases where the *Nearest Neighbour Approach* would produce misleading results. In this context, one of our goals is to show how such composite systems can be used to effectively produce accurate predictions, and to demonstrate how enhancements can be applied to existing systems.

The next section will describe the problem that this work is trying to address, with an illustration and a generalization of the components involved. In addition, it will provide an example of the problem, to give the reader a better understanding of the interactions between the components.

1.1 Problem Description

We address the problem of predicting the effort needed to correct an issue posted on an Issue Tracking System (or any defect repository). We define an issue as being any of: *Bugs*, *Tasks* or *Feature Requests*. A *Bug* represents any defect or fault reported against the software system, which impairs or prevents it from functioning properly. *Tasks* are usually development or assertion activities set by developers to achieve certain goals. A *Feature Request* is any request for a new feature which is not yet included in the system. In literature, the term "Defect" usually includes all of the above types of issues, which is how we will use it for the remainder of this study. Our method measures effort in "man-hours", which is often the standard for defect-correction approaches.

To perform the prediction for a given issue (whether it is a *Bug*, a *Task* or a *Feature Request*), we present the *Effort Prediction System*. There are three major components involved in our defect-correction effort prediction approach (*Effort Prediction System*): *Target-Issue Information* (T), *Historical Issues Information* (H) and the *Prediction Model* (P) (see Figure 1.1). The *Target-Issue* is the issue for which we are required to make the effort prediction. The *Historical Issues* is the set of all the existing (resolved) issues we use to derive our experience from; i.e. training issues. Finally, the *Prediction Model* is the method we use compute the predictions based on the information we were given. Referring to Figure 1.1, we can formalize the problem as follows: given T, a vector containing M distinct variables t_i each describing a certain property of T (*Project Name, Priority, Type,... etc*), H, a matrix of size $N \times M$, where H_{ij} is the j^{th} property of the i^{th} issue. Produce an estimate $\hat{E}$ that is within an acceptable distance of the actual effort.

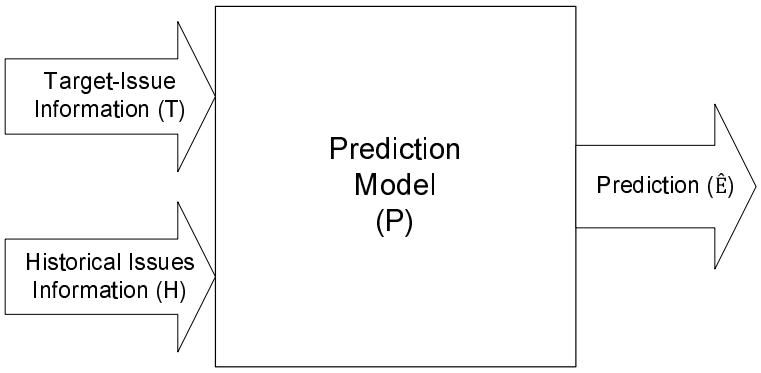

Figure 1.1: A Depiction of an Effort Prediction System.

We use H to learn the behavior, extract rules and identify similarity patterns from resolved issues. This is equivalent to an expert's knowledge; while the expert might have gained that experience in many years, the prediction model P tries learns it in a much smaller interval. T is the information describing the issue for which we need to make the prediction. P will use the experience it gained from H, to produce a prediction for T.

To facilitate a better understanding of the problem, we give an example to show how the above components are used and what each variable means. Let T be of size $M = 3$, where T =[*JBAS, Blocker, Bug*] referring to properties (*Project Name, Priority* and *Type*) in respective order. Let H be a matrix of size $N \times M$, where $N = 2$, meaning we have two issues in our history that we can use to derive the estimate from (H =[*JBAS, Major, Bug* ; *JAXR, Minor, Task*]). Then P would use H to derive an informed estimate $\hat{E}$, based on some criteria set by the prediction method used by P.

In the next section, we will outline the major contributions of the work presented in this thesis. They will provide a better perspective of what the reader should expect out this work, and what goals we are trying to achieve.

1.2 Thesis Contributions

The major contribution of the thesis is to address the problem of defect-correction effort prediction for issues posted on an Issue Tracking System. Our approach proposes the application of an effort prediction framework, combining a *Nearest Neighbour* approach with a *Binary Clustering* approach to predict the effort needed to fix a certain defect. The framework is given information about a target issue and a set of resolved issues, in addition to some specifications and properties, at which point it will generate a correction-effort prediction measured in "man-hours". The proposed effort prediction framework aims to provide a set a improvements to the existing *Nearest Neighbour Base Approach* as follows:

- Develop a way to improve the accuracy of the similarity scoring mechanism, by introducing additional data related to the target issue into the process.
- Design an improvement to the prediction technique by utilizing properties of the data from the defect repository. By introducing an alternative prediction method to the mean (as used by the *Base Approach*), the proposed framework uses the repeating historical effort values more effectively. Since these values are observed to be close to the actual effort, devising a better way to capitalize on this observation will increase the accuracy of the predictions.
- Enhance the practical application of the framework by improving the percentage of issues for which the model makes predictions; achieved by automatically adding more issues into the set of similar historical matches.
- Introduce an approach that does not depend on similarity scores, which helps improve predictions if the scores are deemed misleading. By using additional common information about issues, predictions can be produced independent of the misleading low similarity scores.
- Develop an extensible effort prediction framework, which can be adapted and used with any defect repository to produce accurate effort predictions.

Components can be added or modified to fit the needs of the users and the specific repository in use.

The next section will outline the organization of the remainder of this thesis. It will describe the contents and sections of each chapter, and the general topic of each.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents a survey of the related works. In the first section, it discusses literature in the area of software development effort prediction in general. Then it dedicates another two sections to describing the areas of *Defect Prediction* and *Defect-Correction Effort Prediction*. *Defect Prediction* is the science of predicting the number and/or severity of defects that will affect the system in the future, based on the current software system's properties. *Defect-Correction Effort Prediction* refers to the science of predicting the effort needed to resolve or fix a reported issue in a software system. These two related fields of study are sometimes used in conjunction to predict the effort and costs needed to maintain a software system. Finally, the last section presents a survey of literature in the area of *Similarity-Based Prediction*, exploring different disciplines and domains that also use this method to make predictions.

- Chapter 3 presents the *Base Approach* on which we build our proposed effort prediction framework. First it describes the scoring technique used to generate the similarity scores (used as the distance measure between the issues). Then *Nearest Neighbour* approach is described along with the computation method used to produce the predictions. Next, it describes a more advanced alternative *Nearest Neighbour with Thresholds* approach to predicting effort. Finally, it presents a general process flow diagram combining the different components of the *Base Approach* together, and introducing the new components of the effort prediction framework; leading the reader into the next chapter.

- Chapter 4 describes the effort prediction framework in more detail, along with the underlying components. First it describes the process model of the effort prediction framework in greater detail. Then it describes some weight computation techniques, which are used by the framework in various places to produce predictions. The effort prediction framework components are broken down into two categories: Similarity-Score Dependent Enhancements (*SSDE*), and Similarity-Score Independent Enhancements (*SSIE*. *SSDE* include *Data Enrichment*, *Majority Voting* and *Adaptive Threshold*, while *SSIE*

include *Binary Clustering*. Each enhancement is described in detail and the advantages of each is given. Finally, the a simplification of the effort prediction framework implementation algorithm is presented, combining all of the proposed enhancements and the appropriate components from the *Base Approach*.

- Chapter 5 describes the experimental studies and the evaluation results. It describes the Issue Tracking System and the data sets we extracted from it. Then presents an overview of the implementation tools that were used to perform the experimental studies along with some implementation details. Then the evaluation method and evaluation metrics are described, in addition to a discussion of the obtained results. Finally, a summary is presented to outline what we have learned.

- Chapter 6 presents the thesis contributions, outlines future directions and ends with some concluding remarks.

Chapter 2

Related Works

Software development effort prediction has been studied for a few decades now. It has become an integral part of management's arsenal of planning tools. The increasing growth and complexity of software systems today has pushed management into adopting more structured and informed ways of estimating development effort, namely prediction models. Many prediction models have been developed in an attempt to improve development effort estimation, each with its own slightly different take on the software process model [11, 14, 25, 41, 44, 53, 80]. Due to the complexity of the software development process, thus far no one model can address all of the variability involved. For example, during the course of software evolution, developers may add new features to address emergent customer needs or may need to fix existing defects. In turn, these modifications add to the uncertainty of the process by introducing some unaccounted-for side effects. It is for this reason that some software practitioners have turned to empirical models for effort estimation [3, 10, 39, 59, 71, 73]. Empirical models allow practitioners to overlook any assumptions about the underlying processes. Thereby partially relieving the effect of the software community's incomplete understanding of the global process structure. However, global and analytical models are still useful and are usually quite accurate if calibrated to local environments. Models like COCOMO [10] and SLIM [63] have been used in practice for many years.

The remainder of this chapter will describe the different effort prediction methods used in literature for software development and defect-correction. Section 2.1 presents a general categorization of the most common prediction methods used for software development effort, in addition to discussing some advantages and disadvantages of each. Sections 2.2 and 2.3 discuss more related topics to our work, namely defect prediction, defect-correction effort prediction and similarity-based prediction methods. In particular, Section 2.2 outlines the different work done in the fields of defect prediction and defect-correction effort prediction, outlining some of the problems in current literature. On the other hand, Section 2.3 describes the work done in the area of Similarity-Based predictions across multiple disciplines and academic domains. Since the literature in the area of defect-correction effort

prediction is very limited, we expand our literature search outside the software domain.

2.1 Effort Prediction Methods

In this section, we outline the different methods commonly used in literature for software development effort prediction. The following is a generic categorization of these methods, both empirical and theoretical:

- **Parametric Models:** The functional form of these models is based on theory or experimentation [11]. SLIM [63], CHECKPOINT [40], COCOMO [10], PRICE-S [60], ESTIMACS [66] and the SELECT Estimator [70] are some of the well established parametric models which were extensively used in the past few decades. They are built from the authors' understanding and extensive studies of the software process. Unfortunately, since most of these models are proprietary, they cannot be compared in terms of model structure. The advantages of *Parametric Models* is that they are based on years of professional experience, and are usually developed by dedicated teams that study the many aspects of the software development life cycle. These studies also benefit from an abundant availability of data, since it is usually provided by the sponsoring private companies (rather than public institutions). However, the fact that this data is usually provided by one or few institutions, the results can be misleading. The models can suffer from being too specific to the data provided to them. To address this problem, some models like COCOMO provide a local calibration mechanism to fit the model to the patterns in the new data set.

- **Learning Models:** Learning-oriented techniques either attempt to build models that automatically learn from previous experiences, or use similar historical case studies to produce an estimate. The former attempts to construct a model that describes the relationship between dependent and independent variables of the underlying process. Neural networks are a common example of such a model, and are used by many practitioners to predict software effort [9, 29, 47, 69, 73]. The latter (Also known as Case-Based Reasoning [1] or Instance-Based Reasoning [67]), however, tries to find a range of well documented similar historical cases. In turn, these cases are used (along with their recorded effort) as input into certain justified statistical models to produce an estimate. Jorgensen *et al.* use Case-Based-Reasoning to predict software development effort [43]. However, they use an influencing concept for extreme values known as "Regression toward the Mean", which adjusts estimates towards values of more average projects. Shepperd *et al.* also use Case-Based Reasoning to predict software development effort [71]. In addition, they make use of stepwise regression analysis yielding higher accuracy estimates. Regression-Based techniques could be considered a special class

of learning models, namely statistical learning. They are extensively used in literature due to their simplicity, local adjustability and relatively accurate results. Regression analysis is commonly used in conjunction with parametric techniques to calibrate the unknown parameters. The calibration is performed according to some criterion such as Least Squares Error or Mean Magnitude of Relative Error, as described in [7, 45, 81]. Recently, the application of Artificial Intelligence in the effort prediction field has become increasingly popular. Models like Neural Networks [13] and Genetic Algorithms [15] are proving to be effective due to their adaptive nature. Some very informative Artificial Intelligence model comparisons have been written, such as Bibi *et al.*'s book "Artificial Intelligence Applications and Innovations" [8] and the work done by Tronto *et al.* [80]. Bayesian probabilistic models are another interesting "learning" approach that has been applied to software effort prediction such as the one used by Pendharkar *et al.* in [61]. All these methods use some form of software complexity and size metrics in order to predict effort. Some studies like [58] and [32], try to develop better metrics which either relate to effort more closely or represent the complexity of software more accurately. One advantage of using a learning models to predict effort, is that they can extract rules from the data directly, without requiring additional previous assumptions from the user. This saves practitioners from trying to develop analytical models that can account for the software behavior. However, since these models rely solely on data, any deviation in the learning data set will affect the performance and therefore the accuracy of the model. Therefore, extra care needs to be taken when training these models, by ensuring that the data reflects the real behavior of the software. Techniques like pruning can be used achieve this, by eliminating outliers, we can limit their effect on the performance of the model. Another advantage of using Instance-Based Reasoning methods, is that they perform very well even when data is limited. However, similarity (distance) calculation methods play a big role in the accuracy of this method, and therefore the appropriate calculation method must be implemented to measure similarity as accurately as possible.

- **Expert-Based Models:** Expert-Based techniques rely on the knowledge and experience of practitioners within a domain of interest. The experts provide estimates by synthesizing a set of known outcomes of all past projects known to the expert. This model is useful when there is a lack or shortage of quantified, empirical data. However, since these models rely on expert judgement, they are prone to opinion bias. None the less, practitioners have used them with success and such are examples of these studies [10, 34]. Jorgensen *et al.* have done a great comparison study of different expert estimation techniques [41], in addition to practical guidelines for using such methods [42]. One of the more popular expert-based methods is the Delphi Technique [16]. It is used to guide a group of experts to a consensus of opinion on a particular issue, such as the effort needed to complete a software project. It takes the experts through a number of rounds, where they discuss their estimates and

give their opinions regarding the topic at the end of the round. This technique tries to alleviate the problem of opinion bias in expert-based judgment by ensuring that a panel of expert agree on the estimate, rather than taking the estimate of one individual. However, this technique still suffers from the fact that the decisions made are based more on expert opinion rather than hard empirical data.

- **Composite Models:** Composite models combine two or more techniques to formulate the most appropriate functional form for estimation [11]. For example, some models like COCOMO combine parametric alongside statistical learning techniques such as regression analysis to achieve accurate results. COCOMO uses regression to locally calibrate the different variables to the specific development environment. Composite models are an effective estimation tool; the different methods incorporated into the model allow for greater adjustment flexibility. If the methods are used correctly, they can complement the weaknesses of each other. Something that single models usually lack. However, if the combined methods are not compatible, or could be considered too similar, then such composite models do not benefit from the complementary nature that practitioners seek in these methods.

We have presented a number of software development effort prediction techniques, with an overview of some the advantages and disadvantages of each. The lesson we can take from this summary is that no one technique can be considered superior to all others. The key to getting accurate predictions is to use a variety of techniques and tools and understand why the results might differ from one method to another. If the practitioner understands the differences between the methods in use, then they are more likely to have a good grasp of the underlying costs and factors that affect the development of the project. Thus, making them better equipped to take part in the planning and budgeting undertaken by management.

In the next chapter, we will outline the different defect prediction methods used in literature, in addition to discussing the various defect-correction effort prediction approaches.

2.2 Defect-Correction Effort Prediction

Although our research is in the area of defect-correction effort prediction, it is beneficial to review and understand defect prediction methodologies. This allows us to better understand defects, their properties and how they can be controlled and tracked. In turn, this will help us devise better defect-correction effort prediction methods.

Organizations want to predict faults or defects in software systems before they are deployed in the field, to gauge their quality, stability and effort required to maintain them. A wide range of models have been developed for this purpose, and

most usually rely on complexity and size metrics to predict defects [4, 17, 48, 56]. Others may use a quality measure of the software development process to assess the system [18, 19], or sometimes multivariate analysis [4, 24]. There are also some novel approaches that do not quite fit the commonly known approaches. For example, in [84], the authors describe a method of fitting empirical data to mathematical functions in various ways, yielding promising results. Also, Padberg *et al.* propose the use of Neural Networks coupled with nonlinear regression to estimate defect content in software systems. In [26], Fenton *et al.* provide an excellent review and comparison of the various defect prediction models in literature.

The next step after defect prediction, is to employ some form of a defect-correction effort prediction to assess the time and resources needed to address them. It is interesting to note, that only recently has this area of research begun to attract attention. Again, we give credit to the increasing complexity of software interactions and the need for more formal methods of prediction. Although the number of literature addressing this area is limited, they are nothing short of pioneering works. In [85], Zeng and Rine use a self-organizing neural network approach to predict defect-correction effort. First, they cluster defects from the training set, then they compute the probability distributions of effort from the resulting clusters. These distributions are then compared to the defects of each instance in the test set to derive the prediction error. They use the NASA KC1 data set to evaluate their approach, but unfortunately their only performance measure was Magnitude of Relative Error. In literature, this metric has been believed to be asymmetric [27, 50], and the use of supporting measures is usually desired to avoid any doubts of validity. Additionally, when they apply their prediction method to an external data set (data set not used in training), their MMRE reaches scores of up to 159%, with a maximum MRE of about 373%. This shows that while their technique preformed well when applied to the training data set (with MMRE ¡ 30%), it is not reliably extendable to other data sets. On the other hand, Song *et al.* proposed the use of association rule mining to categorize effort into intervals [77]. To evaluate their approach, they use NASA's well known SEL defect data set. According to their findings, their technique outperformed other methods such C4.5, Naïve Bayes and PART. Another interesting approach was proposed by William Evanco in [23]. Using explanatory variables such as software complexity metrics, fault spread measures and the type of testing conducted; he develops a statistical model that can be used in conjunction with defect prediction methods to give estimated defect fix effort with certain a certain probability measure. In [22], he also proposes a related model using Poisson and multivariate regression analysis to model the effort. Finally, in [82], Weiss *et al.* propose a novel approach using an Instance-Based Reasoning methodology (a type of nonparametric learning method) called the "Nearest-Neighbour Approach" [31, 67]. They use a text-similarity approach to identifying nearest neighbours, which they use to generate informative effort predictions. They test their approach on real data extracted from an Issue Tracking System, by using an evaluation method that is time-line conscious to simulate a practical application of the model. Their model shows promising results,

with Average Absolute Residuals reduced down to about 6 Hours within the actual effort. In [52], Manzoor creates a list of pointers and guidelines for experts to follow when estimating defect-correction effort. He groups them by the type of defect and programming environment, such application architecture, programming practices and object-oriented design. These pointers can prove to be valuable if studied and applied effectively.

The above defect-correction effort prediction studies present various attempts at providing accurate predictions. However, they do not extend their experimental evaluation to additional data sets (with the exception of [85], which we have shown to have produced less than impressive results). This raises questions about the external validity of the proposed approaches. Many approaches can be calibrated to a specific data set, but only a small number can be extended to additional data sets. It is those methods that can be applied to additional external data sets, and still perform well, that can be used in practice. To extend the validity of our proposed effort prediction framework, we evaluate its performance on an additional data set, giving the reader greater confidence that our approach can perform well in practice. In addition, we use multiple performance metrics to assess the accuracy of the framework. Most performance metrics have some kind of bias [50], therefore we opt to using complimentary metrics that show the distribution of error for our predictions in a more comprehensive way.

As we can observe, the literature in the area of defect-correction effort prediction is limited (to the best of our knowledge). Therefore, the next section will explore similarity-based prediction methods from many other disciplines, ranging from medicine to economics. This will give us a better understanding of what techniques and tools are currently being employed in literature and practice, to help us make informed decisions for the implementation of our effort prediction framework.

2.3 Similarity-Based Prediction Techniques

Instance-Based Reasoning methods have been widely used in many domains and disciplines to produce predictions. For example, it has been used in medicine to predict drug toxicity [21] and in economics to predict financial distress of companies [78]. Due to the limited number of works in the area of defect-correction effort prediction, and especially for methods using Instance-Based Reasoning, we will expand our scope into these alternate disciplines. This will allow us explore, leverage and possibly apply foreign Instance-Based Reasoning models into the field of software defect-correction. Our summaries will mainly be concerned with the use of similarity to generate the prediction, rather than with the methods used to compute the similarities themselves. Although many papers do propose some interesting and novel ways of computing the similarities, this area of research is usually domain specific.

We can divide the literature into four major categories according to their method

of using historical similarities to compute an estimate: *Clustering*, *Top-K Regression*, *Top-K Mean* and *Top-K Majority Voting*. The following subsections will describe each in detail, in addition to outlining the different studies performed using each technique.

2.3.1 Clustering

First we will describe the use of *Clustering* to produce estimates. *Clustering* is an unsupervised learning technique which aims to find structure in a set of unlabeled data. It is the process of arranging or dividing objects into groups whose members are similar in some way [54]. Distance-based clustering groups objects into a cluster if these objects are considered *close* according to some distance measure. Conceptual clustering is another kind of clustering which groups objects that share a common concept; i.e. descriptive concept rather than a defined similarity measure [54].

In [62], Phansalkar *et al.* use k-means clustering (as described in [72]) to predict a software program's performance. They use five categories of benchmarks to measure similarity between programs: Instruction Mix, Behavior of branches, Inherent Instruction Level Parallelism, Data locality and Instruction locality. To reduce the dimensionality of data (29 metrics), they use Principal Component Analysis which transforms the data to a new coordinate system. They find the optimal number of clusters by using the Bayesian Information Criterion a described in [72]. Once they identify what cluster the target program goes into, they use the performance measure of the closest program to the center of the cluster as the prediction. Another interesting work is presented by Ben-Dor *et al.* concerning biological Tissue Classification [6]. They use the CAST clustering algorithm to classify tissues with gene expression profiles. To classify an unknown instance, they conduct a majority vote of the element labels in the matching cluster and use the winning label as the predicted classification. In order to test the performance of this clustering classification method, they compare it against a simple kNN approach (k = 1). Their results show that this method is successful in reliably predicting tissue types, and is comparable to existing methods in literature. The advantage of using *Clustering* to classify data and produce effort predictions is that it is a well established and relatively simple mechanism to perform. However, it relies on defining discrete labels to groups or clusters which limits the adaptability of the approach.

2.3.2 Top-K Regression

Regression analysis tries to model a dependent (response) variable (e.g. Effort) as a function of independent variables (e.g. Program Size), their corresponding parameters (constants) and an error term. The parameters are estimated to give a "best fit" of the data. Usually evaluated using the least squares method, although

other methods have been used [7, 81]. The error term is used to represent any unexplained variation in the dependent variable. For a good introduction to regression analysis please refer to [79]. The idea behind *Top-K Regression* is based on limiting the training data set to the most similar instances (candidates) only, as compared to using all of the history. This helps to alleviate the effect of outliers which may affect the model's accuracy. On the other hand, we need to compute a different equation for every target instance (instance for which we need a prediction), since the set of candidates is different. However, the candidate set is usually small and the computation is simpler and less time consuming.

Iwata *et al.* propose a top-30 multiple regression method to predict effort for embedded software development [37]. The use a collaborative filtering system to generate the similarity scores between projects. However, of the 73 projects they use for the evaluation of their approach, 53 of them were missing measurements for some of the metrics. They mitigate this problem by computing the missing values based on project similarities using a weighted mean of the similar projects' metric values. Now they can obtain a full matrix of projects and their corresponding metric values, in addition to the similarity vector relating the target project to each of the projects in the matrix. Then they choose the top 30 most similar projects according to score and scale/size, and use them to generate the regression model and the prediction. To evaluate their approach they compare it to regular multiple regression analysis and to collaborative filtering, concluding that their approach performs favorably. It is beneficial to mention that they use 5 performance metrics for evaluation: Mean Absolute Error, Variance of Absolute Error, Mean Relative Error, Variance of Relative Error and Rratio which they define as the inverse of PRED(15). In [86], Zhu *et al.* use a multiple linear regression model coupled with fuzzy membership evaluation to predict soil properties at specific locations [86]. Given a raster layer of the location, with each pixel represented by a similarity vector, they constrict a multiple linear regression model to predict the soil property value at a location using a regression between soil property and fuzzy membership values in each of the soil classes. They conclude that using this method is best suited for gentle landscapes where the relationships between soil property values and terrain attributes approach linear. Kapadia *et al.* also propose the use of *Top-K Regression* (which they call "Locally Weighted Polynomial Regression") to predict the run-specific resource-usage in a computational grid environment [46]. However, the results indicate that the *Top-K Mean* and the *Top-K Weighted Mean* methods outperform the more sophisticated "Locally Weighted Polynomial Regression". Smith *et al.* use three variants of the *Top-K Regression* model in an effort to improve run time (execution time) predictions for jobs that run on computational grids [74]. Computational grid systems rely on scheduling algorithms to provide queue wait times for jobs. In turn, scheduling algorithms rely on run time (execution time) predictions to update schedules and queue wait times. After experimentation, the authors conclude that the *Top-K Mean* method performed better than any of the three regression approaches (linear, logarithmic and inverse). Similarly, in [75], Smith *et al.* use *Top-K Linear Regression* to predict run times for applications

on computational grid systems. Yet again, their results suggest that *Top-K Mean* performs better than th e *Top-K Linear Regression* approach. Dymo also proposes the use of *Top-K Regression* to predict software development time [20]. However, his work does not show any empirical evaluation of the proposed method, making it difficult to assess its performance. *Top-K Regression* relies on the availability of additional independent variables that can be used as descriptors for the target issues. This limits its application to only data sets that provide such information.

2.3.3 Top-K Mean

Top-K Mean simply computes the mean of the measure (e.g. Effort) describing the Top-K candidates. Ebbles *et al.* suggest the use of Top-3 Mean to predict drug toxicity based on the similarity of their physiological effects on organisms [21]. They generate a multidimensional similarity matrix by using a pre-existing density estimation method called CLOUDS. Then they use the average of the top 3 candidates, chosen from the matrix, as their prediction. Weiss *et al.* also use *Top-K Mean* to predict the time needed to fix a particular software maintenance issue [82] (mentioned as well in Section 2.2). Nassif *et al.* use *Top-K Mean* to predict job completion times for applications running on grid computing environments [57].

Top-K Weighted Mean is also a common variant of this method, where a weight is assigned to each instance in the candidate set based on its similarity score. Li *et al.* propose using *Top-K Weighted Mean* to predict job response times for aid in large-scale grid resource-allocation scheduling [51]. They use application and resource state similarities to compute the distance scores. To assess the performance of *Top-K Weighted Mean*, they compare it to a simple nearest neighbour approach (where they pick the value corresponding to the nearest neighbour as their prediction). In [76], Smith and Wong propose another method using *Top-K Weighted Mean* for predicting application execution times of parallel applications running on computational grids. Their work will also allow for the prediction of application wait times in scheduling queues before being allocated the required resources. Jo *et al.* apply three forecasting techniques to predict firm bankruptcies: Discriminant Analysis, a Case-Based Forecasting System and Neural Networks [38]. Discriminant Analysis is a statistical model, similar to clustering, commonly used in classification. It estimates a linear function which can classify objects based on a group-membership score. Their implementation of the Case-Based Forecasting System uses a *Top-K Weighted Mean* approach to produce the estimate. According to their findings, the Neural Network model outperforms both of the Discriminant Analysis and the Case-Based Forecasting System methods, with both following closely behind. They attribute the under-performance of the Case-Based Forecasting System to the low correlation between the dependent and independent variables used in their approach. However, they believe that Case-Based Reasoning systems are comparatively useful, especially when training data is scarce. They also outline that their future studies will concentrate on developing hybrid systems which integrate various prediction tools, such as those described above. Phansalkar *et al.*

uses *Top-K Weighted Mean* as an alternative to clustering in [62]. They use the same method we described in their implementation of *Clustering* to compute the similarity scores/distances. They define the weights as the reciprocal of the distance to each of the programs. The program with the highest similarity (lowest distance) to the user's application, gets the highest weight. They experiment with three different weighted mean methods to use for the prediction of performance: geometric mean, harmonic mean and arithmetic mean. After examining the average error of each, they decide that the weighted harmonic mean performs the best. Although they conclude that both the *Clustering* and the *Top-K Weighted Mean* approaches show competitive results, if enough data is present, *Clustering* seems to follow the actual data distribution better. In [86], Zhu *et al.* also use *Top-K Weighted Mean* as an alternative to *Top-K Regression* to predict soil properties. They conclude, that unlike the *Top-K Regression* approach, *Top-K Weighted Mean* performs better in areas with steep landscape due to the non-linearity of the relationships.

Since *Top-K Mean* is the simplest predictor approach, it usually preferred by practitioners for implementation. However, since mean is greatly affected by outliers, this method must be used with caution, and data must sometimes be pruned or manipulated to relieve their effect on the accuracy of the predictions. *Top-K Weighted Mean* gives the user more control over the behavior of the predictor, but the chosen weight computation method is crucial to the performance of the predictions.

2.3.4 Top-K Majority Voting

Majority Voting counts the repeating occurrence of a certain value in history as a vote towards its use as the estimate; the value with most votes wins. *Top-K Majority Voting* simply limits the history set to the Top-K candidates. This model is less prone to the effect of outliers, however, in cases of indecision (no majority vote); an alternative method has to be used.

In [78], Sun *et al.* propose the use of *Top-K Majority Voting* to predict the financial distress of companies ahead of time. Using fuzzy logic, they compute the similarity scores between the target company and each of the other companies. After retrieving the k-nearest neighbours, the company class/state with highest ratio out of the k-nearest neighbors becomes their prediction. To evaluate their approach, they use data collected for 135 Chinese companies. They conclude that their method is competitive with Neural Networks, Support Vector Machine, Logit and Multi Discriminate Analysis approaches. Santos *et al.* also propose using *Top-K Majority Voting* to estimate stellar parameters [68]. What is interesting about their approach is the fact that they use a *Top-K Weighted Majority Voting* approach; where the votes of more similar data points have a higher vote-count (influence) than the less similar ones. They report achieving competitive results in their field. Ratanamahatana also uses *Top-K Weighted Majority Voting* to predict potential customers who are likely to switch to the new 3G wireless networks [65].

Ramirez *et al.* use *Top-K Majority Voting* to predict stellar atmospheric parameters [64]. First they use Genetic Algorithms to reduce the size of the data set as to decrease the computation overhead needed to generate the similarity scores. They experimentally deduce that the best value for k is 3, and an increase in k does not significantly improve the predictive accuracy, while slightly increasing the running time. Finally, they use *Top-K Majority Voting* for discrete-valued target functions and *Top-K Weighted Mean* for real-valued target functions. A target-function simply describes the different states for certain atmospheric properties or parameters.

What we found lacking in the above-mentioned works, is any description of the methods' behavior in cases of majority vote indecision. While this may describe a small percentage of cases, the way in which they are handled may improve the results, even if slightly. *Top-K Majority Voting* performs better than *Top-K Mean* when dealing with outliers and extreme values. However, *Majority Voting* can only be applied to discrete values, and if the effort values come from a continuous distribution (i.e. not a limited set of values), *Majority Voting* cannot be used.

An interesting study by Kim *et al.* uses a composite approach to predict interest rates for two countries [49]. They combine Neural Networks with a Case-Based Reasoning method. The composite model makes use of the Case-Based Reasoning method by feeding its prediction into the Neural Network, in addition to the raw variables. In order to evaluate the performance of the composite model, they compare it to the separate Neural Network and Case-Based Reasoning approaches, in addition to a random walk model as a benchmark to overall performance. While, the composite model does not provide an improvement in comparison to the independent Case-Based Reasoning method, it does perform comparatively with the Neural Network approach.

The next section will summarize what we presented in this chapter, outline the lessons learned and how our work plans to solve some of the problems apparent in current literature.

2.4 Summary

In this chapter we outlined the different categories of effort prediction for software development, that are used most commonly in literature. We learned that no one prediction approach can be preferred over all others. The key is to understand their differences and either combine them or use each in their best performance environment. We also discussed the different methods used to predict defects and defect-correction effort. We demonstrated that while current defect-correction effort prediction methods show reasonable performance, they do not extend the validity of their approach to other data sets. In cases were the approach was applied to additional case studies, the prediction performance was compromised. For that reason, our study applies the proposed framework on multiple case studies to extend the

validity of our approach, and show its capability to be applied in practice. Finally, we explored the application of similarity-based effort prediction in other domains, due to the limited number of studies in the area of software and defect-correction. We have learned that prediction methods in this area can be divided into four major categories: *Clustering*, *Top-K Regression*, *Top-K Mean* and *Top-K Majority Voting*. Each prediction method has its own advantages and disadvantages, and can be used effectively if the practitioner understands their strength and weaknesses. For example, while *Top-K Majority Voting* performs better with the existance of outliers, it cannot be used in cases where the effort values are not discrete. On the other hand, *Top-K Mean* is a simpler technique and while prone to outliers, it can be used with continuous effort values.

It can also be noted that current literature does not focus on providing extensible and modular frameworks, that can adapt to the user's needs. We propose a modular framework, where the different components can be updated, or additional modules can be added. This allows the user to customize the framework to suite their particular data set. Also, while current defect-correction effort prediction literature compares the different prediction methods, it does not combine them in one framework. We propose combining two approaches that complement the weaknesses of one another. Our framework does not limit the combination to two methods. Additional methods can be added if deemed appropriate.

The next two chapters will describe our proposed effort prediction framework. First, Chapter 3 will present the *Base Approach* on which we build our effort prediction framework, and a brief outline of the general architecture of the framework. Chapter 4 describes the effort prediction framework in more details, presenting it as a set of enhancements applied to the *Base Approach*.

Chapter 3

Proposed Framework Introduction

To facilitate an easier understanding of our proposed framework, we describe a base approach to which we apply a set of enhancements (which make up the core of the effort prediction framework). In Section 3.1, we present the base approach, by first describing the mechanism used to generate the similarity scores, then describing two prediction methods that use the *Nearest Neighbour Approach*. In Section 3.2, we outline the different components of the effort prediction framework, with the aid of a simplified process model describing the framework. Finally, Section 3.3 outlines what we discussed in this chapter and leads the reader into the next chapter.

3.1 The Base Approach

This section describes the base approach, inspired by the work presented by Weiss *et al.* in [82], to which we apply our enhancements. We propose two prediction approaches: the *Nearest Neighbour Approach* (described in Subsection 3.1.2) and the *Nearest Neighbour Approach with Thresholds* (described in Subsection 3.1.3. *Nearest Neighbour* approaches require a distance or scoring mechanism to assess the similarity between the different instances (in our case issues). Subsection 3.1.1 describes the text-similarity approach along with the tool we use to generate the similarity scores.

3.1.1 Generating Similarity Scores

In obtain the nearest neighbours to target issue, we need to define a distance or similarity measure. We use a text-similarity measuring approach, inspired by [82], and motivated by the fact that most of the information about an issue is available in the title and description fields of the issue report. The text-similarity measuring engine we use is called Lucene; an open source project developed by the Apache Foundation [33]. It has demonstrated its competence through its deployment in many major systems at high profile organizations such as MIT, FedEx and New

Scientist Magazine. Lucene uses a vector-based approach to measuring the similarity between two texts, which it uses to index resolved issue reports into a repository. Before we perform the indexing, we remove all English stop words such *a, an* and *the*, we also remove all symbols such as *+,* - and *()*.

When we need to find the *Nearest Neighbours* for a new target issue, we form a query containing the information known about the issue. Then, compare it to all the existing (indexed) issues in the repository, to generate the similarity scores. Lucene returns scores between 0 and 1; where scores closer to 0 indicate marginal similarity and ones closer to 1 indicate very high similarity. On a more technical note, we use the *Multi-field Query* provided by Lucene to combine the different query fields into a single score. Lucene compares the different fields seperately and combines the results into a single score using a boost factor. We use an equal boost factor for all fields in our experiments, which instructs the search engine to give the same weight to all similarities for the different fields.

The approach in [82] proposes the use of only the *Title* and *Description* fields of an issue; however, we propose to use additional information in the query to increase the accuracy of the similarity scores, and consequently, the performance of the predictions.

3.1.2 The Nearest Neighbours Approach(*kNN*)

The *Nearest Neighbour Approach* (*kNN*) has been widely used in software development to predict the effort or the cost for software projects early in their lifetime [71]. It has also been shown to perform better than traditional prediction methods such as COCOMO and linear regression [71]. A great advantage of *kNN* is its flexibility and ease of use in practice, in addition to its ability to perform well even in situations where data is limited. Therefore, we adopt the *kNN* approach to produce our defect correction effort prediction, based on reasoning adopted from research in the area of software cost estimation; i.e. it is very likely that similar issues will have similar correction times.

We use *kNN* to produce a prediction as follows: the target issue (to be predicted) is compared to the resolved ones in the Issue Tracking System. The distance measure (as defined in Subsection 3.1.1) is used to produce a similarity score between the target issue and every resolved issue. Then, the k most similar issues (candidates) are selected, and their reported efforts are combined to produce a prediction. If the users require a justification of the prediction, we can provide the candidates used to derive it, giving them a better understanding of the prediction. In [82], the authors use the mean of the candidate issues to generate the prediction, however, our approach proposes a multi-step *Top-K Majority Voting* approach which makes better use of the discrete nature of the recorded effort values.

3.1.3 The Nearest Neighbours Approach with Thresholds (α-*kNN*)

To make a prediction, *kNN* chooses the k nearest neighbours according to the similarity scores and computes the mean, with no regard to the score values; i.e. it uses the scores to sort the issues according to similarity. This technique might be inaccurate at times, where even the highest scores are very low, consequently producing misleading results. For example, if the nearest neighbours have scores below 0.1, *kNN* will still use them to make the prediction, although it might be inaccurate. Therefore, to address this problem, we introduce the concept of thresholds, as originally presented in [82]. We define a variant of *kNN* called the Nearest Neighbour with Thresholds (α-*kNN*). α-*kNN* applies a lower bound on the acceptable scores; i.e. if $\alpha = 0.5$, then only issues with scores ≥ 0.5 are allowed as candidates, up to k issues. This implies that for higher values of α, α-*kNN* may not choose any candidates and will return "Unknown" rather than a prediction. However, this is one of the problems resolved by introducing two enhancements in our approach: *Adaptive Threshold* and *Binary Clustering*.

The next section will show how the above components are combined into our proposed framework, and outline their interactions. With the aid of a simplified process model diagram, we will briefly describe the different components and illustrate how the *Base Approach* fits into the effort prediction framework.

3.2 Simplified Framework Composition

This section will outline the different components that make up the composite effort prediction framework. It will give a simplified overview of the process model, including the interactions of the different components. It will also summarize where the *Base Approach* components fit in the framework.

There are five different components/modules that make up the framework, as illustrated by Figure 3.1: Data Enrichment, Text Similarity Engine (Lucene), Nearest Neighbours with Thresholds (α-*kNN*), Adaptive Threshold and the prediction modules (Majority Voting and Binary Clustering). The *Base Approach* components (shown by the black boxed modules in Figure 3.1) are made up of the Text Similarity Engine (as described in Subsection 3.1.1) and the Nearest Neighbours with Thresholds (as described in Subsection 3.1.3). We chose to use α-*kNN* in our framework since the results of using this prediction method outperform the standard Nearest Neighbours approach *kNN*. It also allows us to manipulate the threshold to control the scope of data we would like incorporate in our predictions. The new modules/components (shown in dashed red boxes in Figure 3.1) are the enhancements introduced to the *Base Approach*, which make up the remainder of the effort prediction framework. They are described and illustrated in more details in Chapter 4.

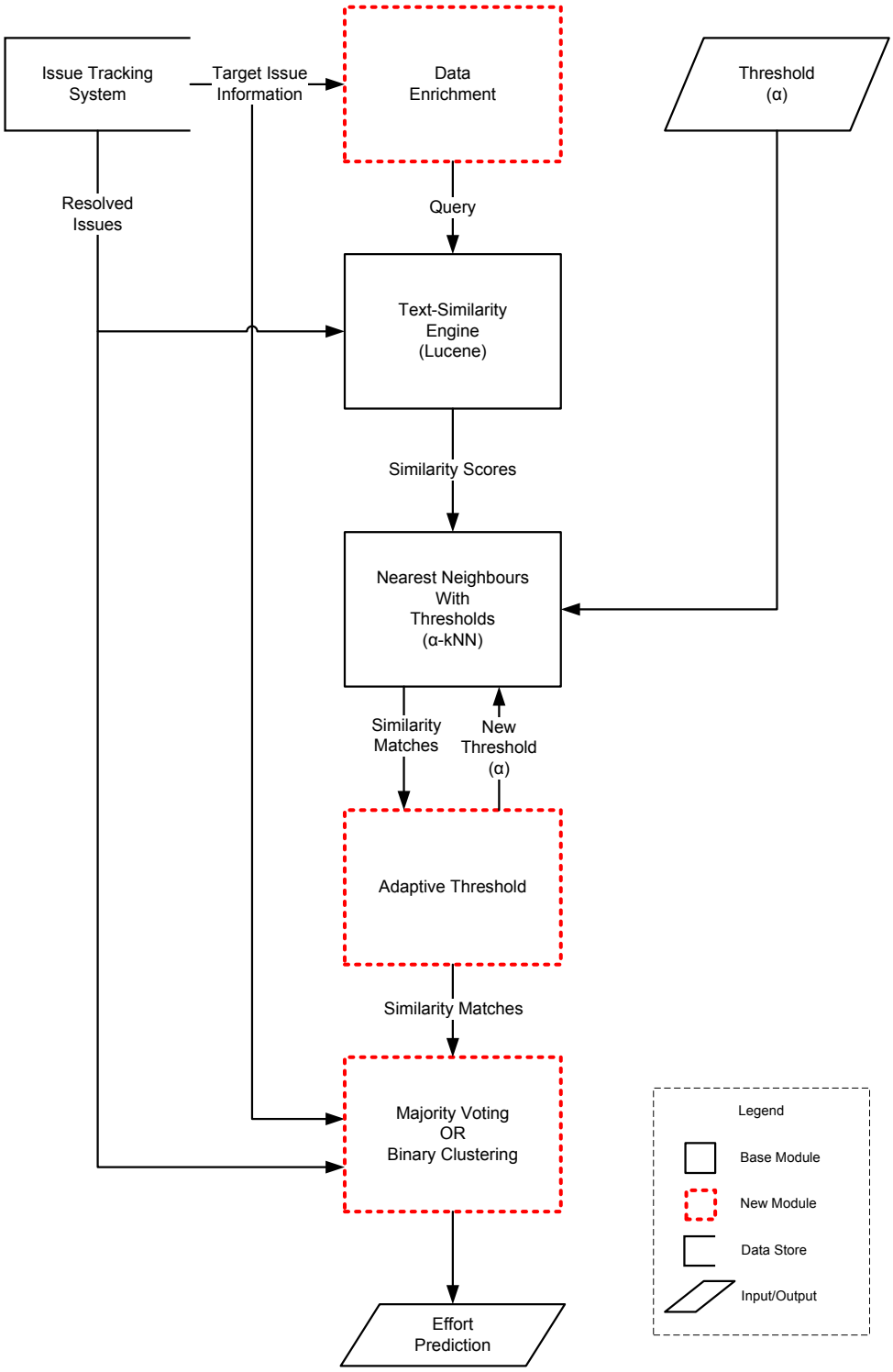

Figure 3.1: Simplified Process Model of the Proposed Effort Prediction Framework

Data Enrichment is an enhancement that incorporates more issue related information into the similarity engine text query. This should generate more accurate similarity scores since the similarity engine can now focus its search to more related issues (assuming similar results have similar actual effort [82]). Adaptive Threshold automatically adjusts the threshold value to a lower limit if no matches are found until we reach a similarity level were scores are deemed misleading. This solves the problem of returning "Unknown" results to user when using α-*kNN*. Rather than leaving the user without a prediction, we provide the *best possible* prediction while still keeping the lowest scores out of the similarity matches. By providing the user with the matches that were used to produce the prediction along with their similarity score, the user can then perform his or her own reasoning based on that information. At the same time, since we provide a minimum limit where the similarity scores are believed to be misleading, and revert to alternative prediction methods, the user can set this limit to whatever threshold value they want. This allows great flexibility when performing the predictions. In cases were the similarity scores are high enough to pass the threshold limit, we use Majority Voting to calculate the effort prediction. Otherwise, if the similarity scores are too low, or we have no matches for a particular target issue, we use an alternative prediction method, called Binary Clustering. Binary Clustering uses other common issue information from the Issue Tracking System to calculate the predictions (independent of the similarity scores). To keep the framework extendable and modular, Binary Clustering can be replaced by any similarity score independent approach. Additional methods can also be added prior or after Binary Clustering to calculate the predictions for those issues which it could not make a prediction. For example, in cases where a cluster contains no members, Binary Clustering cannot make a prediction. For our framework, resolve that problem by using a different clustering criterion, and recalculating the prediction. However, that does not limit users from using a different prediction method instead.

The next section will summarize what we discussed in this chapter and introduce the following chapter which describes the effort prediction framework in more details.

3.3 Summary

In this chapter, we introduced the *Base Approach* to which we apply a set of enhancements. The *Base Approach* is composed of a similarity measuring engine, which calculates the distance between the issues needed to identify the Nearest Neighbours, and the Nearest Neighbour Approach (*kNN* or α-*kNN*), which calculates the predictions based on the similarity scores. We described our effort prediction framework as a set of enhancements to the *Base Approach*, from which we used α-*kNN* for its superior performance. We introduced four enhancement: *Data Enrichment*, *Adaptive Threshold*, *Majority Voting* and *Binary Clustering*. These enhancements make up the core of our proposed composite effort predic-

tion framework. The next chapter will describe the effort prediction framework in more details, and will provide a summary pseudo-code algorithm which shows its implementation.

Chapter 4

Effort Prediction Framework

In this chapter we describe our proposed effort prediction framework for defect correction effort prediction. First, Section 4.1 presents two weight computation techniques that we use in cases where the approach reverts to computing the effort prediction using a *Weighted Mean* method. Then, Section 4.2 describes the proposed enhancements to the *Base Approach*, including when and how they are used. Section 4.3 presents a summary algorithm (pseudo-code) that combines the proposed enhancements, and a corresponding description of its operation. Finally, Section 4.4 summarizes what we discussed, including the different enhancements and weight computation methods presented, and introduces the following chapter of experimental studies.

4.1 Weight Computation Methods

When using *Majority Voting*, we may encounter cases where a majority vote is not reached. In cases where we have multiple similarity matches in the candidate set, but no majority vote, we opt to use a *Weighted Mean* approach. It expresses the prediction as a weighted sum of the effort values, corresponding to the issues in the candidate set. We experiment with two techniques to compute the weights as explained next. These techniques show similar performance, and a small, but noticeable, improvement over simply using the mean (equal weights).

4.1.1 Least Squares Learning

This technique uses the method of least squares, also known as regression analysis, to compute the weights based on minimizing the sum of the squared residuals (a residual is the difference between the actual and the predicted effort). This technique is Similarity-Score Dependent since it relies on similarity-scores to sort and filter training issues. The training set is the group of issues that we use to compute the weights. The key to generating useful and accurate weights is to choose

the training set properly. Our technique relies on the fact that the similarity-scores are a good indicator of similarity [82], and chooses the training set the same way as the candidate set is chosen (a weight is assigned to an issue's position in the sorted candidate set array). Ideally, this should produce a training set that has similar properties to the candidate set. The following is a formalization of how *Least Squares Leaning* is used to compute the weights:

$$\hat{E}_i = \sum_{j=1}^{M} w_j E_{ij} \tag{4.1}$$

$$\begin{aligned} \text{MSE} &= \frac{1}{N}\sum_{i=1}^{N}(E_i - \hat{E}_i)^2 & (4.2)\\ &= \frac{1}{N}\sum_{i=1}^{N}(E_i - \sum_{j=1}^{M} w_j E_{ij})^2 & (4.3) \end{aligned}$$

where M is the number of candidate issues, w_j represents the weight coefficient for the j^{th} candidate issue, E_{ij} represents the actual effort of the j^{th} candidate issue of the i^{th} training issue, $\hat{E}_i$ is the predicted effort for the i^{th} issue, N is the number of training issues and E_i represents the actual effort for the i^{th} issue. In order to minimize the Mean Square Error (MSE), we differentiate w.r.t. w_q for $1 \leq q \leq M$ and set $\frac{\partial MSE}{\partial q} = 0$; resulting in:

$$\sum_{j=1}^{M}\sum_{i=1}^{N} E_{ij}E_{iq}w_j = \sum_{i=1}^{N} E_i E_{iq} \tag{4.4}$$

By computing (4.4) for all q, we obtain a system of linear equations for finding the weights w_j $(1 \leq j \leq M)$.

4.1.2 Historical Value Frequency

This technique benefits from the fact that the recorded effort values are discrete; i.e. effort is usually rounded to the nearest 15 minutes. To compute the weights, this technique counts the number of occurrences of each effort value in the history, and computes the normalized relative weights (estimate of probabilities) accordingly. This is equivalent to minimizing the mean squared residuals (by computing the statistical expectation conditioned on the candidate set) by solely relying on the history to estimate the probability values (ignoring the similarity scores, and consequently, the order in the candidate set).

The following is an example of how these weights are computed: we are given the *Candidate Set* $= [1, 2, 4]$, which contains the effort values for the target issue candidates, and *History* $= [1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5]$, a vector of all effort values

in history (resolved issues). First, we compute the frequency of each effort value in the *Candidate Set* as it appears in *History*, this give us the *Candidate Set Historical Frequencies* $(F) = [count(1), count(2), count(4)] = [3, 5, 2]$. Then, to compute the normalized weights, we divide the resulting frequencies by the sum of the frequencies in the *Candidate Set Historical Frequencies* vector, as shown in *Candidate Set Weights* $= [\frac{3}{sum(F)}, \frac{5}{sum(F)}, \frac{2}{sum(F)}] = [\frac{3}{10}, \frac{5}{10}, \frac{2}{10}] = [0.3, 0.5, 0.2]$. Now that we have computed the weights, we can apply them to the *Candidate Set* by multiplying each entry accordingly, and the final *Weighted Candidate Set* $= [(0.3 \cdot 1), (0.5 \cdot 2), (0.2 \cdot 4)]$ $= [0.3, 1, 0.8]$. Finally, to produce the prediction, we compute the sum of the *Weighted Candidate Set*.

In the next section, we describe the proposed enhancements that make up the core of the framework in detail. In addition, we provide an illustration, similar to Figure 3.1 in Chapter 3, but with a more detailed look into the components the make up each of the enhancements.

4.2 Proposed Enhancements

Our approach could be described as a set of enhancements to the *Base Approach*. To help understand our approach, we divide the four proposed enhancements into two categories: Similarity-Score Dependent Enhancements (*SSDE*) which include *Data Enrichment*, *Majority Voting* and *Adaptive Threshold*, and Similarity-Score Independent Enhancements (*SSIE*) which include *Binary Clustering* as depicted in Figure 4.1. It can be observed that *SSDE* is used when the similarity scores are high, and *SSIE* is used otherwise. This is justified based on studies showing that prediction methods using historical similarity can be more effective in comparison to those which do not (such as COCOMO or regression models [71, 43]). However, at lower scores, *SSDE* could actually negatively affect the accuracy of the model [82]. For this reason, we adopt the *SSIE* in such cases, relying only on issue-related information.

Figure 4.1 shows the process model describing our effort prediction framework, we will give a brief walk-through to help better understand the process. First, we extract the *Target Issue Information* along with the *Resolved Issues* from the Issue Tracking System. Then we form a match query through *Data Enrichment*, where we include a number of issue properties. While the *Base* approach uses the *Title* and *Description* properties only, our approach includes additional issue properties such as *Project Name* and *Issue Type*. Then after feeding the query (along with the *Resolved Issue*) into the text-similarity engine, we obtain the similarity scores with each resolved issue. The scores, in addition to the similarity threshold (α) provided by the user, are then used by α-*kNN* to find the similarity matches (candidates). If no matches are found at the given α value, *Adaptive Threshold* kicks in and decrements α if possible. Then the new α is fed back into α-*kNN*, and the process is restarted. This continues until we either find a match or reach a threshold limit (i.e. predictions based on scores below this threshold are considered misleading), at

which point we switch to *Binary Clustering*. If we find matches, *Majority Voting* is used to compute the prediction, otherwise, *Binary Clustering* uses its own criteria to cluster related issues and produce the prediction.

The following subsections give a description of the enhancements (refer to Figure 4.1 to see where each fits in the process):

4.2.1 Data Enrichment

In Section 3.1.1, we described how we use a text similarity measuring engine to extract similarities between issues. The *Base* approach uses the issue's *Title* and *Description* in Lucene's search query to generate the similarity scores. While that has generated promising results, we believe that incorporating more related data into the search query should better focus the results. For example, on a search engine like Google, if we search for the word *"apple"*, we might get mixed results giving us links related to the fruit and the computer company. However, if we enrich the query by adding more relevant criteria like *"computer"*, the results should be more relevant to what we really are looking for. Nevertheless, we will notice that the number of the results returned for the enriched query will decrease, since we have defined a narrower scope for the search. In most cases, we do not notice the effect of this decrease in the number of results since search engines like Google index millions of documents, and the decrease is minute relative to the number of results.

Similarly, if we apply the above philosophy to our engine's search query; incorporating more data relevant to the target issue into the query, we should obtain more accurate scores. We should see an improvement in the relevancy of the associated issues. A more contextual example would be: if we include the issue's *Project Name* and *Issue Type*, given that those two properties are correlated to the actual effort values, we should see more representative matches higher up in the score array (i.e. more relevant matches should obtain higher scores). However, as mentioned above, there are two unfavorable side effects: 1) in general there will be less matching issues, 2) the number of high scoring matches will also decrease; both side effects are due to the more specific nature of the query. Contrary to the Google case, we will notice these effects on our data set, since our index contains only a few hundred documents. This implies that, while yielding more accurate results, using the α-*kNN* method will have lower *FEEDBACK* values for high α's, (*FEEDBACK* is a performance metric that measures the percentage of issues for which the model makes predictions).

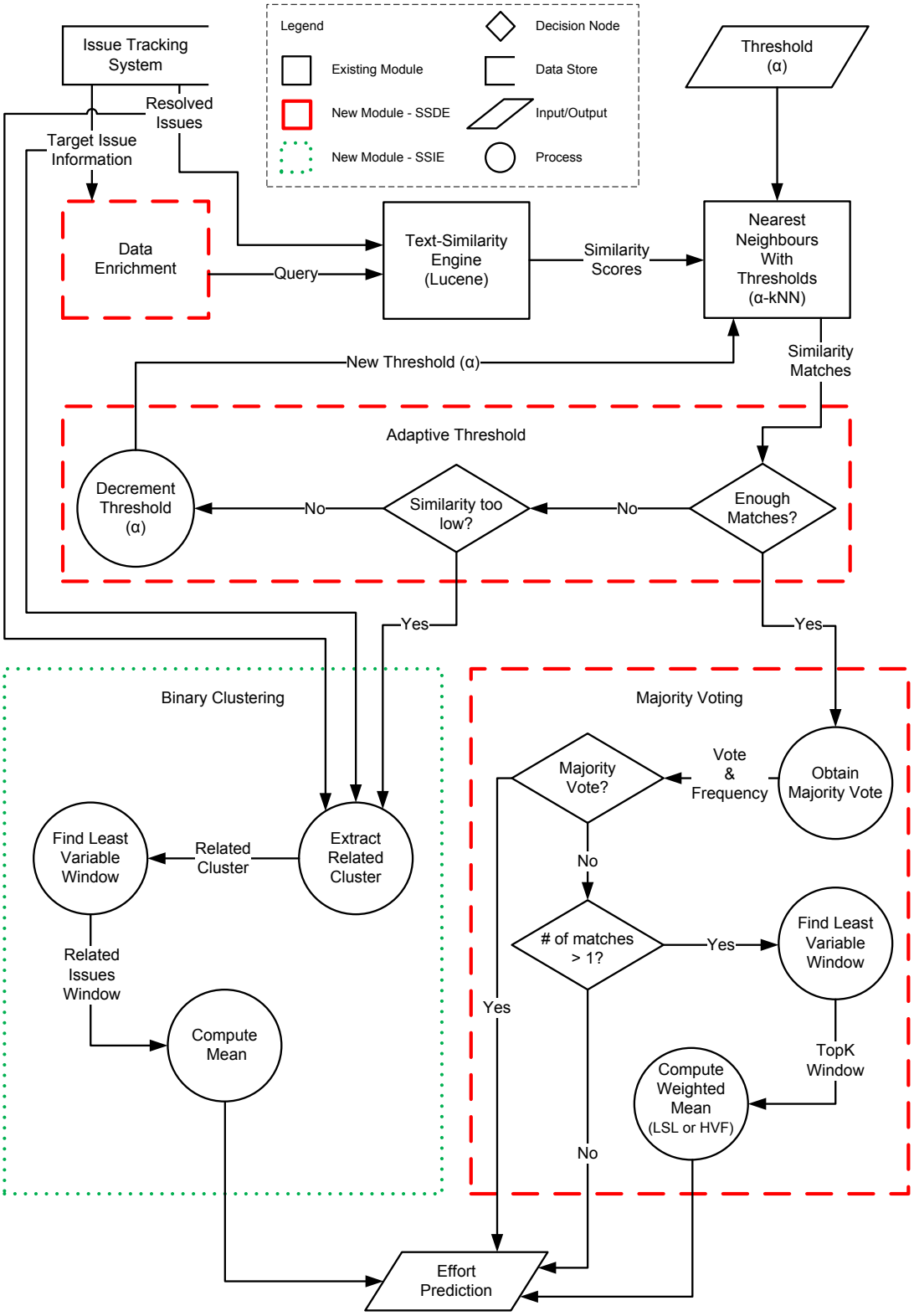

Figure 4.1: Process Model of the Proposed Effort Prediction Framework

To implement this enhancement, we perform a two step analysis to determine which issue properties to include. This process is performed per data set and the determined properties are used universally for all predicted issues for each given data set. Initially, we perform a correlation analysis between each issue property available from the Issue Tracking System, and the actual effort. Then, we choose the properties with highest significant correlation values as our base set. An systematic analysis is then performed to identify the effect of each property on the accuracy of the scoring mechanism. First, we test the effect of enriching the query with each property individually. Once we identify the best group of properties, we again systematically test them together until we arrive at the best set of properties, giving the most accurate results possible. During the testing we also take into consideration the *FEEDBACK* value; as we mentioned above, the *FEEDBACK* decreases as introduce more criteria into the query. In general, we try to avoid cases where *FEEDBACK* reaches the 0% mark; intuitively we would like it be as high as possible, at the same time we want the accuracy of the scoring mechanism to also be high (it is a balancing act). One issue property that we did not use in our study, is the associated files changed. The main reason behind our decision to exclude this information, is the fact that this information is not usually available in the initial stages of the issue's life. In fact, it is usually the end result of the issue resolution cycle. Therefore, using such information in our approach is not justified, and may not prove to be useful if it is to be used as an effort assessment tool for such repositories. Additionally, issues that did have such information constituted only about 12% of the data set, not allowing us a significant ratio to work with.

4.2.2 Adaptive Threshold

As mentioned in Subsection 4.2.1, introducing *Data Enrichment* can negatively influence *FEEDBACK*, the magnitude depending on how many properties are incorporated into the search query. *Adaptive Threshold* is used to compensate this negative effect for α-*kNN*. If for some α, α-*kNN* does not find any candidates, we automatically decrement α. We can also set certain match-goals to trigger such an event; for example, decrement α until we reach a required *FEEDBACK* percentage, or have a majority vote, or simply receive a match (which is the approach we have implemented). Moreover, we can control how fast α decreases, giving us greater control over the range of the score spectrum we would like to include (we used 0.1 as the decrement value, similar to [82]).

We take a number of steps to implement this enhancement. Since this enhancement is Similarity-Score Dependent, it is implemented as a LOOP structure containing the *SSDE*. The user inputs the initial desired α value into the algorithm. If we can find a match (or matches) given the current α value, we submit them to the prediction module (whether it is Mean or *Majority Voting* it is a modular approach). If no match is found, we decrement α by a certain decrement-value and repeat the search for matches given the new α. On the other hand, if α is determined to be too low at to grant a reliable prediction at this point, *Adaptive*

Threshold terminates the Similarity-Score Dependent Enchantments LOOP, and hands the prediction over to the Similarity-Score Independent Enhancements. As we can see, this approach is dependent on two main variables that must calibrated per data set, namely *Threshold Decrement* and *Threshold Limit*. *Threshold Decrement* determines the value by which we decrement α at each interval or step during the *Adaptive Threshold* automatic adjustment procedure. If *Threshold Decrement* is set a low value, the threshold is decremented at a finer level. This forces the algorithm to obtain less but more relevant matches as it decrements; i.e. if we set the *Threshold Decrement* to 0.01 as opposed to 0.1, then the algorithm will try to obtain matches at finer intervals (1.00, 0.99, 0.98, 0.97 ... etc. rather than stepping to 0.9 directly). While finer intervals give more relevant matches higher priority, the fact that we may obtain less matches means that any error in the scoring process could be magnified if the given matches are misleading. This is the reason why we revert to using 0.1 as our preferred decrement value, as also supported experimentally. Also, when applying *Data Enrichment*, we obtain less high scoring matches as described in Subsection 4.2.1, therefore finer *Threshold Decrement* values at high α's will also magnify this effect. We also determine *Threshold Limit* experimentally. *Threshold Limit* defines the point at which we consider the scores misleading; i.e. predictions based on Similarity-Score Dependent Enhancements are considered inaccurate. We compare different limits by assessing the performance of our approach beyond the given threshold. If the Similarity-Score Independent Enhancements perform better beyond the limit, we consider it a cut off point for the *SSDE*. We concluded that a *Threshold Limit* of 0.1 was appropriate for both data sets in our study.

Adaptive Threshold applied to *α-kNN*, can also be used as a superior technique to the simple *kNN* model. For each target issue, we only consider the highest scoring matches to compute the prediction. For example, for some target issue the highest scoring match might be at $\alpha = 0.9$, while another might be at $\alpha = 0.6$. If we set our match-goal to a single match; starting at $\alpha = 1$ and decrementing until we receive at least one match, we ensure that we are using only the most similar matches to compute the prediction. If the scoring mechanism is accurate enough, this method should yield more accurate results than *kNN*.

4.2.3 Majority Voting

In [82], the authors use *Top-K Mean* to predict effort. After studying the data, we observed that about 80% of the issues have repeating values in their candidate sets, and many of those values were close to the actual effort. We also noticed that, often, there are outliers that could greatly skew the mean. These observations led us to using *Majority Voting* in place of the mean, as it is better suited to deal with outliers, and makes better use of the discrete nature of the effort values.

We also observed that that about 91% of the issues have effort $<$ 40 Hours, 78% $<$ 16 Hours and 64% $<$ 8 Hours. This indicates that the data is skewed

towards lower values, and the majority of the issues have effort less than 5 work days. However, this also means that a larger number of the other 9% of effort values could be extreme outliers, which is the case in our data sets. As mentioned above, this can greatly skew the predictions made by *Top-K Mean*, which is gives greater confidence in adopting our proposed *Majority Voting* approach.

When a majority vote cannot be reached, we employ a *Least Variable Window* method which narrows down the candidate set to the group of issues that has the least variation in their corresponding effort values (for a given window size). *Least Variable Window* also takes the similarity scores into consideration; i.e. if two sets have the same standard deviation, the one with the higher score-mean is chosen. If there is only one issue in the windowed candidate set, we use the corresponding effort as the prediction. Otherwise, the prediction is computed using a *Weighted Mean* approach. In this case, we can use *Least Squares Learning* or *Historical Value Frequency* to compute the weights and produce the prediction. We show the performance for using each technique in the effort prediction framework comparison with the *Base* approach.

To describe the *Least Variable Window* method in more details, first we outline two window adjustment methods we experimented with during our studies, namely *Window Size* and *Window Variability*. *Window Size* controls the window by adjusting its size or length; i.e. the number of elements allowed in the window. On the other hand, *Window Variability* controls the window by adjusting the maximum variability (Standard Deviation) of the elements included in the window. First we describe the *Least Variable Window* method using the *Window Size* adjustment technique. Given a list of effort values and the scores associated with each, *Least Variable Window* first sorts the effort values in non-decreasing order, while keeping the score associations intact. Then using the given window size, traverses the new sorted vector from starting from the smallest effort value to highest, producing a list of windows of the specified size (order of traversal is not important since we produce a full list of windows). Next, the variability (Standard Deviation) of each window is computed, and the window with the smallest standard deviation is chosen. If more than one window has the same standard deviation value, the score mean of each window is computed and the window with the highest score is chosen. The *Window Variability* control technique uses the *Window Size* technique to compute its best window. Given a maximum standard deviation threshold, *Least Variable Window* starts with a window size equal to the size of the entire candidate set (the aim is to try to find the maximum window size with variability less than or equal to the threshold), and computes the standard deviation. If variability is higher than the threshold, decrement the window size by 1 and retry. This process is repeated until we reach a window with variability below or equal to the given threshold. If we could not find any window with the required variability, we return the issue (effort value) with the highest score. After experimenting with both methods, we concluded that using the *Window Size* adjustment method gave us better results. We use the *Least Variable Window* method in both *Majority Voting* (as descirbed avbove) and in *Binary Clustering* as described in the next

Subsection 4.2.4. Again, the size of the window must be calibrated for each data set separately, as the variability in the data is different in each case.

4.2.4 Binary Clustering

As mentioned in [82], predictions based on low similarity matches could in fact be misleading. For these cases, we implement a Similarity-Score Independent enhancement called *Binary Clustering*, to produce reasonably accurate predictions. This enhancement uses the basic idea of clustering to group similar issues into clusters. However, we use a simplified binary distance function, which returns "0" if the issue has the same properties as that of the cluster, or "1" otherwise. Only issues with a distance of "0" are accepted into the cluster. Cluster properties can be chosen from whatever information we can extract from the Issue Tracking System. For example, properties like *project name*, *issue type* and *issue priority* could all be used as clustering criteria. The target issue defines the values of the properties for a cluster; i.e. if we use *project name* as the cluster property and the *project name* of the target issue is "FOO", then only historical issues with the *project name* "FOO" are accepted into the cluster.

To decide which properties to use for the clustering, we perform correlation analysis between the different issue properties and the actual effort. If more than one property is found to be highly correlated, then multiple properties are used. This means that issues have to match all properties before entering a cluster. Once the set of properties is narrowed down by the correlation analysis, we perform systematic experimentations to isolate the best group to be used for clustering. First, we determine the effect of clustering using each property individually. Then through combining the best set of properties together and observing the results, we determine the idea group for each data set. Once we decide on the clustering properties, we populate the cluster by applying the binary distance function between the target issue and each issue in the history. Now, we have a vector of related issues, and their corresponding effort values. We use the *Least Variable Window* method to limit the variability of the determined set, and then compute the mean of the resulting window.

Binary Clustering is just one approach to introducing Similarity-Score Independent Enhancements. For example, we can use Regression Analysis or Artificial Intelligence methods like Neural Networks to support the Similarity-Score Dependent Enhancements. Also, we can take a multiple step approach when producing predictions using *SSIE*. For example, in our approach, we use multiple steps of *Binary Clustering* depending on the nature of the data set. In some cases, clustering using certain criteria like *Project Name* does not produce predictions 100% of the time (i.e. some projects only have 1 issue listed under them). Therefore, we implement a second *Binary Clustering* step using a different criterion or set of criteria (e.g. *Issue Type*), where cases that are not predicted by the first step of *Binary Clustering* could be predicted by the second (possibly less accurate) step.

This technique could also be applied with different Similarity-Score Independent Enhancements.

The next section presents a simplified pseudo-code algorithm implementation of the framework. It discusses the different operational features of the algorithm, in addition to explaining the different important variables involved in the running of the code.

4.3 Framework Implementation

In this section, we bring the four enhancements together in an algorithm that describes the process we used to implement our approach, and apply the empirical studies. We outline the enhancements in the comments within the algorithm, and to keep the pseudo-code easy and simply to understand, we name the variables and functions with their corresponding methods as mentioned in previous sections. The code is also partitioned into three sections, *Variables Initialization*, *SSDE Implementation* and *SSIE Implementation*. *Variables Initialization* simply contains all the important variables initialization. The *SSDE Implementation* partition shows the section of the code implementing the Similarity-Score Dependent Enhancements. Our goal is clearly demonstrate how these enhancements interact with one another. Finally, *SSIE Implementation* shows the Similarity-Score Independent Enhancement, namely *Binary Clustering*. However, we would like to emphasize that this where multiple steps of *Binary Clustering* or *SSIE* enhancements would be implemented, by simply introducing additional filters/steps following the first prediction attempt.

The following algorithm shows a simplification of the final composition of our framework, incorporating all of the above enhancements. First, some variable names: $threshold\_decrement$ specifies the granularity at which to decrement the threshold, $lsl\_coefficients$ holds the weights computed by *Least Squares Learning*, $hvf\_coefficients$ holds the weights computed by *Historical Value Frequency*, mv_window_size specifies the *Least Variable Window* size for the *Weighted Mean* alternative in *Majority Voting*, bc_window_size specifies the *Least Variable Window* size for *Binary Clustering*, $issue\_history$ contains all of the resolved issues that are older than the target issue (including the similarity scores).

```
1: {——————Variables Initialization——————}
2: prediction ⇐ null
3: threshold ⇐ 1
4: threshold_decrement ⇐ 0.1
5: threshold_limit ⇐ 0.1
6: mv_window_size ⇐ 3
7: bc_window_size ⇐ 5
8: {——————SSDE Implementation——————}
9: while prediction = null & threshold ≥ threshold_limit do {Adaptive Threshold}
10:    topk ⇐ choose_topk(issue_history, k, threshold) {α-kNN}
11:    if size(topk) = 0 then
12:       threshold ⇐ threshold − threshold_decrement
13:       continue
14:    end if
15:    [vote, frequency] ⇐ majority_vote(topk) {Majority Voting}
16:    if frequency < 2 then {No Majority Vote}
17:       if size(topk) = 1 then {Single Match}
18:          prediction ⇐ topk
19:       else {Multiple Matches}
20:          topk_window ⇐ least_variable_window(mv_window_size, topk)
21:          lsl_coefficients ⇐ least_squares_learning(issue_history)
22:          hvf_coefficients ⇐ historical_value_frequency(topk, issue_history)
23:          prediction ⇐ sum(topk_window · (lsl_coefficients or hvf_coefficients))
24:       end if
25:    else {We have a Majority Vote}
26:       prediction ⇐ vote
27:    end if
28: end while
29: {——————SSIE Implementation——————}
30: if prediction = null then
31:    related_issues ⇐ binary_clustering(target_issue, issue_history) {Binary Clustering}
32:    related_issues_window ⇐ least_variable_window(bc_window_size, related_issues)
33:    prediction ⇐ mean(related_issues_window)
34: end if
```

Lines $2-7$ show the variable initializations. Lines $9-28$ implement the *SSDE*: lines $9, 11-14$ implement the *Adaptive Threshold* enhancement, line 10 represents the α-*kNN* method, line 15 performs the *Majority Voting*, lines $15-24$ are executed if no majority vote can be reached (at which point either a single match is available or multiple matches are found). If a single match is available, the corresponding effort is used as the prediction, otherwise, the *Least Variable Window* method is used to reduce the variability for the multiple matches, then one of the weight computation techniques is used to compute the prediction using the *Weighted Mean*. If there is a majority vote, line 26 is executed and the prediction is returned as the vote. Lines $30-34$ implement the *SSIE*, if no matches were found from the similarity-score dependent method, we execute these lines; at line 31 we perform the *Binary Clustering*, and at line 32 we find the *Least Variable Window* for the related issues obtained from the cluster, then the prediction is computed using the mean of the window at line 33.

Both *bc_window_size* and *mv_window_size* are determined experimentally; we observed that a window size larger than 5 makes the results worse for *Binary Clustering*, and larger than 3 worse when multiple matches exist but no majority vote can be reached. *threshold_decrement* could be set to a smaller value for finer threshold traversal. After numerous experiments with different values, we found that the best results are produced using decrements of 0.1. Any similarity score below the *threshold_limit* is considered too low, so if there are no matches with scores ≥ 0.1, the effort prediction is handed over to *Binary Clustering*. We also experimented with different values for this limit and 0.1 seems to be a good cut off point for both data sets used in this work (this value should be calibrated for different data sets). As for the *SSIE*, we can specify as many filters/steps as required to predict 100% of the issues. In some cases, certain criteria/propoerties form empty clusters for certain target issues, yielding no prediction. To resolve that, we can apply multiple steps of *Binary Clustering*, each using a different criterion for clustering, until we obtain a prediction.

The next chapter will summarize the work we have presented in this chapter, and lead the reader into the contents of the next chapter.

4.4 Summary

In this chapter, we described our proposed composite effort prediction framework in detail. First by introducing the weight computation methods, used when the framework reverts to using a *Weighted Mean* method to predicting effort. We also provided a more detailed look at each of the enhancements, with a more detailed process model diagram describing the interactions between them. Finally, we presented a simplified pseudo-code algorithm that described the implementation of the framework, and the different variables and functions involved.

In the following chapter, we will present the experimental studies performed to evaluate the performance of our framework using case studies from live, operating Issue Tracking Systems. First, we present the case studies, then describe the implementation tools we used used to develop our testing code. Next, we outline the evaluation method and metrics we used to measure the performance of the framework. Finally, we present the results obtained from the evaluation studies.

Chapter 5

Experimental Studies

In this chapter, we present the experimental studies performed to evaluate the performance of our effort prediction framework. Section 5.1 describes the case studies we use to perform our experimental studies, in addition to the Issue Tracking System from which they were extracted. Then, Section 5.2 describes the implementation tools and experiment setup. Section 5.3 presents the evaluation method along with the performance metrics we use to assess our proposed approach. In addition, it shows the results of the evaluation study in addition to a comparison between the effort prediction framework with the *Base Approach*. Finally, Section 5.4 summarizes the results of the evaluation study.

5.1 Case Studies: JBoss and Codehaus

Many Issue Tracking Systems currently exist in the software community. Systems like Bug-Zilla, Mantis, DevTrack and JIRA, all track bugs and issues effectively. However, only a limited number of them provide us with effort information. Since effort data is not directly related to the issue being tracked, most repositories do not keep a record of it. In addition, it takes extra effort on part of the issue tester to keep track of effort. Another common problem with testers recording effort, is the tendency for humans to over state their own effort, rendering the data in accurate.

JIRA is a project management and issue tracking system, which companies can use to manage bugs, features, tasks, improvements or any issue related to their projects. What makes JIRA particulary useful, is the fact it keeps track of the actual effort spent on an issue. While JIRA provides the utility to record the effort, it is not mandatory. Therefore, we need to filter the available projects to obtain the ones which do record the effort values. A number of open source projects are currently being tracked through JIRA such as: Apache, Spring, Flex, Codehaus, JBoss and more. For our purposes, the only projects that contain enough issues with enough recorded actual effort values are JBoss and Codehaus. Weiss *et. al.* [82] use the JBoss data set (with issues tracked up until 05/05/2006) to evaluate their

approach. In order to provide a comparison basis, we use the same data set for the evaluation of our approach. In addition, we use the Codehaus data set (with issues tracked until 01/03/2008) to evaluate our final composite approach, which we also compare against *Base* approach.

JBoss is a division of RedHat that develops and maintains a range certified open source middle-ware projects based on the J2EE platform. The JBoss community projects sit between application code and the operating system to provide services such as persistence, transactions, messaging and clustering. The JIRA JBoss issue tracking branch currently tracks 85 different projects and just under 35,000 issues. The issues can be broken up into about 15,000 bugs, 7,000 Feature Requests, 8,000 tasks and the rest are assorted issues. Codehaus is a project that provides a collaborative environment for other open source projects needed to meet real world needs. The JIRA Codehaus issue tracking branch tracks more than 100 different projects and about 56,000 issues. About 28,000 are bugs, 5,000 are feature requests and another 5,000 are tasks. Table 5.1, summarizes the above statistics describing each case study.

JIRA Branch	Projects	Issues	Bugs	Feature Requests	Tasks
JBoss	85	35,000	15,000	7,000	8,000
Codehaus	> 100	56,000	28,000	5,000	5,000

Table 5.1: Summary of Statistics Describing Case Studies

The JBoss data set we use comprises of about 600 issues, and the Codehaus data set comprises of about 500 issues. Although we rely on the same evaluation method and performance measures as described in [82], we use a smaller set of testing issues. Since the issue time-stamp determines the size of the training set, some test issues will have a smaller training set than others. For that reason, we limit our testing set to the most recent 300 issues. This makes the size of the training sets more uniform across all test issues. In reality, this should not be a problem unless the project is in its initial phases.

JIRA provides the following easily accessible issue information (properties): (*Project Name, Title, Description, Summary, Type, Priority, State, Resolution, Reporter, Assignee*). Conforming to the criteria set by Weiss *et al.* in [82], we consider a limited number of categories for each property. Priority has 5 categories: (*Blocker, Critical, Major, Minor, Optional*), State has 3: (*Closed, Resolved, Reopened*), Resolution has 3: (*Done, Out of Date, Unresolved*) and finally, Type has 4: (*Bug, Task, Sub-Task, Feature Request*).

We use these properties for the *Data Enrichment* enhancement (as query criteria) and for the *Binary Clustering* enhancement (as clustering criteria). After correlation analysis, we found that the following setup gives the best results (as described in Sections 4.2.1 and 4.2.4):

- JBoss Dataset:
  - Data Enrichment: (*Title, Description, Project Name, Type, Priority, State*).
  - Binary Clustering: (*Type*).
- Codehaus Dataset:
  - Data Enrichment: (*Title, Description, Project Name*).
  - Binary Clustering: we use two step clustering, first using (*Project Name*), then issues that do not receive a prediction we cluster using (*Type*). This was due to the fact that some projects had only 1 issue.

5.2 Implementation Tools

To extract the issue information from JIRA [5], we developed PHP [30] (v5.2.6) scripts to crawl, collect and filter the information. Only issues that contained actual effort information were retained into our MySQL [2] (MySQL Community Server v5.0) database. We also, used PHP scripts to populate, extract and format information from the database. PHP simplifies string manipulation, in addition to database interaction, which was the reason we chose this scripting language to perform the described functions.

We use the Java Programming Language [35] (Java SE v6) to generate the similarity scores, mainly due to the fact that Lucene [28] is developed in Java. We used version 2.3 of Lucene, which was the most up to date version available on the Apache Foundation web site. We used the *Multi-Field Query* provided by Lucene to query the different issue information, in addition to the "time" filter (to eliminate issues newer than the target issue) and "stop-word" filter (to eliminate common English stop-words).

Finally, we implemented the proposed approach and performed the data analysis using MATLAB [36] (vR2007b). The way MATLAB handles vectors and large data sets simplifies the implementation of our approach, in addition to the set of useful statistical analysis tools that are embedded into it. To exchange the required data between the different phases described above, we use ".csv" files.

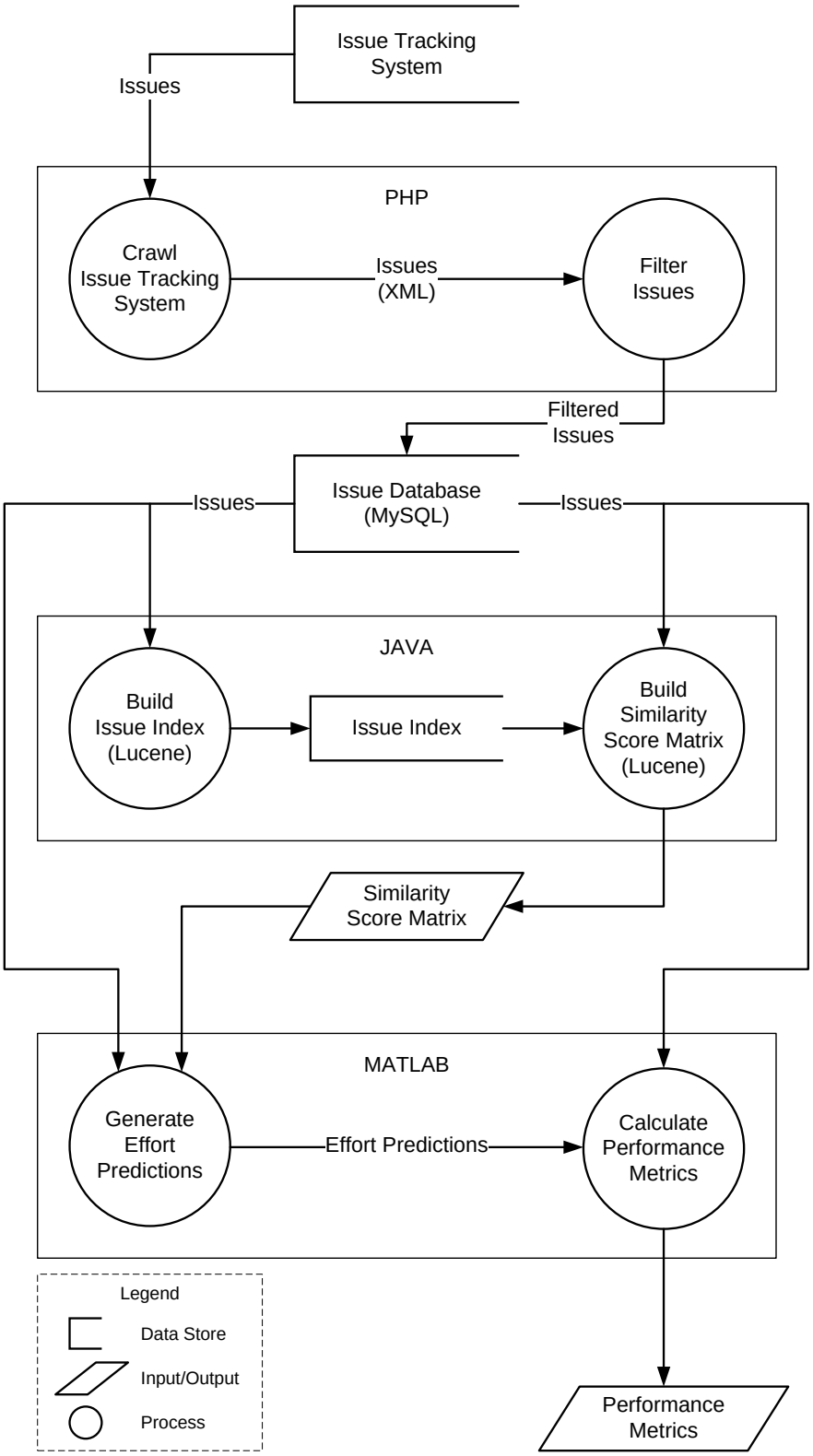

Figure 5.1: Illustration Describing the Implementation Architecture and the Tools

Figure 5.1 describes the general implementation architecture we used to evaluate the proposed framework. It highlights the high level process architecture, grouping the different components by their corresponding implementation language. It also illustrates the interactions between the different processes and the data output of each. The "Filter Issues" process filters out any issues that do not conform to the categories described in Section 5.1, and those that do not have a record of the actual effort. To keep the illustration simple and easy to understand, we emitted some details of intermediate steps that we took to format the data into a readable format for MATLAB. More specifically, the "Issue" data that MATLAB uses from the "Issue Database", goes through a PHP script that formats it into ".csv" files that are easily readable in MATLAB. The "Similarity Score Matrix" is also outputted in ".csv" form, however, it is directly formatted in JAVA.

The following section will present the results of evaluating our proposed effort prediction framework. It will also describe the evaluation method and the performance metrics we use in the study.

5.3 Framework Evaluation

In this section we present the formulation and results of the experimental study performed to evaluate the performance of our proposed framework. In Subsection 5.3.1, we describe the evaluation method in addition to the evaluation measures or metrics used in the study. Subsection 5.3.2 presents the results obtained from performing the experimental study. It illustrates the effect of introducing the enchantments individually, then provides a comparison between the effort prediction framework and the *Base Approach*.

5.3.1 Performance Metrics

To evaluate the performance of the predictions, we retrace the history of the Issue Tracking System. In other words, we consider each issue in the data set as a target issue (test issue), but we only use issues that have been submitted before the target issue in the training set. The training set is then used to search for the nearest neighbours. This means that for the first submitted issue, we cannot make a prediction. This is the reason why we limit our testing set to the most recent 300 issues. We use the following performance measures in our evaluation:

First, we define the *residual* (or error) r_i as the absolute difference between the predicted effort $\hat{e}_i$ and the actual effort e_i reported for an issue i. Clearly, the lower the residual, the more accurate is the prediction.

$$r_i = |e_i - \hat{e}_i| = |\text{Actual Effort} - \text{Estimated Effort}|$$

Average Absolute Residual: also known as the Mean Absolute Error, is defined as follows:

$$AAR = \frac{\sum_{i=1}^{n} r_i}{n}$$

This measure assumes that both over and under estimates are equally bad, which might not be true in all situations. However, it is closely related to the Sum of Absolute Residuals, which considered an unbiased statistic [50].

Percentage of predictions within ±x%: Pred(x) is a commonly used performance metric, which measures the percentage of predictions that lie within $\pm x\%$ of the actual effort value e_i. Once again, this measure does not distinguish between over and under estimates. We use Pred(25) and Pred(50) for the purposes of our experiments.

$$Pred(x) = \frac{|\{i|r_i/e_i \leq x/100\}|}{n}$$

FEEDBACK: this metric measures the percentage of issues for which the approach makes a prediction. In some cases, α-*kNN* reports "Unknown" rather than making a prediction, this metric allows us to measure the percentage of time this occurs. For *kNN* and α-*kNN* with $\alpha = 0$, *FEEDBACK* is 100%.

Interpreting *AAR* values is simple; larger number of good predictions results in a smaller *AAR* value, while a larger number of bad predictions results in a larger *AAR* value. However, *AAR* is easily influenced by outliers; i.e. very large residuals can lead to misinterpretations of the model's performance. For that reason, we also report the Pred(25) and Pred(50) measures, to give a better picture of the residuals' distribution. Higher Pred(x) values mean a better quality of predictions.

In the next subsection we discuss the results obtained from performing the experimental studies on the above mentioned case studies.

5.3.2 Discussions on Obtained Results

The following subsections will describe the results obtained from performing the evaluation study. In the "Enhancement Evaluation" subsection, we will present the results of applying the individual enhancements to the *Base Approach*. This will help the reader understand the effect of each enhancement on the performance of the predictions. Then, in the "Framework Evaluation" subsection, we will provide a performance comparison between the effort prediction framework and the *Base Approach*. We will also present the results of applying the final framework to the additional Codehaus dataset, extending the validity of our work.

Enhancement Evaluation

The following charts will only illustrate the α-*kNN* approach. Although the enhancements do in fact improve the *kNN* approach, our focus is to show improvements on the best results of the *Base* approach, as shown in [82]. Figures 5.2, 5.3, 5.4

and 5.5 show the performance charts describing the our approach using *Historical Value Frequency* for weight computation (based on using the most recent 300 issues).

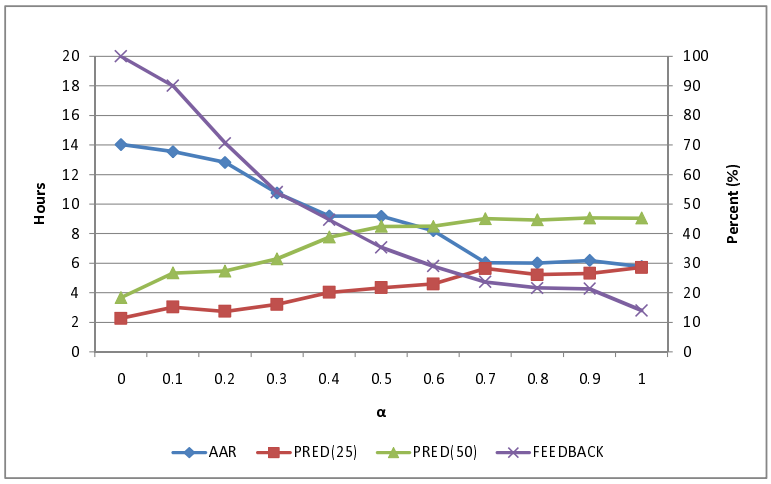

Figure 5.2: Performance Metrics for the Base Approach (JBoss Dataset).

Figure 5.2 shows the performance results for the *Base Approach*, using the *Nearest Neighbours with Thresholds* method (α-*kNN*), with $k = \text{inf}$. In this chart, when $\alpha = 0$, the prediction is basically the mean of the entire training set. We can see that α-*kNN* performs the best at $\alpha = 1$, with $AAR \approx 6$ Hours, *PRED(25)* $\approx 30\%$ and *PRED(50)* $\approx 45\%$. However, *FEEDBACK* reaches as low as 14%. α-*kNN* shows a noticeable improvement as α increases. Starting at $\alpha = 0.1$, with $AAR \approx 14$ Hours, *PRED(25)* $\approx 15\%$ and *PRED(50)* $\approx 25\%$, we get approximately a 15% improvement for *PRED(25)* and a 20% improvement for *PRED(50)*. The most significant improvement is in AAR, where we get an improvement of about 8 Hours.

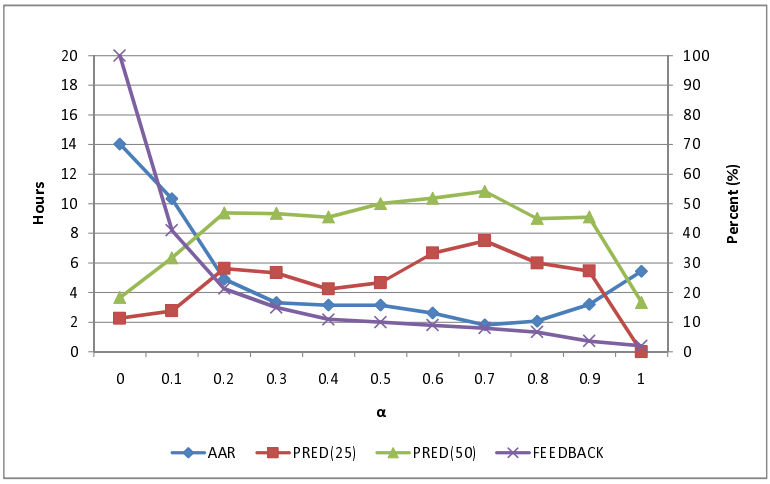

Figure 5.3: Performance Metrics for the Base Approach with Data Enrichment (JBoss Dataset).

From Figure 5.3, we can see that introducing *Data Enrichment* does in fact improve upon the *Base Approach* results. At its best point, *Data Enrichment* has: $AAR = 2$ Hours, *PRED(25)* $= 35\%$ and *PRED(50)* $= 55\%$, compared to the *Base Approach* (shown in Figure 5.2): $AAR = 6$ Hours, *PRED(25)* $= 30\%$ and *PRED(50)* $= 45\%$. This translates into an improvement of 4 Hours for AAR, 5% for *PRED(25)* and 10% for *PRED(50)*. The overall *FEEDBACK* is lower than that of the *Base Approach* (as expected and explained in Section 4.2.1 of Chapter 4). We notice some unexpected behavior for $\alpha > 0.7$; if we look at the *FEEDBACK* beyond that point we notice that it is very low ($< 10\%$). We attribute the sudden change in behavior of the metrics to the low *FEEDBACK*; when it is this low, any errors in the scoring process are magnified and can greatly influence the results. Since *Data Enrichment* is aimed at improving the accuracy of the similarity scores, we can see that reflected in the rapid drop in AAR even for low values of α (< 0.4). AAR is reduced from about 14 Hours to less than 4 hours at $\alpha = 0.3$.

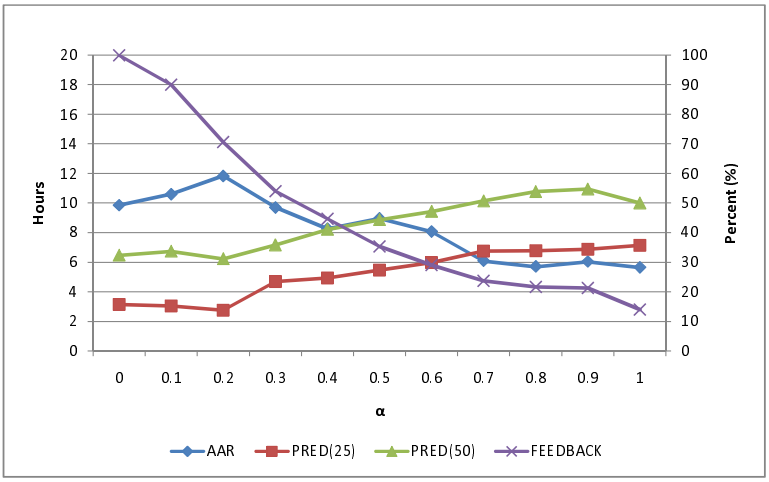

Figure 5.4: Performance Metrics for the Base Approach with Majority Voting (JBoss Dataset).

Figure 5.4 illustrates that introducing *Majority Voting* produces a more consistent improvement over all values of α. It outperforms the *Base Approach* results in terms of AAR for $\alpha < 0.4$, but then levels off to match it for higher values of α. However, the *PRED* measures show good consistent improvement over all values of α. With results reaching a high of about 55% for *PRED(50)*, *PRED(25)* reaching up to 35% and AAR to a minimum of about 6 Hours. We can observe at least a 5% improvement overall due to introducing *Majority Voting* in place of the mean used by the *Base Approach*. The *FEEDBACK* however is the same as that of the *Base Approach*, since using *Majority Voting* does not modify the percentage of issues for which we make predictions. Rather, it just modifies the method used for prediction, for the same issue similarity-scores.

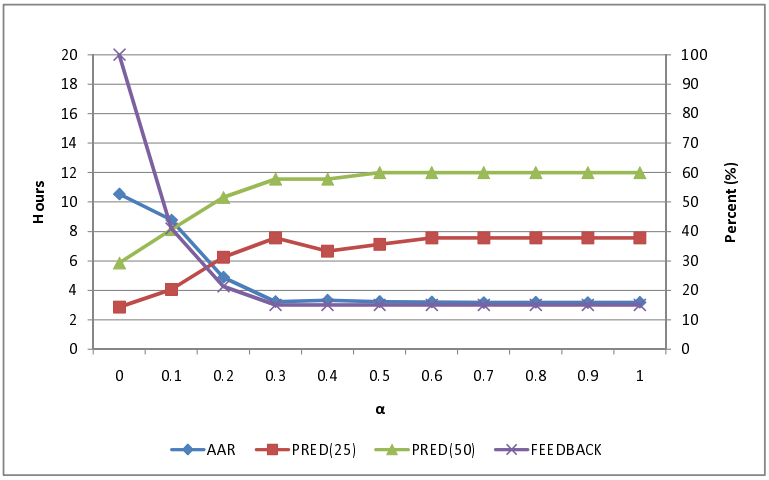

Figure 5.5: Performance Metrics for the Base Approach with Data Enrichment, Majority Voting and Adaptive Feedback (JBoss Dataset).

As shown in Figure 5.5, applying the three *SSDE* to the *Base Approach* shows significant improvement; at the best point we obtain $AAR = 3$ Hours, *PRED(25)* = 38% and *PRED(50)* = 60%. Compared to the best *Base Approach* results ($AAR \approx 6$ Hours, *PRED(25)* $\approx$ 30% and *PRED(50)* $\approx$ 45%); this is about 3 Hours AAR improvement and about 10% improvement for both of the *PRED* measures. Note that with the introduction of *Adaptive Threshold*, the *FEEDBACK* deterioration (caused by the introduction of *Data Enrichment* shown in Figure 5.3) is resolved, as explained in Chapter 3, Section 4.2.

Framework Evaluation

The following will present a comparison between the performance of the effort prediction framework and the *Base Approach*. We will apply the evaluation to both the JBoss and Codehaus datasets to extend the validity of our framework.

Method		AAR	PRED(25)	PRED(50)	RE Max	RE Mean	RE StdDev	FEEDBACK
EPF	LSL	8.9 Hours	23.3 %	41.3 %	79	1.77	5.48	100 %
	HVF	8.9 Hours	23.6 %	41.6 %	79	1.75	5.49	
BA (*kNN*, $k = 1$)		13.8 Hours	15 %	31.7 %	191.00	5.37	19.03	
EPF	LSL	3.3 Hours	35.6 %	57.8 %	6.2	0.82	1.13	15 %
	HVF	3.2 Hours	37.8 %	60.0 %	6.2	0.72	1.07	
BA (α-*kNN*, $\alpha = 1$)		5.8 Hours	28.6 %	45.2 %	12	1.48	2.45	
EPF	LSL	4.9 Hours	34.4 %	53.1 %	9.5	1.03	1.67	21 %
	HVF	4.8 Hours	35.9 %	54.7 %	9.5	0.97	1.65	
BA (α-*kNN*, $\alpha = 0.9$)		6.2 Hours	26.6 %	45.3 %	12	1.17	2.03	

Table 5.2: Comparison between the Effort Prediction Framework and the Base Approach (JBoss Dataset)
EPF - Effort Prediction Framework, BA - Base Approach, LSL - Least Squares Learning, HVF - Historical Value Frequency, RE - Relative Error

The Effort Prediction Framework always produces results with 100% *FEEDBACK*. Therefore, to provide a comparison basis with the α-*kNN Base Approach*, we disable the *Binary Clustering* enhancement, thereby allowing us to control the *FEEDBACK* of the approach using the *SSDE*. We set the *threshold_limit* to a point where *FEEDBACK* is comparable with that returned by the *Base Approach*. As we can see in Table 5.2, there is significant improvement for all *FEEDBACK* cases, even in comparison to the best results produced by the *Base Approach* (α-*kNN*, $\alpha = 1$). The Effort Prediction Framework shows an improvement of more than 3 Hours in *AAR* for the 100% *FEEDBACK* case, and about a 10% improvement for both of the *PRED* measures. As for the other two *FEEDBACK* cases (15% and 21%), we get about 2 Hours of improvement in *AAR* and about a 10% improvement for *PRED(25)*. As for *PRED(50)*, for the 15% *FEEDBACK* case, we see approximately a 15% improvement, and about a 10% improvement for the 21% *FEEDBACK* case. The other metrics (Relative Error Maximum, Relative Error Mean and Relative Error Standard Deviation) are mainly displayed to show the performance difference between the use of the two weight computation methods. For this dataset, we do not see a large difference in performance since the percent of issues that use a *Weighted Mean* is very small as shown in Table 5.3.

FEEDBACK	MV	SM (No MV)	MM (No MV)	BC
100 %	4.3 %	15.3 %	1.7 %	78.7 %
15 %	26.7 %	66.7 %	6.7 %	N/A
21 %	20.3 %	71.9 %	7.8 %	N/A

Table 5.3: Percentage of Issues Predicted by each Method (JBoss Dataset)
MV - Majority Voting, SM - Single Match, MM - Multiple Matches, BC - Binary Clustering

Table 5.3 gives a deeper perspective on the behavior of the Effort Prediction

Framework as described in Table 5.2, by showing the percentage of issues predicted by each method (JBoss Dataset). The methods shown in Table 5.3 refer to the ones described in Chapter 3, and as depicted in Figure 4.1 which describes the process model. We can see four different ways to producing the effort prediction; 1) Majority Voting, 2) Single Similarity Match (when no majority vote is reached), 3) Multiple Similarity Matches (when no majority vote is reached) and 4) Binary Clustering. As reflected by Table 5.3, we can see why the effect of *LSL* vs. *HVF* is not significant for the JBoss Dataset; where the percentage of issues predicted using these methods (for multiple matches when no majority vote can be reached) is less than 10% for all *FEEDBACK* cases, and as low as about 2% for the 100% *FEEDBACK* case.

Method		AAR	PRED(25)	PRED(50)	RE Max	RE Mean	RE StdDev	FEEDBACK
EPF	LSL	5.5 Hours	20.0 %	41.0 %	140.75	1.96	8.71	100 %
	HVF	5.4 Hours	19.3 %	40.7 %	140.75	1.92	8.68	
BA (*kNN*, $k = 2$)		8.8 Hours	18.0 %	35.0 %	261.50	6.34	25.97	
EPF	LSL	8.9 Hours	27.8 %	57.4 %	11.34	1.23	2.13	18 %
	HVF	8.2 Hours	24.1 %	55.6 %	7.08	1.04	1.50	
BA (α-*kNN*, $\alpha = 1$)		12.1 Hours	22.2 %	53.7 %	37.40	2.03	5.44	
EPF	LSL	11.2 Hours	15.39 %	38.5 %	191.00	4.44	19.25	48 %
	HVF	8.8 Hours	12.6 %	37.8 %	265.25	5.06	27.27	
BA (α-*kNN*, $\alpha = 0.35$)		8.5 Hours	14.7 %	35.7 %	261.50	4.52	22.31	

Table 5.4: Comparison between the Effort Prediction Framework and the Base Approach (Codehaus Dataset)
EPF - Effort Prediction Framework, BA - Base Approach, LSL - Least Squares Learning, HVF - Historical Value Frequency, RE - Relative Error

To extend the validity of the Effort Prediction Framework, we apply it to an alternate dataset (Codehaus). From Table 5.4, we can again see clear improvement for all *FEEDBACK* cases. But, we would like to point out that, although we used the same Lucene setup, the scoring was particularly misleading for this dataset. We can see that reflected in the results above for the *α-kNN Base Approach* ($\alpha = 1$), and in the fact that the results using the Least Squares Learning weight computation method performs worse (since it depends on similarity scores). Although the *PRED* measures do show improvement as α increases, the *AAR* metric is affected negatively. This maybe explained by a different style of issue documentation in the Codehaus project; more in depth analysis is needed to identify the exact reason. In this case, *Binary Clustering* greatly helped in improving the results due to its independence of similarity-scores. We can see about a 3 Hour improvement in *AAR* for the 100% *FEEDBACK* case, and about 1% and 5% for the *PRED(25)* and *PRED(50)* measures respectively. For the 18% *FEEDBACK* case, we see about 4 Hours improvement in *AAR* and approximately a 2% improvement for both the *PRED* measures. Finally, for the 48% *FEEDBACK* case, we only see minor but apparent improvement for the *PRED(50)* measure (about 2%), however, the *AAR* and *PRED(25)* suffer slightly (about 0.3 Hours and 1% respectively).

FEEDBACK	MV	SM (No MV)	MM (No MV)	BC
100 %	0.7 %	11.3 %	6.0 %	82.0 %
18 %	3.7 %	63.0 %	33.3 %	N/A
48 %	2.1 %	66.4 %	31.5 %	N/A

Table 5.5: Percentage of Issues Predicted by each Method (Codehaus Dataset) MV - Majority Voting, SM - Single Match, MM - Multiple Matches, BC - Binary Clustering

Table 5.5 shows the percentage of issues predicted by each method for the Codehaus Dataset. In contrast to the JBoss Dataset percentages shown in Table 5.3, we can see that the percentage of issues predicted by *MM (no MV)* is much higher. Especially in the 18% and 48% *FEEDBACK* cases, where it reaches up to 33% and 31%, respectively. This is reflected by the effect of *LSL* and *HVF* on the performance of the predictions in Table 5.4, as compared to Table 5.2 for the JBoss Dataset. Another interesting observation, is the fact that the percentage of issues predicted by *Majority Voting* for the Codehaus Dataset is much lower than that of the JBoss Dataset; Codehaus with a high of about 4% and JBoss with a high of about 27%. This shows the different properties of data sets that the approach must handle. However, the percentage of issues predicted by *Binary Clustering* is very close for both datasets, about 80%.

The following section will summarize what we discussed in this chapter regarding the case studies, implementation tools, evaluation method, performance metrics and the obtained results. It will also summarize the what we learned from the results discussed in this section, and introduce the final chapter of the thesis.

5.4 Summary

In this chapter, we have described the case studies along with the Issue Tracking System (JIRA) we have used to extract them. We also outlined the implementation tools and methods we used to implement our experimental studies. Then, we explained our approach to evaluating the performance of the predictions produced by our approach, along with the performance metrics we used to measure it. We used the same data set (JBoss) used by the Weiss *et al.* study to provide a comparison basis and allow us to assess the relative performance of our approach. Were possible, we compared our framework to the *Base Approach*, by showing the effect of adding each enhancement independently, and in all cases our approach produced competitive results. Finally, we compared the Effort Prediction Framework, showing the effect of using each weight computation technique independently, to the best results produced by the *Base Approach*. We also applied the same comparison to a different data set to extend the validity of our results. In both case studies, our

approach performed better. The results demonstrate that our composite effort prediction framework performs better than a comparable single method approach. The modular nature of the framework allowed us to adapt to the misleading similarity scores and adjust the components to produce more accurate predictions. The enhancements evaluation charts show that while the application of each enhancement individually improved the performance of the predictions, combining the different methods improved it even further.

In the following chapter, we will outline the contributions of the thesis and discuss future directions of this research.

Chapter 6

Conclusion and Future Directions

In this chapter, we summarize the findings of the thesis and outline future directions that could be pursued from this research. Section 6.1 summarizes the contributions of the work presented in the thesis. Then, Section 6.2 outlines some potential future work to extending this research. Finally, Section 6.3 ends with a summary of the findings and some concluding remarks.

6.1 Contributions

This work addressed the problem of predicting effort for defect-correction of issues posted on a defect repository or an Issue Tracking System. We emphasize the fact that the proposed effort prediction framework could be applied to any defect database that records actual effort. Our framework proposes applying a composite method to predicting effort, combining a *Nearest Neighbour* approach with *Binary Clustering*. The framework takes a target issue and a set of resolved historical issues, and generates prediction for the effort needed to resolve the target issue in "man-hours". The major contributions of the framework can be summarized through the enhancements applied to the base *Nearest Neighbours* approach:

- **Data Enrichment:** aims to improve the accuracy of similarity scoring, by introducing additional data related to the target issue into the process.
- **Majority Voting:** uses the repeating historical effort values as a superior predictor to the mean approach. Since these values are observed to be close to the actual effort, this method improves the accuracy of the predictions. This method also improves the framework's resilience against outliers in the data.
- **Adaptive Threshold:** improves the percentage of issues for which the framework makes predictions, by automatically adding more issues into the set of similar historical matches.

- **Binary Clustering:** introduces an approach that does not depend on similarity scores, which helps improve predictions if the scores are deemed misleading. It uses common information about issues to group and use them as a prediction basis.

We also propose two weight computation techniques, which we use in cases where the model reverts to a *Weighted Mean* method for computing the predictions (when multiple similarity matches are available, but no majority vote can be reached). These methods are designed to be modular, and can be replaced by any method the user deems appropriate or more effective. The *Weighted Mean* effort prediction method (as compared to the mean) allows the user greater control over the influence of particular issues on the prediction. The following describes the two techniques used in our study:

- **Least Squares Learning:** uses similar issues to compute weights based on regression analysis, using the Least Squares approach (also known as the Mean Square Error measure). If the similarity scores are accurate, this technique improves the accuracy of the predictions by attempting to minimize the Mean Square Error for historical issues.
- **Historical Value Frequency:** uses a historical count of effort values to compute the corresponding weights. This method relies on the discrete nature of effort values in the Issue tracking System. In cases where the similarity scores are deemed to be misleading, this method computes weights independent of these scores, reducing their negative effect on the prediction.

The framework is also designed to be modular, extensible and applicable in practice. Additional components can be incorporated into the framework, or existing modules can be updated easily. For example, the *Binary Clustering* enhancement can be replaced with an equivalent similarity-score independent method, or added in conjunction. The *Majority Voting* prediction method can also be replaced with an equivalent, such as *Regression* or *Clustering*, as long as the required information is available. Also, since the evaluation studies applied the framework to case studies taken from an operational Issue Tracking System, it can be used in practice with local calibration.

The following section will outline future directions of this research, and describe some the work that could extend on the current framework.

6.2 Future Work

Our approach taps into a relatively unexplored area of defect correction effort prediction, namely the composite approaches. We believe that there are numerous ways to improve our approach; some lie into improving the scoring mechanism and others in exploring alternative Similarity-Score Independent approaches:

- If we can extract additional data from the Issue Tracking System, such as source code of files to be changed early in the life of the issue; methods like the one proposed by Mirarab *et al.* in [55], can be used to predict the magnitude of change caused by correcting the issue in question. In [55], the authors propose the use of Bayesian Belief Networks to predict module or code change propagation in software systems. If we can obtain the code or the estimated code to be changed in the system, we can use such methods to predict the effect of these changes on the system, and empirically derive an estimate of code change and from that an effort estimate. However, currently, the problem with this approach is the fact that information like changed source code is only available later in the life time of an issue, rendering this approach unfeasible.

- Better methods need to be implemented to identify similarity among issues, in order to utilize the full potential benefits of the Similarity-Score Dependent approaches. As for text-similarity approaches, such as the use of Lucene to measure similarities. We need to explore additional, richer ways, of combining the strings in a query. Currently, the query strings are simply OR'ed together, which is the way Lucene implements the Multi-Field Query. However, Lucene allows us to define our own query classes, which we can use to implement more complex and smarter ways of combining the field strings. Although this may seem like a simplistic remark, we believe it will affect the accuracy of the scores significantly, if implemented correctly. Moreover, when considering similarity scores, one can develop a more through model per project by using methods like regression analysis, and use it as a support tool along with the *Nearest Neighbour* approach to arrive at more informed estimates.

- As outlined in Chapter 2, there are a number of approaches used in literature to perform the predictions using the *Nearest Neighbour* approach, one of which we have not explored, namely *Top-K Regression*. Since regression analysis is a widely used technique in software cost estimation, we believe it might prove to be an effective tool when implemented with our approach. Also, other common Artificial Intelligence methods such genetic algorithms and Neural Networks are prime candidates to be used for prediction of defect correction effort, especially as support methods for Similarity-Score Independent approaches. Exploring alternate Similarity-Score Independent enhancements will definitely be beneficial; as we have seen that any errors in the scoring process can have a significant negative impact on the final results.

- We have implemented two weight computation methods, namely *Least Squares Learning* and *Historical Value Frequency*. We believe that further refining of the candidate sets and better filtering algorithms will greatly improve the performance of these two methods. Moreover, additional weight computation methods need to be explored and implemented to maximize the performance of the prediction models, as we have seen a noticeable improvement in the results using the above two techniques. In general, the area of defect effort

prediction is relatively new, and many breakthroughs are awaited before full trust can be developed into such systems.

- Since we have designed the effort prediction framework to be extensible, it is easy to replace or update the different components. For this reason, we can encourage further experimentation into alternative effort predictors and similarity-score independent prediction methods. For example, alternative methods to *Majority Voting* can be investigated, such as *Clustering*. It may prove beneficial to study the behavior of the different prediction methods and their relation to the properties of the case studies.This would give practitioners valuable insight into the effectiveness of the different methods for different data properties. Similarly, additional similarity-score independent prediction methods can be incorporated into the framework to enhance the accuracy of the predictions, in cases where the similarity scores are deemed misleading.

The next section will provide a summary of the framework and findings, in addition to some concluding remarks regarding the research involved in the thesis.

6.3 Conclusion

In this work, we have presented a framework for predicting defect correction effort. We discussed how it can be used to produce the predictions and how such approaches can be superior to others. While it is based on a *Nearest Neighbour* approach, it also benefits from a *Binary Clustering* approach which relies on its own interpretation of the data.

First, we described a *Base Approach* on which we apply a set of enhancements that make up the core of the composite effort prediction framework. The *Base Approach* describes the text-similarity scoring technique we use to generate the distance measures needed for the *Nearest Neighbour* approach. Then two *Nearest Neighbour* prediction approaches are described, from which we use the superior performing *Nearest Neighbours with Thresholds* (α-*kNN*) in our framework. There are four enhancements that make up the effort prediction framework: *Data Enrichment* (improves the accuracy of the similarity scores), *Adaptive Threshold* (improves the percentage of issues for the framework makes predictions), *Majority Voting* (improves the framework's resilience against outliers and capitalizes on the discrete nature of effort values) and *Binary Clustering* (improves the accuracy of the prediction in cases where the similarity scores are considered misleading). In addition to the above enhancements, we propose two weight computation techniques used in cases where the framework reverts to computing the effort using a *Weighted Mean* method. *Weighted Mean* is used when multiple similarity matches are available, but no majority vote can be reached. The two techniques are: Least Squares Learning and Historical Value Frequency. Least Squares Learning uses regression analysis and tries to minimize the Mean Square Error in order to compute the best possible

weights. This method of weight computation is dependent on similarity scores. On the other hand, Historical Value Frequency, relies on the discrete nature of effort values and computes the weights by using the frequency of a certain value in history. This method of weight computation is independent of the similarity scores.

We have evaluated the effectiveness and performance of the effort prediction framework on two open source data sets extracted from an operational Issue Tracking System. Although each data set had its own style of issue records, the composite model performed favorably for both. We also compared our approach with a similar existing study, and again we have shown that our enhancements produced significantly better results. To demonstrate the effect of each enhancement on the performance of the framework, we showed how each influences the predictions and compared the results with the *Base Approach*. It was evident that the proposed enhancements improved the prediction performance significantly; when applying the three enhancements (*Data Enrichment, Adaptive Threshold* and *Majority Voting*), *PRED(50)* and *AAR* were improved by about 50% and *PRED(25)* by 30%. We also compared the effort prediction framework performance to the *Base Approach* for the two data sets, were we also saw noticeable improvement. When the framework is performing at 100% *FEEDBACK*, we saw about 35% improvement in *AAR*, 50% improvement in *PRED(25)* and 30% improvement in *PRED(50)* for the one of the case studies. Similarly, we saw about 30% improvement in *AAR*, 1% improvement in *PRED(25)* and 14% improvement in *PRED(50)* for the other case study. For future work, we can apply the effort prediction framework to additional case studies, as more projects are added to the Issue Tracking System, to extend its validity.

We have shown that using a composite approach to predict effort is indeed an effective strategy. Intrinsically, issues have different properties and features that we should exploit to adapt the best approach for prediction. An important feature of our approach is that it can be applied to any issue database that records effort information. It can leverage any information available to build similarities and predict effort. With the power of multiple prediction methods, the weaknesses of each individual approach are remedied and the approach becomes more adaptive. Adaptive approaches are the next step in effort prediction and we believe more research should be applied to this area.

References

[1] A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994. 8

[2] MySQL AB. Mysql :: The world's most popular open source database, June 2008. http://www.mysql.com/. 39

[3] M. Agrawal and K. Chari. Software effort, quality, and cycle time: A study of CMM level 5 projects. *IEEE Transactions on Software Engineering*, 33(1):145–156, 2007. 7

[4] W. Agresti and W. Evanco. Projecting software defects from analyzing ada designs. *IEEE Transactions on Software Engineering*, 18(11):988–997, 1992. 11

[5] Atlassian. Jira - bug tracking, issue tracking and project management software, June 2008. http://www.atlassian.com/software/jira/. 39

[6] A. Ben-Dor, L. Bruhn, N. Friedman, I. Nachman, M. Schummer, and Z. Yakhini. Tissue classification with gene expression profiles. In *Proceedings of the International Conference on Research in Computational Molecular Biology*, pages 54–64, 2000. 13

[7] R. Berk. *Regression Analysis: A Constructive Critique*. Sage Publications, 2004. 9, 13

[8] S. Bibi and I. Stamelos. *Artificial Intelligence Applications and Innovations*. Springer Boston, 2006. 9

[9] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. 1, 8

[10] B. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, 1984. 1, 7, 8, 9

[11] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches a survey. *Annals of Software Engineering*, 10(1-4):177–205, 2000. 7, 8, 10

[12] B. Boehm and V. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. 1

[13] G. Boetticher. Using machine learning to predict project effort: Empirical case studies in data starved domains. Technical report, Department of Software Engineering, University of Houston, Texas, USA, 2001. http://nas.cl.uh.edu/boetticher/MBRE01_Paper.pdf. 9

[14] L. Briand, K. El Emam, D. Surmann, I. Wieczorek, and K. Maxwell. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the International Conference on Software Engineering*, pages 313–322, 1999. 7

[15] C. Burgess and M. Lefley. Can genetic programming improve software effort estimation? a comparative evaluation. *Information and Software Technology*, 43(14):863–873, 2001. 9

[16] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4):573–583, 1999. 9

[17] B. Compton and C. Withrow. Prediction and control of ADA software defects. *Journal of Systems and Software*, 12(3):199–207, 1990. 11

[18] M. Diaz and J. Sligo. How software process improvement helped motorola. *IEEE Software*, 14(5):75–81, 1997. 11

[19] M. Dyer. *The cleanroom approach to quality software development*. John Wiley & Sons Inc, 1992. 11

[20] A. Dymo. Software development time estimation using similarity theory. Technical report, National University of Shipbuilding, Mykolaiv, Ukraine, 2005. http://dymo.mk.ua/files/similarity_en.pdf. 15

[21] T. Ebbels, H. Keun, O. Beckonert, M. Bollard, J. Lindon, E. Holmes, and J. Nicholson. Prediction and classification of drug toxicity using probabilistic modeling of temporal metabolic data: The consortium on metabonomic toxicology screening approach. *Journal of Proteome Research*, 6(11):4407–4422, 2007. 12, 15

[22] W. Evanco. Modeling the effort to correct faults. In *Selected papers of the sixth annual Oregon workshop on Software metrics*, pages 75–84, 1995. 11

[23] W. Evanco. Prediction models for software fault correction effort. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, page 114, 2001. 11

[24] W. Evanco and W. Agresti. A composite complexity approach for software defect modelling. *Software Quality Journal*, 3(1):27–44, 1994. 11

[25] R. Fairley. Recent advances in software estimation techniques. In *Proceedings of the International Conference on Software Engineering*, pages 382–391, 1992. 7

[26] N. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. 11

[27] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion MMRE. *IEEE Transactions on Software Engineering*, 29(11):985–995, 2003. 11

[28] The Apache Software Foundation. Welcome to lucene, June 2008. http://lucene.apache.org/. 39

[29] A. Gray and S. MacDonell. A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology*, 39(6):425–437, 1997. 1, 8

[30] The PHP Group. Php: Hypertext preprocessor, June 2008. http://www.php.net/. 39

[31] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001. 2, 11

[32] T. Hastings and A. Sajeev. A vector-based approach to software size measurement and effort estimation. *IEEE Transactions on Software Engineering*, 27(4):337–350, 2001. 9

[33] E. Hatcher and O. Gospodnetic. *Lucene in Action (In Action series)*. Manning Publications Co., 2004. 19

[34] O. Helmer. *Social Technology*. Basic Books, 1966. 9

[35] Sun Microsystems Inc. Developer resources for java technology, June 2008. http://java.sun.com/. 39

[36] The MathWorks Inc. Matlab - the language of technical computing, June 2008. http://www.mathworks.com/products/matlab/. 39

[37] K. Iwata, Y. Anan, T. Nakashima, and N. Ishii. Effort prediction model using similarity for embedded software development. In *Proceedings of the International Conference on Computational Science and Applications*, pages 40–48, 2006. 14

[38] H. Jo, I. Han, and H. Lee. Bankruptcy prediction using case-based reasoning, neural networks, and discriminant analysis, expert systems with applications. *Expert Systems with Applications*, 13(2):97–108, 1997. 15

[39] P. Johnson, C. Moore, J. Dane, and R. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6):51–56, 2000. 7

[40] C. Jones. *Applied Software Measurement.* McGraw Hill, 1997. 8

[41] M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004. 7, 9

[42] M. Jørgensen. Practical guidelines for expert-judgment-based software effort estimation. *IEEE Software*, 22(3):57–63, 2005. 9

[43] M. Jørgensen, U. Indahl, and D. Sjoberg. Software effort estimation by analogy and regression toward the mean. *Journal of Systems and Software*, 68(3):253–262, 2003. 8, 27

[44] M. Jørgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007. 7

[45] G. Judge, W. Griffith, and C. Hill. *Learning and Practicing Econometrics.* Wiley, 1993. 9

[46] N. Kapadia, J. Fortes, and C. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, page 6, 1999. 14

[47] N. Karunanithi, D. Whitley, and Y. Malaiya. Using neural networks in reliability prediction. *IEEE Software*, 9(4):53–59, 1992. 1, 8

[48] G. Kenny. Estimating defects in commercial software during operational use. *IEEE Transactions on Reliability*, 42(1):107–115, 1993. 11

[49] S. Kim and H. Noh. Predictability of interest rates using data mining tools: a comparative analysis of korea and the us, expert systems with applications. *Expert Systems with Applications*, 13(2):85–95, 1997. 17

[50] B. Kitchenham, L. Pickard, S. MacDonell, and M. Shepperd. What accuracy statistics really measure. *IEE Proceedings - Software*, 148(3):81–85, 2001. 11, 12, 42

[51] H. Li, D. Groep, and L. Wolters. Efficient response time predictions by exploiting application and resource state similarities. In *Proceedings of the IEEE/ACM International Workshop on Grid Computing (GRID)*, pages 234–241, 2005. 15

[52] K. Manzoor. A practical approach to estimating defect-fix time. 12

[53] E. Mendes, I. Watson, C. Triggs, N. Mosley, and S. Counsell. A comparison of development effort estimation techniques for web hypermedia applications. In *Proceedings of the International Symposium on Software Metrics*, page 131, 2002. 7

[54] Politecnico Di Milano. Clustering, 2002. http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/. 13

[55] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pages 177–188, 2007. 53

[56] K. Moller and D. Paulish. An empirical investigation of software fault distribution. pages 82–90, 1993. 11

[57] L. Nassif, J. Nogueira, A. Karmouch, M. Ahmed, and F. Andrade. Job completion prediction using case-based reasoning for grid computing environments: Research articles. *Concurrency and Computation: Practice and Experience*, 19(9):1253–1269, 2007. 15

[58] P. Nesi and T. Querci. Effort estimation and prediction of object-oriented systems. *Journal of Systems and Software*, 42(1):89–102, 1998. 9

[59] F. Padberg, T. Ragg, and R. Schoknecht. Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering*, 30(1):17–28, 2004. 7

[60] R. Park. The central equations of the PRICE software cost model, 1988. 8

[61] P. Pendharkar, G. Subramanian, and J. Rodger. A probabilistic model for predicting software development effort. *IEEE Transactions on Software Engineering*, 31(7):615–624, 2005. 9

[62] A. Phansalkar and L. John. Performance prediction using program similarity. In *Proceedings of the Standard Performance Evaluation Corporation Benchmark Workshop (SPEC)*, 2006. 13, 15

[63] L. Putnam and W. Myers. *Measures for Excellence*. Yourdon Press Computing Series, 1992. 7, 8

[64] F. Ramirez, O. Fuentes, and R. Gulati. Prediction of stellar atmospheric parameters using instance-based machine learning and genetic algorithms. *Experimental Astronomy*, 12(3):163–178, 2001. 16

[65] C. Ratanamahatana. 3G customer prediction using time series similarity measure. Technical report, Department of Computer Engineering, Chulalongkorn University, Bangkok, Thailand, 2006. http://www.ntu.edu.sg/sce/pakdd2006/competition/Open31.pdf. 16

[66] H. Rubin. *ESTIMACS*. IEEE, 1983. 8

[67] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. 2, 8, 11

[68] J. Santos, A. Carrasco, and J. Martinez. Feature selection using typical testors applied to estimation of stellar parameters. *Computacion Sistemas*, 8(1):15–23, 2004. 16

[69] C. Schofield. Non-algorithmic effort estimation techniques, 1998. 8

[70] SELECT. Estimation for component-based development using SELECT estimator, 1998. http://www.selectst.com. 8

[71] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(11):736–743, 1997. 7, 8, 20, 27

[72] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001. 13

[73] M. Shin and A. Goel. Empirical data modeling in software engineering using radial basis functions. *IEEE Transactions on Software Engineering*, 26(6):567–576, 2000. 1, 7, 8

[74] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, 1998. 14

[75] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 202–219, 1999. 14

[76] W. Smith and P. Wong. Resource selection using execution and queue wait time predictions. Technical report, NASA Ames Research Center, 2002. http://www.nas.nasa.gov/News/Techreports/2002/PDF/nas-02-003.pdf. 15

[77] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006. 11

[78] J. Sun and X. Hui. Financial distress prediction based on similarity weighted voting CBR. *Lecture Notes in Computer Science, Springer*, 4093/2006:947–958, 2006. 12, 16

[79] A. Sykes. An introduction to regression analysis. Technical report, Law School, University of Chicago, 1999. http://www.law.uchicago.edu/Lawecon/WkngPprs_01-25/20.Sykes.Regression.pdf. 14

[80] I. Tronto, J. Simöes da Silva, and N. Sant'Anna. Comparison of artificial neural network and regression models in software effort estimation. Technical report, 2006. http://mtc-m18.sid.inpe.br/col/sid.inpe.br/ePrint%4080/2006/12.08.12.47/doc/v1.pdf. 7, 9

[81] S. Weisberg. *Applied Linear Regression*. John Wiley and Sons, 1985. 9, 13

[82] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*, pages 1–10, 2007. 2, 11, 15, 19, 20, 21, 23, 26, 27, 30, 31, 33, 37, 38, 42

[83] Wikipedia. Software development process, June 2008. http://en.wikipedia.org/wiki/Software_development_process. 1

[84] C. Wohlin and P. Runeson. Defect content estimations from review data. In *Proceedings of the 20th International Conference on Software Engineering*, pages 400–409, 1998. 11

[85] Hui Zeng and David Rine. Estimation of software defects fix effort using neural networks. In *Proceedings of the International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC)*, pages 20–21, 2004. 11, 12

[86] A. Zhu, A. Moore, and J. Burt. Prediction of soil properties using fuzzy membership. Technical report, State Key Lab of Resources and Environmental Information System, Institute of Geographical Sciences and Natural Research, Chinese Academy of Sciences, Beijing, China, 2006. 14, 16