SPRINGER BRIEFS IN COMPUTER SCIENCE

Sandeep Kumar
Santosh Singh Rathore

Software Fault Prediction

A Road Map

Springer

SpringerBriefs in Computer Science

SpringerBriefs present concise summaries of cutting-edge research and practical applications across a wide spectrum of fields. Featuring compact volumes of 50 to 125 pages, the series covers a range of content from professional to academic.

Typical topics might include:

- A timely report of state-of-the art analytical techniques
- A bridge between new research results, as published in journal articles, and a contextual literature review
- A snapshot of a hot or emerging topic
- An in-depth case study or clinical example
- A presentation of core concepts that students must understand in order to make independent contributions

Briefs allow authors to present their ideas and readers to absorb them with minimal time investment. Briefs will be published as part of Springer's eBook collection, with millions of users worldwide. In addition, Briefs will be available for individual print and electronic purchase. Briefs are characterized by fast, global electronic dissemination, standard publishing contracts, easy-to-use manuscript preparation and formatting guidelines, and expedited production schedules. We aim for publication 8–12 weeks after acceptance. Both solicited and unsolicited manuscripts are considered for publication in this series.

More information about this series at http://www.springer.com/series/10028

Sandeep Kumar · Santosh Singh Rathore

Software Fault Prediction

A Road Map

Springer

Sandeep Kumar
Department of Computer Science
and Engineering
Indian Institute of Technology Roorkee
Roorkee
India

Santosh Singh Rathore
Department of Computer Science
and Engineering
National Institute of Technology Jalandhar
Jalandhar
India

ISSN 2191-5768 ISSN 2191-5776 (electronic)
SpringerBriefs in Computer Science
ISBN 978-981-10-8714-1 ISBN 978-981-10-8715-8 (eBook)
https://doi.org/10.1007/978-981-10-8715-8

Library of Congress Control Number: 2018942190

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
part of Springer Nature
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

Software quality assurance (SQA) is a vital and foremost important task to build robust software and to ensure that the developed software meets the standardized quality specifications. There are many parameters/measurements used to measure the quality of the software system. One such measure is the fault-proneness information of the software modules. The presence of faults not only reduces the quality of the software but also increases the development cost of the system. Thus, ensuring lower faults in software system ensures a higher quality of the software. Software fault prediction (SFP) is one such activity, which is used to predict the fault-proneness of the software system prior to the testing process. A large number of software fault prediction models can be found in the literature. Most of these models have used the historical software data, the previously revealed software faults, and metric information to predict the fault-proneness of the software modules. In general, the developed fault prediction model is used to predict that whether a software module is faulty or non-faulty. This book is focused on exploring the use of software fault prediction in building reliable and robust software systems. First, we introduce the basic concepts related to software fault prediction process and discuss its generalized architecture. We also discuss different types of fault prediction models presented in the literature. Subsequently, we discuss different works presented earlier for predicting software modules being faulty or non-faulty. At last, we present an evaluation of different techniques for the software fault prediction and discuss their results. This book also covers the details of the software fault datasets and discusses their different issues with respect to software fault prediction. In addition to various important works reported in this area, some of reported works in this domain are also summarized. The book has been organized as follows. Chapter 1 introduces the basic concepts of software fault prediction and various terminologies. Chapter 2 explains the generalized architecture of software fault prediction process and discusses its different components. Chapter 3 provides the details of types of fault prediction models and discusses the state-of-the-art literature of each model. Chapter 4 describes the software fault datasets and different issues of fault datasets when building fault prediction models. Chapter 5 presents an empirical study to evaluate various fault prediction techniques with reference to

binary class prediction. Chapter 6 presents another study evaluating the techniques for the prediction of number of faults in the software modules. The book concludes with Chap. 7, which provides the summary of the discussed works. The primary contribution of the book lies in presenting a single source of information for software engineers and researchers for learning about the area of software fault prediction. The book can also work as an initial source of information for starting research in this domain. In addition, the book can be useful to the experienced researchers in getting summary of latest work reported in this area. We are hopeful that the book will not only provide a good introductory reference but will also give the readers a breadth and depth of this topic.

Roorkee, India Sandeep Kumar
Jalandhar, India Santosh Singh Rathore

Acknowledgements

First, I would like to extend my sincere gratitude to the Almighty God. I also express my thanks to the many people who provided their support in writing this book, either directly or indirectly. I thank Dr. Sandeep Kumar, for the impetus to write this book. I want to acknowledge my sisters and parents for their blessing and persistent backing. I thank all my friends and colleagues for being a source of inspiration and love throughout my journey. I want to thank the anonymous reviewers for proofreading the chapters and the publishing team.

—Santosh Singh Rathore

I would like to express my sincere thanks to my institute, Indian Institute of Technology Roorkee, India, for providing me a healthy and conducive working environment. I am also thankful to the faculty members of the Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, India, for their constant support and encouragement. I am especially thankful to some of my colleagues, who are more like friends and give me constant support. I am also thankful to Prof. R. B. Mishra of the Indian Institute of Technology, Banaras Hindu University, India, for his guidance. I am grateful to the editor and the publication team of the Springer for their constant cooperation in writing the book. I am really thankful to my wife, sisters, brother, parents-in-law, and my lovely daughter Aastha, who is my life, for their love and blessings. I have no words to mention the support, patience, and sacrifice of my parents. I dedicate this book to God and to my family.

—Sandeep Kumar

Contents

About the Authors

Sandeep Kumar (SMIEEE'17) is currently working as an assistant professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, India. He has supervised 3 Ph.D. theses, about 30 master dissertations, about 15 undergraduate projects, and is currently supervising 4 Ph.D. students. He has published more than 45 research papers in international/national journals and conferences and has also written book/chapters with Springer, USA, and IGI Publications, USA. He has also filed two patents for his work done along with his students. He is the member of board of examiners and board of studies of various universities and institutions. He has collaborations in industry and academia. He is currently handling multiple national and international research/consultancy projects. He has received Young Faculty Research Fellowship Award from MeitY (Government of India), NSF/TCPP Early Adopter Award 2014 and 2015, ITS Travel Award 2011 and 2013, and others. He is the member of ACM and senior member of IEEE. His name has also been enlisted in major directories such as Marquis Who's Who and IBC. His areas of interest include semantic Web, Web services, and software engineering. Email: sandeepkumargarg@gmail.com, sgargfec@iitr.ac.in

Santosh Singh Rathore did his Ph.D. from the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, India. He received his master's degree (M.Tech.) from the Indian Institute of Information Technology Design and Manufacturing (IIITDM), Jabalpur, India. He is currently working as an assistant professor in the Department of Computer Science and Engineering, National Institute of Technology (NIT) Jalandhar, India. His research interests are software fault prediction, software quality assurance, empirical software engineering, object-oriented software development, and object-oriented metrics. He has published research papers in various refereed journals and international conferences. Email: santosh.srathore@gmail.com

Chapter 1
Introduction

Nowadays, software is the key element of function in many modern engineered systems. It can be found in a variety of applications from the refrigerators to cars to the washing machine, etc. Due to the advancement of technology brought about by the IT industry, the significance of the field of software engineering has been continuously increasing. In this growing number of applications of software, one vital challenge is to ensure that system is matching with the quality specifications and is reliable to use. The quality of the software determines its value and reliability ensures the failure-free operation of software for a specified period of time in the given environmental conditions. Quality and reliability of the software depend on the software faults. The more the faults, the lesser the reliability of the software and more efforts are required to maintain the quality of the software. Software quality assurance (SQA) can be thought as an umbrella activity, which incorporates various activities to organize and monitor the software development process and ensure that the final software product is of higher quality and reliability (Menzies et al. 2010; Mishra et al. 2013). The activities that may be included in SQA are pair evaluation, code inspections, code walkthroughs, software testing, checklist, and fault prediction (Johnson and Malek 1988; Adrion et al. 1982; Kundu et al. 2009). However, SQA is a time-consuming process and requires an ample amount of resources.

Software fault prediction (SFP) can be used to help in allocating limited SQA resources in a cost-effective and optimized manner by predicting the fault-proneness of software modules before the testing. It is a process to predict the fault-prone software modules without executing them by using some underlying characteristics of the given software system. A typical fault prediction model is built by training a learning technique for the dataset having some structural properties together with the fault information for a known project. The trained prediction models are subsequently used to predict fault-proneness of the unknown software project (Arisholm et al. 2010). The idea of software fault prediction is laid on the assumption that under certain environmental conditions if a previously developed project was found faulty, then any module currently developing under similar environmental conditions having similar project characteristics will end to be fault-prone (Jiang et al. 2008). The quick and early identification of the faulty modules as a result of

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_1

using software fault prediction can be utilized by the tester or developer to better strategize the software quality assurance efforts.

The lure of early detection of faults and improving the quality of the system has attracted considerable attention of research community to the software fault prediction. A wide range of statistical and machine learning techniques has been used earlier to build the fault prediction models and to predict the fault-proneness of currently developing software system. Moreover, the availability of open-source and publicly accessible software fault dataset repositories such as NASA Metrics Data Program and PROMISE data repository (PROMISE 2015) is allowing researchers to undertake more investigations and is opening up new areas of applications.

1.1 Software Faults, Errors, and Failure Terminologies

Software fault can arise in any phase of the software development including requirements gathering and specifications, designs, code, or maintenance. Depending upon the origin of the fault, the nature of the fault differs. The faults that are translated into the code of the software may lead to system failure, if not identified and removed correctly. Errors, defects, faults, and failures are interrelated terms and often created misunderstanding in their definitions (Huizinga and Kolawa 2007). Here, we follow the IEEE standard 610-1990 to define these terms.

Error: "A human action that produces an incorrect result".
Ex. main()
{
——
——//some code
1. If (a! = b)
2. {
3. c = (a/b);
4. }
———
———//other functionality
}

If programmer misses the semicolon at the end of the third line of code and compiles the program, then an error message will occur. This type of mistake called error.

Bug: "An unexpected result or deviation in actual functionality found out by an author (who wrote the code) after compilation of program and during any testing phase is called bug."

Ex. In the above program, if the author uses some other variable like "d" instead of "b" at third line or uses "+" operator instead "/" operator, then no compilation error occurs but it produces some unexpected results. This is called bug.

Exception: "An unhandled error occurring at run-time of the program is called exception."

Ex. There is no error generated at the compilation time, but at the run-time it throws an exception for b = 0 (third line).

Fault: "An incorrect step, process, or data definition in a computer program that causes the program to perform in an unanticipated manner." It is commonly known as defect also and generally found by moderator (not an author of code).

Ex. In the above program, if the author uses some other variable like "d" instead of "b" at third line or uses "+" operator instead "/" operator, then no compilation error occurs but it produces some unexpected results and this issue is found by moderator. This is called fault or defect.

Failure: "The inability of a software or software component to perform its required functions within specified performance requirements." In other words, the software does not do what the requirements describe.

Ex. Due to the fault occurrence, if any other part of code or module gets affected, this condition is called failure.

Software faults are different from software failures. Software faults are an indication of a quality attribute that explains a condition that the software fails to perform its desired functions. While, software failures are the symptoms, revealed when one or more software faults are executed.

1.2 Benefits of Software Fault Prediction

The potential of software fault prediction to identify the area of code where faults are more likely to occur has drawn a considerable attention of the research community. This early identification of faults can help software testers or developers to narrow down in the risky areas of software design to optimize and prioritize the testing efforts and resources. For example, if there are only limited amount of testing resource left to perform testing, knowing what are the most risky modules in the software system can certainly helpful to the testers to narrow down the testing efforts. Following are some benefits of using software fault prediction in software development process:

(1) *Advantage in testing cost reduction*: In 2013, (Monden et al. 2013) conducted a study to verify the effectiveness of test effort allocation strategies based on fault prediction results in term of cost saving for a telecommunication system. The results of the study revealed that using fault prediction with testing achieved the 25% reduced in the testing efforts without effecting the fault finding efficiency of the testing process. Though, author reported that about 6% of the additional effort are required by the company to collect software metrics, to preprocess data, and to build the fault prediction model. Therefore, total 19% of the efforts could be saved with the best fault prediction model. In another attempt to verify the effectiveness of software fault prediction, (Li et al. 2006) reported a study where authors shared the experience of applying fault prediction model for ABB Inc. software system. The study showed that modules ranked as fault-prone by the software fault prediction model helped the testers to detect additional

faults in the modules that were considered previously to be a low fault-prone. Additionally, modules identified by testers as the most fault-prone are among the top four fault-prone modules identified by the software prediction model.

(2) *Advantages in early flaw detection in software design*: (Jiang et al. 2008) reported an investigation using requirement-level and design-level software metrics. They found out that the early-level software metrics used for fault prediction help in improving the software design and eventually lead to lesser number of faults in the software system. Some other researchers also conducted their studies using design-level and requirement-level software metrics and found that software fault prediction helps to detect flaws in the software design at very early software development phase and eventually leads to lesser faults in the software system (Hall et al. 2012; El Emam et al. 2001; Glasberg et al. 1999).

(3) *Advantage in the development of highly dependable system*: Use of software fault prediction in software quality assurance activity helps the practitioners to uncover risky areas of the software system that may not be reachable by the regular testing process. In such way, it helps in increasing the robustness of the software system.

1.3 Contribution and Organization of the Book

Chapter 2 discusses the architecture of software fault prediction and the various activities involved in the software fault prediction process. The chapter also discusses various components of software fault prediction and reports their importance in building fault prediction models.

Chapter 3 presents the types of software fault prediction. For each type of software fault prediction, we provide the overview of the model building process and discuss the review of some relevant works related to the prediction process. The chapter also presents observations and challenges of software fault prediction for each type of prediction process.

Chapter 4 provides the details of software fault dataset repositories. The chapter first presents the summary of different fault dataset repositories and then presents different issues associated with the fault dataset repositories. At the end of the chapter, the observations and challenges of fault dataset repository have been discussed.

Chapter 5 presents the details of binary class prediction of faults. The chapter first provides the description of different techniques used for building fault prediction models. Subsequently, it presents the evaluation of the techniques for the binary class prediction of faults. The chapter also provides the details of the used performance evaluation measures and experimental analysis process.

Chapter 6 presents details of the number of faults prediction. The chapter provides the details of the techniques that can be used for the prediction of number of faults. Further, the chapter presents the description of used performance evaluation measures and the experimental methodology used for the prediction of number of faults. The

evaluation of different techniques for the prediction of number of faults and the results are also discussed in the chapter.

Chapter 7 concludes the book by summing up the contributions made in the presented work.

1.4 Summary

Software fault prediction is used to detect faults in the software code without execution. It has the potential to optimize the software quality assurance process by accurately identifying the faults. It may also help in an economic software development process. But most of the organizations still do not consider fault prediction techniques while its potential has been validated in a couple of researches. It indicates that there is a need for further research in this field that would emphasize on how it can improve the quality assurance process. In this chapter, we have introduced the underlying fundamentals of the software fault prediction. Basics of software fault prediction, software fault terminologies, and benefits of software fault prediction have been discussed. The chapter further has summarized the organization of the book.

References

Adrion, W. R., Branstad, M. A., & Cherniavsky, J. C. (1982). Validation, verification, and testing of computer software. *ACM Computing Surveys, 14*(2), 159–192.

Arisholm, E., Briand, L., & Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software, 83*(1), 2–17.

El Emam, K., Melo, W., & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software, 56*(1), 63–75.

Glasberg, D., Emam, K. E., Melo, W., & Madhavji, N. (1999). *Validating object-oriented design metrics on a commercial java application*.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Huizinga, D., & Kolawa, A. (2007). *Automated defect prevention: Best practices in software management*. Wiley.

Jiang, Y., Cukic, B., & Ma, Y. (2008). Techniques for evaluating fault prediction models. *Empirical Software Engineering, 13*(5), 561–595.

Johnson, A. M., Jr., & Malek, M. (1988). Survey of software tools for evaluating reliability, availability, and serviceability. *ACM Computing Surveys, 20*(4), 227–269.

Kundu, D., Sarma, M., Samanta, D., & Mall, R. (2009). System testing for object-oriented systems with test case prioritization. *Software Testing, Verification and Reliability, 19*(4), 297–333.

Li, P. L., Herbsleb, J., Shaw, M., & Robinson, B. (2006). Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB inc. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 413–422.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering Journal, 17*(4), 375–407.

Mishra, D., Mishra, A., Colomo-Palacios, R., & Casado-Lumbreras, C. (2013). Global software development and quality management: A systematic review. In *Proceedings of the OTM Confederated International Conferences on the Move to Meaningful Internet Systems* (pp. 302–311).

Monden, A., Hayashi, T., Shinoda, S., Shirai, K., Yoshida, J., Barker, M., et al. (2013). Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering, 39*(10), 1345–1357.

PROMISE. (2015). *The PROMISE repository of empirical software engineering data*. http://openscience.us/repo.

Chapter 2
Software Fault Prediction Process

Accurate detection and early removal of software faults during the software development can reduce the overall cost of software development and can result in the improved software quality product. These inherent advantages of software fault prediction have attracted many researchers to focus on the software fault prediction. Thus, it is a key area to study in the field of software engineering and is subject to many previous studies. The primary goal of software fault prediction (SFP) is to assist the software testing process and to help in the allocation of available software testing and quality assurance resources optimally and economically by raising the alarm for the software code where faults are more likely to occur (Menzies et al. 2010). Typically, software testing process tries to detect and remove the maximum of existing faults in the software before deploying the software to the customer. However, the testing process consumes an ample amount of testing efforts and resources, which are not generally available (Najumudheen et al. 2011). Software fault prediction plays an important role to optimize and streamline the software testing process. It identifies and alarms about the code of the software where faults are more likely lurk and thus helps to detect most of the faults in the current release of the software. SFP models learn from the historic fault data stored in the software repository and having information about the previous software releases or versions. The learned SFP models are then used to predict the fault-proneness of software modules of the current release of the software project. Thus, SFP guides the software testing and helps in increasing its efficiency and effectiveness.

2.1 Architecture of Software Fault Prediction

A software fault prediction model is generally built by establishing the relationship between different structural and procedural measures of the software system also known as software metrics with faults. Figure 2.1 provides the description of the software fault prediction process (Rathore and Kumar 2017).

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_2

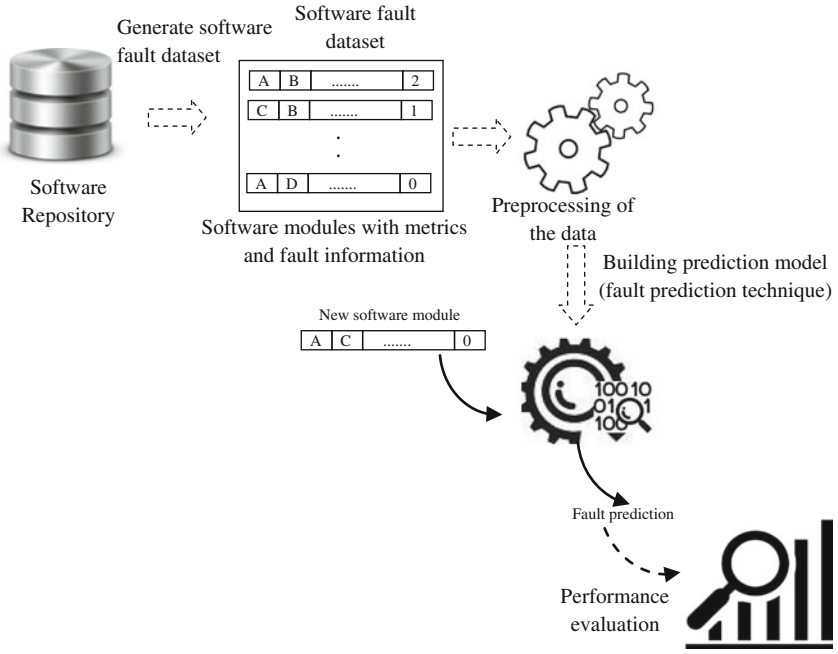


Fig. 2.1 Architecture of software fault prediction process

Kim et al. have provided the details of essential steps required to build a fault prediction model (Kim et al. 2008). These steps are described below.

1. Collecting faults/bugs information: The collection of bug data requires the data retrieval and linking of the source. For each software project, it requires analyzing the source code repository (e.g., SVN or CVS), and based on the log contents, it decides whether a commit is a bugfix or not. A commit is inferred as a bug if it solves an issue reported in the bug tracking system.
2. Extracting features (software metrics) and creating training dataset: This step requires extracting the software metrics information from the source code of the software project or from the log contents of the projects. By combining fault's information and extracted software metrics, a training set is created that is used to train learning techniques for fault prediction.
3. Building prediction models: Generally, machine learning or statistical techniques such as tree-based classifiers or Bayesian network are used to build fault prediction model using training set. Subsequently, the prediction model is then used to predict the fault-proneness of unseen software modules.
4. Assessment: To evaluate fault prediction model, generally a separate testing dataset is used besides training dataset. First, the fault-proneness of software modules in the testing dataset is predicted, and subsequently, the performance of the model is evaluated by comparing the predicted value of fault-proneness and actual value of fault-proneness.

2.2 Components of Software Fault Prediction

The main components of software fault prediction model are software fault dataset (metrics and fault information), fault prediction techniques, and performance evaluation measures. The details of all three components are provided in the upcoming subsections.

2.2.1 Software Fault Dataset

Typically, each software development organization uses the concept of software repository, where it records the information regarding the evolution and progress of software system. During the development of the software, generally, a large amount of data such as source code, documentation and configuration files, change logs is generated and captured for various software components. This data is stored in the software repository. The data/information stored in the software repository can be mined and used by researchers and practitioners in deciding and monitoring the software development policies. This information can be used by the project managers also for controlling and improving the software development process (Kagdi et al. 2007). According to the requirement, software manager or developer can mine and access the information stored in the repositories about the previous releases of the software. Further, this mined information can be converted into the useful knowledge and can be applied to the software currently under development to improve the decision-making process and to revise the development policies (Hassan 2008). The idea of software fault prediction is based on this underlying process. It makes the use of historic fault dataset of the earlier software projects stored in the repositories to predict fault-proneness in the currently developing software project.

Software fault dataset is used as training dataset to train the fault prediction model and as testing dataset to assess the performance of fault prediction model. Primarily, software fault dataset has three sets of information: set of software metrics, fault information such as faulty or non-faulty software module, number of faults per module, and meta information about the software project.

1. *Software fault information*: This information tells about the fault-proneness of the given software artifact or component. Depending upon the software development methodology, component can be a class, a function, a module, or a package. When, during the development or testing, any fault is detected in any software component that fault is reported in the software repository along with the details of the component where it has found. The fault information can tell that whether a given component is faulty or non-faulty, how many faults are found in the component, what is the severity of the faults, etc.?

Generally, depending upon the availability of the repository and the type of software project for which data is collected, fault dataset repositories can be classified into three types (Radjenovic et al. 2013).

(1) Private/commercial dataset repository: In this type of repository, fault dataset and source code are not available publicly. The data available in this type of repositories is maintained by the companies and used within the organization to improve the software development process. Since the dataset is not publicly available, therefore, it may be difficult to repeat the study based on the datasets available in this repository.
(2) Partially public/freeware dataset repository: For the repository of this category, only the source code and log information of the software project are publicly available such as SourceForge. However, the value of different software metrics is usually not available in this type of repository (Radjenovic et al. 2013). To use the data available in these repositories, user first needs to estimate the software metric values from the given source code and subsequently needs to map these metric values to the available fault information. The process of metric value calculation and mapping is required additional care to avoid any bias in the calculation. This process is vital for collecting the unbiased and robust dataset since any error can lead to the biased learning.
(3) Public: For the repository of this category, the software metric values and the corresponding fault information both are publicly available. Examples of these repositories are NASA metrics data program and PROMISE data repository. Usually, some third party extracts and calculates the software metric value and fault information and makes it publicly available to use. The datasets in these repositories are generally corresponding to open-source software projects. The studies performed using datasets from these repositories can be repeatable.

Not all the available fault dataset contained all types of fault information. Some of the fault datasets have information of only faulty and non-faulty software modules. Some other fault datasets have both number of faults and severity of fault information for software modules.

2. *Software metrics*: During the software development, often managers and developers are required to evaluate the quality of the software and the process used to build it. In order to do so, they need to capture various attributes of software artifacts at different phases of software development and need to measure them to evaluate the software and its process. To achieve this, software metrics have been proposed. Software metrics allows to quantify the attributes of software artifacts, and using these quantitative values, software quality can be evaluated.

Software metric can be defined as a measurement-based technique applied to the software process, product, and services to supply the engineering and management based information about the software and can be used to provide feedback to improve the software process, product, and services. Software metrics can be used: to understand the software process and product, to evaluate the software process, product, to control the process flow, to predict the quality of the end product.

In terms of software fault prediction, software metrics can be classified into two broad categories: product metrics and process metrics. These categories are further classified into subcategories. However, sometimes, each category is not mutually

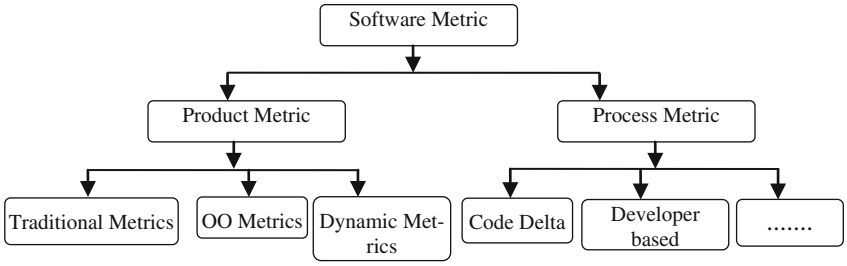


Fig. 2.2 Classification of software metrics

exclusive and there are some metrics, which act to be a part of more than one category. A classification of software metrics is shown in Fig. 2.2.

(a) Product metrics: This is the set of metrics calculated from the end software product. These metrics are generally used to estimate the overall quality of the software such that whether software product confirms norms of ISO-9126 standard or not, etc. Product metrics are further classified as traditional metrics, object-oriented metrics, and dynamic metrics.

1. Traditional metrics: This set of software metrics designed at the early days of software engineering and termed as traditional metrics. This set of metrics mainly contained the following metrics:
   - Size metrics: "Function Points (FP), Source lines of code (SLOC), Kilo-SLOC (KSLOC)."
   - Quality metrics: "Defects per FP after delivery, Defects per SLOC (KSLOC) after delivery."
   - System complexity metrics: "Cyclomatic Complexity, McCabe Complexity, and Structural complexity" (McCabe 1976).
   - Halstead metrics: "number of distinct operators (n1), number of distinct operands (n2), total number of operators (N1), total number of operands (N2), Program vocabulary (η), volume (v), program length (N), Difficulty (D), effort (E), number of delivered bugs (B), time required to program (T)" (Halstead 1977).
2. Object-oriented metrics: This set of metrics are corresponding to the various features of the software developed using object-oriented (OO) methodology such as coupling, cohesion, inheritance (Crasso et al. 2014). Many OO metrics suites have been proposed capturing different OO features. Initially, Chidamber and Kemerer presented a OO software metrics suite known as CK metrics suite (Chidamber and Kemerer 1994). Subsequently, some other authors also presented different OO metrics suites such as (Harrison and Counsel 1998; Lorenz and Kidd 1994; Briand et al. 1997; Marchesi 1998; Bansiya and Davis 2002).
   - CK metrics suite: "Coupling between Object class (CBO), Lack of Cohesion in Methods (LCOM), Depth of Inheritance Tree (DIT), Response for

a Class (RFC), Weighted Method Count (WMC) and Number of Children (NOC)" (Chidamber and Kemerer 1994).
- MOODS metrics suite: "Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (PF), Coupling Factor (CF)" (Harrison and Counsel 1998).
- Wei Li and Henry metrics suite: "Coupling Through Inheritance, Coupling Through Message passing (CTM), Coupling Through ADT (Abstract Data Type), Number of local Methods (NOM), SIZE1 and SIZE2" (Li and Henry 1993).
- Lorenz and Kidd's metrics suite: "PIM, NIM, NIV, NCM, NCV, NMO, NMI, NMA, SIX and APPM" (Lorenz and Kidd 1994).
- Bansiya metrics suite: "DAM, DCC, CIS, MOA, MFA, DSC, NOH, ANA, CAM, NOP and NOM" (Bansiya and Davis 2002).
- Briand metrics suite: "IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, IFMMIC, AMMIC, OM- MIC, FMMEC, DMMEC, OMMEC" (Briand et al. 1997).

3. Dynamic metrics: Dynamic metrics are used to capture the dynamic behavior of the software project. This set of metrics are calculated from a running program and are used to identify the objects that are the most run-time coupled and complex during execution (Tahir and MacDonell 2012). These metrics give different indication on the quality of the software design (Yacoub et al. 1999).
   - Yacoub metrics suite: "Export Object Coupling (EOC) and Import Object Coupling (IOC)" (Yacoub et al. 1999).
   - Arisholm metrics suite: "IC_OD, IC_OM, IC_OC, IC_CD, IC_CM, IC_CC, EC_OD, EC_OM, EC_OC, EC_CD, EC_CM, EC_CC" (Arisholm 2004).
   - Mitchell metrics suite: "Dynamic CBO for a class, Degree of dynamic coupling between two classes at runtime, Degree of dynamic coupling within a given set of classes, RI, RE, RDI, RDE" (Mitchell and Power 2006).

(b) Process metrics: Process metrics capture and measure the features of software development life cycle. These metrics are used to estimate the process of software development. Further, these metrics can be used to make strategic decisions about the software development process. Generally, software managers and developers have used these metrics to measure software process and services that lead to long-term software process improvement (Bundschuh and Dekkers 2008). Process metrics can be further classified into different subcategories as follows:

1. Code delta metrics: "Delta of LOC, Delta of changes" (Nachiappan et al. 2010).

2. Code churn metrics: "Total LOC, Churned LOC, Deleted LOC, File count, Weeks of churn, Churn count and Files churned" (Nagappan and Ball 2005).
3. Change metrics: "Revisions, Refactorings, Bugfixes, Authors, LOC added, Max LOC Added, Ave LOC Added, LOC Deleted, Max LOC Deleted, Ave LOC Deleted, Codechurn, Max Codechurn, Ave Codechurn, Max Changeset, Ave Changeset and Age" (Nachiappan et al. 2010).
4. Developer-based metrics: "Personal Commit Sequence, Number of Commitments, Number of Unique Modules Revised, Number of Lines Revised, Number of Unique Package Revised, Average Number of Faults Injected by Commit, Number of Developers Revising Module and Lines of Code Revised by Developer" (Matsumoto et al. 2010).
5. Requirement metrics: "Action, Conditional, Continuance, Imperative, Incomplete, Option, Risk level, Source and Weak phrase" (Jiang et al. 2008; Byun et al. 2014).
6. Network metrics: "Betweenness centrality, Closeness centrality, Eigenvector Centrality, Bonacich Power, Structural Holes, Degree centrality and Ego network measure" (Premraj and Herzig 2011).

3. *Meta information about the project*: Meta information refers to basic information about software project, which can provide the additional information and can make working with data easier. It may have the following set of information such as the domain for which software is developed, the programming language in which software is written, the number of releases/versions of the software. This meta information of software project helps in establishing the usability of the fault prediction models and also defined the context of the model applicability. The context of fault prediction is an important attribute as with varying the context of the prediction, performance of fault prediction models may also vary. The transferability of fault prediction model between different contexts may affect the prediction results (Hall et al. 2012). Meta information of software project consists the following variables/factors that can be used when building fault prediction models, as given below (Hall et al. 2012):

   - Source of data: Source of the data shows the domain of the software project over which the study was performed. For example, a software project can be of open-source or commercial, can be Web browser or IDE, etc. Sometimes, it also shows that whether fault dataset of the project is publicly available or not. The source of the fault dataset can have an effect on the performance of the fault prediction models. Fault prediction models are suffered from the data shifting issue, where training and testing have been performed on different datasets.
   - Maturity of the system: It shows how old software system is. The maturity of system is measured in terms of number of versions/releases over which software system evolved. Each new release of the software is generally corresponding to the changes occurred in the software or to add new functionality in the software. The mature the system is, the lower the chance of error in

the system. Performance of fault prediction model can get influenced by the maturity of the system.

- Size: Size of the software project is estimated using non-commented lines of code (LOC) attribute. Generally, it is represented in kilo lines of code (KLOC). It is generally analogy that the higher the value of LOC, the higher the faults in the software. Size of the software influences the performance of fault prediction model. Fault prediction model based on large software project generally produced better performance.
- Application domain: It presents the environment and the process under which a software project is developed. A software project can be developed by teams in distributed manner across the world. Since each software project is developed using different development methodology and under different conditions. Therefore, sometimes a fault prediction model built for one application domain may not perform better when application domain changes.
- The granularity of prediction: The granularity of a fault dataset shows the level for which data has been collected. For example, fault dataset can be collected corresponding to the class level for object-oriented software or at function level for procedural software, or at the module level, etc. The granularity for which fault prediction model is built has a notable influence on the performance of fault prediction model.

4. *Performance evaluation measures*: Performance evaluation measures are used to assess the performance of a fault prediction model. A fault prediction model is trained for some fault dataset and then this trained model is applied to unknown testing dataset to assess its performance. Various performance evaluation measures have been used earlier in the context of software fault prediction to assess the model performance. Generally, these performance measures can be grouped into two classes: numeric measures and graphical measures. Figure 2.3 presents the classification scheme of performance evaluation measures used for fault prediction models. Numeric performance evaluation measures belong to the class of measures provide a numeric description about the performance of prediction models. It mainly contains accuracy, precision, recall, f-measure, false positive rate, G-means, false negative rate, J-coefficient, specificity, average absolute error, and average relative error (Maji and Yahia 2014). Graphical performance evaluation measures display information in easy and quickly understandable manner. It mainly contains ROC curve, precision–recall (P-R) curve, and cost curve.

Numeric Measures: The root of all the numeric evaluation measures is the confusion matrix. A confusion matrix contains information about actual and predicted classifications done by a fault prediction technique. It provides four different information.

1. True positive (TP) = a faulty module is correctly predicted as faulty.
2. True negative (TN) = a non-faulty module is correctly predicted as non-faulty.
3. False positive (FP) = a non-faulty module is incorrectly predicted as faulty.

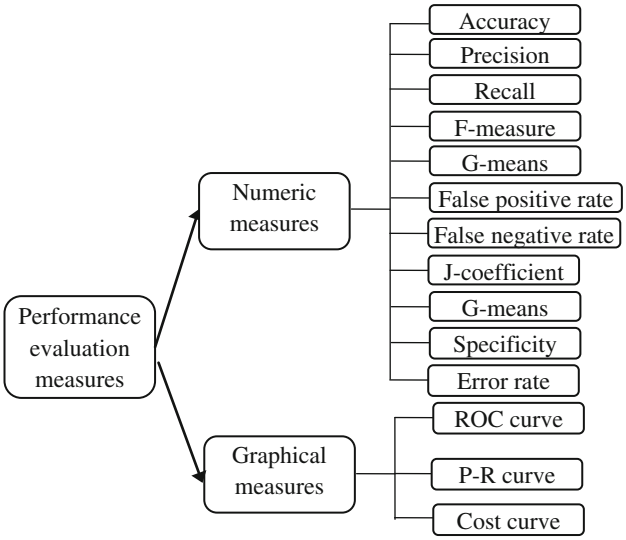


Fig. 2.3 Description of performance evaluation measures

4. False negative (FN) = a faulty module is incorrectly predicted as non-faulty.

Accuracy: It shows the probability of correctly predicted faulty and non-faulty modules for a fault prediction model (Kubat et al. 1998). It is measured as defined in Eq. 2.1. However, accuracy does not provide any information about the incorrect prediction of faulty and non-faulty modules. It does not capture the misclassification cost of the prediction model. Therefore, accuracy is not a suitable measure, if anyone is interested in estimating misclassification cost of faulty and non-faulty modules or if fault dataset is imbalance.

$$\text{Accuracy} = \frac{\text{TN} + \text{TP}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.1}$$

False positive rate (FPR) and False negative rate (FNR):

FPR is calculated as the ratio of non-faulty modules predicted incorrectly as faulty module to the all non-faulty modules in the dataset (Lewis and Gale 1994). It is also known as false alarm rate or type 1 error. It is calculated as defined in Eq. 2.2.

$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{2.2}$$

FNR is calculated as the ratio of faulty modules predicted incorrectly as non-faulty module to the all faulty modules in the dataset (Lewis and Gale 1994). It is also known as type 2 error. It is defined as given in Eq. 2.3.

$$\text{FNR} = \frac{\text{FN}}{\text{FN} + \text{TP}} \tag{2.3}$$

Precision, Recall, and Specificity:

Precision is used to measure the relevancy of the results. It shows the amount of modules predicted correctly as faulty out of all faulty predicted modules (Conte et al. 1986). It is defined as given in Eq. 2.4.

$$\text{Precision} = \frac{\text{TP}}{\text{FP} + \text{TP}} \tag{2.4}$$

Recall is used to measure how many are true out of the relevant results. It shows the amount of modules predicted correctly as faulty out of all the faulty modules in the given fault dataset (Conte et al. 1986). It is also known as probability of detection (PD) and is defined as given in Eq. 2.5.

$$\text{Recall} = \frac{\text{TP}}{\text{FN} + \text{TP}} \tag{2.5}$$

Precision and recall are useful when dataset is imbalance and accuracy measure cannot be used. Using them together can provide useful information about the fault prediction model.

Specificity is calculated as the portion of negative (non-faulty modules) that bare correctly predicted by the fault prediction model (Conte et al. 1986). It is defined as given in Eq. 2.6.

$$\text{Specificity} = \frac{\text{TN}}{\text{FP} + \text{TN}} \tag{2.6}$$

F-measure: It is used to show the trade-off between precision and recall. It is defined as the harmonic mean of precision and recall values of a fault prediction model (Jiang et al. 2008). It is defined as given in Eq. 2.7.

$$\text{F-measure} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.7}$$

G-means and J-coefficient:

In fault prediction environment, sometimes a prediction model is achieving higher prediction accuracy by correctly predicting the non-faulty modules. However, the same prediction model is incorrectly predicting the faulty modules. In this scenario, G-means and J-coefficient can be used to capture the actual prediction performance of the model.

G-mean-1 is calculated as the square root of the precision and recall. G-mean-2 is calculated as the square root of the product of recall and specificity (Conte et al. 1986). They are defined as given in Eqs. 2.8 and 2.9.

$$\text{G-means 1} = \sqrt{\text{Precision} * \text{Recall}} \tag{2.8}$$

$$\text{G-means 2} = \sqrt{\text{Specificity} * \text{Recall}} \tag{2.9}$$

J-coefficient (J-coeff): It provides the information about the fault prediction model performance by combining the results of recall and specificity []. It is defined as given in Eq. 2.10.

$$\text{J-coeff} = \text{Recall} + \text{Specificity} - 1 \tag{2.10}$$

The value of J-coeff = 0 indicates that the chance of identifying a faulty is equal to the false alarm rate and model is not useful to do fault prediction. The value of J-coeff > 0 indicates that model is useful to do fault prediction. The J-coeff = 1 shows perfect fault prediction model, and the J-coeff = −1 shows the worst fault prediction model.

Error rate: It is used to calculate the difference between the predicted values of the faults to the actual value of the faults in a given software module. Generally, two different measures are used to calculate error rate. They are defined below.

Average absolute error (AAE): AAE calculates the absolute difference between the predicted values of the faults to the actual value of the faults in a given module (Veryard 2014). It is defined as given in Eq. 2.11.

$$\text{AAE} = \frac{1}{N} \sum_{i=1} \left| \left(Y_i - \overline{Y_i} \right) \right| \tag{2.11}$$

Absolute relative error (ARE): ARE estimates the magnitude of the absolute error in comparison with the total size of the object measured . It is defined as given in Eq. 2.12.

$$\text{ARE} = \frac{1}{N} \sum_{i=1} \left| Y_i - \overline{Y_i} \right| / (Y_i + 1) \tag{2.12}$$

Graphical Measures: Graphical measures are used to visually depict the performance of a fault prediction model. These measures are also derived from the confusion matrix.

ROC curve:

ROC curve provides a performance estimation by calculating the ratio of the faulty modules predicted correctly to the non-faulty modules predicted incorrectly (Bockhorst and Craven 2005). It gives an estimation about the prediction model's performance by considering the cost of misclassification of faulty and non-faulty modules, if there is an unbalance in the class distribution. ROC curve is represented by a graph with a diagonal line. This diagonal line shows a random model, which has no prediction capability. Figure 2.4 depicts a sample ROC curve. ROC curve starts from the bottom left side and evolves toward the upper right side. The closer the curve to the upper left side, the accurate the prediction model is.

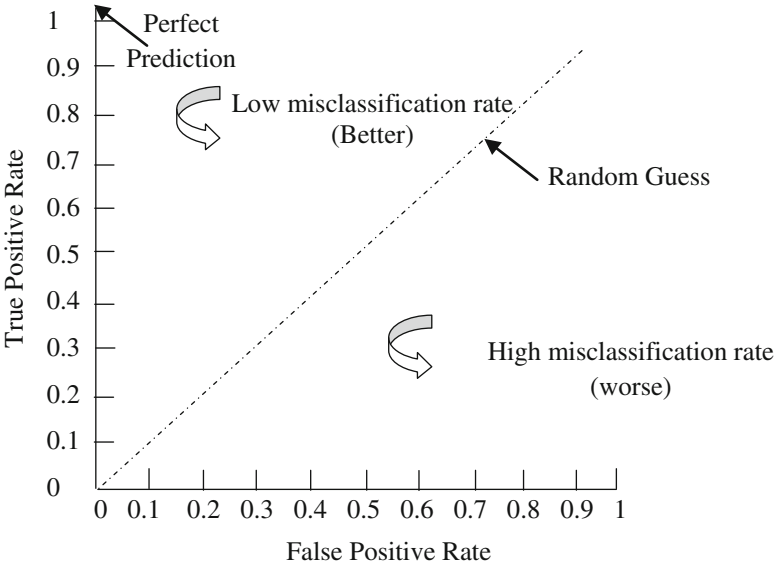


Fig. 2.4 ROC curve: example

Generally, area under the ROC curve (AUC) is used to measure the prediction capability of the model. It is generally calculated by choosing a threshold value. The value AUC = 1 represents 100% prediction capability of the model. The value AUC = 0 represents the 0% prediction capability of the model.

PR curve:

A PR curve shows the trade-off between precision and recall. It is plotted by showing recall value at *x*-axis and precision value at *y*-axis (Bunescu et al. 2005). In PR curve, high value of recall and high value of precision show the best performance. Figure 2.5 shows an example of PR curve. For a prediction model to be perfect, its PR curve must pass through the upper right corner having 100% precision and 100% recall. The closer the PR curve to the upper right corner, the better the prediction model is.

Cost curve:

A cost curve is used to depict the cost of misclassification of software modules of a fault prediction model. *Y*-axis of cost curve plots the probability of detection (PD), also known as recall and *x*-axis plots the probability of false alarm (Drummond and Holte 2006). It shows the difference between the maximum and minimum values of cost for misclassifying the faulty modules. Figure 2.6 shows an example of cost curve. The diagonal line between (0,0) and (1,1) shows that prediction model predicted all the modules as fault-free. The diagonal line between (0,1) and (1,0) shows that prediction model predicted all the modules as faulty. The horizontal line between (0,1) and (1,1) shows that prediction model misclassified all the modules. The horizontal

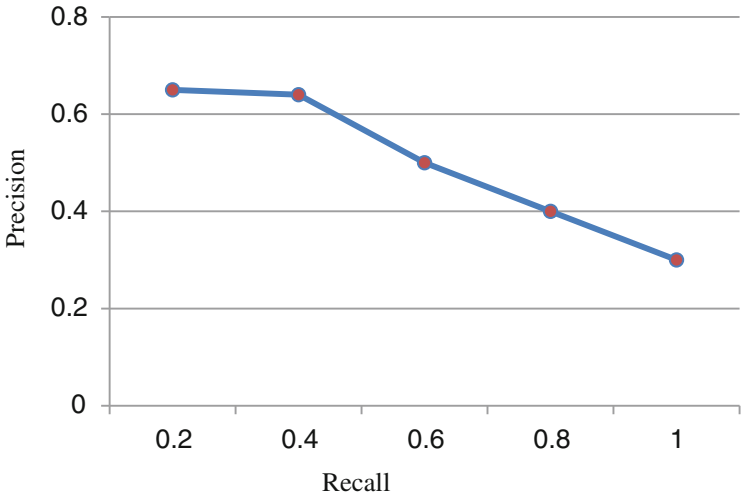


Fig. 2.5 An example of PR curve

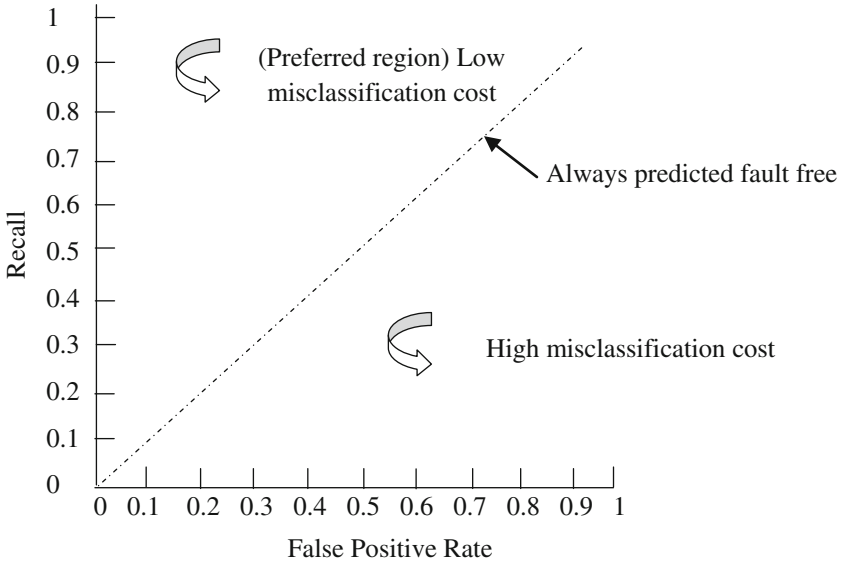


Fig. 2.6 Cost curve: example (Jiang et al. 2008)

line between (0,0) and (0,1) shows that prediction model predicted correctly all the modules. Generally, range of (0,0.5) is sufficient for model evaluation.

2.3 Summary

Software fault prediction helps in reducing fault-finding efforts by predicting faults prior to the testing process. This practical use of software fault prediction has encouraged the researchers to focus on this area. In this chapter, we provided the description of software fault prediction. We also discussed various activities involved in software fault prediction and presented the details of its different components such as software metrics, software fault data, and performance evaluation measures.

References

Arisholm, E. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering, 30*(8), 491–506.

Bansiya, J., & Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4–17.

Bockhorst, J., & Craven, M. (2005). Markov networks for detecting overlapping elements in sequence data. In *Proceedings of the Neural Information Processing Systems* (pp. 193–200).

Briand, L., Devanbu, P., & Melo, W. (1997). An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering* (pp. 412–421).

Bundschuh, M., & Dekkers, C. (2008). *The IT measurement compendium*.

Bunescu, R., Ruifang, G., Rohit, J. K., Marcotte, E. M., Mooney, R. J., Ramani, A. K., et al. (2005). Comparative experiments on learning information extractors for proteins and their interactions [special issue on summarization and information extraction from medical documents]. *Artificial Intelligence in Medicine*, *33*(2), 139–155.

Byun, J., Rhew, S., Hwang, M., Sugumara, V., Park, S., & Park, S. (2014). Metrics for measuring the consistencies of requirements with objectives and constraints. *Requirements Engineering, 19*(1), 89–104.

Chidamber, S., & Kemerer, C. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Conte, S. D., Dunsmore, H. E., & Shen, V. Y. (1986). *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc.

Crasso, M., Mateos, C., Zunino, A., Misra, S., & Polvorín, P. (2014). Assessing cognitive complexity in java-based object-oriented systems: Metrics and tool support. *Computing and Informatics, 32*.

Dallal, J. A., & Briand, L. C. (2010). An object-oriented high-level design-based class cohesion metric. *Information and Software Technology, 52*(12), 1346–1361.

Drummond, C., & Holte, R. C. (2006). Cost curves: An improved method for visualizing classifier performance. In *Proceedings of the Machine Learning Conference* (pp. 95–130).

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Halstead, M. H. (1977). *Elements of software science (operating and programming systems series)*. Elsevier Science Inc.

Harrison, R., & Counsel, J. S. (1998). An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering, 24*(6), 491–496.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of software maintenance (FoSM 2008)* (pp. 48–57).

Jiang, Y., Cukic, B., & Ma, Y. (2008). Techniques for evaluating fault prediction models. *Empirical Software Engineering, 13*(5), 561–595.

Kagdi, H., Maletic, J. I., & Sharif, B. (2007). Mining software repositories for traceability links. In *15th IEEE International Conference on Program Comprehension, ICPC'07* (pp. 145–154).

Kim, S., Whitehead, E. J., Jr., & Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering, 34*(2), 181–196.

Kubat, M., Holte, R. C., & Matwin, S. (1998). Machine learning for the detection of oil spills in satellite radar images. *Machine Learning Journal, 30*(2–3), 195–215.

Lewis, D., & Gale, W. A. (1994). A sequential algorithm for training text classifiers. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 3–12).

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*(2), 111–122.

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics*. Prentice Hall.

Maji, S. K., & Yahia, H. M. (2014). Edges, transitions and criticality. *Pattern Recognition, 47*(6), 2104–2115.

Marchesi, M. (1998). OOA metrics for the unified modeling language. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering* (pp. 67–73).

Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K., & Nakamura, M. (2010). An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (pp. 8–18).

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320.

Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering Journal, 17*(4), 375–407.

Menzies, T., Stefano, J., Ammar, K., McGill, K., Callis, P., Davis, J., et al. (2003). When can we test less? In *Proceedings of the 9th International Software Metrics Symposium* (pp. 98–110).

Mitchell, A., & Power, J. F. (2006). A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Science of Computer Programming, 59*(1–2), 4–25.

Nachiappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). Change bursts as defect predictors. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering* (pp. 309–318).

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering* (pp. 284–292).

Najumudheen, E., Mall, R., & Samanta, D. (2011). Test coverage analysis based on an object-oriented program model. *Journal of Software Maintenance and Evolution: Research and Practice, 23*(7), 465–493.

Olson, D. (2008). *Advanced data mining techniques*. Springer.

Premraj, R., & Herzig, K. (2011). Network versus code metrics to predict defects: A replication study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement* (pp. 215–224).

Radjenovic, D., Hericko, M., Torkar, R., & Zivkovic, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Rathore, S. S., & Kumar, S. (2017). A study on software fault prediction techniques. *Artificial Intelligence Review*, 1–73.

Tahir, A., & MacDonell, S. G. (2012). A systematic mapping study on dynamic metrics and software quality. In *Proceedings of the 28th International Conference on Software Maintenance* (pp. 326–335).

Veryard, R. (2014). *The economics of information systems and software*. Butterworth-Heinemann.
Yacoub, S., Ammar, H., & Robinson, T. (1999). Dynamic metrics for object-oriented designs. In *Proceedings of the 6th International Symposium on Software Metrics* (pp. 50–60).
Yousef, W., Wagner, R., & Loew, M. (2004). Comparison of nonparametric methods for assessing classifier performance in terms of roc parameters. In *Proceedings of the International Symposium on Information Theory* (pp. 190–195).

Chapter 3
Types of Software Fault Prediction

A large number of researchers have presented various fault prediction studies to predict the fault-proneness of the given software system. These fault prediction studies reported the results in term of different–different contexts. Depending upon the context of the results, a fault prediction model can classify a software module into faulty or non-faulty class (binary class classification) or can predict the number of faults in the given software module. Additionally, a fault prediction model can be built by using the fault dataset of other similar software projects (cross-project prediction). A fault prediction model can be employed to identify fault-inducing changes to provide the earlier feedback to the developers (just-in-time prediction). In this chapter, we provide an overview of different types of fault prediction. A detailed discussion on the state of the art has been presented. At the end of each section, we discuss the shortcoming of each type of fault prediction and provide the suggestions to overcome these shortcomings.

3.1 Binary Class Classification of Software Faults

Fault prediction models based on the binary class classification classify the given software modules into faulty or non-faulty classes only. When a fault prediction model predicts more than one fault in the software module then that module is marked as faulty. When a fault prediction model predicts zero faults in the software module then that module is marked as non-faulty. A large number of research on software fault prediction has focused on this type of prediction such as (Gokhale and Michael 1997; Guo et al. 2003; Venkata et al. 2006; Turhan and Bener 2009; Elish and Elish 2008; Kanmani et al. 2007; Menzies et al. 2010; Erturk and Sezer 2015). These studies have used various machine learning and statistical techniques including logistic regression, linear regression, naive Bayes, decision tree to build the fault prediction models for this context. Some semi-supervised learning such as EM algorithm and unsupervised learning techniques such as clustering has also been used to build the fault prediction models for the binary class classification of faults.

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_3

In this type of fault prediction, confusion matrix-based performance evaluation measures have been used to assess the performance of the built fault prediction models. Accuracy, precision, recall, and f-measure have primarily used to assess the performance of the prediction models. Some of the studies have also used AUC (area under ROC curve) and g-means to assess the performance of the prediction models.

Gokhale and Michael (1997) built a fault prediction model using regression tree and density modeling techniques. The study was performed for the fault dataset corresponding to an industrial software system. The results of the study showed that regression tree-based fault prediction model outperformed density modeling-based fault prediction model for all the used performance evaluation measures. In another study, Guo et al. (2003) have used three different techniques, Dempster–Shafer (D–S) belief network, logistic regression, and discriminant analysis to build the fault prediction models. The study was performed for the KC2 fault dataset available in the NASA fault dataset repository. The results were evaluated by using various performance evaluation measures. The results found that D–S belief networks produced better prediction accuracy compared to the other used fault prediction techniques.

Koru and Hongfang (2005) performed an experimental analysis to evaluate the performance of J48 and K-star learning techniques for the software fault prediction. The study was performed for various fault datasets available in the NASA fault data repository. Authors suggested that a better fault prediction performance can be achieved when fault dataset corresponding to larger software project has been used to build the fault prediction model. Additionally, authors found that class level software metrics can help in building better fault prediction models as compared to method level software metrics. Venkata et al. (2006) performed a comparative study to evaluate the performance of various machine learning techniques for software fault prediction. The study includes eleven different techniques such as linear regression, pace regression, support vector regression, neural network, naive Bayes, instance-based learning, etc. The experimental study was performed for four public datasets, and the performance has been evaluated using various performance parameters. The results found that combination of 1-Rule and instance-based learning techniques gives better prediction accuracy compared to the other used techniques.

Turhan and Bener (2009) investigated the use of naive Bayes technique for the software fault prediction. Authors have used seven fault datasets available in the NASA data repository to build the fault prediction model and evaluated the results using various performance parameters. Authors suggested that fault prediction model based on naive Bayes technique and principal component analysis (for attribute selection) performed significantly better for software fault prediction. Additionally, authors found that assigning weights to static code-level metrics can significantly increase the performance of fault prediction model. Elish and Elish (2008) build the fault prediction model using support vector machine (SVM) and compared its performance with various other machine learning techniques. The results of the study indicated that in general SVM-based fault prediction model performed better or at least similar to the other used machine learning techniques. Kanmani et al. (2007) built fault prediction models using two different neural network-based techniques, probabilistic neural network, and back-propagation neural network. The study was performed

for the fault datasets collected from the student's projects developed in a university. The results showed that in general neural network-based fault prediction models performed accurately for software fault prediction. Out of two used techniques, probabilities neural network performed better as compared to the back-propagation neural network. Menzies et al. (2010) performed an experimental investigation to evaluate the performance of naive Bayes with a logNum filter for software fault prediction. The results showed that used techniques performed significantly accurate for the software fault prediction. Catal (2011) presented a comprehensive review of the software fault prediction studies published between 1990 and 2009. Authors reviewed the available works and discussed the current trends of software fault prediction. Additionally, authors highlighted the need for a baseline framework that can be used to assess the effectiveness of fault prediction studies.

These studies showed that a lot of efforts have been made earlier using various techniques for building fault prediction models in the context of binary class classification of faults. However, it is still difficult to draw any generalized argument about the usefulness of the fault prediction techniques. In most of the studies, fault prediction models produced accuracy between 70 and 90% (approx.) with higher misclassification rate. Different techniques produced different performance for the different software projects and none of the technique proved better to the other techniques across different software systems. One possible reason for the abject performance of different techniques is the poor optimization of various control parameters of the technique. Some of the authors have built fault prediction techniques as black box, without examining domain of software system and the distribution of the dataset. Simple fault prediction techniques such as naive Bayes and logistic regression performed better compared to the complex fault prediction techniques such as support vector machine (Hall et al. 2012). Some other issues such as imbalance of faulty and non-faulty software modules, high data dimension, and noisy software modules need to be taken proper care to build effective fault prediction models.

3.2 Prediction of Number of Faults

Prediction of number of faults refers to predict that certain number of faults can be occurred in the given software system. This type of fault prediction model assigns an estimated number of faults to each given software module instead of classifying them into faulty or non-faulty classes. There have been various efforts reported that analyzed the performance of fault prediction techniques when predicting number of faults in the software system (Graves et al. 2000; Gao and Khosgoftaar 2007; Ostrand et al. 2004, 2005; Bell et al. 2006; Yan et al. 2010). Generally, error rate-based performance evaluation measures such as mean absolute error, mean relative error. have been used to assess the performance of the fault prediction models built in this context.

Graves et al. (2000) presented a fault prediction model that uses the fault occurrence history of the previous releases of the software system to predict the faults in

the current release of the software. Authors evaluated the built prediction model for a very large telecommunication system. The results showed that a combination of module age, changes made to the module, and duration of the change as software metrics help in achieving better prediction performance as compared to the size and complexity metrics. Ostrand et al. (2004, 2005), Bell et al. (2006) presented some studies using the negative binomial regression (NBR) to build the fault prediction model for the prediction of number of faults. Lines of code and software module modification history were used as software metrics for the fault prediction model. The results found that NBR-based fault prediction model performed accurately in predicting number of faults.

Yan et al. (2010) presented a fuzzy support vector regression-based fault prediction model to predict the number of faults in the given software system. The study was performed for a medical image system and redundant strapped down unit projects. The results indicated that used fault prediction techniques produced lower error values and detected most of the faults. Liguo (2012) evaluated the performance of negative binomial regression (NBR) to predict fault-proneness on an open-source software. Authors compared the results of NBR technique logistic regression model and found that presented NBR techniques could not outperform logistic regression techniques. Gao and Khosgoftaar (2007) conducted the analysis of different count models for the prediction of number of faults. The study was performed for an industrial software system and results found that count models can be used to build fault prediction models to predict number of faults.

Despite the benefits of building fault prediction model for the prediction of number of faults in terms of prioritizing testing resources, only few authors presented fault prediction model in this context. Initially, Graves et al. (2000) built the fault prediction model for the number of faults prediction; however, validation of the model was limited to one software system. Ostrand et al. (2004, 2005), Bell et al. (2006) performed an analysis of negative binomial regression for predicting number of faults; however, evaluation of results was limited to one performance measure only. Later, Gao and Khoshgoftaar (2007) built count model-based fault prediction models for the prediction of number of faults. Authors reported some useful results, but a thorough analysis of count models for other software system is required to prove the significance of count models for number of faults prediction. One issue with the presented studies related to the prediction of number of faults is that none of the studies performed a thorough and comprehensive analysis of fault prediction techniques to establish their usefulness in the context of number of faults prediction.

3.3 Cross-Project Software Fault Prediction

The types of fault prediction discussed in Sects. 3.1 and 3.2 have used historical fault datasets of the previous releases of the software system to train the fault prediction technique. Generally, most of the big software development organizations collect the data about the software development process and the information about the

faults found in the previous software releases. However, sometimes when fault prediction model needs to be built for the software development organization, which has not recorded any such information or it is a new organization and does not have any historical dataset. Then cross-project fault prediction is used to build the fault prediction model for such software development organization. In this type of fault prediction, fault datasets collected from other similar software systems are used to train the fault prediction technique to predict the faults in the given software system. Cross-project fault prediction can be used to classify the given software module into the faulty or non-faulty class or can be used to predict the number of faults in the given software module. The concept of cross-project fault prediction is relatively new in the software fault prediction area and only a few studies were reported evaluating the performance of fault prediction techniques in this context (He et al. 2012; Ma et al. 2012; Zimmermann et al. 2009; Peter et al. 2013; Canfora et al. 2013).

He et al. (2012) have presented a study to investigate the feasibility of building fault prediction model using cross-project dataset. The study mainly focused on the selection of appropriate training dataset in cross-project fault prediction context. Additionally, authors proposed an approach to automatically select appropriate training dataset for the projects without any historical data. In another study, Ma et al. (2012) explored the use of transfer learning methods to build accurate and effective cross-project fault prediction model. Authors concluded that the knowledge about the distribution characteristics of the given software metrics can help in the selection of suitable training dataset for the new software project.

Zimmermann et al. (2009) performed an extensive study to evaluate the performance of cross-project fault prediction models. The study was performed for the twelve different software projects and total 622 fault prediction models were generated. The results indicated that build cross-project fault prediction model is a challenging task and without the proper mechanism about the selection of training dataset, it is not possible to build accurate fault prediction model. Recently, Peter et al. (2013) proposed a filter named "peter filter" that helps in the selection of appropriate training dataset when building cross-project fault prediction model. Authors performed a thorough comparison of the presented filter with the other cross-project fault prediction techniques and found that the presented filter helps in building better cross-project fault prediction models.

Cross-project fault prediction can be very useful for the projects where there is no historical fault dataset is available to build the fault prediction model. However, earlier studies related to the cross-project prediction showed that there is a very limited success has been achieved in this domain. One of the main challenges in building cross-project fault prediction is the selection of suitable training dataset to train the fault prediction technique. Simple selection of training dataset from similar projects or the projects developed from the same process does not lead to the accurate prediction model. One factor that needs to be analyzed is the domain of the software development for the success of cross-project fault prediction. By the identification of different characteristics of the given software project, we can select the appropriate training dataset efficiently and effectively. Repeating the experiments

for more software projects and analysis of more potential determinants is necessary for the success of cross-project fault prediction.

3.4 Just-in-Time Software Fault Prediction

The traditional fault prediction types such as binary class classification, prediction of number of faults, and cross-project fault prediction identify the fault-prone software module at the design level or later. However, sometimes it is difficult to apply recommendations about the fault-proneness of the given software system practically due to the many possible reasons such as design decisions has been made by different set of people and it may be difficult for a developer to recall or change it, etc. To overcome these problems, just-in-time fault prediction has been introduced. In just-in-time fault prediction, the changes inducing faults are identified at the earlier stage and feedback has been provided to the developer to recall those changes. The advantages of just-in-time fault prediction are: (1) the fault prediction is made at the very fine granularity level, i.e., for a change in a code, and (2) the prediction recommendations can be easily applied, since it identifies fault-inducing changes at the developer level and it is easy for the developer to recall those changes because design decisions are still fresh.

Some of the works reported earlier related to the just-in-time fault prediction for the given software system (Kamei et al. 2016; Fukushima et al. 2014; Yang et al. 2016). Fukushima et al. (2014) performed an experimental study of just-in-time (JIT) fault prediction using cross-project model. The study was performed for eleven open-source software projects having various design-level software metrics. The results found that JIT fault prediction model produced a high performance for the software fault prediction. Additionally, authors suggested that JIT fault prediction based on the cross-project model can be a viable tool for building fault prediction models for the software projects where no historical data is available.

Yang et al. (2016) developed just-in-time fault prediction model using deep learning technique. Authors used six open-source software projects to build and evaluate the performance of fault prediction model. Authors compared the presented fault prediction model with the approach proposed by Kamei et al. (2016) and found that their presented model discovered 32.22% more faults than the Kamei et al.'s approach. Yang et al. (2015) built effort-aware just-in-time fault prediction model using unsupervised fault prediction technique. Authors compared the performance of the presented model with the model developed using supervised techniques and found that simple unsupervised fault prediction techniques base model could perform better than the supervised techniques-based fault prediction models in effort-aware JIT defect prediction. Kamei et al. (2016) investigated just-in-time fault prediction using cross-project model. Authors performed experimental analysis for eleven open-sources and found that just-in-time model performed well in the context of cross-project fault prediction.

Just-in-time (JIT) fault prediction is a relatively new concept in the context of software fault prediction. Some authors have been reported the works in this context and highlighted the usefulness of just-in-time fault prediction in software development context. However, one challenge in building JIT fault prediction model as highlighted by Kamei et al. (2016) also that a large amount of training data is required to train the technique, which may not be available in the initial stage of software development. We need to address this issue to build the effective and efficient JIT fault prediction model.

3.5 Summary

Many software fault prediction models have been proposed and investigated to detect and prevent faults in the early stages of the software development. Depending upon the goal of the prediction, different fault prediction models have been presented and studied. However, the main goal of all fault prediction models is to provide useful hints about the fault-proneness in the given software system. In this chapter, we have discussed different types of software fault prediction and have provided the details about the works done in each category. This can be useful to the researchers to select and build the appropriate fault prediction model based on the characteristic of the given software system.

References

Bell, R. M., Ostrand, T. J., & Weyuker, E. J. (2006). Looking for bugs in all the right places. In *2006 International symposium on software testing and analysis* (pp. 61–72).

Canfora, G., Lucia, A. D., Penta, M. D., Oliveto, R., Panichella, A., & Panichella, S. (2013). Multi-objective cross-project defect prediction. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST '13* (pp. 252–261). Washington: IEEE Computer Society.

Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert System Application, 38*(4), 4626–4636.

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software, 81*(5), 649–660.

Erturk, E., & Sezer, E. A. (2015). A comparison of some soft computing methods for software fault prediction. *Expert System with Applications, 42*(4), 1872–1879.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., & Ubayashi, N. (2014, May). An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 172–181).

Gao, K., & Khoshgoftaar, T. M. (2007). A comprehensive empirical study of count models for software fault prediction. *IEEE Transaction on Software Engineering, 50*(2), 223–237.

Gokhale, S. S., & Michael, R. L. (1997). Regression tree modeling for the prediction of software quality. In *Proceeding of ISSAT'97* (pp. 31–36).

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering, 26*(7), 653–661.

Guo, L., Cukic, B., & Singh, H. (2003). Predicting fault prone modules by the dempster-shafer belief networks. *In Proceedings of 18th IEEE International Conference on Automated Software Engineering* (pp. 249–252).

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

He, Z., Shu, F., Yang, Y., Li, M., & Wang, Q. (2012). An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering, 19*(2), 167–199.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., & Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering, 21*(5), 2072–2106.

Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V., & Thambidurai, P. (2007). Object-oriented software fault prediction using neural networks. *Journal of Information and Software Technology, 49*(5), 483–492.

Koru, A. G., & Hongfang, L. (2005). An investigation of the effect of module size on defect prediction using static measures. *In Proceedings of the 2005 workshop on Predictor models in software engineering, PROMISE '05* (pp. 1–5).

Liguo, Y. (2012). Using negative binomial regression analysis to predict software faults: A study of apache ant. *Information Technology Computer Science, 4*(8), 63–70.

Ma, Y., Luo, G., Zeng, X., & Chen, A. (2012). Transfer learning for cross-company software defect prediction. *Information and Software Technology, 54*(3), 248–256.

Menzies, T., Milton, Z., Burak, T., Cukic, B., Jiang, Y., & Bener, (2010). Defect prediction from static code features: Current results, limitations, new aproaches. *Automated Software Engineering, 17*(4), 375–407.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004). Where the bugs are. *ACM SIGSOFT software engineering notes, 29,* 86–96.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering, 31*(4), 340–355.

Peters, F., Menzies T., & Marcus, A. (2013). Better cross company defect prediction. In *10th IEEE working conference on mining software repositories (MSR'13)* (pp. 409–418).

Turhan, B., & Bener, A. (2009). Analysis of naive bayes' assumptions on software fault data: An empirical study. *Data Knowledge Engineering, 68*(2), 278–290.

Venkata, U. B., Farokh Bastani, B., & Yen, I. L. (2006). A unified framework for defect data analysis using the mbr technique. In *Proceeding of 18th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '06* (pp. 39–46).

Yan, Z., Chen X., & Guo, P. (2010). Software defect prediction using fuzzy support vector regression. In *International symposium on neural networks* (pp. 17–24). Springer.

Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015). Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (pp. 17–26).

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., & Leung, H. (2016). Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 157–168).

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data versus domain versus process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 91–100).

Chapter 4
Software Fault Dataset

Machine learning and statistical techniques are used in software fault prediction to predict the presence or the absence of faults in the given software modules. In order to make the predictions, a software fault prediction learns upon the software fault data having the information about the software system (software metrics) augmented with the fault value. An implicit requirement to perform effective software fault prediction is the availability of reasonable quality fault data. However, obtaining quality software fault data is difficult as in general software development companies are not keen to share their software development information or they are not having any software repository in first place.

Therefore, researchers moved toward the open-source software available publicly. Open-source software systems provide facility to the researchers to collect the dataset and build their own software fault dataset. However, there is one important requirement is that the software system must have been developed using a bug tracking system and have the provision of storing software development information. Once, these requirements are met, researchers can mine these repositories to collect the software fault dataset. The key problem with collecting dataset from the open-source systems is that it consumes ample amount of resources, and it is very difficult to ensure the accuracy of the dataset. Having these practical problems in obtaining software fault data, the use of public domain software fault datasets repositories such as NASA and PROMISE data repositories have gained popularity among the researchers. A significant amount of research using these public data repositories has been published over the last few decades. In this chapter, we first provide the description of software fault data repositories. A detailed discussion on the type of software repositories, how these repositories are organized, is presented. An overview of available software fault dataset repositories is also presented in the chapter. At the end, we discuss various issues associated with the fault dataset and also discuss possible remedies to the listed issues.

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_4

4.1 Description of Software Fault Dataset

Software fault dataset primarily contained software metrics information augmented with fault information. The detailed description of software metrics is provided in Chap. 2. Almost, all the software organization have used configuration management system (CMS) to manage and control the development of software system and to accommodate the changes that are occurred during software development life cycle. These CMSs store various information about the software artifacts (components) produced during the software development life cycle such as detailed design document, source code, user manuals, change log. (Bersoff et al. 1978; Babich 1986). Having all the information, these software repositories can serve as the source of knowledge, which can be used by research community or the developers to control and manage the software development process. The timely and effective mining of these repositories can guide the practitioners/managers in decision making.

During the software development life cycle, various types of software repositories have been maintained to store the software artifacts information. We have describe different types of software repositories commonly used.

1. *Source code repository*: Source code is the most important information about the software and can be used to reveal interesting findings of the system for the analysis. This type of repositories mainly contained the source code for different software artifacts. The open-source software systems use some publicly maintained software repositories such as Sourceforge.net, Google code to store the source code and commercial organizations use some in-house repository to store the same information. Source code repository records and maintains the development trail of the software. It tracks all the changes occurred in the software artifacts augmented with the metadata regarding the changes (Ambriola et al. 1990, Gall et al. 1998). Examples of some the popular source code repositories are Git, control version system (CVS), subversions (SVN), etc.
2. *Bug repositories*: Bug (also known as defect or fault) is a crucial entity in the software system. Bug repositories keep track and maintain the information about the bug life cycle. Generally, bugs are reported by the developers or the users of the software project and are recorded in the bug tracking system. A bug tracking system is a software program that is used to keep the record of different bugs or issues reported in software development life cycle. Usually, each software development organization employs bug tracking system to allow the developers/users to report various types of bugs generated during the software development or during the use. The main advantage of a bug repository is that it provides a centralized and concise information about the bugs reported in different phases of the software development life cycle and their state. The information stored in the bug repository can be used by the managers or stakeholders to take the corrective decisions to improve the software quality and reliability (Malhotra 2016). Bugzilla and Mozilla are the two most commonly used bug tracking systems used by the open-source community.

3. *Archived communication repositories*: These type of repositories are mainly used to keep the record regarding the discussions of various aspects of a software project during its life cycle. It includes the information such as email communications, messages exchanged among the developers, other developer-related information. This information can be mined and used by the managers to better understand the software development practices and environment.

The above-mentioned software repositories provide a vast amount of data and information about the software project and its artifacts. By mining these repositories, useful knowledge can be gathered and can be further used to improve the software quality and reliability. Software fault dataset is composed of bug information and software metrics information. The collection of software fault dataset can be undertaken by following these steps as stated below.

1. Collection of software metrics from the source code or change log information: This step requires the calculation of software metrics from the available software information. Depending upon the type of metrics to be collected, one can use change log information for calculating process metrics or source code version for calculating source code metrics. Various automated tools (open-source and proprietary) are available to calculate software metrics. Some of the tools are *Understand*[1] to calculate source code metrics, *CKJM*[2] to calculate object-oriented metrics, *Source Monitor*[3] to calculate metrics from the software written in C++, C, and C#, *Eclipse Metrics Plugin*[4] to calculate various metrics for code written in Java language, etc.
2. Collection of bug information from the bug tracking system: In bug tracking system, *Bug ID* is used to identify each bug uniquely in the bug repository and each bug is related to source code changes in a source code management repository. When a bug is submitted to the bug tracking system, team of developers and users first discuss the bug and after that bug information is stored with the additional comments and attachments. After the confirmation of bug, a developer fixes the bug and commits the changes to the software version control repository. Sometimes, additional information such as the status of a bug (e.g., unconfirmed, new, open, etc.), resolution of a bug (e.g., fixed, duplicate, invalid, etc.), and severity level of a bug is also stored on the bug tracking system. Mining bug repository involves the collection of all this data about the bug and then search for the commit messages. At least one file that is changed in the commit is associated with the bug. For a software artifact, all the bug information has been collected.
3. Linking of source code artifacts and bug information to generate the final dataset: Each bug reported in bug tracking system is associated with some part of the software system. Thus, linking of each bug with the corresponding component of the system is required. Some automated tools such as FAMIX, Churrasco

[1] Understand for Java, https://scitools.com/.

[2] CKJM, https://www.spinellis.gr/sw/ckjm/.

[3] Source Monitor, http://www.campwoodsw.com/sourcemonitor.html.

[4] Eclipse metric plugin, http://metrics.sourceforge.net/.

can be used to link software component with bugs retrieved from bug tracking system. A file (software component) in the source code repository contains a comment made by the developer and written at commit time. This commit often includes a reference to a bug tracking system. These references are used to link bug with the files of the software system. In some cases, the link between a file and bug is not formally defined. Thus, some techniques such as pattern matching are used to extract a list of bug ids and their corresponding files.

Careful execution of these steps results in the software fault dataset having the information of various software metrics as well as the information of the corresponding bug found in the software components.

4.2 Fault Dataset Repositories

One major problem with the building of software fault dataset is that it consumes a significant amount of time to collect the dataset fault as well as the accuracy of the built dataset is always questionable. The collection of bug information and mapping with them to the software component is often required to check the validity and unbiased nature of the data. This difficulty in obtaining software fault dataset led to the public domain fault dataset repositories. These repositories contained various software fault datasets, which can be used by the researchers and practitioners to perform the software fault prediction related studies. This section provides a brief description of various software fault data repositories available in the public domain. We only provided the details of the software repositories having software fault datasets. Rest of the available software repositories are not included in this study.

1. PROMISE (PRedictOr Models In Software Engineering, http://openscience.us/repo/): PROMISE software dataset repository is one of the most widely used repositories in the software fault prediction community. It contains software fault and other datasets such as effort estimation, refactoring, etc. of various open-source and proprietary software systems. The repository hosts datasets of a large number of applications and software systems such as Apache software project, including jEdit, Lucene, Xerces, and many others, Eclipse, Mozilla. Currently, PROMISE data repository contained fault dataset of 61 different software projects and their different releases. The size of the repository is growing rapidly with many more software datasets are added day by day.
2. Bug Prediction Dataset (BPD, http://bug.inf.usi.ch/index.php): This repository is a collection of various software metrics and the bug information of different software systems. The main aim of this repository is to provide software fault datasets to practitioners and researchers and to allow them to validate the new software fault prediction models. With the availability of the benchmarked datasets, researchers can compare different fault prediction approached and can assess how a new prediction approach is working in comparison to the existing ones. Currently, repository host the software fault dataset of five different Eclipse

projects with their subsequent releases. For each Eclipse project, the repository hosts the datasets having the information of various change metrics, CK metrics, number of pre-release defects, number of post-release faults, complexity metrics, code churn metrics for the object-oriented system, Entropy of CK metrics. The fault dataset is mainly available at the class-level. However, package-level or subsystem-level information can be derived by aggregating the data for each class.

3. Eclipse Bug Data Repository (https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/): Eclipse bug data repository has been primarily design to collect and store fault dataset of Eclipse project. This repository contained the fault datasets of Eclipse project and its components. Currently, repository hosts the dataset of three versions of Eclipse project, Eclipse-2.0, Eclipse-2.1, and Eclipse-3.0. For each version of Eclipse, its provide the information of various source code and structural metrics and pre-release and post-release faults.

4.3 Issues with Software Fault Datasets

In the previous section, we have provided the details of various software fault data repositories available in the public domain. These repositories are very popular among the research communities to assess and validate the fault prediction techniques. Generally, it is assumed that the datasets available in these repositories are of reasonable quality and are of free from any problem. However, this case is not always held for all the available datasets. There are some quality issues associated with the datasets available in the public domain repositories that need the proper handling/cleaning before using them for building software fault prediction model (Gray et al. 2011).

1. Outlier: In fault dataset, outliers are the observations that significantly deviated from the general observations of the dataset. Outliers violate the mechanism that is used to generate the data points (Aggarwal 2013). Careful detection and removal of outliers are particularly important in the fault prediction since there is a gray area between the outlier objects and normal objects. Arbitrarily removal of observation by considering them as outlier may lead to biased learning. Additionally, sometimes outlier observations contained fault information and removing them can cause significant loss of fault information.
2. Missing value: Missing values are the set of values left blank in the dataset. Due to the human error who entered the values or unavailability of the information, sometimes some values are not entered in the dataset. Some of the fault prediction models ignore the observations corresponding to the missing values and learn from the rest of the observation, while some other fault prediction models employ some autocorrective measures to deal with the missing value (Gray et al. 2011).
3. Repeated value: Repeated attributes represent a situation in the dataset where two attributes/features have the same identical values for each observation. This

particular case occurs in the dataset when a single attribute is over-described. For building accurate fault prediction model, it is required to remove one of such attribute, so that each attribute is being represented only once (Gray et al. 2011).

4. Redundant and irrelevant value: Data redundancy is a condition is the dataset where same attribute/feature is held in two or more observations with the same class label. Such data points are problematic for the effective learning of fault prediction since this could lead to the overfitting of the prediction model and could result in the model performance decrement (Gray et al. 2011).
5. Class imbalance: Class imbalance occurs when the number of positive class observations (minor class) is less than the number of negative class observations (major class). It represents a situation where a class of observations is rarely presented in the dataset compared to the other types of observations. In this case, observations of major class dominate the dataset as opposed to the observations of the minor class. This imbalance in the distribution of observations can lead to the biased learning of prediction model toward the observations of major class. The prediction model can produce poor results for the minor class observations (Moreno-Torres et al. 2012).
6. Data shifting problem: In fault dataset, data shifting problem occurs when the training dataset and testing dataset are not following the same joint distribution (Moreno-Torres et al. 2012). In the literature, data shifting also refers to concept shift or concept drift. When data shifting happens, the prediction model having previous knowledge about the dataset is in no use as the new dataset has the change in the class distribution. Data shifting problem can have a serious effect on the performance of the prediction models. Therefore, it is required to handle the data shifting problem for building accurate and effective fault prediction model.
7. High data dimensionality: Dimensionality refers to the number of attributes/features in the given fault dataset. When the number of attributes is higher, then it is known as high data dimensionality. Sometimes high data dimensionality is defined as the situation where number of attributes (P) is greater than the number of observations (m) in the dataset ($p > m$). Earlier, it was found that high data dimensionality could decrease the performance a fault prediction model and could lead to the higher misclassification errors (Kehan et al. 2011; Rodriguez et al. 2012). Higher data dimensionality is also resulted in the higher computational cost and memory usage when building the fault prediction model.

In recent years, the use of machine learning techniques has been increased extensively in software fault prediction (Gokhale and Michael 1997a, b; Guo et al. 2003; Koru and Hongfang 2005; Elish and Elish 2008; Catal 2011). However, the various quality issues associated with public domain datasets are hindering the performance of the fault prediction models. It was found that the earlier studies related to the software fault prediction have not handled data quality issues properly. Some of the studies explicitly handled data quality issues when building fault prediction models (Calikli and Bener 2013; Shivaji et al. 2009); however, they are very few in numbers. Higher attentions have been paid to the data quality issues such as "high data dimension,"

"outlier," and "class imbalance" (Rodriguez et al. 2012; Seiffert et al. 2008, 2009; Alan and Catal 2009), whereas other quality issues such as "data shifting," "missing values," "redundant values" have not been paid significant attention.

4.4 Summary

Software fault prediction makes the use of software process and development dataset having different software metrics and bug information to predict the fault-proneness of the software systems. The public domain fault datasets helped the research community to carry out the fault prediction study over the benchmarked datasets. However, this ease of availability of the dataset can be dangerous as the public domain datasets are stuffed with may quality issues. Unfortunately, the datasets hosting sites generally do not indicate these issues explicitly. Therefore, using these datasets without proper preprocessing/cleaning can lead to the biased learning and inaccurate results. In this chapter, we first discussed the software fault datasets and its components. We presented the details of commonly available public domain fault datasets. We also discussed various quality issues associated with fault datasets to be taken care before building any fault prediction model.

References

Aggarwal, C. C. (2013). *Outlier analysis*. Springer Science and Business Media.

Alan, O., & Catal, C. (2009). An outlier detection algorithm based on object-oriented metrics thresholds. In *Proceedings of the 24th International Symposium on Computer and Information Sciences* (pp. 567–570).

Ambriola, V., Bendix, L., & Ciancarini, P. (1990). The evolution of configuration management and version control. *Software Engineering Journal, 5*(6), 303–310.

Babich, A. W. (1986). *Software confguration management: coordination for team productivity*. Boston, MA: Addison-Wesley.

Bersoff, E. H., Henderson, V. D., & Siegel, S. G. (1978). Software confguration management. *ACM SIGSOFT Software Engineering Notes, 3*(5), 9–17.

Calikli, G., & Bener, A. (2013). An algorithmic approach to missing data problem in modeling human aspects in software development. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering* (pp. 1–10).

Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert System Application, 38*(4), 4626–4636.

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software, 81*(5), 649–660.

Gall, H. Hajek, K., & Jazayeri, M. (1998). Detection of logical coupling based on product release history. In *Proceedings of the 14th International Conference on Software Maintenance* (pp. 190–198), Bethesda: MD.

Gokhale, S. S., & Michael, R. L. (1997a). Regression tree modeling for the prediction of software quality. In *Proceedings of the ISSAT International Conference on Reliability and Quality in Design* (pp. 31–36).

Gokhale, S. S., & Michael, R. L. (1997b). Regression tree modeling for the prediction of software quality. In Proceeding of ISSAT'97, pp. 31–36.

Gray, D., Bowes, D., Davey, N., Sun, Y., & Christianson, B. (2011). The misuse of the NASA metrics data program data sets for automated software defect prediction. In *Proceedings of the 15th Annual Conference on Evaluation and Assessment in Software Engineering* (pp. 96–103).

Guo L., Cukic B., & Singh H. (2003). Predicting fault prone modules by the dempster–shafer belief networks. In *Proceedings of 18th IEEE international conference on automated software engineering*, (pp 249–252).

Kehan, G., Khoshgoftaar, T. M., Wang, H., & Seliya, N. (2011). Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software Practice and Experience, 41*(5), 579–606.

Koru, A. G. & Hongfang, L. (2005). An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the 2005 workshop on Predictor models in software engineering, PROMISE '05* (pp. 1–5).

Malhotra, R. (2016). *Empirical research in software engineering: Concepts, analysis, and applications*. US: CRC Press.

Moreno-Torres, J. G., Raeder, T., Alaiz-Rodriguez, R., Chawla, N. V., & Herrera, F. (2012). A unifying view on dataset shift in classification. *Pattern Recognition, 45*(1), 521–530.

Rodriguez, D., Herraiz, I., & Harrison, R. (2012). On software engineering repositories and their open problems. In *Proceedings of the 1st International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering* (pp. 52–56).

Seiffert, C., Khoshgoftaar, T. M., Hulse, J. V., & Napolitano, A. (2008). Building useful models from imbalanced data with sampling and boosting. In *Proceedings of the 21st International FLAIRS Conference* (pp. 306–311).

Seiffert, C., Khoshgoftaar, T., & Van Hulse, J. (2009). Improving software-quality predictions with data sampling and boosting. *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans, 39*(6), 1283–1294.

Shivaji, S., James Whitehead, E., Akella, R., & Kim, S. (2009). Reducing features to improve bug prediction. In *Proceedings of the International Conference on Automated Software Engineering* (pp. 600–604).

Chapter 5
Evaluation of Techniques for Binary Class Classification

Depending upon the type of fault prediction model, a model can be used to classify software modules into faulty or non-faulty categories or a model can be used to predict the number of faults in the given software system. The former approach is known as the binary class classification of software faults. In this type of prediction, if a given software module has one or more faults, then it is classified as faulty otherwise it is classified as non-faulty. Most the studies available in the literature related to the software fault prediction have focused on the binary class classification of software faults.

Accurate prediction of software faults is one of the holy grails of software testing process. Software fault prediction aims to fulfill this goal by predicting the fault-proneness of the software modules. A large number of studies have been successfully applied various machine learning and statistical techniques to build fault prediction model for the binary class classification of faults. These used techniques varied in terms of accuracy, complexity, and the input data they require. However, there is still not a consensus about the usefulness of different techniques for building fault prediction model. Different studies have evaluated various techniques for different software project datasets and have used various performance evaluation measures to assess their performance. This made the comparison of fault prediction techniques more complicated. Therefore, it is needed to evaluate the performance of frequently used fault prediction techniques for the common software fault datasets and using common performance evaluation measures to reach the consensus about the usefulness of these techniques.

5.1 Description of Different Fault Prediction Techniques

Various researchers have used different set of techniques to build and evaluate the software fault prediction models. It includes several machine learning techniques such as Bayesian-based techniques, tree-based techniques, instance-based techniques and several statistical techniques such as logistic regression, linear regression. In the presented study, we have used fifteen different fault prediction techniques that have

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_5

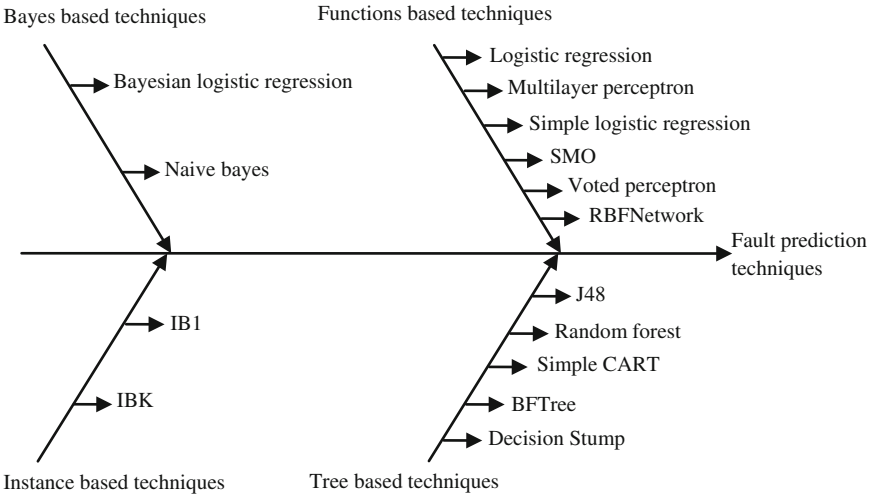


Fig. 5.1 Classification of the fault prediction techniques used in the presented study (Malhotra 2015)

been used mostly used to build the fault prediction models for the binary class classification of software faults (Safavian and Landgrebe 1991; Genkin et al. 2007; (Wang and Li 2010; Boetticher 2005; Gayathri and Sudha 2014). The classification of the used techniques has been given in Fig. 5.1.

The used fault prediction techniques have been classified into four major categories: Bayesian-based techniques, tree-based techniques, instance-based techniques, and function-based techniques. Each category consists of various fault prediction techniques. All the used fault prediction techniques are supervised learning techniques. A technique is called supervised technique when the technique operates under the supervision provided with the actual outcome for each of the training examples. The supervised technique requires known fault measurement data (i.e., the number of faults, fault density, or faulty or non-faulty) for training data. Usually, fault measurement data from previous versions, pre-release, or similar project can act as training data to predict new projects.

In tree-based techniques, a tree type (hierarchical) of structure is created that classifies the given testing example into one of the output classes (faulty or non-faulty) by deducing some learning rules from the given features (software metrics) (Safavian and Landgrebe 1991). The root node is selected by determining a feature that best splits the training data. Similarly, other features are selected as intermediate nodes and leaves. We have used five different tree-based fault prediction techniques. They are—random forest, Simple CART, J48, BF Tree, and decision stump (Athanassoulis and Ailamaki 2014; Moser et al. 2008).

Function-based techniques are the one that can be written down as mathematical equations. We have used six different function-based fault prediction techniques. They are—logistic regression, multilayer perceptron, simple logistic, SMO,

RBFNetwork, and voted perceptron. Logistic regression fits a regression model that minimizes the squared root error between the actual and the predicted value (Moser et al. 2008). It is used to obtain the output using regression if the output class is nominal. SMO technique is an implementation of sequential minimal optimization algorithm for training a support vector machine. It is used polynomial or Gaussian kernels to build the support vectors (Elish and Elish 2008). Simple logistic is an implementation of logistic regression that uses a LogitBoost with simple regression functions as base learners. It determines the number of iterations to be performed using cross-validation, which supports automatic attribute selection (Moser et al. 2008). Voted perceptron is the perceptron algorithm that uses a modified basic perceptron to multiplicative updates (Moser et al. 2008). RBFNetwork is an implementation of Gaussian radial basis function. It uses K-means algorithm to derive the centers and widths of hidden units and uses logistic regression to combine the outputs obtained from the hidden layer (Liu et al. 2013). Multilayer perceptron consists a series of interconnected processing elements in the form of layers govern by some weights. During the training phase, the intermediate layers are updated iteratively until the difference between the actual value and predicted value becomes below certain threshold value (Gayathri and Sudha 2014).

Instance-based fault prediction techniques are the one that learn from the similar examples available in the training dataset to classify a given testing example. These techniques do not generate any explicit knowledge from the training dataset. Based on the value of the similar examples in the training dataset, they classify the given testing example into one of the output classes. We have used two different instance-based fault prediction techniques. They are—IB1 and IBK ($K = 5$) (Koru and Liu 2005).

Bayesian-based techniques are the techniques that use a statistical inference based on Bayes' theorem to update the probability of a hypothesis as more supporting evidence become available. We have used two different Bayesian-based fault prediction techniques. They are—Bayesian logistic regression and naive Bayes. Naive Bayes technique uses Bayes' theorem with the assumption that each pair of attribute (software metrics) is independent of each other (Koru and Liu 2005). Naive Bayes technique assumes that the presence of a particular feature in the dataset does not affect the presence of any other feature. Bayesian logistic regression uses Bayesian inference to determine the logistic regression parameters (Safavian and Landgrebe 1991). It first calculates the likelihood function of the data, and then it calculates the prior distribution of all the unknown parameters. Last, it applies Bayes' theorem to determine the posterior distribution of all parameters.

5.2 Performance Evaluation Measures

In our experiment, we have built the fault prediction models for the binary class classification of software faults. Each fault prediction model has labeled the given software modules as faulty or non-faulty. To assess the performance of built fault

prediction models, we have used accuracy, precision, recall, F-measure, and AUC (area under ROC) curve performance measures. The root of all these performance measures is the confusion matrix. The details about these performance measures have been given in Sect. 2.2, Chap. 2. Most of the studies related to the binary class classification of software faults have used some or all of these performance measures to evaluate the performance of the fault prediction techniques. For this reason, we have used these measures to perform the overall assessment of the used fault prediction techniques.

5.3 Evaluation of the Fault Prediction Techniques

5.3.1 Software Fault Datasets

A large of the studies available in the literature related to the software fault prediction have used the fault datasets available in the PROMISE data repository to build and evaluate the fault prediction models. Therefore, in this work, we have used the software fault datasets available in the PROMISE data repository to evaluate the performance used fault prediction techniques. PROMISE data repository contained the fault datasets of various open-source software projects including various Apache products and others. Currently, it has contained fault datasets of sixty-five different software projects and their different releases. The size of the available fault datasets varies, and some datasets have only few software modules, whereas some other datasets have thousand of software modules. All the available datasets contained the same twenty object-oriented software metrics. They are—weighted method count (WMC), coupling between objects (CBO), response for a class (RFC), depth of inheritance tree (DIT), number of children (NOC), inheritance coupling (IC), coupling between methods (CBM), afferent coupling (CA), efferent coupling (CE), measure of functional abstraction (MFA), lack of cohesion of methods (LCOM), lack of cohesion of methods 3 (LCOM3), cohesion among methods (CAM), measure of aggregation (MOA), number of public methods in a class (NPM), data access metric (DAM), average method complexity (AMC), lines of code (LOC), cyclomatic complexity (CC). The detailed description of these metrics has been given by M. Jureczko (Jureczko and Madeyski 2010). Various studies related to the software fault prediction used object-oriented (OO) software metrics to build fault prediction models. The results of these studies showed that OO metrics performed significantly accurate for the fault prediction (Briand and Wüst 2002; Elish et al. 2011; Shatnawi and Li 2008). In the presented experimental study, we have used forty-six different software fault datasets out of total sixty-five available fault datasets having more than 100 software modules. We have discarded nineteen fault datasets having less than 100 software modules. The rationale behind discarding these datasets is that it is difficult to trust the results of datasets of very small size due to the requirement of some fault prediction techniques of the larger dataset for the training. The description of the used

software fault datasets is given in Table 5.1. For each given software project, we have used its multiple releases to build and evaluate the fault prediction model.

Each of the used fault dataset contained unique ID, path to the software module, twenty OO metrics, and number of faults information. We have preprocessed the datasets and removed the unique ID and path to the software module fields from the datasets. In the presented study, we focus on the binary class classification of software faults. Therefore, we have transformed the number of faults information into faulty and non-faulty information. If a software module has one or more faults, then we labeled that module as faulty otherwise non-faulty. We have applied the same preprocess and transformation for all the used forty-six fault datasets.

5.3.2 Experimental Setup

The software fault datasets as given in Table 5.1 vary in terms of the percentage of faulty modules in the datasets. This percentage of faulty module varies between 3.82 and 98.68%. The performance of fault prediction techniques may vary with respect to the percentage of faulty modules in the fault datasets. Therefore, we have categorized the used fault datasets into four different categories based the percentage of the faulty module and evaluated all the fault prediction techniques for each category. The details about the different categories of fault datasets have been given in Table 5.2. If a fault dataset has less than 10% of the faulty software modules, then we categorized it into category-1. If a fault dataset has faulty software modules between 10 and 20%, then we categorized it into category-2. If a fault dataset has faulty software modules between 20 and 30%, then we categorized it into category-3. If a fault dataset has more than 30% faulty software modules, then we categorized it into category-4. This analysis helps us to find out that whether the performance of fault prediction techniques has been affected by the percentage of faulty modules or not.

5.3.3 Experiment Execution

We have used Weka machine learning tool to build and evaluate different fault prediction models (Witten and Frank 2005). All the reported experiments utilizing different fault prediction techniques have been implemented using Weka tool by using the default values of various parameters of used techniques as given in Weka. We have used a ten-fold cross-validation scheme to build and evaluate the fault prediction models. The original fault datasets has been partitioned into ten different parts. Each time nine parts are used as training dataset, and rest one part is used as testing dataset. This particular process has been repeated for ten times, and results are averaged for all the iterations.

Table 5.1 Description of used software fault datasets

Dataset	Release	# non-commented-LOC	Total number of modules	Total number of faulty modules	% of faulty modules (%)
Ant	Ant-1.7	208KLOC	746	166	22.25
Arc	–	31KLOC	235	27	11.49
Camel	Camel-1.0	33KLOC	340	13	3.82
	Camel-1.2	66KLOC	609	216	35.47
	Camel-1.4	98KLOC	873	145	16.61
	Camel-1.6	113KLOC	966	188	19.46
Ivy	Ivy-1.1	27KLOC	112	63	56.25
	Ivy-1.4	59KLOC	242	16	6.61
	Ivy-2.0	87KLOC	353	40	11.33
Jedit	Jedit-3.2	128KLOC	273	90	32.97
	Jedit-4.0	144KLOC	307	75	24.43
	Jedit-4.1	153KLOC	313	79	25.24
	Jedit-4.2	170KLOC	368	48	13.04
	Jedit-4.3	202KLOC	493	11	2.23
Log4 J	Log4 J-1.0	21KLOC	136	34	25.00
	Log4 J-1.1	19KLOC	110	37	33.64
	Log4 J-1.2	38KLOC	206	189	91.75
Lucene	Lucene-2.0	50KLOC	196	91	46.43
	Lucene-2.2	63KLOC	248	144	58.06
	Lucene-2.4	102KLOC	341	203	59.53
Poi	Poi-1.5	55KLOC	238	141	59.24
	Poi-2.0	93KLOC	315	37	11.75
	Poi-2.5	119KLOC	386	248	64.25
	Poi-3.0	129KLOC	443	281	63.43
Prop	Prop-1	3816KLOC	18472	2738	14.82
	Prop-2	3748KLOC	23015	2431	10.56
	Prop-3	1604KLOC	10275	1180	11.48
	Prop-4	1508KLOC	8719	840	9.63
	Prop-5	1081KLOC	8517	1299	15.25
	Prop-6	97KLOC	661	66	9.98
Redaktor	–	59KLOC	177	27	15.25
Synapse	Synapse-1.0	28KLOC	157	16	10.19
	Synapse-1.1	42KLOC	223	60	26.91
	Synapse-1.2	53KLOC	257	87	33.85
Tomcat	–	300KLOC	859	77	8.96
Velocity	Velocity-1.4	51KLOC	197	147	74.62
	Velocity-1.5	53KLOC	215	142	66.05
	Velocity-1.6	57KLOC	230	78	33.91
Xalan	Xalan-2.4	225KLOC	724	111	15.33
	Xalan-2.5	304KLOC	804	387	48.13
	Xalan-2.6	411KLOC	886	411	46.39
	Xalan-2.7	428KLOC	910	898	98.68
Xerces	Xerces-1.2	159KLOC	441	71	16.10
	Xerces-1.3	167KLOC	454	69	15.20
	Xerces-1.4	141KLOC	589	437	74.19
	Xerces-init	90KLOC	163	77	47.24

Table 5.2 Categorization of used software fault datasets based on the percentage of faulty modules

Category	Percentage of faulty software modules (%)	Number of software fault datasets
Category-1	Less than 10	6
Category-2	10–20	15
Category-3	20–30	5
Category-4	Above 30	20

5.3.4 Results and Analysis

The results obtained from the experimental analysis have been reported in Tables 5.3, 5.4, 5.5, and 5.6. Each table is corresponding to one category of software fault datasets. Each table shows the value of all five performance evaluation measures—accuracy, precision, recall, F-measure, and AUC. For each performance evaluation measure, we have reported the minimum, average, and maximum value of each fault prediction technique. We have presented the summarized results of each used fault prediction techniques for all the datasets fallen into one category.

Results for the Category-1

In this category, we have included the fault datasets have less than 10% of the faulty software modules. Six fault datasets have been fallen into this category. The results for this category are given in Table 5.3. The observations based on Table 5.3 are summarized below.

- With respect to the accuracy measure, all the fault prediction techniques have produced values more than 80% with the highest value of 97.76% using Bayesian logistic regression. The average value for the accuracy measure is above 90% for all the used techniques.
- With respect to the precision measure, all the fault prediction techniques have produced values greater than 0.81 with the highest value of 0.90 using naive Bayes technique. The average value is greater than 0.87 for all the used techniques.
- With respect to the recall measure, all the techniques achieved values greater than 0.80 with the highest value of 0.98 for the Bayesian logistic regression, IBK, and tree-based classifiers. The average value is greater than 0.90 for all the used techniques.
- With respect to the F-measure measure, all the used techniques produced values above 0.83 with the highest value of 0.97 for Bayesian logistic regression and tree-based classifiers. The average value is greater than 0.89 for all the used techniques.
- With respect to the AUC measure, all the techniques produced values above 0.38 with the highest value of 0.81 for the naive Bayes technique. The average value is greater than 0.50 for all the used techniques.

Table 5.3 Results of experiment for category-1 datasets (Datasets have less than 10% of faulty modules, 6 datasets)

Technique		Bayesian logistic regression	Naive Bayes	Logistic regression	Multilayer perceptron	Simple logistic	SMO	Voted perceptron
Accuracy	Min.	90	80.15	88.93	88.48	89.84	90	86.72
	Average	93.12	87.94	92.2	92.26	93.2	93.11	91.32
	Max.	97.76	93.69	96.54	97.35	97.15	97.76	96.95
Precision	Min.	0.81	0.86	0.84	0.85	0.81	0.81	0.81
	Average	0.89	0.9	0.9	0.9	0.9	0.87	0.88
	Max.	0.96	0.97	0.96	0.96	0.96	0.96	0.96
Recall	Min.	0.9	0.8	0.89	0.87	0.9	0.9	0.87
	Average	0.93	0.88	0.92	0.92	0.93	0.93	0.91
	Max.	0.98	0.94	0.97	0.97	0.97	0.98	0.97
F-measure	Min.	0.85	0.83	0.86	0.86	0.85	0.85	0.85
	Average	0.91	0.89	0.91	0.9	0.91	0.91	0.9
	Max.	0.97	0.95	0.96	0.97	0.96	0.97	0.96
AUC	Min.	0.5	0.59	0.48	0.38	0.59	0.5	0.49
	Average	0.5	0.69	0.65	0.62	0.69	0.5	0.5
	Max.	0.51	0.81	0.8	0.77	0.8	0.5	0.53

(continued)

Table 5.3 (continued)

Technique		IB1	IBK ($k = 5$)	J48	Decision stump	Random forest	Simple cart	BF tree	RBFNetwork
Accuracy	Min.	84.54	87.42	89.24	90	86.21	90	90	90
	Average	89.67	92.38	92.56	93.11	91.15	93.15	92.99	93.04
	Max.	95.93	97.76	97.76	97.76	96.95	97.74	97.76	97.76
Precision	Min.	0.86	0.82	0.85	0.81	0.86	0.81	0.81	0.81
	Average	0.9	0.88	0.89	0.87	0.89	0.89	0.89	0.87
	Max.	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96
Recall	Min.	0.85	0.87	0.89	0.9	0.86	0.9	0.9	0.9
	Average	0.9	0.92	0.93	0.93	0.91	0.93	0.93	0.93
	Max.	0.96	0.98	0.98	0.98	0.97	0.98	0.98	0.98
F-measure	Min.	0.85	0.84	0.86	0.85	0.86	0.85	0.85	0.85
	Average	0.9	0.91	0.91	0.91	0.9	0.91	0.91	0.91
	Max.	0.96	0.97	0.97	0.97	0.96	0.97	0.97	0.97
AUC	Min.	0.5	0.55	0.47	0.49	0.54	0.41	0.46	0.48
	Average	0.58	0.69	0.55	0.59	0.68	0.5	0.55	0.63
	Max.	0.64	0.75	0.66	0.7	0.8	0.62	0.71	0.74

Table 5.4 Results of experiment for category-2 datasets (Datasets have 10–20% of faulty modules, 15 datasets)

Technique		Bayesian logistic regression	Naive Bayes	Logistic regression	Multilayer perceptron	Simple logistic	SMO	Voted perceptron
Accuracy	Min.	64.41	57.92	76.36	78.44	77.66	76.36	66.23
	Average	84.46	80.07	85.43	84.71	85.75	85.71	83.32
	Max.	89.8	85.55	89.8	89.61	89.77	90.34	89.17
Precision	Min.	0.42	0.69	0.76	0.75	0.76	0.7	0.7
	Average	0.75	0.81	0.82	0.82	0.82	0.79	0.77
	Max.	0.84	0.88	0.89	0.87	0.89	0.9	0.83
Recall	Min.	0.64	0.58	0.76	0.78	0.78	0.76	0.66
	Average	0.84	0.8	0.85	0.85	0.86	0.86	0.83
	Max.	0.9	0.86	0.9	0.9	0.9	0.9	0.89
F-measure	Min.	0.51	0.58	0.76	0.76	0.74	0.72	0.55
	Average	0.78	0.8	0.82	0.83	0.82	0.8	0.78
	Max.	0.85	0.85	0.9	0.87	0.89	0.89	0.85
AUC	Min.	0.5	0.57	0.63	0.61	0.55	0.5	0.49
	Average	0.51	0.71	0.73	0.72	0.73	0.54	0.52
	Max.	0.53	0.82	0.83	0.85	0.83	0.73	0.61

(continued)

Table 5.4 (continued)

Technique		IB1	IBK ($k = 5$)	J48	Decision stump	Random forest	Simple cart	BF tree	RBFNetwork
Accuracy	Min.	74.81	77.51	76.37	67.27	77.39	78.7	79.27	73.76
	Average	81.98	85.46	85.19	85.02	83.92	85.93	85.67	85.11
	Max.	87.78	89.64	90.01	90.34	89.63	90.34	89.57	89.43
Precision	Min.	0.74	0.73	0.74	0.7	0.75	0.72	0.71	0.7
	Average	0.82	0.82	0.83	0.76	0.82	0.8	0.82	0.78
	Max.	0.87	0.88	0.89	0.9	0.88	0.9	0.88	0.86
Recall	Min.	0.75	0.78	0.76	0.67	0.77	0.79	0.79	0.74
	Average	0.82	0.85	0.85	0.85	0.84	0.86	0.86	0.85
	Max.	0.88	0.9	0.9	0.9	0.89	0.9	0.9	0.89
F-measure	Min.	0.75	0.74	0.75	0.67	0.76	0.73	0.73	0.74
	Average	0.82	0.83	0.83	0.8	0.83	0.82	0.82	0.8
	Max.	0.87	0.88	0.89	0.89	0.88	0.89	0.88	0.86
AUC	Min.	0.55	0.65	0.47	0.55	0.63	0.42	0.41	0.62
	Average	0.64	0.75	0.64	0.65	0.74	0.59	0.62	0.69
	Max.	0.79	0.88	0.78	0.76	0.89	0.82	0.81	0.78

Table 5.5 Results of experiment for category-3 datasets (Datasets have 20–30% of faulty modules, 5 datasets)

Technique		Bayesian logistic regression	Naive Bayes	Logistic regression	Multilayer perceptron	Simple logistic	SMO	Voted perceptron
Accuracy	Min.	72.97	72.52	78.82	73.87	77.92	77.45	68.26
	Average	76.35	78.61	81.1	79.52	81.52	80.24	71.47
	Max.	80.53	84.44	83.33	81.73	83.7	82.96	76.77
Precision	Min.	0.64	0.74	0.77	0.74	0.76	0.76	0.64
	Average	0.72	0.78	0.8	0.79	0.8	0.79	0.7
	Max.	0.78	0.84	0.83	0.81	0.83	0.83	0.77
Recall	Min.	0.73	0.73	0.79	0.74	0.78	0.78	0.68
	Average	0.76	0.79	0.81	0.8	0.82	0.8	0.71
	Max.	0.81	0.84	0.83	0.82	0.84	0.83	0.77
F-measure	Min.	0.64	0.73	0.77	0.74	0.76	0.71	0.65
	Average	0.71	0.78	0.8	0.79	0.8	0.77	0.7
	Max.	0.77	0.83	0.83	0.81	0.83	0.81	0.77
AUC	Min.	0.5	0.71	0.72	0.74	0.72	0.56	0.57
	Average	0.57	0.77	0.78	0.76	0.79	0.63	0.63
	Max.	0.65	0.84	0.82	0.79	0.82	0.69	0.7

(continued)

Table 5.5 (continued)

Technique		IB1	IBK ($k = 5$)	J48	Decision stump	Random forest	Simple cart	BF tree	RBFNetwork
Accuracy	Min.	72.97	74.77	72.52	69.36	73.87	75.67	73.87	75.16
	Average	76.17	78.79	76.31	76.79	78.62	79.68	78.88	78.48
	Max.	79.41	80.71	79.06	83.35	80.93	83.08	82.14	81.48
Precision	Min.	0.73	0.72	0.72	0.67	0.74	0.74	0.72	0.73
	Average	0.76	0.77	0.75	0.76	0.78	0.78	0.77	0.77
	Max.	0.8	0.79	0.78	0.83	0.8	0.83	0.81	0.81
Recall	Min.	0.73	0.75	0.73	0.69	0.74	0.76	0.74	0.75
	Average	0.76	0.79	0.76	0.77	0.79	0.8	0.79	0.78
	Max.	0.79	0.81	0.79	0.83	0.81	0.83	0.82	0.82
F-measure	Min.	0.73	0.72	0.72	0.68	0.74	0.74	0.73	0.74
	Average	0.76	0.77	0.76	0.76	0.78	0.79	0.78	0.78
	Max.	0.8	0.79	0.78	0.83	0.81	0.83	0.82	0.81
AUC	Min.	0.66	0.73	0.62	0.62	0.76	0.65	0.65	0.63
	Average	0.68	0.77	0.66	0.66	0.79	0.68	0.66	0.72
	Max.	0.72	0.81	0.68	0.7	0.81	0.7	0.69	0.76

Table 5.6 Results of experiment for category-4 datasets (Datasets have 30% or above faulty modules, 20 datasets)

Technique		Bayesian logistic regression	Naive Bayes	Logistic regression	Multilayer perceptron	Simple logistic	SMO	Voted perceptron
Accuracy	Min.	49.23	49.06	60.32	61.13	62.39	59.4	48.97
	Average	67.16	64.94	75.21	75.23	76.37	73.76	64.25
	Max.	98.78	83.93	98.12	98.78	98.67	98.78	98.56
Precision	Min.	0.32	0.58	0.6	0.62	0.63	0.6	0.57
	Average	0.59	0.72	0.75	0.75	0.76	0.73	0.69
	Max.	0.98	0.99	0.99	0.98	0.98	0.98	0.98
Recall	Min.	0.49	0.49	0.6	0.61	0.62	0.59	0.49
	Average	0.67	0.65	0.75	0.75	0.76	0.73	0.64
	Max.	0.99	0.84	0.98	0.99	0.99	0.99	0.99
F-measure	Min.	0.4	0.47	0.6	0.61	0.61	0.54	0.46
	Average	0.59	0.64	0.75	0.75	0.75	0.72	0.6
	Max.	0.98	0.9	0.98	0.98	0.98	0.98	0.98
AUC	Min.	0.5	0.57	0.62	0.66	0.5	0.5	0.5
	Average	0.54	0.73	0.75	0.76	0.75	0.65	0.59
	Max.	0.69	0.85	0.92	0.9	0.9	0.77	0.7

(continued)

Table 5.6 (continued)

Technique		IB1	IBK ($k = 5$)	J48	Decision stump	Random forest	Simple cart	BF tree	RBFNetwork
Accuracy	Min.	61.13	59.1	61.53	55.58	60.32	57.08	59.1	56.53
	Average	74.15	74.87	75	72.56	76.43	76.11	75.62	72.46
	Max.	98.99	99.39	99.11	98.78	99.44	99.11	99.11	98.78
Precision	Min.	0.61	0.6	0.61	0.34	0.59	0.56	0.58	0.57
	Average	0.74	0.74	0.75	0.73	0.76	0.75	0.75	0.72
	Max.	0.99	0.99	0.99	0.98	1	0.99	0.99	0.98
Recall	Min.	0.61	0.59	0.62	0.56	0.6	0.57	0.59	0.57
	Average	0.74	0.75	0.75	0.73	0.76	0.76	0.76	0.72
	Max.	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
F-measure	Min.	0.61	0.59	0.61	0.43	0.59	0.57	0.58	0.55
	Average	0.74	0.74	0.75	0.7	0.76	0.75	0.75	0.71
	Max.	0.99	0.99	0.99	0.98	0.99	0.99	0.99	0.98
AUC	Min.	0.6	0.62	0.61	0.5	0.65	0.42	0.57	0.54
	Average	0.7	0.76	0.7	0.66	0.79	0.7	0.71	0.71
	Max.	0.87	0.9	0.91	0.86	0.93	0.88	0.9	0.82

- Overall, we found that the Bayesian-based classifiers and tree-based classifiers have performed better compared to other used fault prediction techniques for this category for all five performance evaluation measures.

Results for the Category-2

In this category, we have included the fault datasets have faulty software modules between 10 and 20%. Fifteen fault datasets have been fallen into this category. The results for this category are given in Table 5.4. The observations based on Table 5.4 are summarized below.

- In terms of accuracy measure, all the used fault prediction techniques have produced values above 57% with the highest value of 90.34% for the decision stump technique. The average value is greater than 80% for all the used techniques.
- In terms of precision measure, all the used fault prediction techniques have achieved values greater than 0.42 with the highest value of 0.90 for SMO and tree-based classifiers. The average value for this performance measure is above 0.75 for all the used techniques.
- In terms of recall measure, the used fault prediction techniques have produced values above 0.58 with the highest value of 0.90 for J48 technique. The average value is above 0.84 for all the used techniques.
- In terms of F-measure measure, the used fault prediction techniques have achieved values greater than 0.51 with the highest value of 0.90 for logistic regression. The average value is above 0.78 for all the used techniques.
- In terms of AUC measure, the used fault prediction techniques have produced values greater than 0.41 with the highest value of 0.89 for the random forest technique. The average value is greater than 0.52 for all the used techniques.
- Overall, we found that tree-based classifiers have performed better compared to the other used techniques for this category for all five performance evaluation measures.

Results for the Category-3

In this category, we have included the fault datasets have faulty software modules between 20 and 30%. Five fault datasets have been fallen into this category. The results for this category are given in Table 5.5. The observations based on Table 5.5 are summarized below.

- For accuracy measure, all the used fault prediction techniques have produced values greater than 68.29% with the highest value of 84.4% for naive Bayes technique. The average value is above 71.4% for all the used techniques.
- For precision measure, all the used techniques have produced values greater than 0.64 with the highest value of 0.84 for naive Bayes technique. The average value is greater than 0.70 for all the used techniques.
- For recall measure, all the used techniques have achieved values above 0.68 with the highest value of 0.84 for the naive Bayes and logistic regression techniques. The average value is greater than 0.79 for all the used techniques.

- For F-measure measure, all the used techniques have produced values greater than 0.64 with the highest value of 0.83 for naive Bayes, logistic regression, and tree-based techniques. The average value is above 0.71 for all the used techniques.
- For AUC measure, all the used techniques have produced values above 0.50 with the highest value of 0.84 for naive Bayes technique. The average value is greater than 0.57 for all the used techniques.
- Overall, we found that naive Bayes technique have performed better compared to the other used techniques for this category for all five performance evaluation measures.

Results for the Category-4

In this category, we have included the fault datasets have more than 30% of faulty software modules. Twenty fault datasets have been fallen into this category. The results for this category are given in Table 5.6. The observations based on Table 5.6 are summarized below.

- With respect to accuracy measure, all the used techniques have produced values above 48.9% with the highest value of 99% for tree-based techniques. The average value is greater than 64% for all the used fault prediction techniques.
- With respect to precision measure, all the used techniques have produced values greater than 0.32 with the highest value of 0.99 for tree-based techniques. The average value is greater than 0.59 for all the used techniques.
- With respect to recall measure, all the techniques have achieved values above 0.49 with the highest value of 0.994 for random forest technique. The average value is greater than 0.65 for all the used techniques.
- With respect to F-measure measure, all the techniques have produced values greater than 0.40 with the highest value of 0.994 for random forest technique. The average value is greater than 0.59 for all the used techniques.
- With respect to AUC measure, all the techniques have achieved values greater than 0.42 with the highest value of 0.93 for random forest technique. The average value is above 0.54 for all the used techniques.
- Overall, we found that random forest technique have performed better compared to the other used techniques for this category for all five performance evaluation measures.

Based on the analysis of the results for all the categories, we found that when a fault dataset has lesser number of faulty software modules Bayesian-based fault prediction techniques have performed the best followed by tree-based techniques. For instance, in category-1 and category-2 Bayesian logistic regression and tree-based classifier have produced better results compared to other techniques. For the fault datasets have larger number of faulty software modules, tree-based techniques followed by naive Bayes technique performed better compared to other techniques. In addition, we found that fault prediction techniques have performed better for the used performance evaluation measures for the fault datasets have lesser faulty software modules (below 30%). Therefore, we suggest that fault prediction can be

useful for the software projects with percentage of faulty module lesser than certain threshold (in our case, it is below 30%).

5.4 Summary

Over the past few years, various researchers have proposed different approaches and have used different fault prediction techniques to build the fault prediction models. These fault prediction approaches varied in terms of complexity of the techniques, used source fault datasets, and used performance evaluation measures. This makes the comparison of these approaches difficult, as no baseline is existed to compare such approaches. In the presented work, we compared fifteen different fault prediction techniques to determine the best fault prediction technique(s). We performed the experiments for the forty-six different fault datasets collected from the PROMISE data repository and used five different performance evaluation measures to assess the results. We evaluated a number of fault prediction techniques from the literature, some of the techniques are novel in the presented study, and others are the variants of the already available techniques. Our experimental study evaluated the performance of fault prediction techniques for four different categories based on the percentage of faulty software modules in the used fault datasets. The results of the analysis showed that simple fault prediction techniques such naive Bayes, Bayesian logistic regression, and tree-based techniques outperformed other used fault prediction techniques for most of the cases. We performed the study for the fault datasets corresponding to the open-source software project to facilitate the reproduction of the experimental findings and to generalize the results. The results of the presented study can be used by the researchers and software practitioners to select the appropriate fault prediction technique to predict the fault-proneness in the early phases of the software development.

References

Athanassoulis, M., & Ailamaki, A. (2014). BF-tree: Approximate tree indexing. *Proceedings of the VLDB Endowment, 7*(14), 1881–1892.

Boetticher, G. D. (2005). Nearest neighbor sampling for better defect prediction. *ACM SIGSOFT Software Engineering Notes, 30*(4), 1–6.

Briand, L. C., & Wüst, J. (2002). Empirical studies of quality models in object-oriented systems. *Advances in Computers, 56,* 97–166.

Challagulla, V. U., Bastani, F. B., Yen, I. L., & Paul, R. A. (2005). Empirical assessment of machine learning based software defect prediction techniques. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (pp. 263–270).

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software, 81*(5), 649–660.

Elish, M. O., Al-Yafei, A. H., & Al-Mulhem, M. (2011). Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of eclipse. *Advances in Engineering Software, 42*(10), 852–859.

Gayathri, M., & Sudha, A. (2014). Software defect prediction system using multilayer perceptron neural network with data mining. *International Journal of Recent Technology and Engineering, 3*(2), 54–59.

Genkin, A., Lewis, D. D., & Madigan, D. (2007). Large-scale Bayesian logistic regression for text categorization. *Technometrics, 49*(3), 291–304.

Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (pp. 9)

Koru, A. G., & Liu, H. (2005). Building effective defect-prediction models in practice. *IEEE Software, 22*(6), 23–29.

Liu, B. T., Huang, P. J., & Zhang, G. X. (2013). The Applied Research of System Subtractive Clustering RBF Algorithm in Eddy Current Testing on the Conductive Structure Defect. *Advanced Materials Research, 712,* 2030–2034.

Ma, Y., Guo, L., & Cukic, B. (2006). A statistical framework for the prediction of fault-proneness. *Advances in machine learning application in software engineering*, pp. 237–265.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing, 27,* 504–518.

Moser, R., Pedrycz, W., & Succi, G. (2008, May). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering* (pp. 181–190).

Safavian, S. R., & Landgrebe, D. (1991). A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics, 21*(3), 660–674.

Shatnawi, R., & Li, W. (2008). The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software, 81*(11), 1868–1882.

Wang, T., & Li, W. H. (2010, December). Naive bayes software defect prediction model. In *2010 International Conference on Computational Intelligence and Software Engineering* (CiSE) (pp. 1–4).

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. US: Morgan Kaufmann.

Chapter 6
Evaluation of Techniques for the Prediction of Number of Faults

The main objective of software fault prediction is to help the detection of maximum software faults with the least possible utilization of the software development resources. The binary class classification of the software faults provides useful hints about the fault-proneness of the software modules. However, having the faulty or non-faulty information only about the software modules is not sufficient, when we need to prioritize the software testing resources. Binary class classification of software faults treats all the faulty software modules equally irrespective of their fault content. Sometimes it may possible that some of the software modules have a large number of faults. Marking them faulty and treating them equally as the software modules have lesser number of fault content can result in the insufficient use of software development resources.

The idea of the prediction of the number of faults is to predict the fault counts in each given software modules instead of labeling them as faulty or non-faulty. This type of fault prediction can be helpful to the software practitioners in narrow down the testing efforts to the software modules have large fault counts and to prioritize the software testing resources. Some of the researchers have reported the studies for the prediction of the number of faults in the given software system. These studies have used regression-based approaches and their variants to build the fault prediction models. However, different studies built fault prediction model for different fault datasets and have evaluated the results using different performance evaluation measures. The absence of common baseline to compare these fault prediction models make it difficult to select the appropriate fault prediction techniques when building fault prediction model for the prediction of number of faults. Therefore, in the present experimental study, we assess the performance of six different fault prediction techniques for fifteen software fault datasets and draw the conclusions.

6.1 Description of the Fault Prediction Techniques

A large number of researchers have used regression-based techniques and their variants to build the fault prediction models for the prediction of number of faults. In the present study, we have used six different fault prediction techniques and

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_6

have evaluated their performance of the prediction of number of faults. The used techniques are linear regression (LR), decision tree regression (DTR), multilayer perceptron (MLP), genetic programming (GP), negative binomial regression (NBR), and zero-inflated Poisson regression (ZIP). Some of the used techniques are novels in the present study such as decision tree regression and genetic programming, and other techniques were already explored by some researchers for the software fault prediction. Out of the used techniques, NBR and ZIP are the count model-based techniques, which estimate the probability of the occurrence of the certain number of faults in the given software module. The other four used techniques are based on the regression analysis, which tries to establish a relationship between input features (software metrics) and a number of faults in the given software modules.

Negative binomial regression is a generalized linear regression model in which the dependent variable is the count of the number of times a certain event occurs (Hilbe 2012; Cameron, A. C. 2013). The model follows negative binomial distribution in which the mean and variance are not equal. The variance of a negative binomial distribution is defined as the function of its mean and using an additional parameter k, which is known as dispersion parameter. Let's say that count of the dependent variable is represented by Y, then the variance of Y is defined as given in Eq. 6.1 (Greene 2011),

$$\text{var}(Y) = \mu + \mu^2/k \tag{6.1}$$

Zero-inflated Poisson regression (ZIP) is used to model a given data that have a large number of zeros. It generates a separate model for the excessive zeros independently and another model for the values greater than zero. ZIP model uses Poisson distribution and logit distribution to model the given data. It produces the possible values of dependent variable as nonnegative integers, 0, 1, 2, … and so on. The details about NBR and ZIP techniques are given in (Lambert 1992).

Linear regression (LR) uses a statistical method to model the occurrences of the dependent variable based on the given one or more independent variables (Cohen et al. 2002). The outcome of linear regression is a linear equation that shows the relation between dependent variable and independent variables, as defined in Eq. 6.2,

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n \tag{6.2}$$

where Y denotes the dependent variable (number of faults) and $X's$ denotes the independent variables. $\beta's$ are the weights associated with independent variables show the contributions of an independent variables in the occurrence of dependent variable. β_0 is the intercept and shows the value of dependent variable that model would made when values of all independent variables are zero (Strutz 2011).

Decision tree regression (DTR) is a type of regression technique that uses a decision tree to model the relationship between dependent and independent variables. The tree grows by calling the linear regression at each intermediate node. The attribute (software metric) which has the maximum reduction in the expected error value is

chosen as the root node, and this process is followed to further grow the tree (Wang and Yao 2013).

Multilayer perceptron is a feed-forward neural network that consists a series of interconnected processing elements to transform the given input values (software metrics) into the required output value (Kotsiantis 2007). During the training phase, based on the information given in the training dataset, it updates the weights associated with the processing elements to reduce the error between the desired value and the predicted value. Generally, it uses back-propagation method to update the weights associated with the processing elements and uses an activation function to generate the output value.

Genetic programming (GP) is a search-based technique that searches for the optimal solution(s) for the given computation task. GP is a specialized case of a genetic algorithm that uses a syntax tree to express the solution of the given task (Smith 1980). The intermediate nodes of the tree show the instructions to be executed, called functions, and leaves show the independent variable of the problem, called terminal. Initially, a random population of potential solutions has been generated from the solution space of the given problem. Subsequently, the initially generated population has been transformed to the next generation until an optimal solution is found. GP uses some operators such as mutation and cross-over to transform the initial population into the optimal solution (Goldberg 1989).

6.2 Performance Evaluation Measures

In the present study, we have built the fault prediction models to predict the number of faults in the given software modules. We have compared the predicted value of number of faults with the corresponding actual value of the number of faults and calculated the difference in the values. For this, we have used two error rate-based measures, average absolute error (AAE) and average relative error (ARE). The description of these performance measures has been given in Sect. 2.2, Chap. 2. Additionally, we have used prediction at *level l* measure to assess the performance of the fault prediction model. Prediction at *level l* or pred(l) measures the percentage of predicted number of faults lying in the *l*% range of the actual number of faults. It is the ratio of the number of modules having ARE value less than *l* to the total number of modules in the given software. Here, *l* is the threshold value. Value of *l* is generally kept less than 30%. Pred(l) gives the count for the number of cases, where ARE values are within *l*% of the actual value. It is defined by Eq. 6.3 (Veryard 2014),

$$\text{Pred(I)} = \text{w/n} \tag{6.3}$$

where *w* represents the number of cases, where ARE values are less than or equal to *l*. *n* is the total number of examples. *l* is the threshold value.

6.3 Evaluation of the Techniques for the Prediction of Number of Faults

6.3.1 Software Fault Datasets

We have used three different fault datasets available in the Eclipse bug data repository to build the fault prediction models and to evaluate the performance of used fault prediction techniques for the prediction of number of faults. Eclipse bug data repository contains the fault information of three successive releases of Eclipse project available at the file-level. They are Eclipse-metric-files-2.0, Eclipse-metric-files-2.1, and Eclipse-metric-files-3.0. Each fault dataset contains various source code and structural software metrics and the information about the number of faults in each software module. The software metrics are pre, NOM_sum, NSM_avg, ArrayCreation, ArrayInitializer, ArrayType, CharacterLiteral, ConditionalExpression, ContinueStatement, DoStatement, FieldAccess, Javadoc, LabeledStatement, ParenthesizedExpression, PrefixExpression, QualifiedName, ReturnStatement, SuperMethodInvocation, SwitchStatement, ThisExpression, ThrowStatement. The details of the fault datasets are given in Table 6.1. In the present experimental study, we have selected number of faults in a software module as the dependent variable. A detailed description of the used datasets is given in (D'Ambros et al. 2010).

6.3.2 Experimental Setup

The fault prediction models have been developed in the following ways:

Modeling of Count Models

Probability distribution functions are used to estimate the expected value of count models (NBR and ZIP) in terms of fault occurrences for each software module. To estimate the parameters of count models, the maximum likelihood function is used. Some of the software metrics have relatively larger values. To mitigate this issue, a square root transformation of software metrics and logarithmic transformation of

Table 6.1 Description of the used software fault datasets

Dataset	Total number of modules	Non-commented LOC	No. of faulty modules	Distribution of faults in %
Eclipse-metric-file-2.0	6730	796KLOC	976	14.50
Eclipse-metric-file-2.1	7889	987KLOC	855	10.83
Eclipse-metric-file-3.0	10594	1305KLOC	1569	14.81

Table 6.2 Values of different control parameters of genetic programming (Rathore and Kumar 2017)

Values of Control Parameters used in experimentation for Genetic Programming	Population size=200, Number of generation=1000, Termination condition = Limited to maximum number of generations, Function set ={+, −, ∗, sin, cos, /, log}, Terminal set = {x, software metrics}, Tree initialization = Ramped half-and-half, Genetic operator = Crossover, mutation, Selection method = Roulette wheel, Elitism = Replace

LOC metric has been performed. The construction of count models has been done using STATA tool.[1]

Modeling of Linear Regression, Multilayer Perceptron, and Decision Tree Regression

We have implemented linear regression and multilayer perceptron using the Weka machine learning tool. To estimate the parameters of the linear regression, we have used least square method, and rest of the parameter values of linear regression have been initialized to their default values as available in Weka. We have used back-propagation algorithm and sigmoid function to train the multiplayer perceptron. For the rest of the parameters, default values as defined in Weka are used. Weka implementation of decision tree regression (as known as M5P) has been used to build the fault prediction model. The different parameter values are initialized to their default values as available in Weka (Witten and Frank 2005).

Modeling of Genetic Programming

We have used GPLAB toolbox version 3.0[2] to implement genetic programming for building fault prediction models. GPLAB toolbox contained the implementation of genetic programming in the MATLAB programming language. The values of various parameters have been initialized as given in Table 6.2, and the rest of the parameter values are kept default as available in GPLAB toolbox. The used fitness function is defined as the square root of the difference between the actual value of the number of faults and the predicted value of the number of faults.

6.3.3 Results and Analysis

Results obtained from the experimental analysis with respect to the AAE and ARE measures have been given in Tables 6.3 and 6.4, respectively. Lower values of AAE and ARE show the better performance of fault prediction techniques.

[1] Stata: Data Analysis and Statistical Software. http://www.stata.com/.

[2] GPLAB, A Genetic Programming Toolbox for MATLAB, http://gplab.sourceforge.net/.

- With respect to the AAE measure, linear regression and decision tree regression have produced better values as compared to the other used fault prediction techniques. For most of the techniques, the AAE value is below 0.50 except negative binomial regression, where the value is relatively higher.
- With respect to the ARE measure, multilayer perceptron has produced better values as compared to the other used fault prediction techniques followed by decision tree regression and linear regression. Again, negative binomial regression produced relatively higher value for the used performance measure.

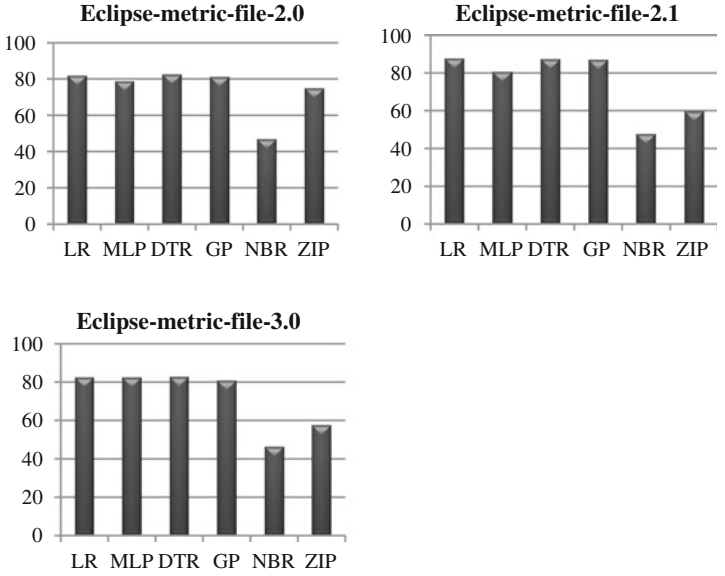
The results of the used fault prediction techniques for pred(0.3) analysis have been shown in Fig. 6.1. In each graph, x-axis shows the used fault prediction techniques and y-axis shows the value of pred(0.3) analysis of each technique. From figure, it is observed that for most of the cases, fault prediction techniques have achieved pred(0.3) value greater than 80%. Exception has been found for negative binomial regression, where the value is lower than 50% for used fault datasets. The top slots have been occupied by decision tree regression and genetic programming.

Table 6.3 AAE analysis of six fault prediction techniques for all used datasets

Datasets	Linear regression	Multilayer perceptron	Decision tree regression	Genetic programming	Negative binomial regression	Zero-inflated Poisson regression
Eclipse-metric-file-2.0	0.24	0.29	0.24	0.30	0.64	0.33
Eclipse-metric-file-2.1	0.15	0.16	0.16	0.15	0.57	0.44
Eclipse-metric-file-3.0	0.23	0.23	0.24	0.30	0.62	0.32

Table 6.4 ARE analysis of six fault prediction techniques for all used datasets

Datasets	Linear regression	Multilayer perceptron	Decision tree regression	Genetic programming	Negative binomial regression	Zero-inflated Poisson regression
Eclipse-metric-file-2.0	0.14	0.14	0.13	0.19	0.53	0.18
Eclipse-metric-file-2.1	0.08	0.07	0.08	0.10	0.52	0.37
Eclipse-metric-file-3.0	0.13	0.11	0.13	0.19	0.52	0.17



*GP=Genetic Programming, MLP=Multilayer Perceptron, LR=Linear Regression, NBR=Negative Binomial Regression, ZIP= Zero-Inflated Poisson Regression, DTR= Decision Tree Regression

Fig. 6.1 Pred(0.3) of each fault prediction technique for all datasets

Based on the analysis of the results, we found that out of the used fault prediction techniques, linear regression and decision tree regression techniques have outperformed other techniques for the prediction of number of faults. Multilayer perceptron and genetic programming are the third and fourth best fault prediction techniques for the prediction of number of faults. The performance of the used count models (negative binomial regression and zero-inflated Poisson regression) is relatively poor compared to the other fault prediction techniques.

6.4 Summary

Prediction of the number of faults in the given software modules can be more rewarding in terms of better utilization of testing resources than the binary class classification of the software faults. In the present study, we explored the use of six different fault prediction techniques for the prediction of number of faults. We evaluated the performance of the used techniques through an experimental analysis by building fault prediction models for the Eclipse project fault datasets. We assessed the results using AAE, ARE, and pred(0.3) performance evaluation measures. Results of the experimental analysis showed that linear regression and decision tree regression outperformed other used fault prediction techniques for the prediction of number of faults. Among the used fault prediction techniques, linear regression, decision tree

regression, multilayer perceptron, and genetic programming occupied the top four position, when predicting number of faults. The finding of the present study can be beneficial to the researchers and the software practitioners to select the appropriate fault prediction technique(s) when building fault prediction model for the prediction of number of faults.

References

Cameron, A. C., & Trivedi, P. K. (2013). *Regression analysis of count*. Cambridge: Cambridge University Press.

Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2002). *Applied multiple regression and correlation analysis for the behavioral sciences* (3rd ed.). London: Routledge.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In*7th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 31–41.

Goldberg, D. E. (1989). *Genetic algorithms in search optimization and machine learning* (1st ed.). Boston: Addison-Wesley Longman Publishing Co.Inc.

Greene, W. H. (2011). *Econometric analysis* (7th ed.). New York: Pearson.

Hilbe, J. M. (2012). *Negative binomial regression* (2nd ed.). California: Jet Propulsion Laboratory California Institute of Technology and Arizona State University.

Kotsiantis, S. B. (2007). Supervised machine learning: a review of classification techniques. In*Proceedings of the 2007 conference on emerging artificial intelligence applications in computer engineering: Real word AI systems with applications in e health, HCI, Information Retrieval and Pervasive Technologies*, The Netherlands, pp. 3–24.

Lambert, D. (1992). Zero-inflated poisson regression, with an application to defects in manufacturing. *Technom J, 34*(1), 1–14.

Rathore, S. S., & Kumar, S. (2017). An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Computing, 21*(24), 7417–7434.

Smith, S. F. (1980). A learning system based on genetic adaptive algorithms. Ph.D. thesis, Pittsburgh, PA, USA. AAI, No. 8112638.

Strutz, T. (2011). *Data fitting and uncertainty*. New York: Vieweg and Teubner Verlag Springer.

Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability, 62*(2), 434–443.

Witten, I. H., & Frank, E. (2005). Data mining: Practical machine learning tools and techniques. Morgan Kaufmann.

Veryard, R. (2014). *The economics of information systems and software*. Amsterdam: Elsevier Science.

Chapter 7
Conclusions

In this chapter, we conclude the work presented in the book. This main aim of this book is to provide the basic information about the software fault prediction and its various components. The matter presented in this book can be beneficial to the new researchers starting the work in the area of the software fault prediction to get the overall depiction of the various topics in this area.

Chapter 1 presented the basic definition of the software fault prediction (SFP) and the need of SFP in development of efficient and robust software system. The chapter also provided the details about the various terminologies of SFP and highlighted the benefits of SFP in software development. The chapter further listed the contributions of the presented work.

Chapter 2 presented the architecture of software fault prediction (SFP) and provided the details about the SFP process. Additionally, chapter provided the relationship between different SFP components and highlighted the significance of each component is the development of effective and efficient software fault prediction model.

Chapter 3 described various types of software fault prediction. The chapter highlighted the merits of each software fault prediction type and provided details about the state-of-the-art of each type of fault prediction. Further, for each type of software fault prediction, chapter listed various disadvantages and provided some suggestions to overcome these disadvantages.

Chapter 4 presented details of the software fault dataset. The chapter provided the details of various available software fault data repositories and highlighted issues associated with different repositories. The chapter also provided the details of issues with the software fault datasets and the redeems need to be used to build the effective and accurate software fault prediction model.

Chapter 5 presented an experimental study of the evaluation of fifteen different fault prediction techniques for the binary class classification of faults. The used forty-six software fault datasets have been divided into four different categories according to the percentage of faulty modules in the given software system, and each fault prediction technique has been evaluated for each category. The evaluation of fault prediction techniques in the chapter concluded that generally, the performance of different varies with the used datasets and with the amount of faulty modules

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8_7

in the given software system. Overall, we found that simple techniques such as naive Bayes and tree-based techniques have performed better compared to other used techniques. The chapter further concluded that amount of faulty modules in the given software system has influenced the selection of a particular technique to build the fault prediction model.

Chapter 6 presented an experimental study to assess the performance of six fault prediction techniques for the prediction of number of faults. The experimental study was performed for three Eclipse project datasets, and error rate-based performance measures were used to assess the capabilities of the used techniques. The results found that generally, the performance of techniques varies with the used software projects. However, linear regression and decision tree regression outperformed other used techniques for the prediction of number of faults.

Closing Remarks

The material presented in this book is mix of basic topics and research works published at various venues. This book mainly provides summary of information available in public domain on the topic of software fault prediction and provides easy pointers to the corresponding detailed documents. The book also includes the summary of various works published by the authors and their other team members who have worked with them previously. The book starts with the basic topics of the software fault prediction and covers till recent advancements such as the topics on the predictionof number of faults. So, it can prove to be a fruitful study material for beginners in this domain to get wide coverage of topics. It is authors' expectation that readers will find the book easy to read and understand. Various tables and figures will also help in betterment of reading experience. Efforts have been done to create better connectivity of various chapters to increase interest of readers.

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8

Index

S. Kumar and S. S. Rathore, *Software Fault Prediction*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-981-10-8715-8