Understanding the impact of experimental design Choices on machine learning classifiers in software analytics

by

Gopi Krishnan Rajbahadur

A thesis submitted to the School of Computing

in conformity with the requirements for the

Degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

September 2020

Abstract

SOFTWARE analytics is the process of systematically analyzing software engineering related data to generate actionable insights that help software practitioners make data-driven decisions. Machine learning classifiers lie at the heart of these software analytics pipelines and help automate the process of generating insights from large volumes of low-level software engineering data (e.g., static code metrics of software projects). However, the generated results from these classifiers are extremely sensitive to the various experimental design choices (e.g., choice of feature removal techniques) that one makes when constructing a software analytics pipeline. Despite that prior studies only explore the impact of a few experimental design choices on the results of classifiers and, the impact of many other experimental design choices on generated results remains unexplored. It is critical to further understand how the various experimental design choices impact the generated insights of a classifier. Such an understanding enables us to ensure the accuracy and validity of the generated insights from a classifier.

Therefore, in this PhD thesis, we further our understanding of how several previously unexplored experimental design choices impact the results that are generated by a classifier. Through several case studies on various software analytics datasets and contexts, 1) we find that the common practice of discretizing the dependent feature could be avoided in some cases (where the defective ratio of the dataset is <15%) by using regression-based classifiers. 2) In cases where the discretization of the dependent feature cannot be avoided, we propose a framework that the researchers and practitioners can use to mitigate its impact on the generated insights of a classifier. 3) We find that interchangeable use of feature importance methods should be avoided as different feature importance methods produce vastly different interpretations even on the same classifier. Based on these findings we provide several guidelines for future software analytics studies.

Acknowledgments

COMPLETING this thesis would not have been possible without the help of several amazing people; I am deeply indebted to each and every one of them.

First and foremost, I would like to thank my supervisor Prof. Ahmed E. Hassan for the constant support and guidance these four years. Ahmed always gave me the freedom, resources and the opportunity to pursue my interests. His honest critique and unending encouragement every step of the way helped me become a better and more confident researcher. Throughout my research, Ahmed provided me with many opportunities to see the world and interact with some of the best minds in software engineering research. Furthermore, the help and support he provided me during my times of personal need is something for which I will eternally be grateful. Above all, thank you Ahmed, for your willingness to take a chance on me and being in my corner. You are an amazing supervisor and a wonderful human being.

Next, I would like to sincerely thank Prof. Shaowei Wang, Prof. Cor-Paul Bezemer and Prof. Yasutaka Kamei. They were like elder brothers to me guiding me in every step of my academic journey and providing me with endless encouragement and invaluable advice. In particular, thanks Shaowei for all the casual chats and dinners with your family. Yasu, for all the coffee breaks and great conversations. Cor-Paul for all the fitness and presentation advice. Every time, I make a presentation, I always think what would Cor-Paul say. Also special thanks to Dr. Gustavo A. Oliva for all the advice, feedback and encouragement on both personal and professional fronts.

I would also like to thank Dr. Andrew Walenstein and Dr. Andrew Malton in BlackBerry for time and again providing me with great opportunities and guidance. I am truly grateful to have you guys as my mentors. I also thank Prof. Nancy Salay and Prof. Jurgen Dingel for all the interesting and invigorating conversations.

Special thanks to Dr. Satya Sai Prakash and Gayathri Manohar for being my mentors and cheer leaders. Gayathri, you have proven to be a true friend over the years. I would never have dreamed of doing a PhD without your faith in me during my formative years.

I am also extremely fortunate to continuously rub shoulders with some of the best minds in software engineering at SAIL lab. Thank you all for the engaging conversations, valuable feedback and all the lunch and coffee breaks over the years. You guys made this journey a lot more enjoyable and fruitful.

My sincere thanks to all the friends I made in Canada who made my life enjoyable here and helped me maintain my sanity even as I worked away on my research. Thank you Sudharshan, Harshith, Sieglinde, Akash, Abarnath, Sai, Dayi, Jiayuan, Dan, Aadi and Shahnaz. I am extremely grateful for all the fun times and the support you have given me over the years. Life would have been poorer without you folks being in it.

Finally, this thesis would not have been possible if not for the unconditional love and the unwavering support of my family: my mom (amma) Uma Maheswari, my little sister Bala kumari, my brother-in-law Arun Kumar and my friends Arvindh, Swami, Kasthuri and Vidhya, who are like family to me. I am glad we made it here, for this journey is every bit yours as it is mine.

Dedication

I dedicate this thesis to my amma Uma Maheswari and my sister Balakumari for their unconditional love, sacrifices and support.

I also dedicate this thesis to the loving memory of my grandfather Bala krishnan, for the love, kindness, late night cricket matches, game nights, stories and for always having my back.

Co-authorship

IN all chapters and related publications of the thesis, my contributions are: drafting the initial research idea; researching background knowledge and related work; collecting data; conducting experiments and data analysis; and writing and polishing drafts.

Earlier versions of the work in the thesis were published as listed below:

1. The impact of using regression models to build defect classifiers.

   Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei and Ahmed E. Hassan. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pages 135-145. Buenos Aries, Argentina. May, 2017.

2. Impact of Discretization Noise of the Dependent variable on Machine Learning Classifiers in Software Engineering.

   Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei and Ahmed E. Hassan. Transactions on Software Engineering (TSE), 2019, In Press.

3. The impact of feature importance methods on the interpretation of defect classifiers.
   Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo A. Oliva, Yasutaka Kamei and Ahmed E. Hassan. Transactions on Software Engineering (TSE), 2020, Under review.

The following publications are not directly related to material provided in this thesis, however they were produced in parallel to research performed in this thesis

1. A Survey of Anomaly Detection for Connected Vehicle Cybersecurity and Safety
   Gopi Krishnan Rajbahadur, Andrew J. Malton, Andrew Walenstein and Ahmed E. Hassan. In Proceedings of the Intelligent Vehicles Symposium (IV), pages 421-426. Hangzhou, China. June, 2018.

2. Pitfalls Analyzer: Quality Control for Model-Driven Data Science Pipelines
   Gopi Krishnan Rajbahadur, Gustavo A. Oliva, Ahmed E. Hassan and Juergen Dingel. In Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pages 12-22. Munich, Germany. Sept 15, 2019.

3. An Empirical Study of the Characteristics of Popular Minecraft Mods
   Daniel Lee, Gopi Krishnan Rajbahadur, Dayi Lin, Mohammed Sayagh, Cor-Paul Bezemer and Ahmed E. Hassan. Empirical Software Engineering (EMSE), 2020, In Press.

4. Revisiting the Impact of Dependency Network Metrics on Software Defect Prediction

Lina Gong, Gopi Krishnan Rajbahadur and Ahmed E. Hassan. Transaction on Software Engineering (TSE), 2020, Under review.

5. Is my transaction done yet? An empirical study of transaction processing times in Ethereum
Michael Pacheco, Gustavo A. Oliva, Gopi Krishnan Rajbahadur and Ahmed E. Hassan. Transaction on Software Engineering and Methodology (TOSEM), 2020, Under review.

Statement of Originality

I, Gopi Krishnan Rajbahadur, hereby declare that I am the sole author of this thesis. All ideas, inventions and discoveries attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

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

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

SOFTWARE analytics is the process of systematically analyzing software engineering related data to generate actionable insights that help software practitioners make data-driven decisions (Buse and Zimmermann, 2012; Menzies and Zimmermann, 2013). For instance, many software practitioners analyze the defects from earlier versions of a software system to flag potentially defect-prone modules in the current release of the system. Such analytics are routinely used by many large software corporations (Shihab et al., 2012; Zimmermann et al., 2009; Lewis et al., 2013; Shimagaki et al., 2016; Li et al., 2020; Thakkar et al., 2008) and researchers (Tian et al., 2015; Chen et al., 2018; Nam et al., 2018).

Machine learning classifiers are at the heart of modern software analytics. These classifiers typically classify data points (e.g, software modules in a current release) into target classes (e.g., defective and non-defective) based on labelled historic data of the same form (e.g., software modules from an earlier version along with labels indicating whether they were defective or not). By doing so, they help automate the process of generating insights from large volumes of available low level software engineering data (Agrawal et al., 2019, 2020; Fisher et al., 2012; Ghotra et al., 2015; Nam et al., 2018; Moser et al., 2008; Tian et al., 2015; Bavota et al., 2014; Jiang et al., 2013; Zimmermann et al., 2012; Shihab et al., 2013; Chen et al., 2018; Herzig et al., 2016).

In software analytics, classifiers serve two key purposes. First, to predict the target class of unlabelled data points. Second, to understand why the classifier assigns a particular target class to a given data point. For example, Shihab et al. (2013) built a Decision Tree classifier to predict whether a given bug fix would be re-opened or not. In addition, they also interpreted this classifier and found features of a bug fix that influence its re-opening likelihood. For instance, if it is found that bugs logged with high-severity are more likely to be re-opened, then practitioners can use this insight to ensure that these bugs and their fixes get more comprehensive reviews before they are closed.

Figure 1.1 presents an overview of the software analytics pipeline that uses a classifier to extract insights from software engineering related data (i.e., data source). As Figure 1.1 shows, first, the data that is to be analyzed for insights is mined from a relevant data source. Following which a dataset is created by extracting the dependent (i.e., class labels) and independent features (i.e., metrics that characterise each data point) from the collected data. However, when building a classifier, if pre-defined class labels are not available and only a continuous dependent feature could be observed, these values are discretized into artificial target classes. Similarly, if the independent

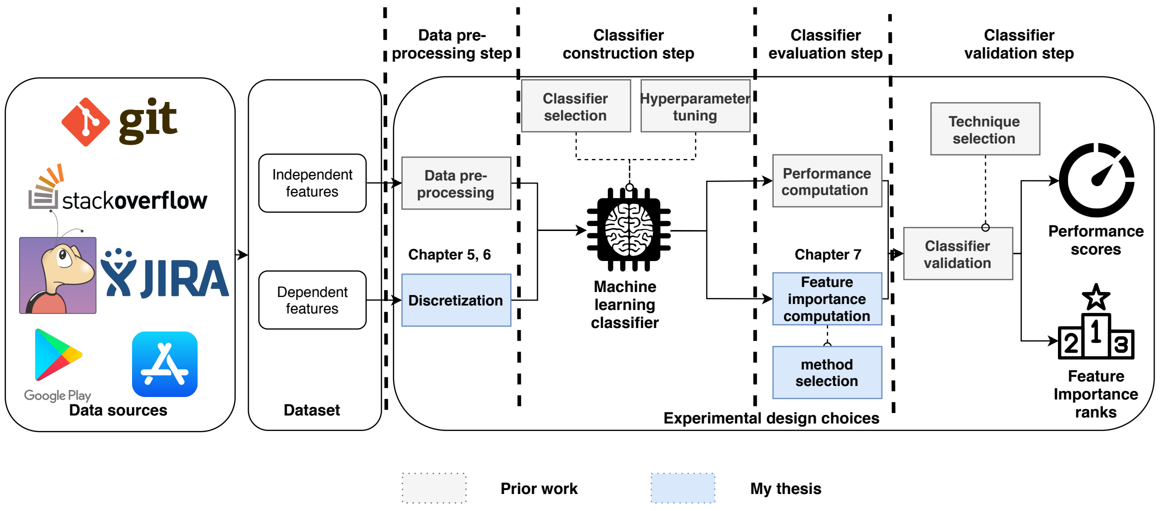


Figure 1.1: An overview of software analytics pipeline.

features exhibit any inconsistencies they are typically pre-processed (e.g., removing missing values). These independent and dependent features together are used to construct classifiers. Once a classifier is constructed, performance measures (e.g., accuracy) are used to quantify the predictive capability of the classifier. Similarly, feature importance methods are used to compute a feature's importance (i.e., feature importance ranks) in predicting the outcome. Finally, the computed results are validated using a validation method (e.g., cross-validation) to ensure the statistical robustness of any derived observations.

Several recent studies raise concerns that the computed performance measures and feature importance ranks of a classifier **are extremely dependent on the experimental design choices** (Song et al., 2010; Ghotra et al., 2015; Panichella et al., 2014; Tantithamthavorn et al., 2018a; Fu et al., 2016; Tantithamthavorn et al., 2018b) that one makes in a software analytics pipeline. For instance, choosing to use a Decision tree classifier, instead of a Random Forest classifier or not tuning the hyper parameters of a classifier could have an adverse effect (Ghotra et al., 2015; Fu et al., 2016; Tantithamthavorn et al., 2018b). In summary, constructing a software analytics pipeline involves a large number of experimental design choices and all of them could potentially impact the generated insights.

However recent studies have only explored the impact of ***few*** experimental design choices on the computed performance measures and feature importance ranks of a classifier and the impact of ***many*** other experimental design choices remain unexplored. Figure 1.1 highlights the experimental design choices whose impact on classifiers are relatively well explored as well as the ones whose impact remains unexplored. In the context of software analytics, it is extremely pivotal to further our understanding of how the various experimental design choices impact the results of software analytics studies.

Though such impact may have been explored in other domains (e.g., data mining), it is essential to explore it in the context of software analytics. Because, prior studies show, software engineering datasets differ from non-software engineering datasets (Menzies, 2019; Ray et al., 2016; Binkley et al., 2018). Therefore, findings from other fields might not generalize in the context of software analytics. For instance, Ray et al. (2016) show that software code has properties that are vastly different from natural language. Consequently, language models (e.g., n-gram models) used in other fields like data mining, when applied to software engineering data might yield spurious results (Menzies, 2019).

1.1 Thesis Statement

This thesis aims to understand the impact of a number of experimental design choices on the computed performance and feature importance ranks of a classifier and in turn, provide actionable guidelines for future software analytics studies.

> Experimental design choices impact the results of software analytics studies. However, the impact of several experimental design choices remain unexplored. Understanding how these unexplored experimental design choices impact the results of software analytics studies enables the creation of actionable guidelines for future software analytics studies.

In this thesis, we examine the impact of **discretizing the dependant feature** and the **interchangeable use of feature importance methods** on the computed performance and feature importance ranks of a classifier in software analytics. We choose these two experimental design choices in particular over other design choices for the following reasons. First, prior studies show that the discretization of the continuous dependent feature to generate artificial target classes incurs information loss (Altman and Royston, 2006; Austin and Brunner, 2004; Cohen, 1983; MacCallum et al., 2002; Royston et al., 2006; Dawson and Weiss, 2012; Rucker et al., 2015). However, the discretization of the dependent feature is still widely practiced in software analytics (Tian et al., 2015; Guo et al., 2010; Jiang et al., 2013). Therefore, we investigate the impact of the discretization of the dependent feature along the following two directions. 1) We study if the harmful discretization of the dependent feature could be avoided in the first place (atleast in some cases) by building regression-based classifiers (Chapter 4), 2)In cases where the discretization of the dependent feature cannot be avoided, we devise a framework to understand and mitigate the impact of discretizing the continuous dependent feature (Chapter 5).

Second, we know that different feature importance methods work differently. For instance the Permutation feature importance method computes feature importance ranks differently than the Gini feature importance method (Strobl et al., 2007). However, software analytics studies tend to use different feature importance methods interchangeably (Treude and Wagner, 2019; Yu et al., 2019). Such interchangeable use of feature importance methods could impact the computed feature importance ranks of a classifier and thereby affect the generated insights. Therefore, we study the impact of interchangeably using feature importance methods on the computed feature importance ranks of a classifier and provide guidelines for the future software analytics studies (Chapter 6).

1.2 Thesis Overview

In this thesis, we examine the impact of a number of experimental design choices on the generated insights of a classifier in software analytics. We do so by first providing a background on the experimental design choices that we explore in this study. Following which we survey the prior studies that study the impact of several experimental design choices on the generated insights of a classifier. We then highlight the critical gap that exists in the literature and set out to address this critical gap in the remainder of this thesis.

Below, we provide a brief summary of each chapter of our thesis.

1.2.1 Background and Motivation (Chapter 2)

We present the relevant background about the discretization of the dependent feature and feature importance methods in this Chapter. We also present the key arguments from prior studies that motivate us to explore the impact of the discretization of the dependent feature and interchangeable usage of feature importance methods on the generated insights of a classifier.

1.2.2 Literature survey (Chapter 3)

In this Chapter, we surveyed prior studies in software analytics that investigated the impact of various experimental design choices on generated insights of a classifier. We group the surveyed results based on the step of the software analytics pipeline in which the investigated experimental design choice is typically made. We grouped the results along the four key steps of the software analytics pipeline: *Data pre-processing, Classifier construction, Classifier evaluation, Classifier validation.* We find that, prior studies do not investigate the impact of the discretization of the dependent feature and the impact of interchangeably using the feature importance methods on the generated insights of a classifier. This thesis aims to fill this critical gap.

1.2.3 Avoiding the discretization of the dependent feature by using regression-based classifiers (Chapter 4)

The Discretization of the dependent feature before constructing a classifier is a common practice in software analytics. Particularly, in the filed of defect prediction where the continuous defect counts are often discretized to defective and non-defective classes before building a classifier (i.e., discretized classifiers). However, several prior studies show that such discretization of the dependent feature causes information loss that is akin to discarding a part of the data (Cohen, 1983; Altman and Royston, 2006). Such information loss could potentially impact the performance and the interpretation of constructed classifiers. We could potentially avoid such discretization of the dependent feature by using regression-models then discretizing the predicted defect counts into defective and non-defective classes (i.e, regression-based classifiers).

In this chapter, we analyze if such regression-based classifiers could be used to avoid the discretization of dependent feature. We do so by comparing the performance and interpretation regression-based classifies and discretized classifiers using six commonly used machine learning techniques and 17 defect datasets. Our findings suggest that, future studies should consider building regression-based classifiers to avoid the discretization of the dependent feature (particularly when the defective ratio of the dataset is less than 15%).

1.2.4 Mitigating the impact of discretizing the dependent feature (Chapter 5)

From Chapter 4, we find that only in some cases, the discretization of the dependent feature can be avoided. Therefore, in other cases the impact of the discretization of the dependent feature needs to be understood and mitigated.

In this chapter, we propose a framework to help researchers and practitioners systematically analyze and mitigate the impact of the discretization of the dependent feature. We further demonstrate the usefulness of our framwork on 7 software analytics datasets. We find that the discretization of the dependent feature impact different performance measures of a classifier differently. Therefore, we suggest that future studies should use our framework to estimate the exact amount of data points around the discretization threshold that one needs to discard in order to avoid the harmful effects of discretizing the dependent feature.

1.2.5 The impact of interchangeably using feature importance methods (Chapter 6)

Feature importance methods are widely and often interchangeably used by prior studies to determine the feature importance ranks from a classifier in software analytics. However, these feature importance measures compute feature importances ranks differently. Therefore, the computed feature importance ranks are likely to be different. Hence such interchangeable use of feature importance methods could result in conclusion instabilities unless the feature importance ranks computed by the different feature importance methods are similar.

To understand if such interchangeable usage of feature importance methods to compute feature importance ranks is acceptable, we conduct a case study on 18 defect datasets through six commonly used classifiers. We find that the feature importance ranks computed by different feature importance methods indeed vary significantly even among the features reported in the top-3 ranks. Therefore, we suggest that practitioners avoid the use of feature importance methods interchangeably (especially when replicating or validating a study), since the use of different feature importance methods leads to extremely different conclusions.

1.3 Thesis Contribution

In this thesis, we investigated the impact of discretizing the dependent feature and interchangeably using the feature importance methods on the generated insights of a classifier in software analytics. In doing so, we make several contributions to the field of software analytics. We highlight some of the key contributions as follows:

1. We show that the discretization of the dependent feature and in turn the associated information loss could be avoided in some cases by using regression-based classifiers (Chapter 4).

2. We provide a framework that helps estimate and mitigate the impact of the discretization of the dependent feature for any given classifier and dataset (Chapter 5). In addition, our proposed framework recommends the exact amount of data points that one needs to discard from their analysis to avoid the harmful impact of the discretization of the dependent feature.

3. We are the first study to empirically demonstrate the harmful impact of the discretization of the dependent feature in software analytics (Chapter 5).

4. We are the first study show that feature importance methods are interchangeably used and demonstrate that such interchangeable usage of feature importance could lead to conclusion instabilities in software analytics studies (Chapter 6).

CHAPTER 2

Background and Motivation

IN this chapter, we provide a brief background on the discretization of the dependent feature and feature importance methods used in software analytics. Further, we summarize the key arguments from prior studies that motivated us to analyze the impact of these two experimental design choices on the generated insights of classifiers over other experimental design choices.

2.1 Discretization Of The Dependent Feature

2.1.1 What is discretization?

Discretization is the process of turning numerical data into discrete data with finite intervals Garcia et al. (2013).

2.1.2 Discretization of the dependent feature

Typically, when class labels for data points are not available and instead only the continuous dependent feature is available, that continuous dependent feature is discretized using a threshold to generate class labels. For instance, consider example of building a classifier to predict if a given module in the current release of a software project might be defective or non-defective. To construct such a defect classifier, we collect the historic data about these modules from their past releases with class labels indicating if they were defective or non-defective in the prior releases. However, typically only the number of defects associated with each module in their past release is available. Therefore, many studies typically discretize the defect counts into "Defective" and "non-Defective" classes then use the discretized defect counts to build a classifier.

2.1.3 Discretization of the dependent feature in software analytics

Discretization of the dependent feature is a common practice in software analytics. Many software analytics studies like (Guo et al., 2010; Gay et al., 2010; Jiang et al., 2013; Schumann et al., 2009; Jalali et al., 2008; Wang et al., 2018; Tian et al., 2015; Hassan et al., 2018) discretize the dependent feature to generate class labels. These class labels are later used to train the machine learning classifiers. For instance, Tian et al. (2015) discretizes the app-ratings (a continuous dependent feature) of mobile apps in Google play store into to “high-rated” and “low-rated” apps before building a classifier. They used the constructed classifier to study what features influence the popularity of a mobile app in the Google playstore. Similarly, before constructing their classifier Wang et al. (2018) discretized the time it takes to receive an accepted answer for a question posted in StackOverflow into “fast” and “slow” classes. They use the constructed classifier to understand which features determine how fast a question posted in StackOverflow website gets an accepted answer.

Of these, some of the studies like (Wang et al., 2018; Tian et al., 2015; Hassan et al., 2018) discard the data around the discretization threshold due to its ambiguous class loyalties and use only the top and bottom x% to train their classifiers. Whereas some other studies like (Guo et al., 2010; Gay et al., 2010; Jiang et al., 2013; Schumann et al., 2009; Jalali et al., 2008) split the continuous dependent feature using various criterion and include all the data points.

2.1.4 Arguments against discretizing the dependent feature

Many researchers have actively argued against the practice of the discretization of the dependent feature. For instance, Altman and Royston (2006) and Cohen (1983) pointed out that discretization at the median of a continuous feature leads to a loss of information, and discretization at any other cut points away from the center lead to a much greater information loss. In addition, several prior studies argued against the use of any data-driven cutpoint to discretize dependent feature as it introduced noise and bias (Rucker et al., 2015; Dawson and Weiss, 2012; Royston et al., 2006). MacCallum et al. (2002). DeCoster et al. (2009) further state that discretization is rarely ever justifiable and it is almost always safer not to discretize. In summary, a majority of the research states that discretizing the dependent feature is not a safe practice and should be avoided. However, as we show in the previous Section 2.1.3, the discretization of the dependent feature is still widely practiced in software analytics.

Takeaway: The Discretization of the dependent feature is a harmful practice. Therefore, it is pivotal to devise strategies to avoid/mitigate the impact of discretization of the dependent feature on the computed insights of a classifier in software analytics.

2.2 Feature Importance Methods

2.2.1 What are feature importance ranks?

Feature importance ranks is a ranked list that lists the features of the dataset in the order of their influence on the classification.

2.2.2 What are feature importance methods?

Feature importance methods are used to determine the importance of a given feature to the classifier. These importances are then used to compute ranking of feature importances (i.e., the feature importance ranks) for a given classifier. These methods can be divided into two categories as follows:

Classifier Specific (CS) methods. A CS method makes use of a given classifier's internals to measure the degree to which each feature contributes to a classifier's predictions. For instance, prior studies use Type 1/2 ANOVA to compute the feature importance ranks from logistic regression classifiers Bird et al. (2011, 2009); Nagappan et al. (2006); Nagappan and Ball (2005); Zimmermann et al. (2007). Therefore, a CS method could only be used to compute the feature importance ranks of the classifier(s) for which it was designed.

Classifier Agnostic (CA) methods. A CA method is not reliant on the classifier's internals to measure the feature importance ranks. Therefore, CA methods could be used to compute the feature importance ranks of any classifier. CA methods typically measure the contribution of each feature towards a classifier's predictions. For instance, some CA methods like permutation feature importance method measures the contribution of each feature by effecting changes to that particular feature in the dataset and observing its impact on the outcome.

2.2.3 Usage of feature importance methods in software analytics

Both CA and CS methods have been widely used by the software analytics researchers to compute feature importance ranks. For instance, McIntosh et al. (2016) and Morales et al. (2015) construct regression models and use ANOVA (a CS method) to understand which aspects of code review impact software quality. In turn, Fan et al. (2018) use the CS methods that are associated with a random forest classifier to identify the features that distinguish merged and abandoned code changes. Similarly, various CS methods that are associated with the random forest classifier have been used to identify features that are important for identifying: who will leave a company (Bao et al., 2017), who will become a long time contributor to an open source project (Bao et al., 2019), code metrics that signal defective code (Guo et al., 2004), popularity of a mobile app (Tian et al., 2015), likelihood of an issue being listed in software release note (Abebe et al., 2016), and many other software analytics contexts. Furthermore, CS methods that are associated with logistic regression and decision trees have also been used to generate insights on similar themes (Briand et al., 1998; Cataldo et al., 2009; Calefato et al., 2019; Gay et al., 2010; Bird et al., 2011, 2009). Correspondingly, previous studies also use CA methods to interpret classifiers. For example, Tantithamthavorn et al. (2018a) use the permutation CA method to study the impact of data pre-processing on a classifier's feature importance ranks. Furthermore, Dey and Mockus (2018) use partial dependence plots (PDP) to identify why certain metrics are not important for predicting the change popularity of an npm package. whereas Mori and Uchihira (2019) use PDP to compute the feature importance of random forest classifiers.

2.2.4 Arguments against the interchangeable usage of feature importance methods

As we observe from the previous Section 2.2.3, both CS and CA methods are used interchangeably to compute insights from a classifier. For example, we observe that Bao et al. (2017) used a CS method, while Mori and Uchihira (2019) used a CA method to compute the feature importance ranks of a random forest classifier. However, both CS and CA methods work differently. Therefore such interchangeable use of feature importance methods is acceptable only if the feature importance ranks computed by these methods do not differ from each other.

Moreover, several prior studies from other fields hint that different feature importance methods generate different feature importance ranks. For instance, Grömping (2009) compared the feature importance ranks computed by CS methods associated with random forest and linear regression learners and found significant differences. Similarly, several prior studies (Calle and Urrea, 2011; Nicodemus, 2011; Strobl et al., 2008) show that different CS methods associated with random forest classifier computes different feature importance ranks. Despite such evidence, feature importance methods (both CS and CA methods) are used interchangeably to compute feature importance ranks in software analytics. We provide a detailed discussion of how frequently such interchangeable usage of feature importance methods is practiced in software analytics studies in Section 6.2 of Chapter 6.

Despite the inherent differences between the workings of CA and CS methods, they are used interchangeably to compute feature importance ranks in software analytics studies. Therefore, it is important to ascertain the impact of such interchangeable usage of feature importance methods on the computed insights of a classifier to provide guidelines for future software analytics studies.

CHAPTER 3

Literature survey

THIS chapter surveys prior work that examined the impact of various experimental design choices on the computed performance and feature importance ranks of classifiers used in software analytics. A software analytics pipeline (as shown in Figure 1.1) is typically comprised of four key steps: *Data pre-processing, Classifier construction, Classifier evaluation and Classifier validation.* Each of these steps requires one to make several experimental design choices that impact the computed performance and feature importance ranks of a classifier. Therefore, in this chapter, we group the existing literature that investigates the impact of experimental design choices into four categories along these aforementioned steps.

In this chapter, we first explain our literature selection process, then discuss the existing studies along the four aforementioned categories.

3.1 Literature Selection

Several prior studies have investigated the impact of various experimental design choices on the conclusions of the software analytics studies. Tantithamthavorn's PhD thesis (Tantithamthavorn, 2016b) presents a comprehensive survey of literature between 2007 to 2017, that investigates the impact of various experimental design choices on the insights generated by classifiers in software defect prediction. We use that as a starting point and we further search for papers published between 2017 and 2020 in major software engineering journals and conferences (please see Table 3.1) with a particular focus on studies from software defect prediction (SDP) community. We do so, as the SDP community is one of the most prominent, mature and advanced users of machine learning classifiers in the field of software engineering. Though we focus particularly on studies from the SDP community, we also present other relevant studies (not from the SDP community) that investigate the impact of various experimental design choices on the generated insights of a classifier.

To ensure that we survey all related literature, we followed the citations of each of the reviewed paper. Finally, we summarize and present the results of all the relevant papers between 2007 to 2020. These presented papers investigate the impact of various experimental design choices on the computed performance and feature importance ranks of a classifier in software analytics.

Table 3.1: Research venues used as a starting point to conduct our literature survey

Venue Type	Venue Name	Abbrevation
Journal	IEEE Transactions on Software Engineering	TSE
Journal	ACM Transactions on Software Engineering and Methodology	TOSEM
Journal	Empirical Software Engineering	EMSE
Conference	ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference	FSE/ESEC
Conference	International Conference on Software Engineering	ICSE
Conference	International Conference on Automated Software Engineering	ASE
Conference	International Conference on Software Maintenance and Evolution	ICSME
Conference	International Conference on Software Analysis, Evolution, and Reengineering	SANER
Conference	International Conference on Mining Software Repositories	MSR

3.2 Data Pre-processing Step

The data pre-processing step typically involves a variety of experimental design choices (e.g., Noise removal and Feature transformation). Prior studies indicate that all such design choices could impact the performance of a classifier.

Noise removal. Several prior studies note that noise in defect prediction datasets impact the performance of a classifier (Bachmann et al., 2010; Tantithamthavorn et al., 2015; Kim et al., 2011). Similarly, Huang et al. (2017) also show the impact of missing values in a dataset and the choice of data imputation techniques on such missing values on the performance of a classifier.

Feature selection. Many prior studies remark about the impact of the choice of feature selection technique on the performance of classifiers in software engineering (Menzies et al., 2006; Hall and Holmes, 2003; Muthukumaran et al., 2015; Xu et al., 2016;

Jiarpakdee et al., 2018; Ghotra et al., 2017; Kondo et al., 2019). More recently, Ghotra et al. (2017) and Kondo et al. (2019) suggest that for supervised classifiers correlation-based feature selection methods yield the best performing classifiers in defect prediction. For unsupervised classifiers, Kondo et al. (2019) suggest using neural-network based feature reduction techniques. Jiarpakdee et al. (2019) investigated how correlated features in software engineering datasets impact the interpretation of defect classifiers. They assert that the presence of correlated features yields spurious feature importance ranks and suggest removing the correlated features before using them in the software analytics pipeline.

Feature transformation. Several prior studies investigate how different feature transformation techniques impact the generated insights from the classifiers in software analytics (Zhang et al., 2017; Jimenez et al., 2018; Biswas et al., 2019; Peters et al., 2017; Kang et al., 2019). For instance, Zhang et al. (2017) investigate how different feature transformation techniques impact the results of cross-project defect prediction studies. Whereas, several other studies (Biswas et al., 2019; Peters et al., 2017; Kang et al., 2019) explore how different feature transformation techniques affect the performance of a classifier when the input features are comprised of natural language and software code rather than tabular data.

Class rebalancing. The impact of various class rebalancing techniques to address the class imbalance problem in software engineering datasets and in turn the impact of this problem on the computed performance and feature importance ranks of the classifiers has been extensively studied (Jing et al., 2016; Malhotra and Khanna, 2017; Song et al., 2018; Tantithamthavorn et al., 2018a; Agrawal et al., 2020; Hall et al., 2011; Wang and Yao, 2013; Pelayo and Dick, 2012; Peters et al., 2013). All of these prior studies agree that class rebalancing improves the performance of the classifiers, though different studies champion different class rebalancing strategies. However, recent studies by both Tantithamthavorn et al. (2018a) and Agrawal et al. (2020) agree that hyperparameter tuned SMOTE class rebalancing strategy yields the most optimal performance gains. In addition, Turhan (2012) and Tantithamthavorn et al. (2018a) report that rebalancing the dataset changes the computed feature importance ranks of a classifier.

Discretization. Several studies in software engineering discretize the independent features as part of data pre-processing before constructing classifiers (Menzies et al., 2011; Ma et al., 2012). For instance, Jiang et al. (2008) find that while the discretization of independent features improves the performance of some classifiers, it does not universally benefit all classifiers (Jiang et al., 2008). Nam and Kim (2015) use a median based discretization of independent features to assign class labels for un-labeled data points to train a cross-project defect prediction model. Though these aforementioned studies analyze the impact of discretizing the independent features on the results generated by a classifier none of the prior studies explore the impact of the discretization of the dependent feature on the computed performance and feature importance ranks of classifiers in software analytics.

3.3 Classifier Construction Step

The classifier construction step typically entails two key experimental design choices: Classifier selection and hyperparameter tuning

Classifier selection. A remarkable number of studies investigate impact of choosing different machine learning classifiers and its impact on the results of the software analytics studies (Ghotra et al., 2015; Shepperd et al., 2014; D'Ambros et al., 2010; Arisholm et al., 2007; Seiffert et al., 2009; Song et al., 2010; Agrawal et al., 2020, 2019; Chen et al., 2018). All of these studies universally agree that the choice of classifier impact the computed performance and feature importance ranks of a classifier. In contrast, Lessmann et al. (2008) argue that the choice of classifier does not matter and all classifiers yield similar performance scores. However, Ghotra et al. (2015) later show that is not the case. More recently, Chen et al. (2018) and Agrawal et al. (2019) compare the performance scores obtained by several latest machine learning classifiers and find that Fast Frugal Trees yield the best performance for several software analytics tasks.

Hyperparameter Tuning. Tantithamthavorn et al. (2018b) point out that hyperparameter tuning significantly impact the computed performance and feature importance ranks of a classifier. They used a random search technique to hyperparameter tune their classifier. Since then, the choice of hyperparameter technique and its subsequent impact on the performance of the classifiers has generated considerable interest in software analytics (Agrawal et al., 2019, 2020; Fu et al., 2016). Fu et al. (2016) show that the differential evolution technique for hyperparameter tuning generated better performing classifiers over standard gird search. Later, Tu and Nair (2018) show that there was no one hyperparameter tuning technique that always performs well. However recently, Agrawal et al. (2019) show that a technique called DODGE finds the best hyperparameters for classifiers in software analytics.

3.4 Classifier Evaluation Step

Performance computation. Both threshold-dependent performance measures like Accuracy, Precision, Recall, F-Measure and threshold-independent performance measures like AUC, Brier score are widely used to compute the performance of classifiers in software analytics (Song et al., 2010; Seiffert et al., 2009; Arisholm et al., 2007; Ghotra et al., 2015). However, Lessmann et al. (2008) argued that threshold-dependent performance measures are unreliable as identifying the right threshold to compute the threshold-dependent performance measures requires knowledge of class and cost distribution of a given dataset. Furthermore, they argue that such information is not typically available for software analytics datasets and using a default threshold or fixed threshold biases the computed performance measures. In addition, several other studies (Agrawal et al., 2019; Menzies et al., 2007a) also argue against the use of threshold-dependent performance measures to estimate the performance of classifiers. Therefore, usage of threshold-independent performance measures is typically recommended to evaluate the performance of classifiers in software analytics.

Feature importance ranks computation. A recent study by Jiarpakdee et al. (2020) assert that the computed feature importance ranks varies when using instance level feature importance methods. To the best of our knowledge this is the only study that explores the impact of feature importance methods on the computed feature importance ranks. However, this study only explores that impact on a limited number of instance level feature importance methods. Therefore, much of the impact of using different feature importance methods interchangeably on the computed feature importance ranks of a classifier remains largely unexplored.

3.5 Classifier Validation Step

The computed performance measures and feature importance ranks of a classifier may not be statistically robust as they typically produce estimates that are unrealistic. Therefore, classifier validation methods like k-fold validation are typically used to validate the computed performance and feature importance ranks of a classifier (Lessmann et al., 2008; Tu and Nair, 2018; Jiarpakdee et al., 2020; Ghotra et al., 2015). However the choice of classifier validation method can severely impact the conclusions obtained from a classifier (Krstajic et al., 2014; Jiang et al., 2009; Mende, 2010). Recently, Tantithamthavorn et al. (2017) perform a comprehensive large scale comparison of different classifier validation methods and assert that out-of-sample bootstrap method provides the most stable performance estimates of a classifier.

Takeaway: Prior studies highlight the impact of several experimental design choices on the computed performance and feature importance ranks of a classifier in software analytics. However, there exists a limited understanding of how the discretization of the dependent feature and the choice of feature importance method impact the computed performance and feature importance ranks of a classifier. Future research needs to address this critical gap. Therefore, in this thesis we explore the impact of the aforementioned experimental design choices on the computed performance and feature importance ranks of a classifier in software analytics.

CHAPTER 4

Avoiding the Discretization of the Dependent Feature by Using Regression-based Classifiers

Classifiers that identify the defect-prone modules (i.e., Defect classifiers) are extensively used in software analytics. When constructing these classifeirs, it is common practice to discretize the continuous defect counts into defective and non-defective classes and use these two classes as a dependent feature when building defect classifiers (discretized defect classifiers). However, this discretization of continuous defect counts leads to information loss that might affect the performance and interpretation of defect classifiers. We could avoid such discretization of the dependent feature by using regression models and then discretizing the predicted defect counts into defective and non-defective classes. In this chapter, we compare the performance and interpretation of defect classifiers that are built using both approaches (i.e., discretized classifiers and regression-based classifiers) across six commonly used machine learning techniques and 17 datasets. We find that, in contrast to common practice, building a defect classifier using discretized defect counts (i.e., discretized classifiers) does not always lead to better performance. Hence we suggest that future defect classification studies should consider building regression-based classifiers and avoid discretizing the dependent feature.

An earlier version of this chapter is published in Proceedings of the 14th International Conference on Mining Software Repositories (MSR), 2017. (Rajbahadur et al., 2017)

4.1 Introduction

Classifiers that identify the defect-prone modules (i.e., Defect classifiers) are extensively used in software analytics (Shihab et al., 2012; Zimmermann et al., 2009; Lewis et al., 2013; Chen et al., 2018; Nam et al., 2018). This is because finding and fixing defects consume a significant amount of the total budget of a software project. These costs can be reduced significantly if the defects are identified and fixed early on (Arar and Ayan, 2015; Ceylan et al., 2006; Fagan, 1999; Moser et al., 2008; Mullen and Gokhale, 2005; Shull et al., 2002).

Defect classifiers assist in software quality assurance efforts and in prioritizing process improvement efforts. In particular, defect classifiers can identify defect-prone modules (Hassan, 2009; Kim et al., 2007; Wang and Yao, 2013; Zimmermann et al., 2007), in turn helping quality assurance teams allocate their limited resources to these modules (e.g., packages, files, or classes). Moreover, the trained defect classifiers can be used to understand the impact of the various features (e.g., process or product metrics) on the defect-proneness of a module, in turn helping practitioners (through process improvement efforts) avoid pitfalls that have led to defective modules in the past.

The most common approach to build a classifier is through the discretization of the continuous defect counts into "defective" and "Non-defective" classes and using these classes as a target feature (i.e., *discretized defect classifiers*) (Cataldo et al., 2009; Mockus, 2010; Lessmann et al., 2008). However, the discretization of continuous features (i.e., defect counts in this case) into two classes often leads to a significant loss of information (Altman and Royston, 2006; Cohen, 1983; Royston et al., 2006) and introduces undesired false positives or false negatives (Austin and Brunner, 2004).

To avoid the information loss because of discretization, one possible solution is to perform classification via regression (Hou et al., 2013; Singh et al., 2006; Xiang et al., 2010). Classification via regression first builds a regression model using the non-discretized defect counts then uses the predicted defect counts to identify the presence or absence of defect (i.e., *regression-based defect classifiers*). However, it is not clear which classifier building approach leads to better performing defect classifiers. It is also not clear whether these two approaches would produce classifiers which are influenced by different set of features (e.g., product versus process metrics). If the regression-based classifiers produce better performing classifiers than discretized classifiers, then the discretization of the dependent features could be avoided.

In this chapter, we examine the use of regression models to build defect classifiers in terms of performance (i.e., Area Under the receiver operator characteristic Curve (AUC)) and model interpretation (i.e., influential features that impact the defect-proneness of a module). We do so to study if the discretization of the dependent feature could be avoided. We conduct our study on six commonly used learners (i.e., linear/logistic regression, random forest, KNN, SVM, CART, and neural networks) and using 17 Tera-PROMISE defect datasets (Sayyad Shirabad and Menzies, 2005). We conduct our study through the following research questions:

- **RQ1. How well do regression-based classifiers perform?**

  In contrast to current practices in our field, building classifiers using discretized defect counts does not always lead to better performance. Regression-based classifiers outperform discretized classifiers when the defective ratio of the modeled dataset is low ($< 15\%$) and the pattern reverses when the defective ratio is high ($> 35\%$).

- **RQ2. Are discretized and regression-based classifiers influenced by the same set of features?**
  The most influential features (i.e., Rank 1 features) do not vary significantly between both approaches for building a defect classifier (when using a random forest based learner). However, we note significant variances for features at lower ranks.

Thus we suggest that future defect prediction and in turn software analytics studies should consider building regression-based classifiers (in particular when the defective ratio of the modelled dataset is low, in other words, when the class imbalance of the modelled dataset is high). Moreover, we suggest that both approaches for building classifiers should be explored, so that 1) Discretiztion of the dependent feature could be avoided in certain cases 2) the best-performing classifier can be used when determining the most influential features.

4.2 Experiment Setup

This section describes the data collection, and gives an overview of our study approach.

4.2.1 Data collection

We use the data from Tera-PROMISE Repository (Sayyad Shirabad and Menzies, 2005). Tera-PROMISE contains 101 software projects data, and the types of these projects are diverse. Using data from Tera-PROMISE helps us draw more general observations across different datasets. We select datasets based on following two criteria which are similar to a prior study by Tantithamthavorn et al. (2018b):

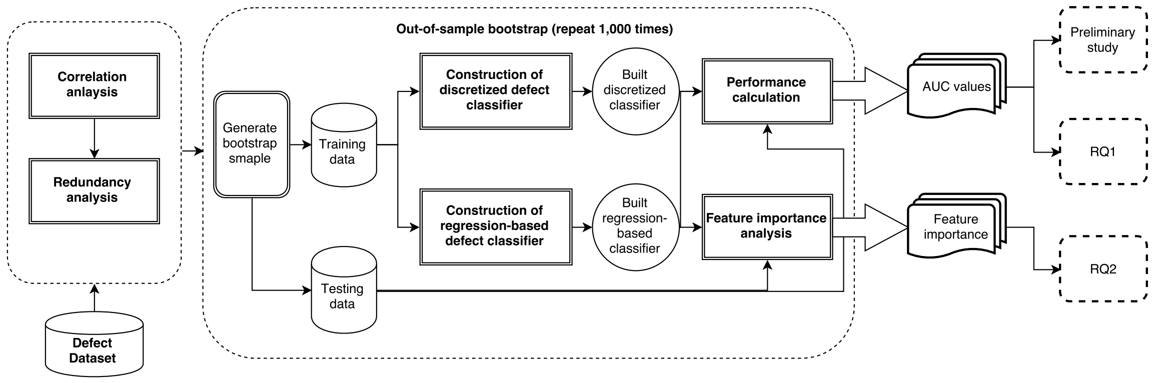

Correlation anlaysis
Redundancy analysis
Defect Dataset
Out-of-sample bootstrap (repeat 1,000 times)
Generate bootstrap smaple
Training data
Testing data
Construction of discretized defect classifier
Construction of regression-based defect classifier
Built discretized classifier
Built regression-based classifier
Performance calculation
Feature importance analysis
AUC values
Feature importance
Preliminary study
RQ1
RQ2

Figure 4.1: An overview of our study approach.

Table 4.1: An overview of the datasets that we use to study whether the discretization of the dependent feature could be avoided by using regression-based classifiers.

Project	DR(%)	#Files	#Features	#MACRA	EPV
Eclipse-2.0	14.5	6,729	32	12	30
Eclipse-2.1	10.8	7,888	32	12	30
Eclipse-3.0	14.8	10,593	32	12	49
Camel-1.2	35.5	608	20	12	11
Mylyn	13.2	1,862	15	8	16
PDE	14.0	1,497	15	9	14
Prop-1	14.8	18,471	20	15	137
Prop-2	10.6	23,014	20	14	122
Prop-3	11.5	10,274	20	15	59
Prop-4	9.6	8,718	20	15	42
Prop-5	15.3	8,516	20	14	65
Xalan-2.5	48.2	803	20	14	19
Xalan-2.6	46.4	885	20	13	21
Lucene-2.4	59.7	340	20	13	10
Poi-2.5	64.4	385	20	12	12
Poi-3.0	63.6	442	20	13	14
Xerces-1.4	74.3	588	20	11	22

DR - Defective Ratio; MACRA - Features After Correlation and Redundancy Analysis

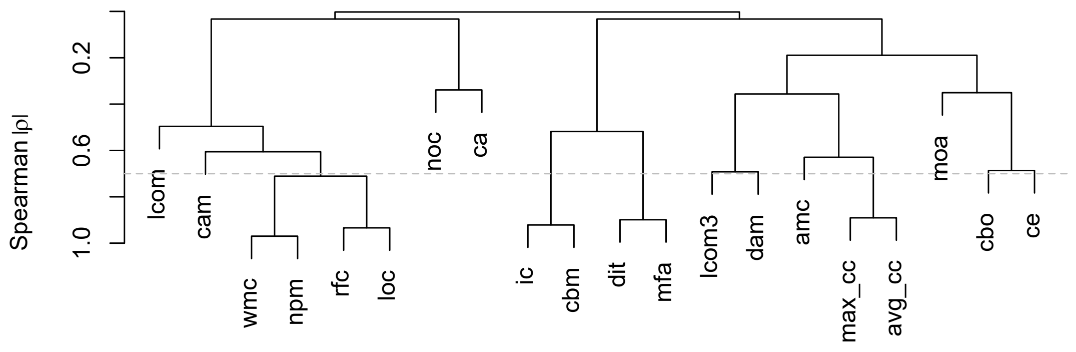


Figure 4.2: Clustered features after correlation analysis for Poi-3.0.

Criterion 1: Remove datasets with an EPV that is larger than 10. Events Per Variable (EPV) is defined as the ratio of the frequency of the least occurring class in the outcome feature to the number of features that are involved in training of a classifier. Prior studies show that the EPV value has a significant influence on the performance of defect classifiers (Peduzzi et al., 1996; Tantithamthavorn et al., 2017). In particular, defect classifiers trained with datasets with a low EPV value yield unstable results (Tantithamthavorn et al., 2017, 2018b). To ensure the stability of our results, we select datasets with an EPV value that is larger than 10 (Peduzzi et al., 1996). We calculate the EPV for our datasets using the steps provided by Tantithamthavorn et al. (2017).

Criterion 2: Remove datasets that have more than 80% defective modules. We choose datasets that have less than 80% defective modules, because it is highly unlikely for any software project to have modules with defects that much more than clean modules.

Among the 101 Tera-PROMISE datasets, we excluded 78 datasets since they had an EPV value that is less than 10. To satisfy criterion 2, we eliminate the Xalan-2.7 project and end up with 22 datasets that satisfy our criteria. We had to eliminate another 5 datasets as they did not have the actual defect counts for each module (they only had the module class, i.e., defective or not defective). We end up with 17 datasets for our analysis. Table 4.1 shows the selected datasets for our study along with basic characteristics about each dataset.

4.2.2 Overall approach

An overview approach of our study is presented in Figure 4.1. First, we perform correlation analysis and redundancy analysis. Then, we build two defect classifiers, one using the non-discretized defect counts and the other using the discretized defect count (i.e., “Defective” or “non-Defective”), respectively. After the classifiers are built, we calculate their performance using the *Area Under the receiver operator characteristic Curve (AUC)* and compute the feature importance for each classifier. We repeat this process 1,000 times using out-of-sample bootstrap validation to ensure that our drawn conclusions are statistically robust as suggested by Tantithamthavorn et al. (2017). In each iteration of the bootstrap, we compute the AUC values and feature importance for the discretized and regression-based classifiers. We use the computed AUC and feature importances to conduct our preliminary study in Section 4.3 and answer our research questions in Section 4.4.

The individual steps of our approach are explained in detail below.

4.2.3 Correlation analysis and redundancy analysis

Correlation analysis: To avoid multicollinearity problems in our classifiers, we perform a correlation analysis to remove highly correlated features. We use a feature clustering analysis technique to construct a hierarchical overview of the Spearman correlations among features. For sub-hierarchies of features with correlations larger than 0.7, we select only one feature from the sub-hierarchy for inclusion into our classifiers. When selecting the feature for inclusion, we select the feature that is simplest to interpret and compute. We use the **varclus** function from the **Hmisc** R package in this Chapter. For example, Figure 4.2 shows the hierachical clustering of the features of the Poi-3.0 project. We observe that the features “wmc”, “npm”, “nfc” and “loc” have correlation values larger than 0.7. We choose “loc” for inclusion in our classifier as it is relatively easy to compute and explain. We repeat a similar process for other correlated features.

Redundancy analysis Correlation analysis handles multicollinearity, but it does not remove redundant features, which are features that do not add additional information with respect to other features (Yu and Liu, 2004). The presence of these features distorts the relationship between features and the target feature. Hence, it is important to remove redundant features prior to classifier construction. We use the **redun** function from **rms** R package to remove redundant features. The function drops features iteratively until either no previously constructed model of features achieved an R^2 above a chosen cutoff threshold (0.9 in our case). Table 4.1 shows the number of remaining features in our datasets after employing feature selection on each dataset.

4.2.4 Classifier construction

In our experiments, we use two approaches to build defect classifiers to predict whether a module has defects or not: a traditional defect classifier that is built with discretized defect counts (referred as a *discretized defect classifier*) and a classifier that is built using a regression model that is built with non-discretized defect counts (referred as a *regression-based defect classifier*).

Construction of discretized defect classifier. The continuous defect counts are discretized to defect classes "Defective" and "Non-Defective" based on the condition: If a module's defect count is greater than or equal to 1, it is classified as "Defective"; otherwise it is classified as "non-Defective". During the training phase, the defect classes are then treated as the target feature and are feed to a classification technique (e.g., random forest) along with the collected features to build the discretized classifier. During the testing phase, the trained discretized classifier is tested on unseen testing data to compute the performance (i.e., AUC) of the classifier and its most influential features.

Construction of regression-based defect classifier. Different from the construction of a discretized classifier, during the training phase, we use the non-discretized defect counts (i.e., the actual values) to build a regression model then the predicted counts of the model are transformed into two classes ("Defective" and "non-Defective") based on a threshold which is not necessary to be 1. Then the model performance and feature importance are computed.

4.2.5 Performance calculation

We use the *Area Under the receiver operator characteristic Curve (AUC)* as the measure of the performance when comparing between the discretized and regression-based

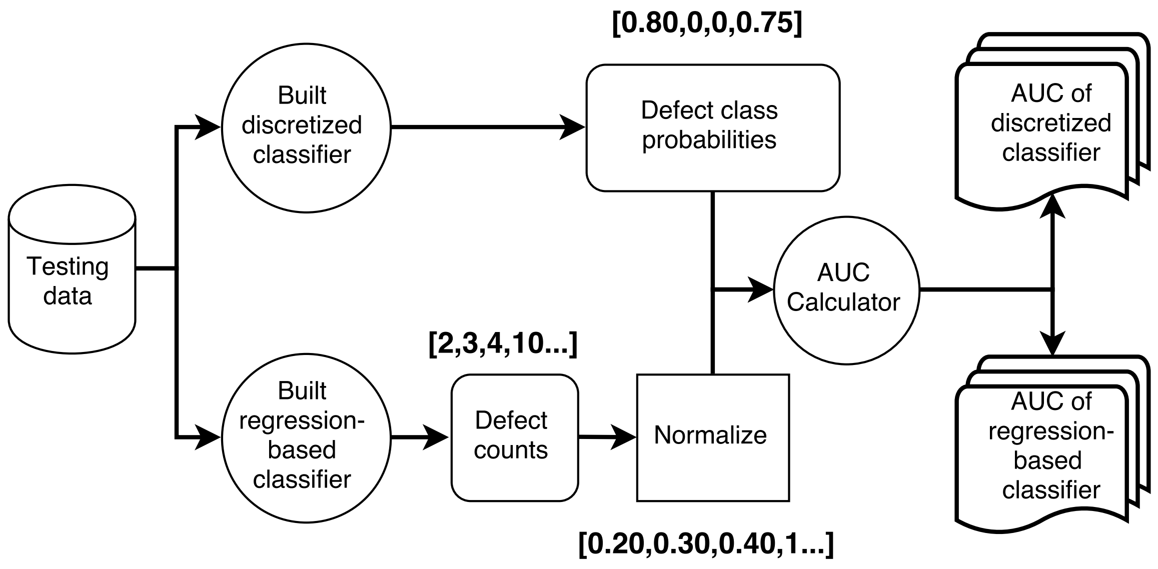

[0.80,0,0,0.75]
Built discretized classifier
Defect class probabilities
AUC of discretized classifier
Testing data
AUC Calculator
[2,3,4,10...]
Built regression-based classifier
Defect counts
Normalize
AUC of regression-based classifier
[0.20,0.30,0.40,1...]

Figure 4.3: An overview of performance evaluation.

defect classifiers (Lessmann et al., 2008). AUC is computed by plotting the ROC curve, which maps the relation between True Positive Rate (TPR) and False Positive Rate (FPR) at all thresholds. We choose AUC because of following reasons:

1. AUC measures the performance across all the thresholds. When calculating the AUC of both discretized and regression-based defect classifiers, a threshold needs to be set up to classify the outcome as “Defective” if the predicted value is above that threshold and “non-Defective” otherwise. It is challenging to decide this threshold. To avoid the problem of threshold setting, we select AUC since AUC measures the performance on all the thresholds (i.e., from 0 to 1) and precludes our analysis from the peculiarities of setting up thresholds.

2. The AUC is insensitive to cost and class distributions (Lessmann et al., 2008) so that the imbalance inherent to the software datasets is automatically accounted for and provides a score that is objective. An AUC score close to 1 means the

classifier's performance is very high and a classifier that has an AUC score of 0.5 is no better than random guessing.

We now briefly discuss the calculation of the AUC for the regression-based defect classifier. Figure 6.1 depicts how the performance calculation component of Figure 4.1 works. The AUC calculation is accomplished by normalizing the predicted defect counts to fall within the 0 and 1 range to mimic the class probabilities generated by a discretized defect classifier. We use the normalized score along with the actual classes to compute the AUC. In this way, we are comparing the discretized defect and regression-based defect classifiers on a common ground. This also ensures that the regression-based defect classifier is tested for its classification prowess rather than its regression performance.

4.2.6 Feature importance analysis

We use permutation feature importance method (Altmann et al., 2010) (i.e., a CA method) as a means of measuring the importance of a given feature. Understanding the importance of each feature on a classifier, helps practitioners in their process improvement activities for avoiding future defects. We use permutation importance method in lieu of the built-in feature importance method of each classification and regression technique (i.e., the CS method associated with each technique) since permutaton importance method gives us a way of conducting feature importance ranks estimation in an unbiased setting.

The permutation importance method works by randomly permuting the values of one feature at a time so that the original relationship between the feature and the target feature is disturbed. Then this permuted feature along with the other non-permuted features are used to classify the testing data, and performance of the classifier is computed. If the computed performance of the classifier that is built using the permuted feature decreases significantly from the classifier that is built with non-permuted feature, then such performance decrease signifies the importance of this feature. This process is repeated for each of the features and they are ranked based on the degree of decrease in their performance once they are permuted.

4.2.7 Out-of-sample bootstrap

In order to ensure that the conclusions that we draw about our classifiers are robust, we use the out-of-sample bootstrap validation, which has been shown to yield the best balance between the bias and variance in a recent study (Tantithamthavorn et al., 2017). The out-of-sample bootstrap is conducted along the following steps:

1. A bootstrap sample of size N is randomly drawn with replacement from the original dataset, which is also of size N.

2. Discretized and regression-based defect classifiers are trained using the bootstrap sample (i.e., training data). On average, 36.8% of the data points will not appear in the bootstrap sample, since it is drawn with replacement (Tantithamthavorn et al., 2017).

3. We calculate the AUC value and feature importance for each classifier on unseen testing data that are data points that do not appear in the bootstrap sample.

The out-of-sample bootstrap process is repeated 1,000 times. After the bootstrap, 1,000 AUC values and 1,000 lists of feature importance are generated. We perform further analysis on these AUC values and feature importance to answer our research questions.

4.3 Preliminary Study

Prior studies have compared the performance of different machine learning techniques when building discretized defect classifiers (Guo et al., 2004; Lessmann et al., 2008). However, there is no knowledge about the performance of regression-based classifiers. Hence we focus our preliminary study to examine the performance of regression-based classifiers. Our goals are two folds: 1) To replicate prior findings in order to understand whether prior findings for discretized defect classifiers would hold for regression-based classifiers, 2) To help focus our analysis in the following sections on the top performing regression-based classifiers.

Approach: We choose one representative technique from each widely used machine learning technique families that are listed by Lessmann et al. (2008) by satisfying following criteria:

1. A classifier could be built based on both discretized and non-discretized defect counts.
2. One is widely used in prior defect prediction studies.

Table 4.2 shows the techniques chosen for our analysis. All of these techniques are used at their default settings. While a recent study by Tantithamthavorn et al.

Table 4.2: Techniques that are selected from each machine learning family.

Family	**Classification technique**	**Regression technique**
Statistical	Logistic regression (Log-Reg)	Linear regression (Lin-Reg)
Nearest neighbor	K-NN classification (KNN-C)	K-NN regression (KNN-R)
Support-Vector machines	SVM classifier (SVM-C)	SVM regression (SVM-R)
Neural networks	Neural Networks classifier (NN-C)	Neural Networks regression (NN-R)
Decision tree	Classification tree (CT)	Regression tree (RT)
Random forest	Random forest classification (RF-C)	Random forest regression (RF-R)

(2018b) shows that parameter optimization could improve the performance of some techniques (e.g., C5.0), the techniques that are used in our study are not significantly sensitive to parameter optimization. Thus, we use the default settings here.

We use the overall experiment setup that is outlined in Section 4.2 with all the six families of chosen techniques. We start with data collection, then correlation and redundancy analysis on the datasets. We then build the discretized and regression-based defect classifiers using the six families of chosen techniques. The generated classifiers are validated and the performance of each classier is evaluated using the AUC that is obtained from the out-of-sample bootstrap as explained in the Section 4.2.

Once the AUC values are computed, we use a Scott-Knott Effect size clustering (SK-ESD) (Tantithamthavorn et al., 2017) to rank the techniques based on the AUC values. SK-ESD uses the effect size as computed by Cohen's Δ (Cohen, 1988) to merge statistically similar groups into the same rank. These ranks are obtained for both discretized and regression based defect classifiers on all 17 datasets for each of the six families of

Table 4.3: Average ranks of various discretized and regression-based classification techniques

Project	Lin-Reg		Log-Reg		RF-R		RF-C		NN-R		NN-C		RT		CT		SVM-R		SVM-C		KNN-R		KNN-C	
	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A
Eclipse-2.0	2	0.80	2	0.83	1	0.84	1	0.84	6	0.63	3	0.71	5	0.71	6	0.71	4	0.75	5	0.76	3	0.79	4	0.78
Eclipse-2.1	1	0.79	1	0.79	2	0.78	2	0.77	3	0.73	4	0.67	3	0.73	4	0.67	5	0.62	5	0.62	4	0.72	3	0.71
Eclipse-3.0	1	0.80	1	0.80	1	0.80	1	0.80	5	0.62	2	0.67	3	0.73	4	0.67	4	0.71	5	0.71	2	0.75	3	0.74
Camel-1.2	3	0.60	2	0.61	2	0.63	1	0.64	5	0.55	4	0.56	3	0.55	4	0.56	1	0.64	5	0.64	2	0.59	3	0.58
Mylyn	3	0.68	1	0.70	1	0.74	2	0.68	6	0.54	2	0.60	5	0.62	4	0.60	4	0.65	3	0.65	2	0.70	1	0.70
PDE	2	0.69	1	0.72	1	0.71	2	0.71	5	0.56	5	0.64	4	0.64	4	0.64	4	0.64	6	0.65	3	0.66	3	0.65
Prop-1	4	0.71	2	0.74	1	0.79	1	0.77	5	0.62	4	0.61	5	0.62	4	0.61	3	0.73	3	0.72	2	0.76	1	0.76
Prop-2	4	0.66	3	0.71	1	0.84	1	0.81	5	0.57	5	0.50	5	0.57	5	0.50	3	0.68	4	0.68	2	0.76	2	0.75
Prop-3	3	0.68	2	0.71	1	0.72	4	0.69	6	0.54	1	0.50	5	0.58	6	0.50	4	0.64	5	0.64	2	0.70	3	0.70
Prop-4	2	0.73	1	0.75	1	0.77	2	0.72	5	0.63	5	0.58	5	0.63	5	0.58	4	0.67	4	0.66	3	0.71	3	0.71
Prop-5	3	0.66	2	0.71	1	0.73	3	0.70	5	0.58	1	0.51	4	0.63	6	0.51	3	0.66	5	0.66	2	0.69	4	0.69
Xalan-2.5	5	0.63	3	0.65	1	0.75	1	0.76	6	0.62	3	0.66	4	0.64	3	0.66	2	0.72	4	0.72	3	0.70	2	0.70
Xalan-2.6	3	0.78	2	0.80	1	0.82	1	0.84	5	0.66	3	0.77	4	0.77	3	0.77	2	0.80	4	0.81	3	0.79	2	0.81
Lucene-2.4	2	0.75	2	0.74	1	0.77	1	0.77	5	0.50	5	0.67	4	0.67	4	0.67	3	0.72	6	0.73	3	0.72	3	0.71
Poi-2.5	5	0.76	3	0.81	2	0.85	1	0.89	6	0.50	3	0.80	4	0.79	3	0.80	1	0.86	4	0.86	3	0.82	2	0.84
Poi-3.0	5	0.75	2	0.84	1	0.82	1	0.89	6	0.50	4	0.82	4	0.75	3	0.82	3	0.80	5	0.85	2	0.81	2	0.85
Xerces-1.4	5	0.86	1	0.94	1	0.91	1	0.96	6	0.50	4	0.91	5	0.86	3	0.91	3	0.90	5	0.91	2	0.90	2	0.92
Avg.	3.12	0.73	1.82	0.76	**1.17**	**0.78**	**1.52**	**0.78**	5.29	0.58	3.41	0.66	4.23	0.68	4.18	0.66	3.11	0.72	4.59	0.72	2.52	0.74	2.52	0.74

R - Rank; A - AUC

chosen techniques. The average ranks for each technique over 17 datasets are calculated and the technique with the lowest rank across both the approaches is considered as the best technique.

Results: Random forest technique has the best performance across both discretized and regression-based defect classifiers. The ranks for each technique are listed in Table 4.3. Random forest techniques have the best performance across both discretized (i.e., average rank is 1.17) and regression-based defect classifiers (i.e., average rank is 1.52). This is compatible with the findings of prior studies that suggest that a random forest classifier outperforms other learners for building discretized defect classifiers (Ghotra et al., 2015; Lessmann et al., 2008). More specifically, the random forest family performs the best in 14 out of 17 studied datasets for regression-based defect classification and in 11 out of 17 studied dataset for discretized defect classification.

The next best technique is K-Nearest Neighbor family which has an average rank of 2.52 for both the regression-based and discretized defect classifier, followed by the Statical classifiers (i.e., Linear and Logistic regressions) which have an average rank of 3.12 for discretized defect classifier and 1.82 for the regression-based defect classifier.

We also report the average AUC for each technique on the studied datasets in Table 4.3. We find that random forest family has an average AUC of 0.78 across both discretized and regression-based defect classifiers, which is the highest among all considered techniques. The average AUC for linear and logistic regression is 0.73 for regression-based defect classifiers and 0.76 for discretized defect classifiers, which is followed by the KNN family at 0.74 for both types of classifiers.

Random forest techniques perform the best across both discretized and regression-based defect classifiers. Hence, we primarily focus our analysis in the following sections on random forest techniques.

4.4 Case Study Results

4.4.1 RQ1. How well do regression-based classifiers perform?

Motivation: Both regression-based and discretized classifiers can identify defect-prone modules. Prior research in software engineering has primarily used discretized classifiers for identifying defect-prone modules. However the discretization that is performed by discretized classifiers leads to information loss. Hence, regression-based classifiers might be a viable option that could help us avoid the discretization of the dependent feature. In this research question, we investigate the performance of regression-based defect classifiers.

Approach: To answer this research question, we construct discretized and regression-based classifiers on the 17 studied datasets. Based on our observations in Section 4.3, we use random forest classifiers since they outperform other types of machine learning techniques.

We then compare the performance of the discretized random forest classifiers (DRFC) and regression-based random forest classifiers (RBRFC) using the AUC values. To measure the differences between the two types of classifiers, we use a Wilcoxon signed-rank test (Wilcoxon, 1945) since it does not need the data to follow a normal distribution and it tests paired results. To quantify the magnitude of the performance differences between DRFC and RBRFC, we use Cohen's d effect size test (Cohen, 1988).

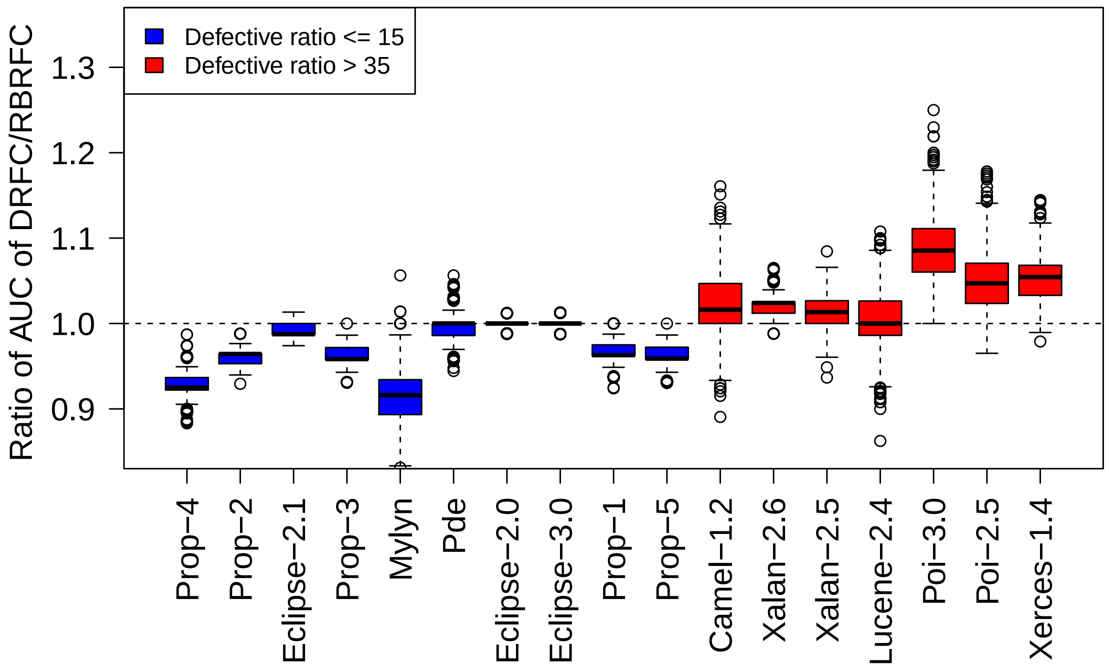


Figure 4.4: Ratio of AUC of discretized/regression-based random forest classifiers across different datasets. The datasets are ordered based on their defective ratio from low to high.

The threshold for analyzing the magnitude is as follows: $|d| \leq 0.2$ means magnitude is negligible, $|d| \leq 0.5$ means small, $|d| \leq 0.8$ means medium and $|d| > 0.8$ means large.

Results: In contrast to prior studies, building a defect classifier using discretized defect counts does not usually lead to better performance. The comparison of DRFC and RBRFC of all datasets are provided in Table 4.4. Overall, the Wilcoxon signed-rank test results show that the differences between DRFC and RBRFC are significant on all datasets. Cohen's d results show that the performance differences between DRFC and RBRFC are not negligible among 14 datasets (82%). More specifically, on 7 out of these 14 datasets, DRFC outperforms RBRFC; while, in contradiction to the intuition, RBRFC outperforms DRFC on another 7 datasets.

Table 4.4: Performance comparison of discretized and regression-based random forest classifiers.

Project	**Avg. AUC of DRFC**	**Avg. AUC of RBRFC**	***p*-Value**	**Cohen's *d***	**DR(%)**
Prop-4	0.72	**0.77**	0	4.27 (L)	9.6
Prop-2	0.81	**0.84**	0	4.32 (L)	10.5
Eclipse-2.1	0.77	**0.78**	0	0.46 (S)	10.8
Prop-3	0.69	**0.72**	0	2.67 (L)	11.5
Mylyn	0.68	**0.74**	0	2.45 (L)	13.2
PDE	0.71	0.71	0	0.12 (N)	14.0
Eclipse-2.0	0.84	0.84	0	0.22 (S)	14.5
Eclipse-3.0	0.80	0.80	0.04	-0.03 (N)	14.8
Prop-1	0.77	**0.79**	0	2.93 (L)	14.8
Prop-5	0.70	**0.73**	0	2.97 (L)	15.3
Camel-1.2	**0.64**	0.63	0	-0.39 (S)	35.5
Xalan-2.6	**0.84**	0.82	0	-0.96 (L)	46.4
Xalan-2.5	**0.76**	0.75	0	-0.39 (S)	48.2
Lucene 2.4	0.77	0.77	0	-0.16 (N)	59.7
Poi 3.0	**0.89**	0.82	0	-2.42 (L)	63.6
Poi 2.5	**0.89**	0.85	0	-1.46 (L)	64.4
Xerces 1.4	**0.96**	0.91	0	-2.43 (L)	74.3

L- Large, S- Small, N- Negligible, DR - Defective Ratio

Regression-based random forest classifiers outperform discretized random forest classifiers when the defective ratio of the dataset data is less than 15% and this trend is reversed when the defective ratio is greater than 35%. To understand how the performance varies among different datasets, we plot the ratio of AUCs of DRFC and RBRFC on the y-axis of Figure 4.4 and we sort the datasets from low defective ratio to high defective ratio on the x-axis. We observe a consistent trend of RBRFC outperforming DRFC for datasets with a low defective ratio (< 15%) and DRFC outperforms RBRFC when the defective ratio is greater than 35%. It should also be noted that though the difference in average AUC is only a few percentage points, it is statistically significant as highlighted in Table 4.4, hence the impact on actual prediction practice will be significant.

One possible reason for DRFC having poorer performance than RBRFC on datasets with low defective ratios is that discretized random forest classifiers are known to be impacted by the imbalance in the dataset (Chen et al., 2004). It is the fact in our case, our findings are suggestive of the fact that, these imbalanced datasets can be better handled by using a regression-based random forest defect classifier in lieu of the traditionally-used discretized classifiers.

The trend of discretized classifiers outperforming regression-based defect classifiers is unique to random forest classifiers. No similar trend is observed for other types of classifiers(e.g., LogReg, LinReg, and KNN). When using other types of classifiers, the discretized classifiers either outperform or perform as good as the regression-based classifiers. For example, Figure 4.5 shows the ratio of AUCs between discretized and regression-based statistical classifiers. We observe that the medians of the ratios are all above the dashed line which indicates that discretized logistic classifiers always

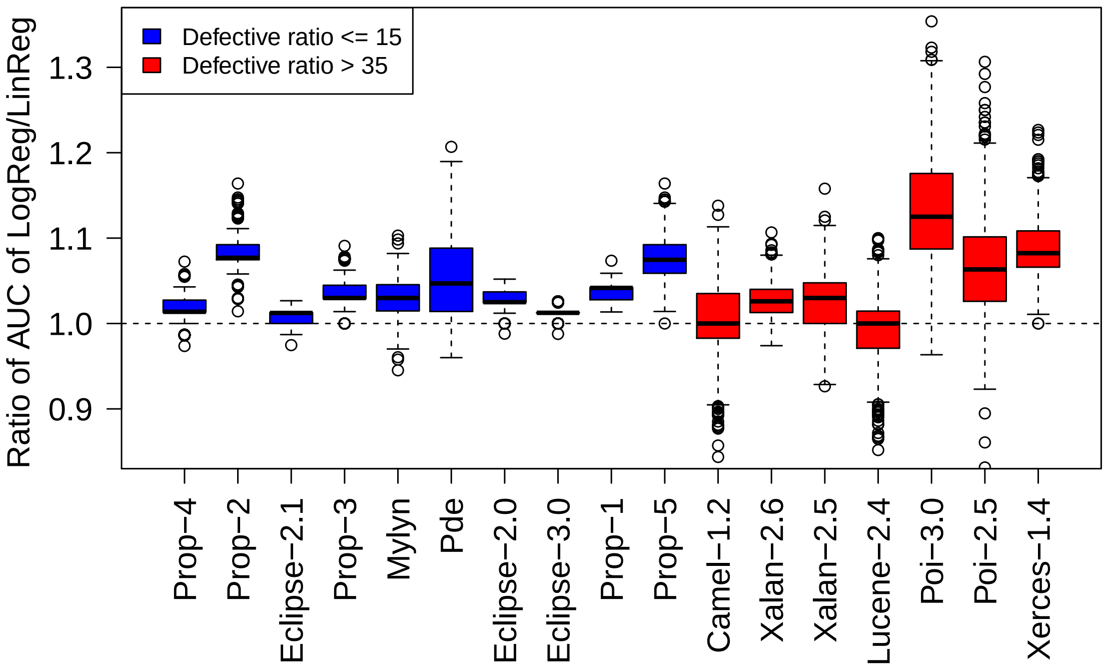


Figure 4.5: Ratio of AUC of discretized/regression-based defect statistical classifiers across different datasets. The datasets are ordered based on their defective ratio from low to high.

outperform or perform equally with regression-based linear classifier. More detailed results are available in Table 4.3.

In summary, the common intuition of building a classifier using discretized defect counts is not always correct. To achieve high performance and enhance quality assurance efforts, we advise the use of RBRFC instead of its discretized alternative on datasets with a low defective ratio (i.e., less than 15%) and the use of DRFC on datasets with a high defective ratio. For datasets with a defective ratio between 15% and 35%, we cannot provide suggestions on which classifier to use, since we do not have datasets in that range of defective ratio. To alleviate this problem, we revisit this point in Section 4.5 where we simulate datasets with different defective ratios.

In constrast to the common practice, building a defect classifier using discretized defect counts does not always lead to better performance. Regression-based random forest classifiers outperform discretized random forest classifiers when the defective ratio of the dataset is low ($< 15\%$) and the pattern reverses when the defective ratio is high ($> 35\%$).

4.4.2 RQ2. Are discretized and regression-based classifiers influenced by the same set of features?

Motivation: Prior studies use defect classifiers to understand the impact that various features (e.g., software metrics) have on the likelihood of a module containing a defect (Cataldo et al., 2009; McIntosh et al., 2014; Mockus, 2010). Understanding the most influential features helps practitioners identify process improvement plans and act on them quickly so that the defects could be avoided in future version of a software systems. In this Chapter, we propose a new approach to build defect classifiers (i.e., regression-based classifiers). In RQ1, we find that such an approach might lead to better performing classifiers than the traditionally used approach for building classifiers (i.e., discretized classifiers). Hence, in this RQ we wish to examine if these different approaches to build classifiers might produce conflicting information about the most influential features that impact the quality of a software module.

Approach: Similar to previous research question, we focus on exploring this RQ with DRFC and RBRFC. However, we will provide insights about the other machine learning techniques whenever appropriate. We follow the approach in Figure 4.1. We build DRFC and RBRFC on the 17 studied datasets as in RQ1. But instead of generating the AUC values of the classifiers, we compute the feature importance ranks for

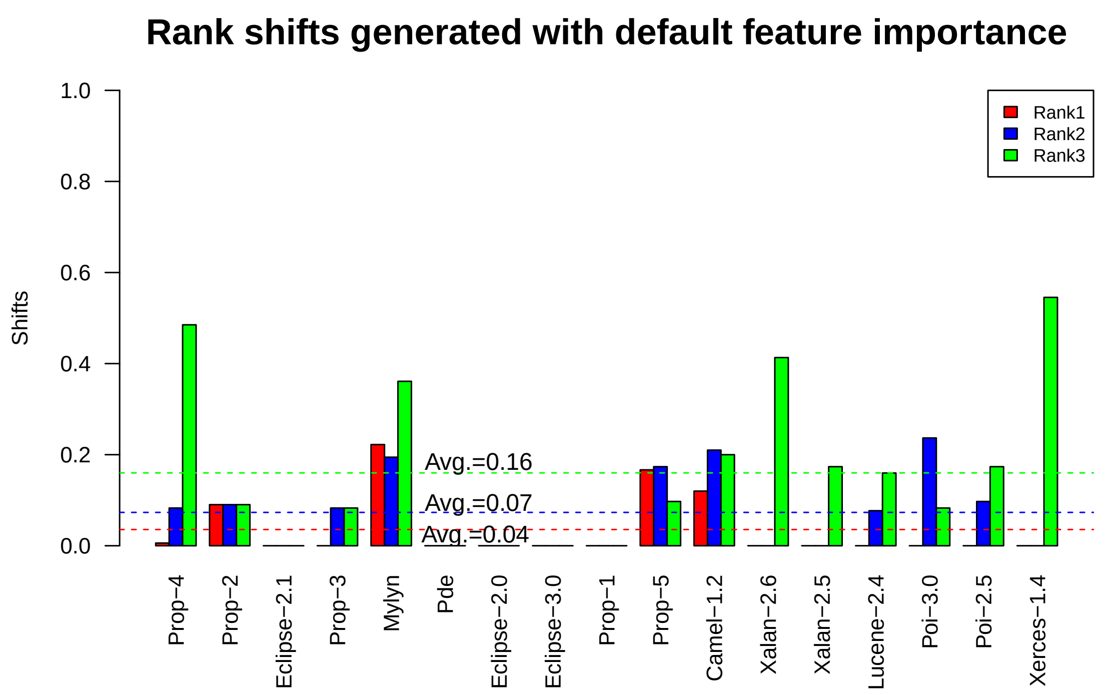


Figure 4.6: Rank Shifts Between DRFC and RBRFC in terms of permutation feature importance across the datasets ordered by defective ratio of the dataset. The mean values of rank shifts are marked with dashed lines.

each feature in each dataset to understand the importance of each feature on identifying the defect-proneness of modules. We first use a permutation importance method for generating feature importance scores for both DRFC and RBRFC as outlined in section 4.2. Once the feature importance scores are generated, we rank features using the Scott-Knott ESD test (Tantithamthavorn et al., 2017). We then compare the computed feature importance ranks of DRFC and RBRFC.

To estimate the impact of DRFC and RBRFC on model interpretation, we compute the shifts in the ranks of the features that appear in the top three ranks for both the DRFC and RBRFC classifiers on each dataset. We define rank shift as the amount that a feature shifts its rank between the two classifier in relation to the total number of features in the dataset. Suppose $DRFC(k) = \{var_1, var_2, ..., var_n\}$ and $RBRFC(k) = \{var_1, var_2, ..., var_m\}$ are the features that appear at rank k of $DRFC$ and $RBRFC$. Let PN be the number of features in the given dataset under consideration. We compute the $Shifts(k)$ of a features on rank k between two classifier for a given dataset using the equation 4.1.

$$\begin{aligned} Shifts(k) = (&\sum_{var \in DRFC(k)} |k - Rank_{RBRFC}(var)| \\ &+ \sum_{var \in RBRFC(k)} |k - Rank_{DRFC}(var)|)/PN \end{aligned} \tag{4.1}$$

In Equation 4.1, $Rank_{RBRFC}(var)$ denotes that rank of var from RBRFC and $Rank_{DRFC}(var)$ denotes the rank of var from DRFC. For example, if the Rank 1 features in the RBRFC are $RBRFC(1) = \{cbo, loc\}$ (i.e., Coupling Between Objects and Lines Of Code), Rank 1 features for $DRFC$ is $DRFC(1) = \{loc\}$, $Rank_{DRFC}(cbo)$ is 2 and NP is 13 for the dataset, we then compute the $Shifts(1)$ between both classifiers as $1/13 = 0.076$, since only the feature cbo has different ranks across both classifiers. We compute the feature importance shifts for all datasets in a similar fashion. These rank shifts capture the difference in the influential features across the two approaches to build classifiers.

Results: Rank 1 features do not vary significantly between the DRFC and RBRFC classifiers, however the influential features vary significantly at the lower ranks. Figure 4.6 shows the rank differences between he DRFC and RBRFC classifiers. The DRFC and RBRFC classifiers have exactly the same rank 1 features in 12 out of the 17 datasets (80%). The features at rank 2 and 3 vary drastically since only 8 (47%) and 5 (29%) datasets have the same features at rank 2 and 3, respectively. In terms of rank shift, we observe that the shift between features in rank 1 (i.e., average shift is 0.04) is small

Table 4.5: Rank shifts between discretized and regression-based classifiers on various techniques.

Technique	**Rank**	**Average Shifts**	**Variance**
Random forest	Rank 1	0.04	0.004
	Rank 2	0.07	0.007
	Rank 3	0.16	0.03
Statistical	Rank 1	0.11	0.22
	Rank 2	0.07	0.005
	Rank 3	0.22	0.039
KNN	Rank 1	0.01	0.001
	Rank 2	0.02	0.002
	Rank 3	0.07	0.008

and the feature importance varies slightly at rank 2 (i.e., average shift is 0.07). But from rank 3 (i.e., 0.16), the shifts start becoming drastic (see the dashed horizontal lines that represent the average shift in each rank between the features in Figure 4.6). We also performed a paired Wilcoxon signed-rank test between the observed shifts and ideal no shift case in which each shifts value is 0 for all datasets. The results show that the shifts at rank 1 are not statistically significant (p-value > 0.05) and shifts at rank 2 and 3 are significant.

We also investigate the rank shifts between discretized and regression-based classifiers when using techniques other than random forest. We present the findings of statistical and KNN classifiers in Table 4.5 as they are the next best techniques after random forests in terms of performance. We find that KNN classifiers exhibit a similar pattern as the random forest classifiers. The feature importance of Rank 1 features does not vary significantly, nevertheless the feature importances start to vary significantly from Rank 2. However, when using statistical classifiers the importance of features shift significantly even at Rank 1.

In summary, although the rank shifts of features appear to be technique dependent, random forest has the most stable ranks for its features across both approaches for building defect classifiers. Nevertheless, we recommend that computed feature importance ranks of the best performing classifier should be used instead of relying solely on the computed feature importance ranks results that are produced by the discretized classifiers (since such classifiers might fail to accurately capture the studied datasets as observed in RQ1)

The importance of Rank1 features does not vary significantly between discretized and regression-based random forest classifiers. However, lower ranked features vary considerably between types of classifiers. Thus, we suggest practitioners to employ caution on the feature importance variation and use the classifiers with superior performance for model interpretation.

4.5 Discussion

4.5.1 How does the performance of discretized and regression-based random forest classifiers vary across different defective ratio?

In RQ1, we observe that the RBRFC outperforms DRFC on data with defective ratio larger than 35% and the observations reverse on the data with defective ratio less than 15%. However, we have no idea how the performance of DRFCs and RBRFCs varies on the data with defective ratio between 15% and 35%. In addition, recent studies typically rebalance the dataset before building a classifier (Tantithamthavorn et al., 2018a). Such resampled datasets might have a defective ratio between 15% and 35%. To fill this gap and better understand the relation between the defective ratio and the performance difference between both approaches for building defect classifiers, we generate datasets with defective ratio ranging from 5% to 50% with 5% interval by re-sampling the studied datasets with replacement (i.e. while keeping the data size fixed). We do this by repeatedly and randomly sampling the datasets with replacement until the datasets have the required defective ratio for our simulation study. Once the datasets at all defective ratios are generated, we followed the experiment setup of RQ1 and analyzed the performance of the generated DRFC and RBRFC on various defective ratios.

We find that as the datasets defective ratio increases the DRFC classifiers start to outperform the RBRFC classifiers. Table 4.6 shows the spearman correlation (ρ) between the ratio of $AUCofDRFC(dataset)/AUCofRBRFC(dataset)$ and the defective ratio of each dataset. For most datasets (94% – 16 out of 17 datasets), there is indeed a positive and strong correlation (i.e., > 0.5) between the defective ratio and the ratio of the AUC in 16 out of 17 datasets.

For example, we show how the DRFC outperforms RBRFC as the defective ratio of the dataset increases using the "Prop-5" dataset in Figure 4.7. After the defective ratio crosses 40%, the DRFC classifier outperforms the RBRFC classifier. This study of variation in performance of AUC between DRFC and RBRFC reaffirms our findings in the RQ1 that RBRFC classifiers perform better for datasets with low defective ratio whereas the DRFC classifiers perform better as the defective ratio of the dataset increases.

Also, we find that the specific point where DRFC classifiers start outperforming RBRFC classifiers is dataset specific. But as a general rule of thumb, DRFC classifiers outperform RBRFC classifiers as the defective ratio in the dataset increases. Finally, for

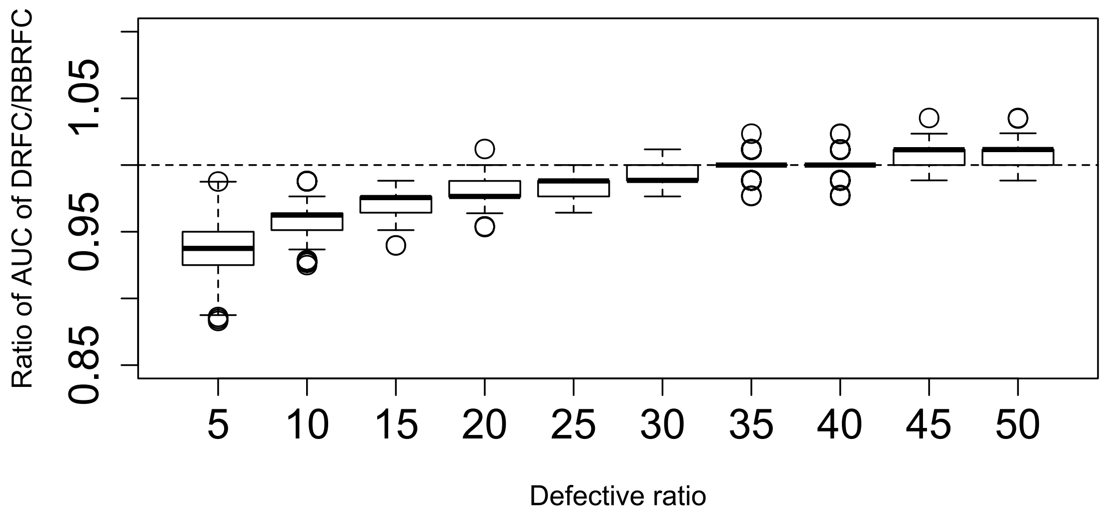


Figure 4.7: Boxplot of the ratio between AUC of DRFC/ AUC of RBRFC on Prop-5 dataset.

Table 4.6: Correlation between defective ratio and ratio of AUC of DRFC/RBRFC.

Dataset	**Correlation**	**Dataset**	**Correlation**
Eclipse-2.0	0.90	Eclipse-2.1	0.81
Eclipse-3.0	0.84	Camel-1.2	0.89
Mylyn	0.79	Pde	0.78
Prop-1	0.90	Prop-2	0.88
Prop-3	0.92	Prop-4	0.91
Prop-5	0.95	Xalan-2.5	0.39
Xalan-2.6	0.86	Lucene-2.4	0.51
Poi-2.5	0.78	Poi-3.0	0.56
Xerces-1.4	0.54		

datasets with defective ratio between 15% and 35% we suggest practitioners to try both DRFC and RBRFC and use the classifier with superior performance.

4.5.2 Does the R^2 regression fit score impact the performance of regression-based classifiers?

The R^2 regression score explains the variability in the data that is captured by the regression model. Prior studies consider that higher R^2 is usually associated with better performance and more accurate model interpretation (Nagappan et al., 2006). But as we are using the regression-based classifier for the purposes of classification (regression-based classifier), we find that this assumption no longer holds.

Low R^2 scores for the regression model does not imply that a regression-based classifier will have a low AUC. There is no correlation between R^2 and a classifier's predictive power (i.e., AUC). A low R^2 score does not indicate poor classification performance. Table 4.7 presents the values of AUC, R^2 and the correlation between the AUC and R^2 of the RBRFC classifier. Overall, the average AUC across all dataset is good (i.e., 0.78), while the average R^2 is poor (i.e., 0.19). We find that in most of the datasets (88%), the correlation between AUC and R^2 is considered weak (i.e., less than 0.4) (Boslaugh and Watters, 2008). Only two datasets, the correlation between AUC and R^2 is considered as moderate (Boslaugh and Watters, 2008). For example, the RBRFC classifier achieves a high AUC 0.84 on Prop-2, while its R^2 is only 0.12 and the correlation between the AUC and the R^2 is 0.17.

Therefore, future studies should use the classification performance measures, instead of regression performance measures when assessing the viability of using a regression-based classifier. From Table 4.7 we observe a higher AUC even when

Table 4.7: Correlation between AUC and R^2 for the RBRFC classifiers.

Dataset	**Correlation**	**Avg. R^2**	**Dataset**	**Correlation**	**Avg. R^2**
Eclipse-2.0	0.14	0.29	Eclipse-2.1	0.33	0.19
Eclipse-3.0	0.19	0.31	Camel-1.2	0.32	0.04
Mylyn	0.28	0.05	Pde	0.21	0.02
Prop-1	0.08	0.05	Prop-2	0.17	0.12
Prop-3	0.21	-0.05	Prop-4	0.18	0.11
Prop-5	0.21	0.10	Xalan-2.5	0.41	0.16
Xalan-2.6	0.45	0.34	Lucene-2.4	0.13	0.33
Poi-2.5	0.37	0.41	Poi-3.0	0.11	0.32
Xerces-1.4	0.07	0.44			

the regression model that was used to construct a regression-based classifier has a low R^2 score. We argue that such a result is because classification is an inherently simpler problem than regression. In other words, a regression model has to predict the number of defects as closely as possible. Whereas for the same regression model to function as a classifier, the regression-based classifier only has to distinguish enough between the two “Defective” and “Non-Defective” classes.

4.5.3 Permutation feature importance vs. Default feature importance?

In RQ2, we propose permutation feature importance method as the method of choice for generating feature importance scores. However, researchers primality use the default feature importance method (i.e., the CS method) that comes along with the implementation of their classifiers. We examine here whether our findings for RQ2 would hold when the default feature importance method is used. We use random forest classifiers for our investigation here. However, the observations hold for all the studied family of classifiers.

Figure 4.6 and Figure 4.8 show the rank shifts between permutation and default feature importance methods. We observe that the rank shifts of Rank 1 between DRFC and RBRFC are low (i.e., average shift is 0.11) with default feature importance method. However, compared with the rank shifts from permutation feature importance method as shown in Figure 4.6, the default feature importance has a higher average rank shift at rank 1. We also conduct the Wilcoxon signed-rank test and it suggests that the feature importances that are computed with default method vary significantly from rank 1. The findings that are observed from permutation and default feature importance methods are still hold from rank 2, although the findings are different in rank 1.

4.6 Threats to Validity

We discuss the threats to the validity of our study.

Construct Validity. Threats to construct validity relates to the suitability of our evaluation measures. We have used AUC to evaluate the performance of defect classifiers in our study. While we have explained our reasons for choosing this metric, other evaluation measures may lead different conclusions. However, AUC is a well-known metric to evaluate classification models and also widely used in prior studies (Lessmann et al., 2008; Tantithamthavorn et al., 2017). Furthermore, AUC is a threshold insensitive metric (Lessmann et al., 2008) and therefore AUC allows us to derive results that generalize over all potential thresholds. We have also performed statistical tests

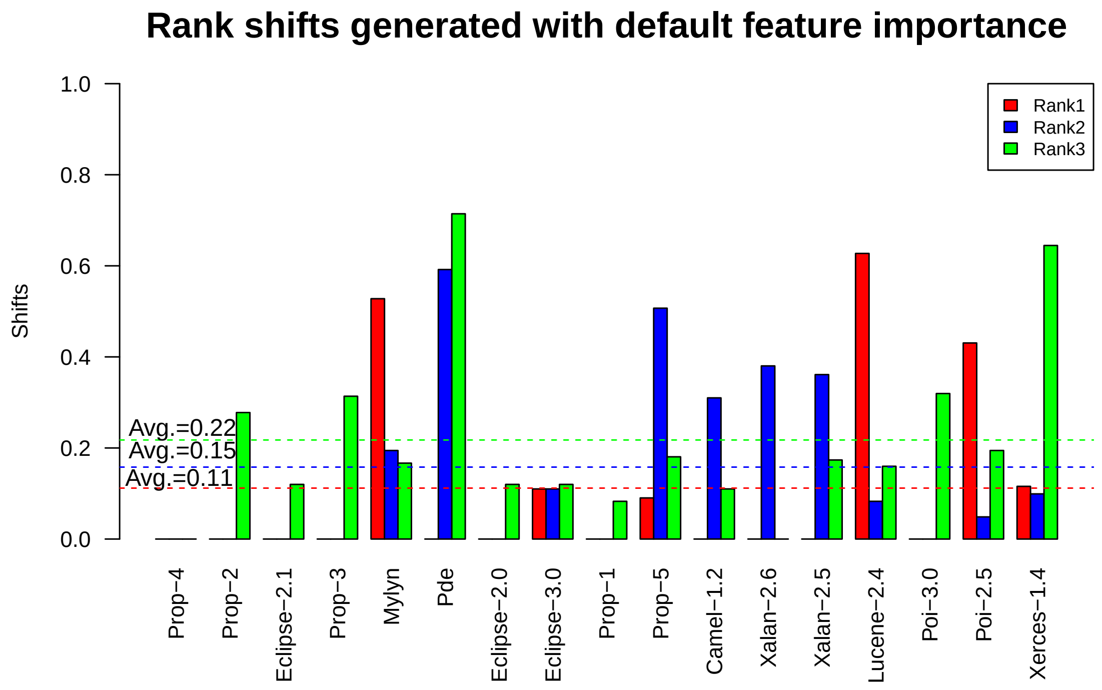


Figure 4.8: Rank Shifts Between DRFC and RBRFC in terms of default feature importance across the datasets. The mean values of rank shifts are marked with dashed lines.

and effect size tests to check if the performance differences between different classifiers are significant and substantial.

In this study we did not optimize the hyper-parameters for studied classifiers except neural networks and CART, as most of the studied classifiers do not get a significant performance boost with parameter optimization (Tantithamthavorn et al., 2017). However, to reduce this threats, future studies should examine the impact of optimized parameters on our findings.

Similarly, some prior studies rebalance the datasets so that the “Defective” and the “Non-Defective” classes are similarly represented in the training data using various class rebalancing techniques (Tantithamthavorn et al., 2018a). However, we do not investigate the impact of using regression-based classifiers on such a rebalanced dataset. We do not do so as Tantithamthavorn et al. (2018a) showed that rebalancing the dataset impacts the computed feature importance ranks. Since we wanted to observe how the regression-based classifiers impact both the performance and feature importance ranks, we consider the studied datasets as is. However, we think that it would be fruitful for the future work to investigate how well the regression-based classifiers perform when compared to discretized classifiers on artificially rebalanced datasets.

In our study we use Cohen’s d effect size test, which is a parametric test that assumes the groups it tests to be normally distributed. However, we do not ensure if the obtained performance scores are normally distributed before using the Cohen’s d effect size test. We acknowledge that this is a threat and future studies should revisit our findings by using a non-parametric effect size test.

Internal Validity. Prior work shows that incorrect data influences the conclusions drawn from software defect classifiers and potentially biases the results (Ghotra et al., 2015). Even though we tried to control the purity of datasets by imposing conditions on data collection, we cannot ensure that our datasets are correct. To reduce the internal validity, future studies should investigate the correctness of the data further.

External Validity. Threats to external validity relate to the generalizability of our results. In this study, we study 17 datasets and our results may not generalize to other datasets. However, the goal of this Chapter is not to show a result that generalizes to all datasets, but rather to show that there are datasets where regression-based classifiers would outperform the commonly used discretized classifiers. Nonetheless, additional replication studies may prove fruitful.

4.7 Chapter Summary

Defect classifiers are extensively used in software analytics. Traditionally, software defect classifiers are built by discretizing the continuous defect counts of modules into "Defective" and "Non-Defective" classes. However the discretization of continuous features leads to a considerable loss of information. To avoid such information loss, we consider a regression-based classifiers which uses the continuous defect counts as the target feature for identifying defect-prone modules.

In this Chapter, we compare discretized and regression-based defect classifiers by applying six machine learning techniques on 17 open datasets from Tera-PROMISE. We observe that in contrast to current practices in our field, the discretization of the dependent feature could be avoided in some cases (especially when the defective ratio is <15%). We make such an assertion as we find that building classifiers using discretized defect counts does not always lead to better performance. Hence future studies should explore both approaches for building classifiers – Given the simplicity of building both types of classifiers, we believe that our suggestion is a rather simple and low-cost suggestion to follow. Moreover, the most influential features vary between the different approaches to build classifiers. Hence future studies should examine the influential factors using the best performing classifier (i.e., discretized or regression-based) instead of simply using discretized classifiers.

CHAPTER 5

Mitigating the Impact of Discretizing the Dependent Feature

Researchers usually discretize a continuous dependent feature into two target classes by introducing an artificial discretization threshold (e.g., median). However, such discretization may introduce noise (i.e., discretization noise) due to ambiguous class loyalty of data points that are close to the artificial threshold. Previous studies do not provide a clear directive on the impact of discretization noise on the classifiers and how to mitigate the impact of such noise. In this Chapter, we propose a framework to help researchers and practitioners systematically estimate and mitigate the impact of discretization noise on classifiers in terms of its impact on various performance measures and the interpretation of classifiers. Through a case study of 7 software analytics datasets, we find that: 1) discretization noise affects the different performance measures of a classifier differently for different datasets; 2) Though the interpretation of the classifiers are impacted by the discretization noise on the whole, the top 3 most important features are not affected by the discretization noise. Therefore, we suggest that practitioners and researchers use our framework to understand the impact of discretization noise on the performance of their built classifiers and estimate the exact amount of discretization noise to be discarded from the dataset to avoid the negative impact of such noise.

An earlier version of this chapter is published in the Transaction on Software Engineering Journal (TSE) (Rajbahadur et al., 2019).

5.1 Introduction

MACHINE learning classifiers are widely used throughout software analytics studies. Some of the most common uses of classifiers include predicting defects (Ghotra et al., 2015; Krishna et al., 2017; Nam et al., 2018), bug-fix times (Jiang et al., 2013), understanding the features that impact the defect proneness of a software system (Cataldo et al., 2009; McIntosh et al., 2014; Mockus, 2010; Menzies et al., 2007b).

Usually, classifiers are trained on labeled data points and are used to predict the target class of the unlabeled data points. In the absence of pre-defined class labels and the availability of only the continuous dependent feature, researchers usually discretize the continuous **dependent** feature into artificial target classes. Such discretization might be based on domain knowledge (Guo et al., 2010), a phenomenon that the study wishes to observe (Jiang et al., 2013), or in many cases when there are no imposed target classes, an artificial discretization threshold is used to discretize the target feature into binary (or n-ary) classes (de Almeida et al., 1998; El-Emam et al., 2001; Tian et al., 2015; Wang et al., 2018).

However a plethora of prior studies note that the discretization of the continuous dependent feature could be detrimental to the performance of classifier and may produce misleading results (Cohen, 1983; Dawson and Weiss, 2012; Royston et al., 2006; DeCoster et al., 2009). One alternative approach to avoid such discretization noise would be to train regression models on the continuous dependent feature and then discretize the predicted outcome afterwards as we outline in Chapter 4. But as we observe in Chapter 4, only when there is a significant class imbalance in the dataset, classification through regression would yield better results. Therefore, the discretization of continuous dependent feature with artificial discretization thresholds is still widely practiced in software analytics as evidenced by (Guo et al., 2010; Gay et al., 2010; Jiang et al., 2013; Schumann et al., 2009; Jalali et al., 2008; Wang et al., 2018; Tian et al., 2015; Hassan et al., 2018). The impact of such discretization of the dependent feature needs to be mitigated in cases where it cannot be avoided.

The other problem with such a discretization approach is that the data points that are very close to the discretization threshold (e.g., median) get class labels that might not be reflective of the true class to which they belong. While many previous studies explore the harmful impacts of discretizing the continuous dependent features on the performance of the classifiers (Cohen, 1983; Dawson and Weiss, 2012; Royston et al., 2006; DeCoster et al., 2009), the problem of data points with ambiguous class labels and its impact on the classifiers is completely unexplored. For instance, consider the example of determining whether a bug was closed fast or slow. A domain-expert might decide that any bug closed within a week is a "fast-closed" bug. However, such a discretization rule for the dependent feature would lead to noise for data points close to that 7-days threshold. For instance, a bug that is closed within 7 days and 1 min would be considered as a "slow-closed" bug. Such discretization introduces noise in the data that is used for training the classifiers. We define such noise as the *discretization noise* and the data points whose ambiguous class labels that generate the discretization noise as the *noisy points*.

In summary, the discretization of continuous dependent feature is both problematic and generates discretization noise. However, the practice of discretizing the continuous dependent feature still remains a widely used practice in software analytics without any consideration to the generated discretization noise.

Therefore the main goal of our study is two-fold: first, to introduce awareness among the software analytics researchers and practitioners about previously unexplored discretization noise. Second, we provide them with a framework - a systematic and rigorous method for exploring the impact of discretization noise on their classifier of choice for any given dataset.

We highlight the capability of our framework by conducting our study on four different types of software analytics datasets (Q&A websites data (4 websites), Linux patch acceptance time data, bug-fix delay data, mobile app ratings data), as a binary classification problem, where the discretization noise is caused by the discretization of a continuous dependent feature around artificial discretization thresholds. We applied our proposed framework to four different families of classifiers (i.e., Random forest classifier (RFCM), Logistic regression (LR), Classification and Regression Trees (CART), and K-Nearest Neighbors (KNN)) to analyze the impact of discretization noise on the various common performance measures (i.e., *Accuracy, Precision, Recall, Brier score, AUC, F-Measure, MCC*). In addition, we also analyze the impact of discretization noise on the interpretation of classifiers in terms of the computed feature importance ranks. We highlight our findings and suggestions as follows:

1. **The impact of discretization noise is inconsistent across multiple performance measures for different datasets across all the studied classifiers.** Though the impact on Recall is the most pronounced (up to 139%), other performance measures - Precision, Brier score, F-Measure, and MCC are also impacted at least up to 43.19%

due to the inclusion or exclusion of discretization noise (both positively and negatively). **Therefore, we urge the researchers and practitioners to use our framework to analyze if (and how much) discretization noise exists and how to address it.**

2. **Though the overall computed feature importance ranks are impacted by discretization noise, the importance ranks of the top three important features are not affected.** Therefore, in absence of any impact on the interpretation of the top x features detected by our framework, one could include or discard the discretization noise in their datasets as recommended by our framework. Especially without being worried about the discretization noise's impact on the interpretation of the classifier.

Finally, we also provide the framework as an R package (and guidelines for using our framework) to provide automated support to others who wish to revisit their prior results or to consider discretization in the future studies.

Table 5.1: Details of datasets used in the study

Dataset	**#Size**	**#Features**	**R(dependent feature)**
Stack Overflow	55,853	28	0-9,981.40 mins
Mathematics	70,336	27	0-30,073.72 mins
Ask Ubuntu	7,134	26	0-31,638.55 mins
Super User	10,776	27	0-51,376.33 mins
Patch	20,000	22	0-1,266.92 days
Bug-delay	2,434	23	-1,319.06*-1,990.27 days
App-rating	7,365	22	1.41-4.97 stars

*The dependent feature has negative value since some developers started fixing a bug before the bug was reported.
R(x) - Range(x)

5.2 Data Collection

In this study, we collect data based on two criteria: 1) the dependent feature is continuous; 2) the dependent feature lacks a clear-cut threshold for discretization. Based on these two criteria, we collect 4 types of data (7 datasets): Q&A websites data (Wang et al., 2018), Linux patch acceptance time (Patch) data (Jiang et al., 2013), Bug-fix delay time (Bug-delay) data (Zhang et al., 2012), and Mobile app rating (App-rating) data (Tian et al., 2015). Table 5.1 contains basic information about the number of data points and the independent features of each of the studied datasets.

Q&A websites data: We reuse datasets from a prior study by Wang et al. (2018), which investigated the features that potentially impact the needed time to receive an accepted answer to a question on Q&A websites. The datasets, which were originally collected from the Stack Exchange sites, are comprised of all of the questions that were posted on Stack Overflow in the year 2015 and all the questions ever posted on Mathematics, Ask Ubuntu, and Super User until December 21st 2015. The datasets have 55,853, 70,336, 7,134, 10,776 data points for Stack Overflow, Mathematics, Ask Ubuntu, and Super User, respectively. We use features that pertain to questions, answer, askers, and answerers in the datasets as outlined by Wang et al. (2018). We treat the time that it took for a question to get an accepted response as the dependent variable in our study. We discretize this dependent variable at a chosen threshold into "fast" (class 1) and "slow" (class 2) to study how fast did a posted question get an accepted response.

Linux patch acceptance time (Patch) data: We use a dataset from a prior study by Jiang et al. (2013). The dataset records the features that impact the acceptance time of patches that are submitted to the Linux kernel's source code. The dataset contains

251,418 data points. We use the patch acceptance time as our dependent variable, which we discretized to “fast” (class 1) and “slow” (class 2) to study how fast did a submitted patch was accepted.

Bug-fix delay time (Bug-delay) data: We use the dataset from a prior study by Zhang et al. (2012). The dataset is obtained from the merging of the bug reports that are available in the Bugzilla tracker for three open source projects Mylyn, Platform (run-time provider of Eclipse), and PDE. The dataset contains 2,435 data points. We use the DelayBeforeChange (DBC), the interval between the time when a bug is assigned to a developer and the time when the developer starts to fix the bug, as the dependent variable of interest in our study. We discretized the variable into “fast” (class 1) and “slow” (class 2) classes.

Mobile app rating (App-rating) data: We use the dataset from a prior study by Tian et al. (2015) on various features of 7,365 mobile apps spanning 30 categories. The dataset was collected to understand the features of highly rated mobile apps. We use the rating of the mobile apps as our dependent variable and discretize them into highly rated apps (class 2) and apps with low rating (class 1).

Table 5.2: Estimated discretization threshold, limits and % of data points in the noisy area for the datasets considered in the study.

Dataset	**MT**			**CT**			**RTT**			***step_size***
	Threshold	**Noisy area (%)**	**Limit**	**Threshold**	**Noisy area (%)**	**Limit**	**Threshold**	**Noisy area (%)**	**Limit**	
SO	21.83 Mins	29	55	136.18 Mins	34	85	214.81 Mins	53	95	5
MA	30.28 Mins	41	70	154.48 Mins	38	85	502.98 Mins	44	95	5
AU	39.74 Mins	38	70	329.28 Mins	54	95	338.93 Mins	53	95	5
SU	30.14 Mins	31	60	193.06 Mins	62	95	262.53 Mins	56	95	5
PH	1.31 Days	10	30	0.06 Days	4	40	9.48 Days	22	60	5
BD	0.67 Days	6	50	0 Days	*	*	47.29 Days	54	100	5
AR	4.03 Stars	43	7	3.86 Stars	50	10	3.74 Stars	63	15	0.5

*The automated noisy area estimation algorithm found no data points in the noisy area

MT- **M**edian based discretization **T**hreshold, **CT**- Univariate **C**lustering based discretization **T**hreshold, **RTT**- CA**RT** based discretization **t**hreshold

Datasets: SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BD- Bug-delay, AR- App-rating

5.3 Framework for Understanding the Impact of Discretization Noise

An overview of our framework for understanding the impact of discretization noise is presented in Figure 5.1. The framework consists of six steps.

We detail the steps below (with more explanations where needed) with a running example in Appendix A to better demonstrate the use of our framework.

The individual steps of our approach are explained in detail below.

5.3.1 Step 1: Correlation and Redundancy Analysis

We perform correlation and redundancy analysis on the independent features of a studied dataset to remove correlated and redundant features from the dataset, thereby not biasing our feature importance results (Tantithamthavorn and Hassan, 2018). We perform correlation and redudancy analysis as we explain in Section 4.2.3. We do so instead of using other common and state of the art dimensionality reduction techniques like PCA, since, dimensionality reduction techniques like PCA combine and transform the original features into principal components, which are no longer directly interpretable. Finally, a recent MSR study by Ghotra et al. (2017) showed that correlation-based feature selection is very robust for software analytics datasets. Though we use and recommend correlation and redundancy analysis, our framework supports the use of other methods. We do not pre-process the independent features of the dataset any further in our study. However, practitioners can perform other data pre-processing steps like imputation if required.

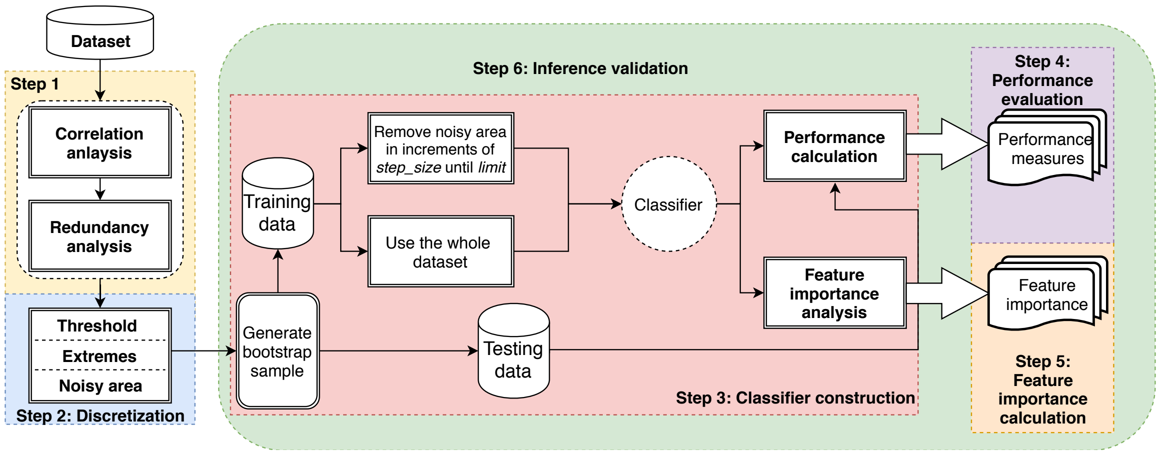

Dataset
Step 1
Correlation anlaysis
Redundancy analysis
Threshold
Extremes
Noisy area
Step 2: Discretization
Step 6: Inference validation
Training data
Remove noisy area in increments of *step_size* until *limit*
Use the whole dataset
Classifier
Generate bootstrap sample
Testing data
Performance calculation
Feature importance analysis
Step 3: Classifier construction
Step 4: Performance evaluation
Performance measures
Feature importance
Step 5: Feature importance calculation

Figure 5.1: Overview of our framework.

5.3.2 Step 2: Discretization

Threshold estimation: The primary objective of our framework is to understand and mitigate the impact of discretization noise. We discretize the dependent feature with respect to an artificial threshold (a.k.a a cutpoint) into two response classes: “class1” and “class2”. We then assign the “class1” class label to all the data points with a dependent feature that has a value that is less than or equal to the chosen discretization threshold (e.g., median). The remaining data points are assigned the class label “class2”.

The artificial threshold for such a discretization could be chosen in multiple ways. The threshold could be domain specific and be defined by the experts (e.g., ideal bug-fix time for a specific project as defined by the software engineers working on the project). Alternatively, in the absence of such an established domain specific discretization threshold, many of the prior studies have resorted to various heuristic, intuitive and alternate thresholds for discretization (Altman and Royston, 2006; de Almeida et al., 1998; El-Emam et al., 2001; Tian et al., 2015; Wang et al., 2018). But irrespective of the choice of the discretization threshold, the data points close to the discretization threshold produce discretization noise. Our framework analyzes the impact of discretization noise generated by any such discretization threshold.

In this study, to demonstrate the generalizability and applicability of our framework, we use three artificial discretization thresholds.

***M**edian based discretization **T**hreshold (**MT**):* Many prior studies use median for discretizing the dependent feature into binary classes (de Almeida et al., 1998; El-Emam et al., 2001) and it is often used in the absence of explicit domain knowledge about the classes of a dependent feature (Altman and Royston, 2006).

*Univariate **C**lustering based **T**hreshold (**CT**):* Univariate clustering is an automated technique for discretization. Univariate clustering splits the dependent feature into multiple groups in an optimal fashion. We use Wang and Song's implementation *optimal k-means clustering in one dimension* (ckmeans.1d.dp[1]) here (Wang and Song, 2011). The ckmeans.1d.dp divides data in one dimension into k clusters so that the sum of squares of within-cluster distances from each element to its corresponding cluster mean is minimized (Wang and Song, 2011). We set k equal 2 since we wish to divide the dependent feature into two classes.

*CA**RT** based discretization **T**hreshold (**RTT**):* We use the regression tree approach as described by Breiman (2017).[2] Here, we use the continuous dependent feature of our dataset as both the independent and the target feature for the regression tree. We then use the generated regression tree's root node as the threshold for discretization since we attempt to split the dependent feature into two classes.

[1]https://cran.r-project.org/web/packages/Ckmeans.1d.dp/index.html
[2]https://cran.r-project.org/web/packages/rpart/index.html

The generated discretization threshold is used for discretizing the continuous dependent feature into binary classes.

Noisy area estimation: Once the dataset is discretized, we need to define the area of the dataset which contains discretization noise as the noisy area. Domain experts could determine a specific range of values around the discretization threshold to be noisy and this could be used as the noisy area. But as we lack deep domain expertise of the datasets considered in this study (which might be the case for many practitioners), we present Algorithm 1: an automated algorithm for estimating the noisy area in a given dataset.

We define the *noisy area* as the data points whose class loyalties are hard to discern due to their proximity to the artificial discretization threshold. Such a hypothesis follows from the rationale of prior studies where the data points around the discretization threshold were discarded to provide better class separation in the training data (Tian et al., 2015; Wang et al., 2018; Judd et al., 2009; Abdelwahab and Busso, 2015; Abdelmoez et al., 2012). Algorithm 1 takes the dataset, cutpoint (i.e., the discretization threshold), and *step_size* as input parameters. *step_size* controls the granularity of the analysis (i.e., the size of the increment from the cut point) - a smaller value of the *step_size* allows for a finer estimation of the noisy area, whereas a larger value provides a coarse estimation of the noisy area. The *step_size* used for all the datasets in this study is given the Table 5.2.

Line 1 to 3 of the algorithm establishes the initial candidate noisy area, by selecting the area around the cutpoint. More specifically, we consider the points within the area $cutpoint \pm cutpoint * 100\%$ as the candidate noisy area. We do so for two reasons: 1) most of the discretization noise would be concentrated around the discretization threshold due to its proximity to the threshold. 2) if we consider more data, we might not be able to ascertain if the impact of the noisy area on the performance and interpretation of a classifier is due to discretization noise or the high volume of data that is lost. Through line 4 to line 9 we incrementally subset the dataset into the quantum of size given by $cutpoint \pm cutpoint * setp\_size$ and compute the non-linearity of the quantum.

Non-linearity is one of the complexity measures defined by Ho and Basu (2002). Non-linearity score attempts to quantify how hard it might be for a classifier to classify the data points (please refer Section 6.5 and Table B.6 in the Appendix for more details about complexity measures). Once we establish the non-linearity for all the quanta, we take the quantum with the maximum non-linearity as the noisy area for our analysis and the step_size that yielded the quantum as the $limit$, which we use to demarcate the noisy area. We use the maximum non-linearity to demarcate the noisy area as it indicates the quantum with the highest data complexity (thereby harder for the classifier). Figure 5.2 demonstrates how the limit value is used to demarcate the noisy area in a dataset. We present the $limit$ that is generated for demarcating the noisy area of all the studied datasets for various discretization thresholds in Table 5.2.

Extremes estimation: Finally we establish the data points with the least discretization noise as the extremes. These data points typically have high discriminative power as they are the furthest away from the discretization threshold. Extremes are typically the data points that are associated with the top and bottom x% of the sorted continuous dependent feature. Prior studies usually consider the top and bottom x% as data points that are devoid of noise and use them for constructing the classifier (Wang et al.,

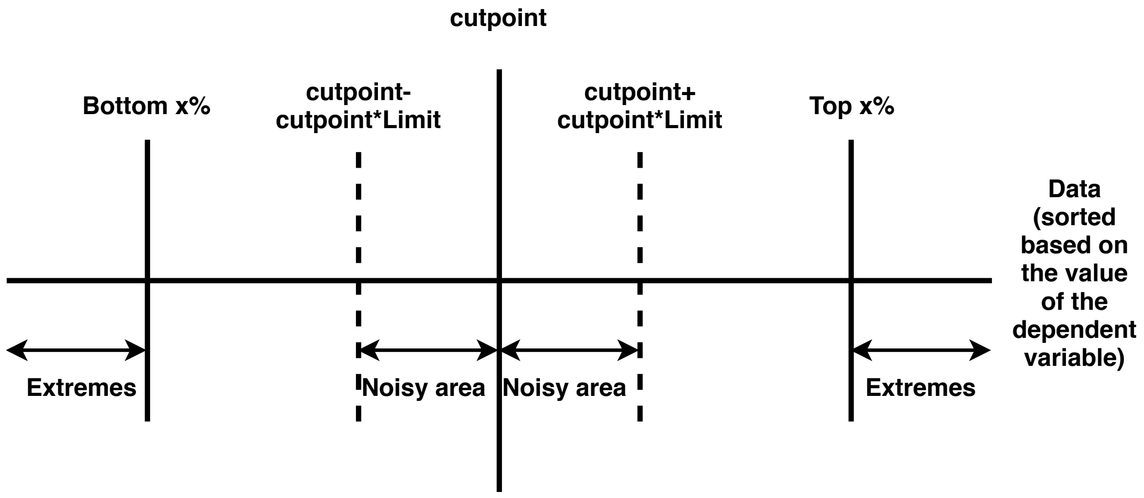


Figure 5.2: Extremes and noisy area definitions of a dataset.

2018; Tian et al., 2015). We use x as 10% in this study. But the framework allows using any value without any further change to the overall methodology.

5.3.3 Step 3: Classifier construction

To study the impact of discretization noise, we construct a classifier on the whole dataset and on the dataset with the noisy area removed. One can choose any classifier of their choice in this step. In our study, we consider the 6 classifiers considered by Rajbahadur et al. (2017). From the 6 classifiers, we choose the classifiers that have a default feature importance computation method as our framework studies the impact of discretization noise on both the performance and feature importance. Therefore we demonstrate the capability of our framework to analyze the impact of discretization noise on random forest classifier (RFCM), Logistic Regression (LR), Classification and Regression Tree (CART), and K-Nearest Neighbour (KNN). All of the chosen classifiers are hyper parameter tuned to ensure the best and stable performance. We used the

Algorithm 1: Automated noisy area estimation algorithm

```
Input: dataset, cutpoint, step_size
Output: limit
Result: Estimates the noisy area in the data automatically by computing the
        limit
1  $lower\_limit = cutpoint - cutpoint * 100\%$
2  $upper\_limit = cutpoint + cutpoint * 100\%$
3  $noisy\_area = SUBSET(dataset, lower\_limit, upper\_limit)$
4  while ($(cutpoint \pm cutpoint * step\_size) \leq upper\_limit\ AND \geq$
     $lower\_limit$) do
5      $quanta = \text{SUBSET}(noisy\_area,\ cutpoint - cutpoint *$
         $step\_size,\ cutpoint + cutpoint * step\_size)$
6      $nl\_score = \text{COMPUTE\_NON\_LINEARITY}(quanta)$
7      $results[step\_size] = (nl\_score)$
8      $step\_size \mathrel{+}= step\_size$
9  end
10 $limit = Indexof\ \text{MAX}(results)$
```

method used by Tantithamthavorn et al. (2018b) to hyper-parameter tune all of our classifiers.

Though we use the four aforementioned classifiers, one can use other classifiers instead of these classifiers without any changes to the other steps in the framework.

5.3.4 Step 4: Performance evaluation

In this step, the desired classifier performance evaluation measures are chosen. In this study, we observe and evaluate the performance of the constructed classifiers on *Accuracy, Precision, Recall, Brier score, Area Under the receiver operator characteristic Curve (AUC), F-measure, and Mathew's Correlation Coefficient (MCC)*, since many prior studies studied the performance of classifiers using these measures (Zhang et al., 2014;

Boughorbel et al., 2017; Brier, 1950). We calculate these measures with "class 1" as the relevant (positive) class.

Though we demonstrate our framework on the aforementioned performance measures, our framework allows users to use any performance evaluation measures (by themselves or in combination with other measures).

5.3.5 Step 5: Feature importance calculation

We use the default feature importance calculation technique (i.e., the CS method) that is associated with each of the studied classifiers to compute the feature importance for each classifier. We use the feature importance computation method **VarImp()** of **caret** package to compute the feature importance ranks of the studied classifiers.

5.3.6 Step 6: Inference validation

To ensure that the conclusions that we draw about our classifiers are statistically robust, as we do in Section 4.2.7 of Chapter 4, we use the 100 out-of-sample bootstrap validation technique. We do so as Out-of-sample bootstrap yields an optimal balance between the bias and variance as suggested in the recent study of Tantithamthavorn et al. (2017).

The out-of-sample bootstrap process is repeated 100 times. After the bootstrap validation, 100 performance measures and 100 lists of computed feature importance ranks are generated. We carry out further analysis on these generated performance measures and the computed feature importance ranks to investigate our research questions.

5.3.7 Framework deployment

We use our framework of 6 steps on any given dataset to analyze the impact of discretization noise (as demonstrated in Section 6.4) along with performance and interpretation on the chosen classifier. Step 1 removes the correlation and redundancy among the features in a dataset, while step 2 is pivotal for estimating the noisy area and extremes for a chosen discretization threshold. Steps 3 to 6 are repeated by incrementally discarding data points in increments of the $step\_size$ parameter (smaller *step_size* enables finer analysis and vice versa) in the noisy area around the threshold until all of the data points in the noisy area are discarded. Such an incremental analysis helps the framework identify the impact of discretization noise and determine the exact amount of data from the noisy area that needs to be discarded for a given dataset, discretization threshold and classifier of choice. We also provide an R package[3] of our framework to enable others and practitioners automated support to use our framework with trivial effort.

5.4 Understanding the Impact of Discretization Noise on the Performance and Interpretation of a Classifier

5.4.1 Studying the impact of discretization noise on the performance of a classifier

Motivation: It is intuitive to expect that discretization noise might impact the performance of a classifier. Ferri et al. (2009) show that different performance measures are

[3]https://github.com/SAILResearch/suppmaterial-19-gopi-discretization_noise_impact

impacted differently by different types of noise in a dataset. Therefore, first, it is essential to establish if discretization noise impact the performance of a classifier like other noises. Second, if the discretization noise does impact the performance of a classifier, we need to analyze how the discretization noise in a dataset impact the performance of a classifier (either positively/negatively) in different performance measures. Finally, it is essential to establish how much data do we have to discard to avoid/mitigate the impact of discretization noise (as opposed to using only the top and bottom x%).

In order to enable researchers and practitioners to perform such an analysis in a generalizable fashion, we propose our framework. Our framework enables researchers and practitioners to examine the impact of discretization noise on the performance of various classifiers using a variety of software analytics datasets, across a multitude of performance measures.

Approach: We employ our proposed framework (see Section 5.3) to perform an incremental analysis as mentioned in Section 5.3.7 to estimate the impact of discretization noise on the performance of a chosen classifier. We specifically draw attention to the classifier construction (step 3) of the framework (see Figure 5.1). In order to ascertain the performance impact of discretization noise on the classifiers, we train the chosen classifier on data after excluding incremental amounts of discretization noise. More specifically, we discard data points in windows which are defined as $cutpoint \pm cutpoint * x/100$ and use the retained data as the training data to build a classifier, where x varies from 0 to *limit* in increments of $step\_size$ (as mentioned in Step 2 of our framework (See Section 5.3.2)). The $step\_size$ can be different for different datasets depending on the *limit* used to define the noise area for a particular

Table 5.3: Percentage of improvement in median performance of various classifiers with the noisy area removed over classifiers with no data removed across various performance measures (The x value for which the performance impact first occurs for the given measure is also provided).

Classifier	Dataset	ACC (%)		PRC (%)		RCL (%)		BS (%)		AUC (%)		F-M (%)		MCC (%)	
		Mag	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x
RF	SO	**-0.65**	50	**5.07**	15	**-12.42**	15	**-8**	5	0	0	**-3.69**	25	-0.054$^{\#}$	55
	MA	**-3.5**	45	**8.96**	15	**-32.2**	15	**-11.77**	5	-1.23$^{\#}$	70	**-11.61**	20	**-6.38**	50
	AU	**-7.76**	40	**12.09**	30	**-99.02**	25	**-7.79**	5	0	0	**-43.19**	25	**-18.64**	45
	SU	-0.53§	0	**5.88**	25	**-20.4**	20	**-7.82**	5	0§	45	**-7.59**	30	0.36	0
	PH	**-1.64**	10	**1.36**	25	**-6.64**	10	**-2.04**	5	**-2.22**	10	**-2.62**	10	**-4.08**	10
	BD	0.47§	0	0.72	0	0.75	0	-0.66$^{\#}$	40	0§	0	0.74§	0	1.81§	0
	AR	-0.78§	0	**-4.23**	3	**13.17**	2	**-2.89**	0.5	0	0	**4.52**	2	-2.93§	0
LR	SO	-0.76$^{\#}$	55	**6.24**	15	**-20.81**	20	**-7.73**	5	0§	0	**-7.04**	25	-0.83	0
	MA	**-1.78**	15	**12.33**	10	**-46.87**	10	**-12.02**	5	0$^{\#}$	70	**-15.28**	15	**-2.59**	25
	AU	**-5.54**	50	**11.47**	30	**-139.29**	25	**-10.42**	10	1.56§	0	**-59.78**	25	**-16.3**	60
	SU	**-1.17**	60	**6.42**	30	**-48.2**	20	**-5.87**	10	0	0	**-18.92**	20	**-4.61**	60
	PH	-0.08§	0	**2.23**	15	**-4.68**	10	**-2.59**	5	0 §	0	**-0.97**	15	**-1.28**	20
	BD	-0.2	0	-0.39	0	1.63§	0	-0.78$^{\#}$	40	0	0	0.98	0	-0.18	0
	AR	-0.36§	0	**-2.83**	4	**10.82**	2	**-8.03**	0.5	0	0	**3.95**	2	0.44	0
CART	SO	**1.41**	10	**8.58**	15	**-16.77**	20	-2.77$^{\#}$	30	**3.85**	20	**-3.42**	35	**5.36**	15
	MA	**-1.67**	10	**11.26**	20	**-37.16**	20	**-6.54**	60	0	30	**-11.93**	30	-1.27$^{\#}$	10
	AU	-0.56§	0	**7.34**	30	**-40.49**	30	**-9.55**	50	0	50	**-15.97**	40	2.47§	0
	SU	**1.46**	20	**6.93**	25	**-21.36**	45	**-7.32**	55	**2.99**	25	**-7.05**	50	**9.52**	20
	PH	**-1.12**	10	**1.76**	25	**-6.77**	15	-1.79§	0	**-1.79**	15	**-2.45**	10	**-3.21**	10
	BD	1.2§	0	1.33§	0	0.67	0	-0.77§	0	1.54§	0	1.69§	0	6.53§	0
	AR	0.33	0	-1.06$^{\#}$	6.5	**9.92**	3.5	-3.62§	0	1.64§	0	**4.35**	3.5	3.79§	0
KNN	SO	**2.64**	10	**6.02**	10	**-9.88**	25	**-3.72**	5	**4.29**	5	**-1.81**	10	**12.14**	10
	MA	**1.55**	10	**10.45**	10	**-30.51**	20	**-6.74**	5	**4.29**	10	**-9.91**	10	**11.61**	10
	AU	-0.15	0	**4.12**	50	**-46.12**	25	**-4.52**	10	1.75$^{\#}$	55	**-21.44**	30	4.56§	0
	SU	**1.78**	20	**4.28**	20	**-19.06**	30	**-3.04**	15	**1.69**	15	**-7.01**	40	**17.31**	20
	PH	**-1.13**	20	-0.33	0	**-4.09**	15	**-0.59**	10	**-1.37**	15	**-2.18**	15	**-4.61**	20
	BD	0.77	0	0.86	0	0.44	0	-0.22§	0	1.85	0	0.67	0	17.43	0
	AR	0.16	0	-0.62§	0	**11.54**	2	**-1.19**	1.5	0	0	**5.6**	2	2.05	0

1. **Performance Measures:** ACC- Accuracy, PRC- Precision, RCL- Recall, BS- Brier Score, F-M- F-Measure
2. **Datasets:** SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BD- Bug-delay, AR- App-rating
3. Mag- Magnitude of the performance impact |x- % of data of the noisy area when dropped starts statistically impacting the given performance measure
4. **Cohen's d effect size:** Negligible - No formatting, Small -§, Medium -$^{\#}$, Large - **bold**
5. '−' indicates performance measure decreases due to removal of noisy area; '+' indicates performance measure increases due to removal of noisy area
6. All the values with small, medium or large effect size are statistically significant with $p \leq 0.05$
7. '−'in cases of Brier score indicates an actual increase in the Brier score and '+' a decrease in Brier score (Lower the Brier score, the lesser the error)

dataset. Table 5.2 presents the various limit and *step _ size* values used for different datasets in our study. We perform this incremental analysis (see Fig B.1 of Appendix) on all the studied datasets for the three different discretization thresholds (MT, CT, and, RTT) considered in the study as given in Section 5.3.2 for all the four chosen classifiers (RFCM, LR, CART, and, KNN).

To measure whether the performance of a chosen classifier is impacted by removing data points in the noisy area (data containing discretization noise), we use a Wilcoxon signed-rank test (Wilcoxon, 1945), since it is a non-parametric test without any assumptions about the underlying distribution. Furthermore, to quantify the magnitude of the performance differences between the performance of the classifier with no data points removed and the classifier with data points in the noisy area removed, we use Cohen's d effect size test (Cohen, 1988). The threshold for analyzing the magnitude is as follows: $|d| \leq 0.2$ means magnitude is negligible, $|d| \leq 0.5$ means small, $|d| \leq 0.8$ means medium and $|d| > 0.8$ means large.

We perform these statistical tests between the performance measures of the classifier that is constructed on the whole data and the classifier constructed on each step of the incremental analysis (where noisy points in the noisy area is incrementally removed). We do so to estimate how much data needs to be discarded to observe a statistically significant impact on the performance of a classifier. The x value (of the $cutpoint \pm cutpoint * x/100$ used for discarding data in the noisy area) for which different performance measures get significantly impacted is reported. If discarding the whole of the noisy area does not create a statistically significant impact for a particular performance measure then 0 is reported instead of x to signify that the discretization noise does not have a significant impact on that particular performance measure for the studied classifier.

Results: The impact of discretization noise on the performance of different classifiers varies across datasets. Similar performance impacts could be observed for the discretization noise generated by all the discretization thresholds considered in the study. Therefore, we only report the impact of discretization noise generated by the median based discretization threshold (MT) on various performance measures for all the four classifiers in Table 5.3 for brevity (See Table B.1 and Table B.2 of the appendix for the performance impact due to other discretization thresholds on the studied classifiers).

Table 5.3 shows that the discretization noise impacts different classifiers differently. For instance, in the case of both CART and KNN classifiers, for all the datasets except for the patch dataset, the removal of discretization noise improves the performance in terms of AUC. However, for the patch dataset removal of discretization noise negatively impacts the AUC measure. For the same classifiers, even while the AUC and the Precision measures are positively impacted by the removal of discretization noise for (5/7 for CART and 6/7 for KNN), the Recall measure is negatively impacted for 5/7 datasets. Similarly, for LR classifier, while we observe no large impact on the AUC, we could observe up to 139% impact on the Recall as we can observe from Table 5.3. Furthermore, while the removal of discretization noise negatively impacts the accuracy of the RFCM and LR classifier, for the Stack Overflow dataset, it positively impacts the accuracy of the CART and KNN classifiers. Such a varied impact on the different performance measures for the different classifiers can be observed throughout (see Table 5.3). **Therefore we assert that different classifiers are impacted differently (either positively or negatively) on the studied performance measures and datasets.**

Additionally, in Table 5.3 we also provide the x, which tells us the percentage of data from the noisy area, that when dropped, starts statistically impacting the given performance measure. This value aids the users of our framework to know how much data they need to discard in order to avoid the performance impact of the noise on a particular performance measure for a studied classifier.

We find that the removal of discretization noise has both a positive and negative impact on different performance measures for different classifiers. In addition, we do not observe a generalizable trend in how the discretization noise affects different performance measures. For instance, from Table 5.3, we see that the removal of the noisy area from datasets negatively impacts the performance measures of an RFCM for 6/7 datasets on Accuracy, Recall, Brier score, F-measure, 2/7 datasets on AUC, and 5/7 datasets on MCC, most of which in a statistically significant fashion with and a large effect size. However, for 6/7 datasets, removal of noisy area positively impacts the Precision in a statistically significant fashion with a large effect size (in most cases). Furthermore, while removal of noisy area negatively impacts the performance measures for most datasets, it improves the Accuracy, Precision, Recall, F-Measure, and MCC for the App-rating dataset in a statistically significant fashion with a large effect size. These varied impacts of the discretization noise on the different performance measures highlight the need for our frameworks to study the peculiarities of each case study as **the discretization noise affects the different performance measures differently**

Finally, we also note that **magnitude of the performance impact due to discretization noise varies for different performance measures.** For instance, in all of the classifiers, for most of the datasets, we could see that while Recall is impacted heavily (up to 139% for LR in Ask Ubuntu), other performance measures even if impacted significantly, are not at the same magnitude (e.g, only -5.54% for accuracy in LR for Ask Ubuntu).

Discussion: We find that the magnitude of the impact of some performance measures is much greater than other performance measures. Also, from Table 5.3 we note that for a given dataset and a classifier, some performance measures are impacted even for small amounts of discretization noise (given by the x in the Table 5.3). Whereas, some other performance measures are more resilient. For instance, in the case of the Mathematics dataset for the RFCM classifier, discarding 15% of the data in the noisy area significantly impacts both Precision and Recall with a large magnitude varying from 8.96% to -32.2%. Whereas even with a drop of 70% data from the noisy area the AUC gets impacted marginally by -1.23%, which indicates that **some performance measures are more resistant to discretization noise than others**.

The different degrees to which different performance measures are impacted can be attributed to the different nature of each performance measure and what they seek to capture. For instance, Precision, Recall, and F-measure focus on capturing how good a classifier performs in predicting one of the classes (Sokolova et al., 2006). Therefore, a potential imbalance in the dataset caused by discarding different amounts of data in the noisy area, even if it is discretization noise (as such noisy points contain useful information too), could impact these measures greatly. Furthermore, Flach (2003) shows that these measures are easily impacted by class imbalance. These explain the large impacts that we observe for the Precision, Recall, and F-measures (as opposed to Accuracy which takes both the classes into consideration). Especially, with all of our datasets having a skewed distribution of the dependent feature (average Skewness of the dependent feature of all of the studied datasets is 12.99 and average Kurtosis is 309.4). For example, let us consider the Ask Ubuntu dataset with MT as the discretization threshold. When the whole dataset is used (without any removal of data), the number of data points belonging to each class are equal. However, discarding the data in the noisy area according to Table 5.2 for MT induces a class imbalance in the dataset. The number of data points belonging to "class1" only makes up 35% of the dataset. Such class imbalance produced by discarding data induces a high degree of impact on class-specific performance measures for the Ask Ubuntu dataset as shown in Table 5.3.

However, the more balanced measures like AUC and MCC are more robust and insensitive to class distributions (Lessmann et al., 2008). Therefore, AUC and MCC are impacted much less severely by the discretization noise in the dataset. As we can observe from Table 5.3, the magnitude to which measures like AUC and MCC are impacted is lesser than the class-specific measures like Precision, Recall, and F-measure. For instance, while Recall is impacted by as much as 139%, AUC is impacted only by at most 4.29% across all the datasets and classifiers. Even for the Ask Ubuntu dataset, while the impact on class-specific performance measures is high, the AUC is seldom impacted. A similar trend could be observed for MCC as they consider both false positives and false negatives, even though they are slightly more sensitive to the discretization noise than AUC. As Huang and Ling (2005) and Mossman (1994) show balanced measures like AUC are very stable and insensitive to noise hence even a small impact in terms of magnitude (if the effect size is large) could be significant. Therefore though the absolute magnitude of the impact is small for some performance measures like AUC and Accuracy, the true nature of the impact needs to established by the practitioner. Our framework seeks to provide a means for finding and measuring the impact. Finally, similar to other balanced measures we observe that the impact on Brier score is moderate (within 13%) when compared to class-specific performance measures. Brier score being an error metric calculates the mean squared difference between the actual outcome and the assigned probability, which elucidates the distance between the classifier's predictions and the actual classes in probability scale. Table 5.3 shows that Brier score is universally negatively impacted for a RFCM (signifying an increase in Brier score). Because, though the noisy points contain discretization noise, they also contain useful information (see Section 6.5.1) and the removal of such information could universally affect the predicted probability scores of classifiers, especially when they are robust to noise (Folleco et al., 2008). However the Table B.1 and Table B.2 provided in Appendix (for performance impact due to discretization noise that is generated by CTT and RT) shows a varied but moderate impact (within 14%) for LR, CART and KNN, which is similar to other balanced performance measures. **In summary, unlike the class-specific performance measures, balanced measures are impacted moderately by the discretization noise.**

To conclude, discretization noise impacts different performance measures differently for various datasets with varying magnitudes for different classifiers (0-139%) as shown in Table 5.3. We also note that **balanced performance evaluation measures are more resilient to the discretization noise whereas class-specific performance**

evaluation measures are greatly impacted by the discretization noise in the dataset. Thus the inclusion of discretization noise in the construction of any chosen classifier could be either beneficial or detrimental depending on the performance measure of interest and the dataset at hand. **Such unpredictability demonstrates the need for our framework to better understand when it is advisable to remove the noisy points, and how much of the data in the noisy area is to be removed to avoid impact on the studied performance measure for a chosen classifier.**

The impact of noisy discretization data points is inconsistent across multiple performance measures for different datasets and different classifiers. Though the impact on class-specific measures (Precision, Recall,F-measure) is the most pronounced (up to 139%) other performance measures are also consistently impacted at least up to 60% due to the inclusion or exclusion of discretization noise (both positively and negatively). Hence, it is advisable to use our framework beforehand to carefully understand if the noisy discretization data points are to be used or discarded and how much of them needs to be discarded.

5.4.2 Studying the impact of discretization noise on the interpretation of a classifier

Motivation: Many prior studies use classifiers to understand the impact of features on the dependent feature (McIntosh et al., 2016; Thongtanunam et al., 2016). From Section 5.4.1, we observe that the data from the noisy area sometimes impacts the performance of a classifier differently for different datasets. This could be because the data in the noisy area contains useful information along with the discretization noise.

Therefore removing/including such data may also lead to a misleading interpretation of a classifier. Therefore, we seek to observe how the discretization noise impacts the computed feature importance with our proposed framework to decide if such noisy data points can be safely discarded or should they be included nevertheless.

Approach: We demonstrate the capability of our framework to analyze the impact of discretization noise on the computed feature importance ranks. We adopt the same incremental analysis approach that we adopted in the Section 5.4.1. But instead of measuring the performance of the classifiers that are constructed by incrementally discarding data from the noisy area, we note the feature importance values of these classifiers (see step 3 of Section 5.3). We perform an incremental analysis on all the studied datasets for the three different discretization thresholds (MT, CT, RTT) considered in the study in Section 5.3.2 (see Fig B.1 of Appendix) for all the four chosen classifiers (RFCM, LR, CART, KNN).

We measure the computed feature importance values of the chosen classifier trained on various data configurations. We use the Scott-Knott ESD test to rank (Tantithamthavorn et al., 2017, 2018b; Li et al., 2017; Ghotra et al., 2015) the features with their feature importance values. To observe whether the computed feature importance ranks vary significantly, we compare the computed feature importance ranks of the classifier that is trained on the whole data and the classifier with $x = limit$ (see Table 5.2) data points removed from the noisy area for each of studied dataset across all the discretization thresholds. We compute the difference between the computed feature importance ranks for each feature in the dataset and compare them to the null distribution (where the rank difference of each feature is zero) to see if removing the

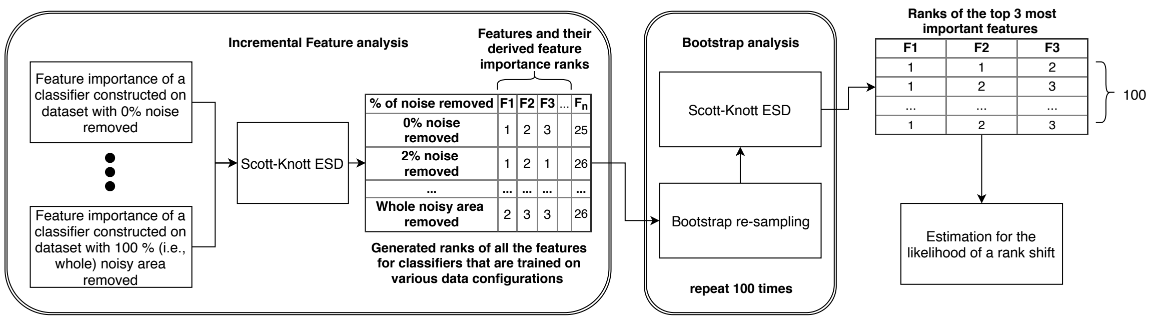

Incremental Feature analysis
Feature importance of a classifier constructed on dataset with 0% noise removed
Feature importance of a classifier constructed on dataset with 100 % (i.e., whole) noisy area removed
Scott-Knott ESD
Features and their derived feature importance ranks
% of noise removed F1 F2 F3 ... Fn
0% noise removed 1 2 3 25
2% noise removed 1 2 1 26
Whole noisy area removed 2 3 3 26
Generated ranks of all the features for classifiers that are trained on various data configurations
Bootstrap analysis
Scott-Knott ESD
Bootstrap re-sampling
repeat 100 times
Ranks of the top 3 most important features
F1 F2 F3
1 1 2
1 2 3
1 2 3
100
Estimation for the likelihood of a rank shift

Figure 5.3: The procedure for estimation of the likelihood of a rank shift.

noisy area impacts the computed feature importance ranks of a classifier (Rajbahadur et al., 2017).

For instance, lets consider the Stack Overflow dataset with MT as the discretization threshold when used for training an RFCM. We wish to study the impact of discretization noise on the interpretation of the constructed RFCM. For simplicity, lets consider that the Stack Overflow dataset has only 5 features. For the RFCM that is intially trained on the whole dataset, a computed feature importance ranks is generated for its 5 features ($F_{\text{whole}} = 3,1,5,4,2$). Following which, an RFCM is trained on the Stack Overflow dataset devoid of the noisy area ($F_{\text{noisy\_area\_removed}} = 2,1,3,2,4$). The difference between F_{whole} and $F_{\text{noisy\_area\_removed}}$ is $\textit{difference} = 1,0,2,2,2$. If the F_{whole} and $F_{\text{noisy\_area\_removed}}$ are the same, then the *difference* generated would be zero and would imply that the discretization noise does not impact the interpretation of a classifier. If the difference is not 0 (as in our example), we compare the *difference* against a zero distribution (*null_ distribution*= 0,0,0,0,0) with a Wilcoxon-signed rank test and Cohen's effect size test to determine whether if the difference is significant or otherwise.

In addition, even if the discretization noise impacts the overall interpretation of a classifier, most researchers and practitioners care only about the top x most important features (Hassan and Holt, 2005; Lewis et al., 2013). Therefore, in this section we present the results of the top 3 features as a showcase. To measure how likely the rank of a feature could shift due to the removal of the discretization noise, we compute the likelihood of rank shifts for top 3 ranks. The importance rank of a feature is ascertained by the median rank of the computed feature importance ranks for a feature of the dataset.

We use a bootstrap analysis to compute the likelihood of rank shifts similar to prior study by Tantithamthavorn et al. (2018b). Figure 5.3 outlines the process of estimation of the likelihood of a rank shift. The feature importance ranks that are generated from each classifier in the incremental feature analysis are taken as the input. These computed ranks are then re-sampled with replacement (i.e., Bootstrap re-sampling). The bootstrap re-sampled feature rank distribution is generated for all the features of each of the studied datasets and they are re-ranked using the Scott-Knott ESD test as shown in the “Bootstrap analysis” part of Figure 5.3. This process is repeated 100 times. The intuition behind such a procedure is that the bootstrap re-sampling and re-ranking would alleviate possible minor and insignificant fluctuations in the computed feature importance ranks while highlighting the pattern of significant fluctuations in the computed feature importance ranks for a feature, and thereby bringing out its true rank. Now we have 100 computed feature importance ranks for all the features in each of the studied datasets.

These 100 ranks (the output of the “Bootstrap analysis” part of Figure 5.3) for a feature are used for estimating the likelihood of a rank shift. The likelihood of a rank shift for rank x feature is computed as the percentage of how many ranks are not equal to x. For example, for rank 1 feature, out of 100 times 2 ranks are not equal to 1, then the likelihood of a rank shift for that feature is 0.02. The estimation for the likelihood of

Table 5.4: The likelihood of rank shifts in the top 3 most important ranks (column A) and the comparison of the computed feature importance ranks of an RFCM trained on the whole dataset (Rank$_W$) and the dataset with the noisy area removed (Rank$_{NR}$) (column B).

Dataset	**Rank shift likelihood (A)**			**Rank$_W$ vs. Rank$_{NR}$ (B)**	
	Rank 1	**Rank 2**	**Rank 3**	**p-value**	**Cohen's d**
Stack Overflow	0	0	0	0	-1.29 (L)
Mathematics	0	0	0	0	-2.14 (L)
Ask Ubuntu	0	0	0	0	-2.84 (L)
Super User	0	0	0	0.01	-0.74 (M)
Patch	0	0	0	0.03	-0.62 (M)
Bug-delay	0	0	0	0	-1.39 (L)
App-rating	0	0	0	1	-0.30 (S)

rank shifts is done for the features in the top 3 ranks of all the studied datasets. If the likelihood values are high for a particular rank in a dataset, it indicates that discretization noise impacts the computed feature importance and the feature(s) reported at that rank should be interpreted with caution in that dataset.

Results: The overall computed feature importance ranks are impacted by the discretization noise for most of the studied datasets. We only report the impact of discretization noise generated by median based discretization threshold (MT) on the computed feature importance ranks for RFCM due to space constraints, However similar results are noted on all the other classifiers (LR, CART, KNN) on all the studies discretization thresholds (please refer to Table B.3, B.4, and B.5 of Appendix for other results). We show a comparison of the computed feature importance ranks of an RFCM that is trained on the dataset with and without the noisy area removed in Table 5.4. The results highlight that for most of the studied datasets (6/7 datasets in the case of the RFCM classifier and for all the datasets for other classifiers), the overall computed feature importance ranks of all the features in a dataset, are significantly impacted (i.e., $p-$value < 0.05) with a non-negligible effect size.

However, the computed feature importance ranks of the top 3 features are not impacted by the discretization noise. To specifically understand how much of an impact that the discretization noise has on the most important features, we computed the likelihood of a rank shift for the top 3 important features. We present the results in Table 5.4. **For all the datasets, the ranks of the most important features (i.e., all features at rank 1, rank 2 and rank 3) are not impacted due to discretization noise**. We further observe that these trends are not specific to RFCM and the discretization noise generated by MT. From Table B.3, B.4, and B.5 (please see Appendix) we observe that the most important features are not impacted by the discretization noise generated in the dataset for any of the studied classifiers.

In summary, though the overall ranks of computed feature importance are impacted by the discretization noise in the dataset, the top 3 (most important) features that most researchers and practitioners focus on (Hassan and Holt, 2005; Lewis et al., 2013; Rajbahadur et al., 2017; Tantithamthavorn et al., 2015) are not impacted by the discretization noise. **Therefore, we suggest that the decision of either including or removing the data in the noisy area could be exclusively arrived at from the results of Section 5.4.1 without being worried much about its impact on interpretation**. Nevertheless these results might vary for other settings (e.g., other datasets or classifiers) and our framework is able to provide a case by case guidance.

The discretization noise (generated by our studied discretization thresholds) does not impact the computed feature importance of any of the top 3 features yet it impacts the overall computed feature importance ranks.

Table 5.5: Median performance (AUC) of the RFCMs on the different regions of the data.

Training data → Testing data	**Stack Overflow**	**Mathematics**	**Ask Ubuntu**	**Super User**	**Patch**	**Bug-delay**	**App-rating**
Noisy area → Extremes	0.96	0.96	0.86	0.86	0.98	0.69	0.76

5.5 Discussion

5.5.1 Why does a classifier trained on the whole dataset (with discretization noise) sometimes perform better than the classifier trained on data devoid of discretization noise?

From the Section 6.4, we observe that excluding data from the noisy area, i.e., data with discretization noise, sometimes negatively impacts the performance of a classifier. We seek to understand this counter intuitive phenomena. Furthermore, in this section, we also remark about why the top 3 important features of a classifier are not impacted by discretization noise.

We hypothesize that a classifier in some cases is able to capture the signal from the noisy points in spite of the discretization noise. Doing so, allows the classifier to capture more information from the noisy data points, in addition to information available from the clean data. Therefore, discarding those noisy points negatively impacts the performance of a classifier. To examine our hypothesis, we start by constructing classifiers with data points from the noisy area (generated with MT) and test them on the data from the extremes and noisy areas. Such an experiment helps us understand whether the data with the discretization noise contains any useful information.

We follow the steps outlined in our framework (see Section 5.3) to construct the classifiers on the noisy area and generate the out-of-sample test sets from the extremes and noisy area separately. For this experiment, we construct an RFCM and observe its performance on the AUC measure. We do so just on RFCM as our intention is only to analyze if the noisy area contains useful information and our results on RFCM could help us test it succinctly. Furthermore, in this section we report only the AUC measure as from Table 5.3 we could observe that AUC measure is the most resilient performance measure that has the least impact across all the studied classifiers and we wish to report the impact on the most resilient measure. A high AUC would therfore objectively justify the presence of useful information. However, all the other performance measures follow the same trend.

The noisy area contains useful information that might help improve the performance of RFCMs. From Table 5.5, we observe that the RFCMs that are trained on the noisy area perform extremely well on the extremes for 5 out of the 7 studied datasets. Three datasets have an AUC that is larger than 0.95. For instance, in the Stack Overflow dataset, the RFCMs that are trained on the noisy area have an AUC of 0.96 when tested on the extreme areas.

While the previous experiments show that noisy area contains useful information, we cannot conclusively establish if the contained information in the noisy points amidst the discretization noise could be successfully used by the classifiers. To test if the studied classifiers can use the information contained in the noisy area, we construct classifiers that are trained with extremes and data from the noisy area (in contrary to the previous experiment, where we trained only on the noisy area) and then add increasing amounts of data from the noisy area. If the performance of a classifier does not degrade significantly (some degradation should be expected) with the increased amount of noise, it may indicate that the classifier is capable of capturing a signal as long as there is enough information in the data. Therefore, we could infer that despite the presence of discretization noise, the data points in the noisy area provide an additional signal to the classifier and the exclusion of the noisy area, negatively impacts the performance of the classifier. On the other hand, if there is a drastic and significant performance degradation of the classifier, it would then invalidate our hypothesis that classifiers are able to learn an additional signal in spite of discretization noise.

We perform the experiment by setting up a simulation study with the help of our framework similar to the previous experiment, where we train all the classifiers on the extremes with different amount data from the noisy area. We train our classifiers on four different data configurations which are given by $(extremes + (noisyarea + over\_sample\% * (noisyarea))$, where the $over\_sample$ takes values of $0, 100, 200, 300$. We oversample different amounts of data from the noisy area while keeping the amount of the data from extremes constant. We build the classifiers on four data configurations i.e., $(extremes + (noisyarea + 0\% * (noisyarea))$, $(extremes + (noisyarea + 100\% * (noisy\_area))$, $(extremes + (noisyarea + 200\% * (noisy\ area))$, and $(extremes + (noisyarea + 300\% * (noisy\ area))$. For instance, in the Ask Ubuntu dataset from our study for MT, the extremes have 1,427 data points and the noisy area has 2,711 data points as shown in the Table 5.2. Therefore for the first configuration, we would have 4,138 data points, of which 65% is comprised of noisy points. Therefore for Ask Ubuntu dataset, our four data configuration consist of 65%, 79%, 85% and 88% noisy points respectively along with the clean data from the extremes. (See Figure B.2 of Appendix explaining the overall experimental setup)

The performance of a classifier that is constructed on the aforementioned data configurations is evaluated on the out-of-sample test data from the extremes. Note that the out-of-sample test data that is obtained from the extremes, is not used in the training phase and is only used for testing the constructed classifier. The experimental setup for constructing a classifier is similar to that of our framework, as outlined in Section 5.3. Finally, we also capture the computed feature importance ranks and observe the likelihood of rank shifts for the top 3 most important features as outlined in Section 5.3 and Section 5.4.2.

Adding data from the noisy area to the training data does not greatly impact the performance of a classifier. We report the results in Table 5.6. The columns in Table 5.6 correspond to different data configurations that we discussed earlier. We observe that the median AUC of a classifier that is trained on the dataset without noise is quite close to the AUC of a classifier that is trained on data with 300% noise. For RFCM, LR and CART classifiers, even an addition of 300% of data from the noisy area only impacts the AUC within 6% as we can observe from Table 5.6. Even in the case of the KNN classifier, which is an instance-based classifier that is traditionally more sensitive to noise in the data (Aha and Kibler, 1989), gets impacted only by 11% in terms of AUC even with the addition of up to 300% data points from the noisy area. These results signify that the performance of the constructed classifiers on the extremes does not degrade significantly even when there is 300% (at least X% of the data is noisy) data from the noisy area in addition to data from the extremes.

Furthermore, we also note that the likelihood of rank shifts for top 3 ranks between classifiers that are trained on the first configuration (0%) and the last configuration (300%) is 0. Which further reinforces the validity of our hypothesis that the classifiers are able to capture the signal in the noisy points despite the discretization noise and the most important features contributed by the true signal in the underlying data are not perturbed by the discretization noise.

Table 5.6: Performance comparison (in AUC) of classifiers that are trained on different data configurations.

Classifier	Dataset	0% Noise	100% Noise	200% Noise	300% Noise
RF	SO	0.96	0.96	0.96	0.96
	MA	0.96	0.96	0.96	0.96
	AU	0.86	0.85	0.84	0.84
	SU	0.86	0.86	0.87	0.85
	PT	0.99	0.99	0.99	0.99
	BD	0.69	0.68	0.66	0.65
	AR	0.76	0.77	0.76	0.76
LR	SO	0.92	0.93	0.92	0.92
	MA	0.93	0.92	0.92	0.92
	AU	0.78	0.77	0.76	0.76
	SU	0.79	0.79	0.78	0.77
	PT	0.98	0.98	0.98	0.97
	BD	0.68	0.68	0.66	0.66
	AR	0.72	0.72	0.71	0.71
CART	SO	0.89	0.86	0.84	0.83
	MA	0.87	0.86	0.87	0.85
	AU	0.74	0.69	0.67	0.66
	SU	0.72	0.70	0.69	0.68
	PT	0.94	0.89	0.90	0.90
	BD	0.64	0.62	0.60	0.60
	AR	0.64	0.63	0.62	0.62
KNN	SO	0.80	0.75	0.71	0.69
	MA	0.80	0.75	0.72	0.70
	AU	0.62	0.61	0.59	0.58
	SU	0.69	0.64	0.62	0.61
	PT	0.83	0.82	0.81	0.80
	BD	0.55	0.52	0.51	0.50
	AR	0.60	0.57	0.56	0.55

Datasets: SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BD- Bug-delay, AR- App-rating

In summary, We establish that the noisy area does contain some useful information. Further, we observe only a maximum performance drop of 11% across 7 datasets for all of the studied classifiers (with less than 6% performance drop for RFCM, LR and CART) with the addition of as much as 300% of data from the noisy area, where at least more than 67% of the dataset is noisy. This suggests that a classifier is able to capture the information in the data in spite of the noise, thereby explaining why the exclusion of data points from the noisy area sometimes impacts the performance of a classifier. In addition, the likelihood of rank shift for the top 3 most important features is 0 for all the classifiers, signifying that the discretization noise even in such high quantities does not impact the interpretation of the classifiers.

5.5.2 Why does inclusion of discretization noise sometimes negatively impacts the performance of a classifier?

Contrary to Section 6.5.1, in this section, we seek to understand why the inclusion of discretization noise negatively impacts the performance of some classifiers. From Table 5.3 we observe that for all the studied classifiers, the inclusion of discretization noise sometimes negatively impacts the performance of a classifier even though Section 6.5.1 shows that data in the noisy area has useful information and classifiers are capable of leveraging it. From Table 5.3 we also observe that for all the studied classifiers and datasets, at least one of the performance measure is negatively impacted. We hypothesize that such a negative impact could be due to the high complexity (less discriminative power) of the noisy points around the discretization threshold, despite containing useful information. We arrive at such a hypothesis as prior studies show

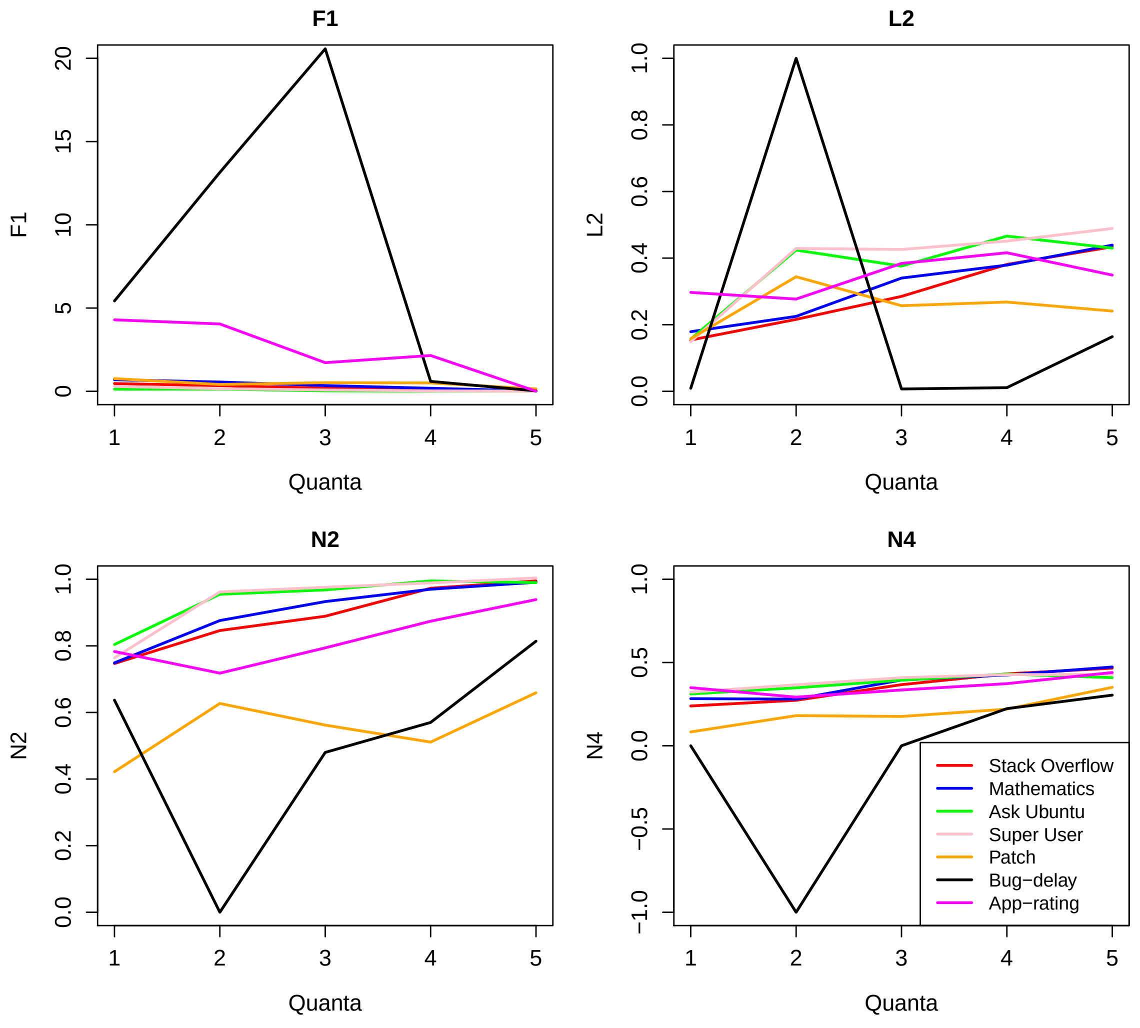


Figure 5.4: Data complexity across quantum for the studied datasets.

that it is difficult for the classifiers to perform well if the complexity of the data is high, irrespective of the contained information (Alm et al., 2005; Ho and Basu, 2002). Thus, we are interested in exploring if the negative impact in the performance of a classifier due to the inclusion of discretization noise is because of the high complexity of the data points around the discretization threshold (noisy area).

Ho and Basu (2002) provide complexity metrics to measure the complexity of data. We use these complexity metrics to measure the complexity of different regions of our data. From the multiple methods that are proposed by Ho and Basu (2002), we choose Fisher's discriminant ratio (F1), linear separability (L2), mixture identifiability (N2) and nonlinearity (N4) as they are simple to explain and easy to interpret. We briefly explain the metrics that we choose in the Table B.6 of Appendix. See the study by Ho and Basu (2002) for more details about the computation of these metrics.

We use the abovementioned measures to compute how the complexity of the dataset changes across data points as we move from the extremes to the noisy areas. We first discretize the data into "class 1" and "class 2" classes as outlined in Section 5.3.2. We then transform the continuous dependent feature using the Box-Cox transformation (Sakia, 1992) to alleviate the skew and increase the spread of the distribution of the dependent feature. We then split the data into 5 quanta for each class using the **bin** function in R. We do so to compartmentalize the data in relation to the continuous dependent feature and analyze the changes to the complexity of the data points as we move closer to the discretization threshold (we choose MT) for our case study. We would not be able to observe how the complexity changes in different areas of the data without such compartmentalization of the data. The choice of using 5 quanta is so that the compartmentalization is neither too granular nor too encompassing. The 1stquantum contains most of the data from extremes and the 5th quantum contains most of the data from the noisy area, whereas 2nd to 4th quantum roughly contain an equivalent amount of data points in between. Finally, we compute the above-mentioned data complexity metrics for the data points in each of these quantum and plot the results. (See Figure B.3 of Appendix explaining the overall experimental setup)

From Figure 5.4, we observe that as we move from the extremes (1st quantum) towards the noisy area (5th quantum), we see a steady increase in data complexity across all complexity measures for all the datasets except for the Bug-delay dataset. We see that all four complexity measures (i.e., Fischer's discriminant ratio, linear separability, mixture identifiability, and nonlinearity) are very high for the data points in the noisy area compared to the data points in the extremes, and the inclusion of such complex data makes it very hard for the classifiers to perform well. The steady increase in data complexity as we move across the quantum can be attributed to the steady increase of the discretization noise in the dataset as we move from the 1st quantum to the 5th quantum (the extremes to the noisy area). Therefore, when the discretization noise (the data points in the noisy area with high complexity) is discarded, the performance of some of the classifiers increases.

The lower complexity in the 2nd quantum for the Bug-delay dataset does not impact our findings. It is due to the way the dataset is split, the BoxCox transformation aims to spread the dependent feature sufficiently so that the class-wise binning yields data in all quantum. But for the Bug-delay dataset, when we split the data into quantum, we observe that the 2nd quantum has data points that only belong to "class 2" and not "class 1" because the quantum 2 for the Bug-delay dataset contains only data points belonging to "class 2", its complexity is very low, which is reflected in Figure 5.4. But this phenomenon has no bearing on our findings that the quantum containing high volumes of discretization noise (q5) is more complex than the quantum containing extremes data (q1) and thereby discarding them sometimes improves the performance of the classifiers.

Hence, the presence of a high volume of discretization noise in the noisy area increases the data complexity, which in turn results in the decreased performance of a classifier that is trained with discretization noise, despite containing useful information. Therefore, in some cases, the performance of a classifier benefits from discarding the data points with from the noisy area.

5.6 Guidelines for Using Our Framework

We explain in detail our framework in Section 5.3 and demonstrate how it is used to study the impact of discretization noise on the performance and interpretation of a classifier in Section 6.4. Furthermore this section, we provide practical guidelines on how to use our framework and the best practices to follow.

Figure 5.5 shows the involved steps, step-wise outputs, user considerations at each step and the overall workflow of our framework. A user can follow the steps one by one when they are given a dataset to study.

5.6.1 Performance impact estimation

A classifier constructed with increasing amounts of discretization noise being removed is constantly compared against the classifier constructed on the whole dataset with a Wilcoxon signed-rank test and a Cohen's effect size test as outlined in Section 5.4.1. If for the chosen performance measure, the impact is statistically significant with non-negligible effect size, then the amount of noisy points to be discarded and the magnitude of the performance impact due to the discretization noise is reported to the user. If the discretization noise does not impact the chosen performance measure then our

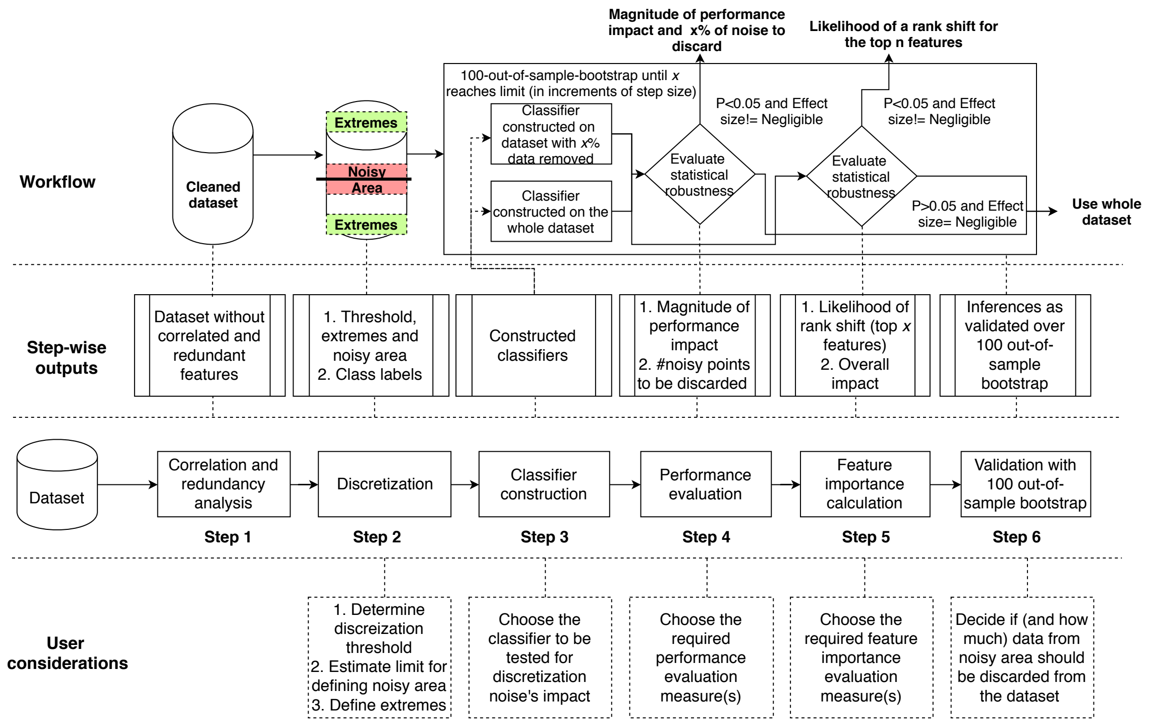

Magnitude of performance impact and x% of noise to discard
Likelihood of a rank shift for the top n features
100-out-of-sample-bootstrap until x reaches limit (in increments of step size)
Classifier constructed on dataset with x% data removed
Classifier constructed on the whole dataset
P<0.05 and Effect size!= Negligible
Evaluate statistical robustness
P>0.05 and Effect size= Negligible
Use whole dataset
Workflow
Cleaned dataset
Extremes
Noisy Area
Extremes
Step-wise outputs
Dataset without correlated and redundant features
1. Threshold, extremes and noisy area 2. Class labels
Constructed classifiers
1. Magnitude of performance impact 2. #noisy points to be discarded
1. Likelihood of rank shift (top x features) 2. Overall impact
Inferences as validated over 100 out-of-sample bootstrap
Dataset
Correlation and redundancy analysis
Discretization
Classifier construction
Performance evaluation
Feature importance calculation
Validation with 100 out-of-sample bootstrap
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
User considerations
1. Determine discreization threshold 2. Estimate limit for defining noisy area 3. Define extremes
Choose the classifier to be tested for discretization noise's impact
Choose the required performance evaluation measure(s)
Choose the required feature importance evaluation measure(s)
Decide if (and how much) data from noisy area should be discarded from the dataset

Figure 5.5: User considerations and workflow that are associated with each of the steps of our framework.

framework would output 0 (suggesting no data needs to discarded) and recommend the use of the whole dataset as outlined in the workflow of Figure 5.5.

However, the choice of the performance measure to focus on and how much of an improvement/impact that one should consider actionable depends entirely on the context. For instance, in a dataset of 100 data points, if 90 data points belong to "class 1" and 10 data points belong to "class 2", then accuracy (w.r.t "class 1") would be 90% even if the classifier always predicts "class 1" for all the 100 data points. Therefore, other balanced performance measures like AUC might be required.

5.6.2 Interpretation impact estimation

The computed feature importance ranks of the classifiers constructed on datasets with varying amounts of discretization noise being removed is computed. These computed ranks are compared to see if the computed feature importance ranks change between the classifiers that are constructed on the whole dataset and the ones that are constructed on the dataset without discretization noise (please see Section 5.4.2 and Figure 5.5). Our framework then checks if the differences of the computed feature importance ranks between the classifiers that are trained on the dataset with and without the discretization noise are statistically significant. If they are, our framework also calculates the likelihood of rank shifts for the top n features (between the classifier trained on the whole dataset and the data with the framework recommended amount of data from noisy area removed). In summary, our framework reports if there is an impact of discretization noise on the overall interpretation and the likelihood of rank shifts in for the top n features to the user.

5.6.3 Best practices

In this section, we recommend the key best practices for others to follow when they are discretizing the data using an artificial threshold. From Section 5.4.1 we note that performance of all the classifiers is impacted across all performance measures differently and that class-specific performance measures (e.g., Precision and Recall) are more sensitive to discretization noise than others. Therefore, we recommend the following best practices for the researchers and practitioners:

1. Irrespective of the choice of a classifier, if one intends to discretize the continuous dependent feature into artificial classes, one should use our framework to analyze if they should use the whole dataset or discard the discretization noise.

2. Class-specific performance measures are more sensitive to discretization noise. Therefore, instead of discarding a fixed amount of data like some of the previous studies (Wang et al., 2018; Tian et al., 2015; Hassan et al., 2018), we recommend the use of our framework to estimate how much discretization noise that one should discard to avoid any negative impacts on their performance measure of choice. Hence, one could avoid the unwanted loss of data.

3. If our framework reports a high likelihood of a rank shift for one of the top n features, we recommend not to trust the feature importance rank for that particular feature and seek the opinion of the domain expert. However, if our framework detects any impact in the overall interpretation along with the performance, then we recommend the use of the interpretation of the best performing model.

5.7 Threats to Validity

External Validity Many of the prior studies highlight that different classifiers have different performance on the same data (Rajbahadur et al., 2017; Ghotra et al., 2015). So the choice of classifiers might impact the findings of our study, as we only use four classifiers (RFCM, LR, CART, and KNN) in our analysis. However, the chosen classifiers represent a diverse range of families: statistical family, nearest neighbor family, Decision tree family and ensemble family, i.e., 4/6 of the common classifier families as outlined by Lessmann et al. (2008). We left out representative approaches from the neural networks family and the support vector machine family. We did so, as classifiers from these families typically do not have a default feature importance measure.

Construct Validity Threats to construct validity pertains to the suitability of the measures that are used in our study. In our study, we study the impact of discretization noise that is generated in the dataset by using three different discretization thresholds as mentioned in Section 5.3 and the results might vary when another threshold is used. However, the three chosen discretization threshold computation methods (MT, CT, RTT) discretize the dependent features differently and represent the most common ways of unsupervised discretization of the dependent feature. Our framework enables others to explore other discretization thresholds in a systematic manner.

In this chapter, we only observe the impact of discretization noise on the performance and interpretation of hyperparameter tuned classifiers. However, as Tantithamthavorn et al. (2018b) highlight in their study, many prior software analytics studies do not hyperparameter tune their classifiers. Therefore, the magnitude of the impact of discretization noise on the performance and interpretation of classifiers whose hyperparameters were not tuned remains unknown. We suggest that future studies should use our framework to investigate the impact of discretization noise on the classifiers whose hyperparameters are not tuned. Such an investigation could shed light on which of the insights from the prior studies that use an artificial threshold to discretize the continuous dependent feature needs to be revisited.

Another construct validity in our study is the choice of the $limit$ parameter for deciding the size of the noisy area in each dataset. We used the limit values that are generated by our automated noisy area estimation algorithm as given in Section 5.3.

Though such an algorithm estimates the noisy area quantitatively based on a complexity measure, it might not consider the inherent dataset characteristics and bias. We acknowledge that it could be a potential threat and we urge researchers to explore various limits, or with limits that are established by domain experts and ratify our findings. Future studies should use our framework to test different values for the limit.

The other construct validities of our study pertains to the statistical tests that we use in our study. In our study, though we conduct multiple statistical tests after removing different amounts of noisy data points to ascertain the impact of discretization noise, we do not perform any p-value correction. We acknowledge that our statistical procedure (without any p-value correction) might produce falsely inflated results and is in turn a potential threat to the validity of our findings. To mitigate such a threat we use the effect size to quantify the magnitude of the observed impact of discretization noise. Nevertheless, future studies that use our framework to decide the amount of noisy data points to discard could benefit from correcting the p-values using a procedure such as Bonferroni correction (Bonferroni, 1936). Also, we use Cohen's d effect size test, which is a parametric test that assumes the groups it tests to be normally distributed. However, we do not ensure if the obtained performance scores are normally distributed before using the Cohen's d effect size test. We acknowledge that this is a threat and future studies should revisit our findings by using a non-parametric effect size test.

Finally, in this section, we wish to reiterate to the readers that our framework enables the researchers and practitioners to fiddle with any components and try a variety of combinations. We only define the needed analysis that is to be done, so that the drawn observations are valid.

5.8 Chapter Summary

In this Chapter, we propose a framework to systematically and rigorously analyze the impact of discretization noise on the performance and interpretation of a classifier within the context of their own domain. We perform a case study on a variety of software analytics datasets and we find that:

1. Discretization noise impacts the different performance measures of classifiers differently across the different datasets. We observe that discretization noise leads to an up to 139% performance differences across various performance measures across all the studied classifiers. Hence it is very important for researchers and practitioners to use our framework to analyze the impact of discretization noise on the classifier's for before either including or discarding it in their analysis.

2. When discretization noise negatively impacts the performance of a classifier, our framework provides a systematic and statistically robust way to estimate exactly how much data should be discarded to avoid discretization noise without incurring unwarranted data loss.

3. Though discretization noise impacts the overall computed feature importance ranks of a classifier, it does not impact the computed feature importance ranks of the top 3 ranks for our case studies. Our framework provides a case by case guidance for others who wish to explore its use for their own case studies.

R package and User Guideline: We provide an R package to enable others to use our framework to analyze the impact of discretization noise. Furthermore, we provide a user guideline, a step-by-step walkthrough and the best practices of using our framework in Section 5.6.

CHAPTER 6

The Impact of Interchangeably Using Feature Importance Methods

Classifier specific (CS) and classifier agnostic (CA) feature importance methods are widely used (often interchangeably) by prior studies to derive feature importance ranks from a classifier. However, different feature importance methods are likely to compute different feature importance ranks even for the same dataset and classifier. Hence such interchangeable use of feature importance methods can lead to conclusion instabilities unless there is a strong agreement among different methods. Therefore, in this Chapter, we evaluate the agreement between the feature importance ranks associated with the studied classifiers through a case study of 18 software projects and six commonly used classifiers. We find that: For a given classifier and a dataset, 1) the feature importance ranks computed by the CS and CA methods do not always strongly agree with each other; 2) the feature importance ranks produced among the two studied CA methods also vary s'ignificantly. Furthermore, 3) on a given dataset, the more commonly used CS methods produce vastly different feature importance ranks, even when considering the most important feature. In addition, we find that the top-3 features computed by different feature importance methods (though are different) are similarly discriminative (i.e., yielded similar classification capability). We show that such a result could possibly be due to the feature interaction that exists in the datasets. In light of our findings, we recommend that 1) When replicating a study, one should use the same feature importance method as the original

study since the use of a different method is likely to lead to different conclusions; 2) one should specify the exact feature importance method that they use in a study; 3) report the feature interactions along with the feature importance ranks if the dataset exhibits feature interactions.

An earlier version of this chapter is currently under review in Transactions on Software Engineering journal.

6.1 Introduction

As we highlight in Chapter 4, defect classifiers are widely used by many large software corporations (Lewis et al., 2013; Zimmermann et al., 2009; Caglayan et al., 2015; Shihab et al., 2012) and researchers (Zhang et al., 2016; Shihab et al., 2011; Chen et al., 2018). Defect classifiers are commonly interpreted to uncover insights to improve software quality. Such insights help practitioners formulate strategies for effective testing, defect avoidance, and quality assurance (Jiarpakdee et al., 2019; Theisen et al., 2015). Therefore it is pivotal that these generated insights are reliable.

When interpreting classifiers, prior studies typically employ either as CS or CA method to compute the feature importance ranks (Jiarpakdee et al., 2019; Herzig et al., 2016; Guo et al., 2004; Jahanshahi et al., 2019; Mori and Uchihira, 2019) (as we detailed in Chapter 2). CS methods typically make use of a given classifier's internals to measure the degree to which each feature contributes to a classifier's predictions. For instance, prior studies use Type 1/2 ANOVA to compute the feature importance ranks from logistic regression classifiers (Bird et al., 2011, 2009; Nagappan et al., 2006; Nagappan and Ball, 2005; Zimmermann et al., 2007). Similarly, other studies use the Gini importance method to derive feature importance ranks from random forest classifiers (Gousios et al., 2014; Guo et al., 2004; Kamei et al., 2012). We note, however, that a CS method is not always readily available for a given classifier. For example, complex classifiers like SVMs and deep neural networks do not have a widely accepted CS method Chakraborty et al. (2017).

For cases such as those, or when a universal way of comparing the feature importance ranks of different classifiers is required (Rajbahadur et al., 2017; Tantithamthavorn et al., 2018a), CA methods are used. Such CA methods measure the contribution of each feature towards a classifier's predictions as we explain in Chapter 2. The primary advantage of CA methods is that they can be used for any classifier (i.e., from interpretable to black box classifiers).

Despite computing feature importance ranks using different ways, CS and CA methods are indiscriminately and interchangeably used in software analytics studies (Table 6.1). For instance, to compute feature importance ranks for a random forest classifier, Treude and Wagner (2019) and Yu et al. (2019) use CS methods: the Gini importance and the Breiman's importance methods respectively. On the other hand, Mori and Uchihira (2019) and Herzig (2014) use CA methods: Partial Dependence Plot (PDP) and filterVarImp respectively. Since these methods compute feature importances differently (see Section 6.3.3), different CS or CA methods are likely to compute different feature importance ranks for the same classifier. Yet, we observe that the rationale for choosing a given feature importance method is rarely motivated by prior studies (Section 6.2).

The interchangeable use of feature importance methods is acceptable only if the feature importance ranks computed by these methods do not differ from each other. For instance, consider the study Ghaleb et al. (2019), where they use mixed-effect logistic regression classifier and ANOVA to find that caching, rerunning failed commands and time of the builds are the most important features that determine the long build duration in a continuous integration setting. Consider if one replicates the study by constructing the same mixed-effect logistic regression classifier. However, instead of using ANOVA, they interpret the coefficients of the mixed effect logistic regression classifier and find that caching is not as important anymore. Such insight would be reflective of how the feature importance method calculates the feature importance ranks rather than identifying the features that drive the empirical relationship and raises concerns about the stability of conclusion across the replicated study.

Therefore, in order to determine the extent to which the importance ranks computed by different importance methods agree with each other, we conduct a case study on 18 commonly used software defect datasets using classifiers from six different families. We compute the feature importance ranks using six CS and two CA methods on these datasets and classifiers. The list of CS methods is summarized in Table 6.4. The two CA methods are: permutation importance (Permutation) and partial dependence plot importance (PDP). Finally, we compute Kendall's Tau, Kendall's W, and Top-k ($k \in \{1, 3\}$) overlap to quantify the agreement between the computed feature importance ranks by the different studied feature importance methods for a given classifier and dataset. While Kendall's measures compute differences across the different feature importance ranks, the Top-K overlap measure focuses on the top-k items of these rankings (more details in Section 6.3.5). We answer the following research questions:

- **RQ1: For a given classifier, how much do the computed feature importance ranks by CA and CS methods differ across datasets?** *The computed feature importance ranks by CA and CS methods do not always strongly agree (i.e., Kendall's $|\tau| \leq 0.6$ (refer Section 6.3.5)) with each other.* In particular, even the most important feature tends to differ for two of the six studied classifiers.

- **RQ2: For a given dataset and classifier, how much do the computed feature importance ranks by the different CA methods differ?** *The two studied CA methods produce significantly different feature importance ranks.* The feature importance ranks computed by the PDP and Permutation CA methods do not have a large median Top-3 overlap (i.e., top-3 overlap > 0.75 (refer Section 6.3.5)) on any of the 16 studied datasets.

- **RQ3: On a given dataset, how much do the computed feature importance ranks by different CS methods differ?** *Different CS methods produce vastly different importance ranks.* We observe that on a given dataset, none of the feature importance ranks produced by the different CS methods strongly agree with each other (i.e., Kendall's $|\tau|$ ≤ 0.6). The top-3 overlap between CS methods is only small (i.e., top-3 overlap ≤ 0.25)) at best and occurs in only three of the studied datasets.

Based on the aforementioned results, we hypothesized whether any specific feature importance method produces a top-3 feature set that is more discriminative (i.e., yielded higher classification capability) than the top-3 features produced by the other methods. Such a result would indicate if any of the studied methods consistently computed best top-3 features and is inherently better than the other studied methods. However, we observed that the top-3 features computed by CA and CS methods are all equally discriminative (even though each of them compute different top-3 features) for a given classifier, thereby indicating that none of the studied feature importance methods are inherently superior or inferior to one another. Therefore to investigate why different feature important methods compute different top-3 features, that are equally discriminative, we examined if the independent features of our studied datasets interact. We do so as several prior studies (de González et al., 2007; Freeman, 1985; Fisher et al., 2018; Devlin et al., 2019; Lundberg et al., 2018) indicate feature interactions might impact the feature importance ranks computed by different feature importance methods. We find that, the presence of feature interactions in the studied datasets, also acts as a confounder affecting the computed feature importance ranks. Therefore, both the feature importance methods themselves and the feature interactions play a role in the computed rankings. In light of these findings, we suggest that future research on defect classification should:

1. When replicating a study in a new context, one should employ the same feature importance method and learner as the original study.

2. One should always specify the used feature importance method to increase the reproducibility of their study and the generalizability of its insights.

6.2 Motivation

In this section, we motivate our study based on how prior studies employed feature importance methods.

We conduct a literature survey of the used feature importance methods in prior studies. To survey the literature, we searched Google Scholar with the terms “software engineering", “variable importance", “feature importance” and the name of each classifier that is studied in this Chapter (Section 6.3.4). We searched the Google Scholar multiple times, once for each studied classifier. We eliminated all the papers that were from before the year 2000 to restrict the scope of our survey to recent studies. We read each paper from the search results in order to check if they employed any feature importance method(s) to generate insights. We consider all the studies presented in the

Table 6.1: Different feature importance methods used for interpreting various classifiers in the software engineering literature

Classifier Family	**Papers using CS**	**Used CS methods**	**Papers using CA**	**Used CA methods**
Statistical Techniques	(Subramanyam and Krishnan, 2003)Herbsleb and Mockus (2003)✓(Zimmermann and Nagappan, 2008; Angelis et al., 2001; Mori and Uchihira, 2019; Nagappan and Ball, 2005; Morales et al., 2015; Ghaleb et al., 2019; Kononenko et al., 2015)	Regression coefficients, ANOVA	(Calefato et al., 2019; Herzig, 2014; Herzig et al., 2016; Premraj and Herzig, 2011)	Boruta✳, filterVarImp†
Rule-Based Techniques	(Gay et al., 2010)✓,(Othmane et al., 2017)	Interpreting rules, varImp✇	(Calefato et al., 2019)	Boruta✳
Neural Networks	(Santos and Belo, 2013)✖ (Ma et al., 2018)✓	MODE(Ma et al., 2018)	(Calefato et al., 2019)	Boruta✳
Decision Trees	(El-Emam et al., 2001)(Knab et al., 2006)✓(Malgonde and Chari, 2019)	Decision branches, Gini importance	(Calefato et al., 2019; Herzig, 2014; Herzig et al., 2016; Premraj and Herzig, 2011)	Boruta✳, filterVarImp†
Ensemble methods-Bagging	(Treude and Wagner, 2019; Yu et al., 2019; Haran et al., 2007) (Guo et al., 2004)✖ (Gousios et al., 2014; Niedermayr and Wagner, 2019) (Martens and Maalej, 2019)✖ (Fan et al., 2018; Bao et al., 2019; Jahanshahi et al., 2019) (Dey and Mockus, 2018)✖	Permutation importance, Gini importance	(Mori and Uchihira, 2019; Calefato et al., 2019; Herzig, 2014; Herzig et al., 2016; Premraj and Herzig, 2011; Dehghan et al., 2017; Blincoe et al., 2019)	Boruta✳, filterVarImp†, PDP, Marks method(Marks et al., 2011), BestFirst★
Ensemble methods-Boosting	-	-	(Calefato et al., 2019; Herzig, 2014; Herzig et al., 2016)	Boruta✳, filterVarImp†

✖ - The used method for computing the feature importance ranks is not mentioned
✓ - Papers in which the rationale for choosing a given feature importance method is specified
✳ - https://cran.r-project.org/web/packages/Boruta/index.html
† - https://www.rdocumentation.org/packages/caret/versions/6.0-84/topics/filterVarImp
★ - https://www.rdocumentation.org/packages/FSelector/versions/0.31/topics/best.first.search
✇ - https://www.rdocumentation.org/packages/caret/versions/6.0-84/topics/varImp

google scholar and do not filter based on venues. However, we do not include the papers in which the either author or the supervisor of the thesis was involved to avoid potential confirmation bias. A summary of our literature survey is shown in Table 6.1.

We observe that studies rarely specify the reason for choosing their feature importance method – only four out of the 29 surveyed studies provide a rationale for choosing their used feature importance method. We also note that both CA and CS methods are widely used. For instance, from Table 6.1, we see that both Gini importance and filterVarImp have been used to interpret a random forest classifier. Although the specific reason for choosing a given feature importance method over another is typically not specified, we observe that CA methods are generally used in studies with multiple classifiers. In turn, CS methods are used in studies with a single classifier or a small number of classifiers. However, given that feature importance methods are (i) typically used to generate insights and that different methods (ii) compute the feature importance using different approaches, such interchangeable usage of methods on a given classifier in prior studies is troublesome.

For instance, Zimmermann and Nagappan (2008) used an F-Test on the coefficients of a logistic regression classifier (a CS method) to show that there exists a strong empirical relationship between Social Network Analysis (SNA) metrics and the defect proneness of a file. Later, Premraj and Herzig (2011) used filterVarImp (a CA method) and logistic regression classifier to show that empirical relationship between SNA metrics and the defective files are negligible. Given that CS and CA methods can produce different feature importance ranks, it is unclear whether the aforementioned conflicting result is due to absence of an empirical relationship in the data or simply due to the differing feature importance methods and as such leads to conclusion instability.

More generally, the interchangeable use of feature importance methods (i.e., CS and CA methods), when replicating a study, is acceptable only if the computed ranks correlate reasonably well. That is, if the feature importance methods that are being interchangeably used, assign similar feature importance ranks to the different features of a given dataset and not differ greatly. If the different feature importance methods compute vastly different feature importance ranks on a given dataset, it raises concerns about the stability of conclusions across the replicated studies. Hence, we investigate the following research question:

(RQ1) For a given classifier, how much do the computed feature importance ranks by CA and CS methods differ across datasets?

Similar concerns exist regarding the interchangeable use of different CA methods, even for the same classifier. From Table 6.1, we observe that, within each classifier family, different studies use different CA methods. The rationale for choosing a given CA method (for instance, filterVarImp) over another (for instance, PDP) is rarely provided. For instance, none of the studies using a CA method in Table 6.1 provide reasons for choosing one CA method over another. Yet, the extent to which these CA agree with each other is unclear. Such a concern becomes particularly relevant with the recent rise of complex classifiers for defect prediction (Dam et al., 2018; Chen et al., 2018; Hihn and Menzies, 2015), as these classifiers do not have a universally agreed-upon or popular CS method. That is, CA methods are commonly employed to compute feature importance ranks in those cases. Hence, we investigate the following research question:

(RQ2) For a given dataset and classifier, how much do the computed feature importance ranks by the different CA methods differ?

A number of general prior studies already note that feature importance ranks differ vastly between CS methods. For example, Strobl et al. (2007) show that different CS methods for random forest classifiers yield different feature importance ranks. However, such a comparison among CS methods pertaining to different classifier (i.e., CS method associated with a decision tree classifier and a random forest classifier) has not been studied, in the context of defect prediction and software engineering. Such a study is extremely important to understand the limits of reproducibility and generalizability of prior studies.

For instance, Jahanshahi et al. (2019) replicate the study of McIntosh and Kamei (2017) using random forest (and the CS methods of random forest classifier) as opposed to the non-linear logistic regression classifier and its associated CS method as the original study does. Jahanshahi et al. (2019) observe that their feature importance ranks differ from those of the original study. They claim that the size feature might not be as important for just-in-time defect prediction. Unless we know that the feature importance ranks produced by the different feature importance methods do not differ greatly, we cannot ascertain which of the two studies produced the correct insight. In particular, different CS methods are likely to compute feature importances differently and the difference in insight could be attributed to the used CS method rather than the underlying phenomena (e.g., just-in-time defect prediction) that is being studied. Therefore, we study the following research question along with the previous ones:

(RQ3) On a given dataset, how much do the computed feature importance ranks by different CS methods differ?

6.3 Case Study Setup

In this section, we describe our case studied datasets (Section 6.3.1), studied classifiers (Section 6.3.2) and feature importance methods (Section 6.3.3). Following which, we detail our study approach (Section 6.3.4), as well as the evaluation metrics that we employ (Section 6.3.5).

6.3.1 Studied datasets

We use the software project datasets from the PROMISE repository Sayyad Shirabad and Menzies (2005). The data set contains the defect data of 101 software projects that are diverse in nature. Use of such varied software projects in our study helps us successfully mitigate the research bias identified by Shepperd *et al.*Shepperd et al. (2014); Tantithamthavorn et al. (2016). In addition, similar to the prior study by Tantithamthavorn *et al.*Tantithamthavorn et al. (2018a), we further filter the datasets to study based on two criteria mentioned by Tantithamthavorn et al. (2018a). We remove the datasets with EPV less than 10 and the datasets with defective ratio less than 50. After filtering the 101 datasets from PROMISE with the aforementioned criteria, we end up with 18 datasets similar to Tantithamthavorn et al. (2018a) for our study. Table 6.2 shows various basic characteristics about each of the studied datasets in this study.

6.3.2 Studied classifiers

We construct classifiers to evaluate our outlined research questions from Section 6.2. We choose the classifiers based on two criteria. First, the classifiers should be representative of the eight commonly used machine learning families in Software Engineering

Table 6.2: Overview of the datasets studied in our case study

Project	**DR**	**#Files**	**#Fts**	**#FACRA**	**EPV**
Poi-3.0	63.5	442	20	12	14.05
Camel-1.2	35.53	608	20	10	10.8
Xalan-2.5	48.19	803	20	11	19.35
Xalan-2.6	46.44	885	20	11	20.55
Eclipse34_debug	24.69	1065	17	10	15.47
Eclipse34_swt	43.97	1485	17	9	38.41
Pde	13.96	1497	15	6	13.93
PC5	27.53	1711	38	13	12.39
Mylyn	13.16	1862	15	7	16.33
Eclipse-2.0	14.49	6729	32	9	30.47
JM1	21.49	7782	21	7	79.62
Eclipse-2.1	10.83	7888	32	9	26.69
Prop-5	15.25	8516	20	12	64.95
Prop-4	9.64	8718	20	12	42
Prop-3	11.49	10274	20	12	59
Eclipse-3.0	14.8	10593	32	9	49
Prop-1	14.82	18471	20	13	136.9
Prop-2	10.56	23014	20	13	121.55

DR: Defective Ratio, FACRA: Features After Correlation and Redundancy Analysis, Fts: Features litterature as given by Ghotra et al. (2015), to foster generalizability and applicability of our results. Second, the chosen classifiers should have a CS method. We only choose classifiers with CS method so that we can compare and evaluate the computed feature importance ranks by the CA methods against the CS feature importance ranks for a given classifier (RQ1) and the computed feature importance ranks between different classifiers (RQ3). After the application of these criteria, we eliminate three machine learning families (clustering based classifiers, support vector machines, and nearest Neighbour), as the classifiers from those families do not have a CS method. Furthermore, we split the ensemble methods family given by Ghotra et al. (2015) into two categories to include classifiers belonging to both bagging and boosting families.

Table 6.3 shows the studied classifiers and machine learning families to which they belong. We choose one representative classifier from each of the machine learning family from the **caret**[1] package in R. Table 6.3 also shows the caret function that was used to build the classifiers. The selected classifiers have a CS method, that is given by the **varImp()** function in the caret package.

Inherently-interpretable classifiers (e.g., fast-and-frugal trees and simple decision trees) do not benefit as much from feature importance methods. Hence, such classifiers are out of the scope of this study. Nevertheless, we strongly suggest that the future studies should also explore the reliability of the insights that is derived from simple interpretable classifiers.

6.3.3 Studied feature importance methods

[1]https://cran.r-project.org/web/packages/caret/index.html

Table 6.3: Studied classifiers and their hyperparameters

Family	**Classifier**	**Caret method**	**Hyperparameters**
Statistical Techniques	Logistic Regression	glm	None
Rule-Based Techniques	C5.0 Rule-Based Tree	C5.0Rules	None
Neural Networks	Neural Networks (with model averaging)	avNNet	size, decay, bag
Decision Trees	Recursive Partitioning and Regression Trees	rpart	K, L, cp
Ensemble methods- Bagging	Random Forest	rf	mtry
Ensemble methods- Boosting	Extreme Gradient Boosting Trees	xgbTree	nrounds, max_depth, eta, gamma, colsample_bytree, min_child_weight, subsample

Classifier Specific feature importance (CS) methods

The CS methods typically make use of a given classifier's internals to compute the feature importance scores. These methods are widely used in software analytics to compute feature importance ranks as evidenced from Table 6.1.

We use six CS methods that are associated with the six classifiers that we study. Table 6.4 provides a brief explanation about the inner working of these CS methods on a given classifier. For a more detailed explanation we refer the readers to Kuhn (2012).

Classifier Agnostic feature importance (CA) methods

CA methods compute the importance of a feature to the classifications of a given classifier by treating the classifier as a "black-box", i.e., without using any classifier specific details. In this study, we use the permutation feature importance (Permutation) and Partial Dependence Plots feature importance (PDP) CA methods. We use these two methods as they are commonly used by prior studies Tantithamthavorn et al. (2018a); Mori and Uchihira (2019); Avati et al. (2018); Janitza et al. (2013). We use Permutation CA method instead of the other CA methods highlighted in Table 6.1of the chapter because, Permutation method is more widely used in both software engineering and other communities than the other CA methods.

Partial Dependence Plot feature importance (PDP). We use the method outlined by Greenwell *et al.*Greenwell et al. (2018). The PDP computes a curve which depicts, how each feature affects the outcome probability, as each feature varies over its marginal distribution (over all the values of other features in the dataset) Goldstein et al. (2015); Greenwell (2017). Greenwell *et al.*Greenwell et al. (2018) later showed the association between the flatness of the PDP curve and the importance of the feature. The flatter

Table 6.4: Brief explanation about the working of caret's CS methods that are used in our study.

CS method	**Brief explanation**
Logistic Regression FI (LRFI)	Classifier coefficient's t-statistic is reported as the feature importance score
C5.0 Rule-Based Tree FI (CRFI)	The number of training data points that are covered by the leaf nodes, created from the split of a feature is given as the feature importance score for that feature. For instance, the feature that is split in the root node will have a 100% importance as all the training samples will be covered by the terminal nodes leading from it.
Neural Networks (with model averaging) FI (NNFI)	The feature importance score is given by combining the absolute weights used in the neural network
Recursive Partitioning and Regression Trees FI (RFI)	The feature importance score is given by the sum of the reduction in loss function that is brought about by each feature at each split in the tree.
Random Forest FI (RFFI)	Average of difference between the Out-of-Bag (OOB) error for each tree in the forest where none of the features are permuted and the OOB error where each of the features is permuted one by one. The feature permutation's impact on the overall OOB error is reported as the feature importance score
Extreme Gradient Boosting Trees FI (XGFI)	Feature importance score is given by counting the number of times a feature is used in all the boosting trees of the xgboost tree.

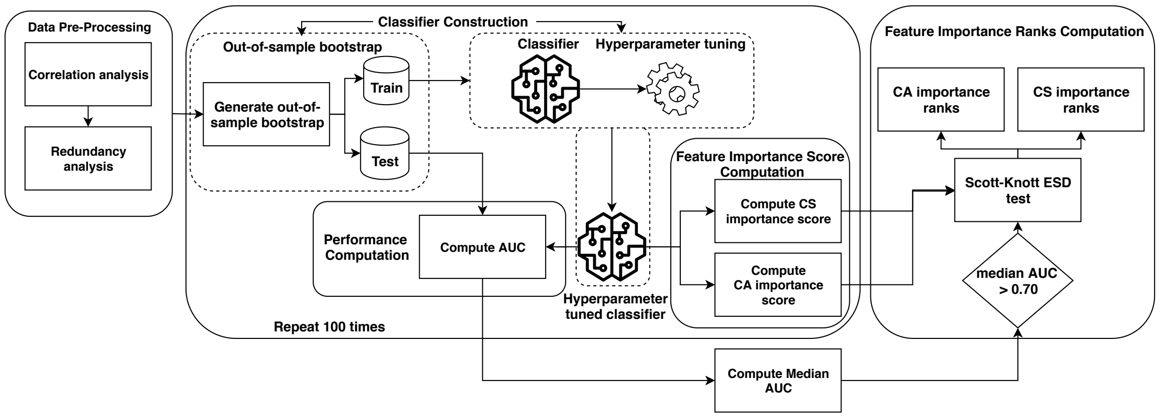


Figure 6.1: Overview of our case study approach.

the curve is, the lesser the importance of that feature is and vice versa. Therefore, the PDP feature importance method, ascertains the importance of each feature, on a given classifier, by computing the flatness of the associated PDP curve. We use the *vip* [2] R package for computing the feature importance scores in our study.

Permutation feature importance (Permutation). We use the same permutation feature importance method used in Section 4.2.6 of Chapter 4.

6.3.4 Approach

Figure 6.1 provides an overview of our case study approach. We use this approach to answer all of our aforementioned research questions in Section 6.2.

Data pre-processing

Correlation and redundancy analysis. We perform correlation and redundancy analysis on the independent features of the studied defect datasets, since the presence of

[2] https://cran.r-project.org/web/packages/vip/index.html

correlated or redundant features impacts the interpretation of a classifier and computes unstable feature importance ranks (Jiarpakdee et al., 2019; Tantithamthavorn and Hassan, 2018; Harrell Jr, 2015)

Classifier construction

Out-of-sample bootstrap. To ensure the statistical validity and robustness of our findings, we use an out-of-sample bootstrap method similar to Section 4.2.7 of Chapter 4 with 100 repetitions to construct the classifiers (Tantithamthavorn et al., 2017; Efron, 1983). More specifically, for each studied dataset, every classifier is trained 100 times on the 100 resampled *train* sets, then these classifiers are used for computing the 100 feature importance scores. The performance of these trained classifiers are also evaluated on the 100 out-of-sample *test* sets.

Classifier construction with hyperparameter tuning. Several prior studies (Fu et al., 2016; Tantithamthavorn et al., 2018a) show that hyperparameter tuning is pivotal to ensure that the trained classifiers fit the data well. Furthermore, Tantithamthavorn et al. (2018a) show that feature importance ranks shift between hyperparameter tuned and untuned classifiers. Therefore, we tune the hyperprarameters for each of the studied classifiers using random search (Bergstra and Bengio, 2012) in every bootstrap iteration using `caret` R package (Kuhn et al., 2008). The specific hyperparameters of the studied classifiers are given in Table 6.3.

Performance computation

Similar to several prior studies (Ghotra et al., 2015; Lessmann et al., 2008; Rajbahadur et al., 2019), and as advised by Tantithamthavorn and Hassan (2018), we use AUC (Area

Under the Receiver Operator Characteristic Curve) to measure the performance of our classifiers. Unlike several prior studies (Tantithamthavorn et al., 2018a; Rajbahadur et al., 2019), we only use this single performance measure (AUC), as the focus of our study is to evaluate the interpretation of the defect classifiers rather than their performance.

Computation of feature importance scores

We use both the CS and CA methods to computer feature importance scores, as detailed in Section 6.3.3, for all the studied classifiers in each bootstrap iteration. For CA methods: we use the `vip` package and the method outlined by Rajbahadur et al. (2017) to compute the PDP and Permutation CA methods feature importance scores respectively. For the CS computation, we use the `VarImp()` function of the `caret` R package (Kuhn, 2012).

Computation of feature importance ranks

We use the Scott-Knott Effect Size Difference (SK-ESD) test (v2.0) (Tantithamthavorn, 2016a) to compute the feature importance ranks from the feature importance scores computed in the previous step, as done by prior studies (Rajbahadur et al., 2019; Jiarpakdee et al., 2019). For each dataset and studied classifier, three feature importance scores are computed (one CS score and two CA scores) for each bootstrap iteration. The SK-ESD test is applied on these scores to compute three feature importance rank lists (one CS rank list and two CA rank lists) for all the 6 studied classifiers on each dataset. The process of feature importance rank computation from the feature importance scores is depicted in the right-hand side of Figure 6.1.

Also, we note that we only compute the feature importance ranks on a dataset, when each of the 6 studied classifiers have a median AUC greater than 0.70. We do so, as Chen et al. (2018) and Lipton (2016) argue, a classifier should have a good operational performance for the computed feature importance ranks to be trusted. Due to this constraint, the datasets `Xalan-2.5` and `Camel-1.2` were discarded.

6.3.5 Evaluation metrics

We measure the difference between the different feature importance rank lists by measuring how much they agree with each other.

Kendall's Tau coefficient (τ) (Kendall, 1948) is a widely used non-parametric rank correlation statistic that is used to compute the similarity between *two* rank lists (Kitchenham et al., 1995; Bachmann and Bernstein, 2010). Kendall's τ ranges between -1 to 1, where -1 indicates a perfect disagreement and 1 indicates a perfect agreement.

We use the interpretation scheme suggested by Akoglu (2018):

$$\text{Kendall's } \tau \text{ Agreement} = \begin{cases} \text{weak,} & \text{if } |\tau| \leq 0.3 \\ \text{moderate,} & \text{if } 0.3 < |\tau| \leq 0.6 \\ \text{strong} & \text{if } |\tau| > 0.6 \end{cases}$$

Kendall's W coefficient (Kendall, 1948) is typically used to measure the extent of agreement among *multiple* rank lists given by different raters (CS methods in our case and raters ≥ 2). The Kendall's W ranges between 0 to 1, where 1 indicates that all classifiers agree perfectly with each other and 0 indicates perfect disagreement. We use the Kendall's W in RQ3 to estimate extent to which the different feature importance ranks that are computed by CS methods agree across all the studied classifiers for a given dataset . We use the same interpretation scheme for Kendall's W as we use for Kendall's Tau.

Top-3 overlap is a simple metric that computes the amount of overlap that exists between features at the top-3 ranks in relation to the total number of features at the top-3 ranks across n feature importance rank lists. This metric does not consider the ordinality of the features in the top-3 ranks, i.e., order in which a given feature appeared in the top-3 ranks. Rather, it only checks if a given feature appeared in both of the top-3 rank lists. Top-3 overlap is adapted from the popular Jaccard Index (Jaccard, 1901) for measuring similarity. We compute the top-3 overlap among n feature importance lists (for in RQ1 and RQ2, $n = 2$, whereas in RQ3, $n = 6$) with the equation 6.1 ($k = 3$).

$$Top-k\ overlap = \frac{\bigcap_{i\geq 2}^{n} Features\ at\ top\ k\ ranks}{\bigcup_{i\geq 2}^{n} Features\ at\ top\ k\ ranks} \tag{6.1}$$

We define the interpretation scheme for Top 3 overlap as follows, which aims to enable easier interpretation of the results:

$$\text{Top-3 Agreement} = \begin{cases} \text{negligible,} & \text{if } 0.00 \leq \text{top-3 overlap} \leq 0.25 \\ \text{small,} & \text{if } 0.25 < \text{top-3 overlap} \leq 0.50 \\ \text{medium,} & \text{if } 0.50 < \text{top-3 overlap} \leq 0.75 \\ \text{large} & \text{if } 0.75 < \text{top-3 overlap} \leq 1.00 \end{cases}$$

For example, assume that the top-3 features for CS and CA on a given dataset and classifier are $Imp_{CS}(Top3) = \{cbo, loc, pre\}$ and $Imp_{CA}(Top3) = \{loc, lcom3, dit\}$ respectively. Then the top-3 overlap corresponds to $1/5 = 0.2$ (as $n = 2, k = 3$).

Top-1 overlap is analogous to the Top-3 overlap metric (Equation 6.1, with $k = 1$). We define the interpretation scheme for Top-1 overlap as follows: if top-1 overlap is ≤ 0.5 then agreement is low, otherwise agreement is deemed high.

6.4 Case Study Results

In this section, we detail the results of our case study with regards to our research questions from Section 6.2.

6.4.1 (RQ1) For a given classifier, how much do the computed feature importance ranks by CA and CS methods differ across datasets?

Approach: For each of the six constructed classifiers, we compare the feature importance ranks that are computed by the CA and CS methods across the 16 studied datasets (where each of the constructed classifiers have an AUC ≥ 0.70). For each classifier, on a given dataset, we compare the feature importance ranks computed by PDP and Permutation CA methods with the feature importance ranks that are computed by the studied CS method of a classifier. We quantify the agreement between the two rank lists in terms of Top-1 overlap, Top-3 overlap and Kendall's Tau. We compute the Top-1 and Top-3 overlap in addition to the Kendall's Tau because some of the prior work primarily examines the top x important features (Hassan and Holt, 2005; Lewis et al., 2013; Chen et al., 2018). Finally, we aggregate the comparisons with respect to each classifier across the studied datasets.

For instance, for the avNNet classifier, we first compare the feature importance ranks that are computed by PDP (CA method) with those that are computed by the CS method of avNNet (i.e. NNFI, see Table 6.4) on the `eclipse-2.0` dataset. Next, we determine the agreement between the two lists according to Top-1, Top-3 overlap, and Kendall's Tau. We then repeat this step for every dataset and plot the distribution for each agreement metric. An analogous process is followed in order to compare the Permutation method with the NNFI method. The goal of this RQ is to determine the

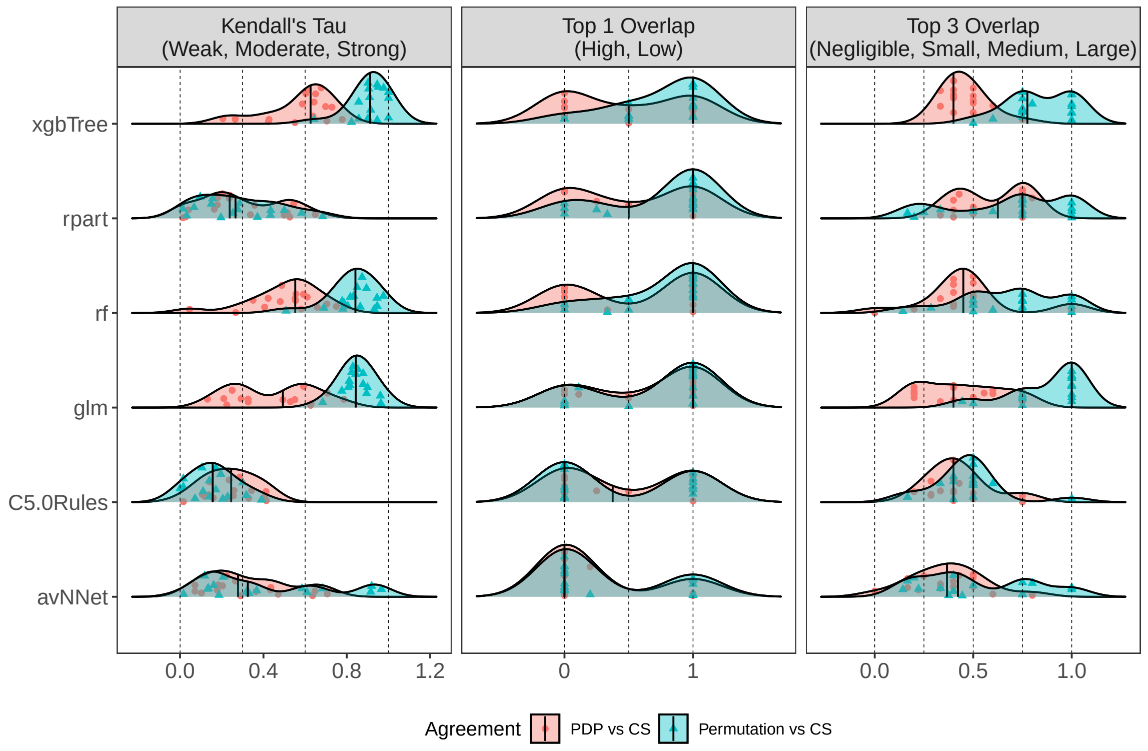


Figure 6.2: A density plot of Top-1 Overlap, Top-3 Overlap, and Kendall's Tau between CA and CS methods for each classifier across the studied datasets. The circles and triangles correspond to individual observations. The dotted lines correspond to the metric-specific interpretation scheme outlined in Section 6.3.5. The vertical lines inside the density plots correspond to the median of the distributions.

extent to which the feature importance ranks that are computed by CA methods differ from the more widely used and accepted CS methods for each classifier. If the studied CA methods consistently have a high agreement with the CS methods for each classifier and across all the studied datasets, then one can use both CS methods and CA methods interchangeably.

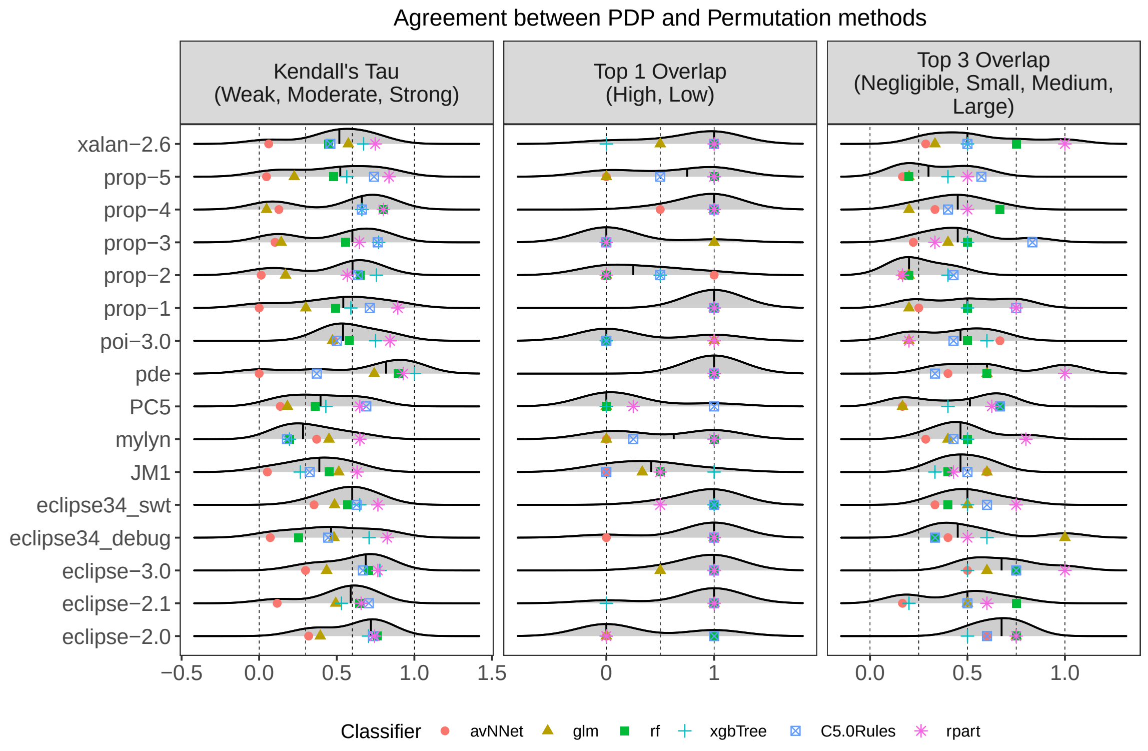


Figure 6.3: A density plot over all six Top-1 Overlap, Top-3 Overlap, and Kendall's Tau values between the PDP and Permutation CA methods for each of the studied classifiers. The dotted lines correspond to the metric-specific interpretation scheme outlined in Section 6.3.5. The vertical lines inside the density plots correspond to the median of the distributions.

Results: The PDP and Permutation CA methods have a low median top-1 overlap with the CS methods of two of the six studied classifiers. The leftmost lane in Figure 6.2 shows the top-1 overlap between the feature importance rank lists that are computed by the CS and CA methods for each classifier and across all the studied datasets.

We observe that the median top-1 overlap between the Permutation method and the CS method of a classifier is low for two classifiers, namely C5.0Rules and avNNet. In turn, the median top-1 overlap between PDP method and CS method is low for four of the six studied classifiers. We also note that both PDP and Permutation CA methods have a low median top-1 overlap for C5.0Rules and avNNet. In other words, even the most important feature varies between the rankings that are computed by the CA and CS methods for two of the studied classifiers.

Both CA methods have a small median top-3 overlap with CS methods on two of the six studied classifiers. We see from the middle lane of Figure 6.2 that the features that are reported at the top-3 ranks by the PDP method do not exhibit a large overlap with the feature importance ranks that are computed by the CS method for *any* of the studied classifiers. The situation for the Permutation CA method is better, as the top-3 reported features by the Permutation CA method and the studied CS methods have a large median overlap on four out of the six studied classifiers. However, from Figure 6.2, we observe that even on cases where the median overlap is large, the spread of the density plot is also large (i.e., several datasets exhibit small and even negligible top-3 overlap).

For half of the studied classifiers, the Kendall's Tau agreement between CA and CS methods is only moderate at best. The Kendall's Tau values between the feature importance ranks that are computed by CA and CS methods for each classifier and across all the studied datasets are depicted as density distributions in Figure 6.2. When comparing the PDP method to the studied CS method of each classifier, we observe that the median Kendall's Tau indicates a strong agreement for only one of the six studied classifiers, namely xgbTree (note the vertical bars inside the density plots in the right-most lane).

The feature importance ranks that are computed by the Permutation method have a strong agreement with those computed by the CS methods of half of the studied classifiers (rf, glm, xgbTree). For two classifiers (rpart and C5.0Rules), both the PDP and Permutation CA methods only weakly agree with the CS methods associated with the studied classifiers. In particular, for completely black-box classifiers like avNNet, we only observe a weak to a moderate agreement with their CS methods and the CA methods (i.e., both the PDP and Permutation methods).

In summary, the CA and CS methods do not always exhibit strong agreement for the computed feature importance ranks across the studied classifiers. Therefore, we discourage the interchangeable use of CA and CS methods in general and suggest that, whenever possible, future defect prediction studies should preferably choose the same feature importance method when replicating or seeking to validate a prior study. If not possible, then the defect prediction study should acknowledge that the difference in insights compared to the original study could be due to the choice of feature importance method (e.g., as a threat to internal validity). Furthermore, in cases such as those, researchers should specify their reasons for choosing a different feature importance method over the one that is used in the study that they seek to replicate or validate.

The computed feature importance ranks by CA and CS methods do not always strongly agree with each other. For two of the six studied classifiers, even the most important feature varies across CA and CS methods.

6.4.2 (RQ2) For a given dataset and classifier, how much do the computed feature importance ranks by the different CA methods differ?

Approach: For each of the studied datasets (where all the constructed classifiers have an AUC $\geq$ 0.70), we check the extent to which the feature importance ranks computed by PDP and Permutation CA methods agree with each other for all the six studied classifiers (see Table 6.3). Similarly to the previous RQ, in order to quantify agreement, we compute the Top-1 Overlap, Top-3 Overlap, and Kendall's Tau measures.

Results: Both PDP and Permutation CA methods have a low median Top-1 overlap for six out of the 16 studied datasets. From Figure 6.3, we observe that even when focusing on the top feature, PDP and Permutation CA methods do not agree for six out of 16 datasets. More importantly, for different datasets, different classifiers have a low Top-1 overlap, indicating that disagreement between the feature importance ranks that are computed by PDP and Permutation CA methods is not a classifier-specific characteristic, but rather a consequence of how different CA methods capture feature importance ranks differently. Such a result highlights that the two classifier-agnostic feature importance methods ascertain feature importance differently and thus cannot be used interchangeably.

PDP and Permutation CA methods do not have a large median Top-3 overlap on any of the 16 studied datasets (see Figure 6.3). Furthermore, on 7 of the 16 datasets, the exhibited median Top-3 overlap is only moderate. On the `prop-2` dataset, the Top-3 overlap is negligible. Even if the Top-1 overlap between the CA and CS are low, if the

Top-3 overlap is high, they could still be interchangeably used, as sometimes practitioners might not care about the ordinality of the influential features (Chen et al., 2018; Tan and Chan, 2016). However, from Figure 6.3, we see that, even for interpretable and additive classifiers like glm, the overlap is generally small. Therefore, such small median Top-3 overlap on the studied datasets further reiterates that CA methods (PDP and Permutation methods) compute vastly different feature importance ranks and cannot be used interchangeably.

For 10 out of the 16 studied datasets, the computed feature importance ranks by PDP and Permutation CA methods only moderately agree at best. From Figure 6.3, we observe that for 10 out of 16 datasets, the median level of agreement (using Kendall's Tau) is moderate at best (note the vertical bars inside the density plots in the left-most and the middle lane). Such a result indicates that PDP and Permutation CA methods ascertain different feature importance ranks even on the same dataset across different classifiers. Furthermore, from Figure 6.3, we see that PDP and Permutation CA methods do not consistently agree on the feature importances for any studied classifier besides rpart. In particular, for avNNet, a complex neural network based classifier for which CA methods are typically employed (Avati et al., 2018; Putin et al., 2016), PDP and Permutation CA methods only exhibit a weak agreement for 11 out of the 16 studied datasets (and no strong agreement in any dataset). In turn, for a simple classifier like rpart, the PDP and Permutation CA methods show a consistently strong agreement for 15 out of the 16 studied datasets.

The computed feature importance ranks by the studied CA methods rarely ascertain the same feature importance ranks in a dataset for a given classifier, including the top-1 and the top-3 most important feature(s).

6.4.3 (RQ3) On a given dataset, how much do the computed feature importance ranks by different CS methods differ?

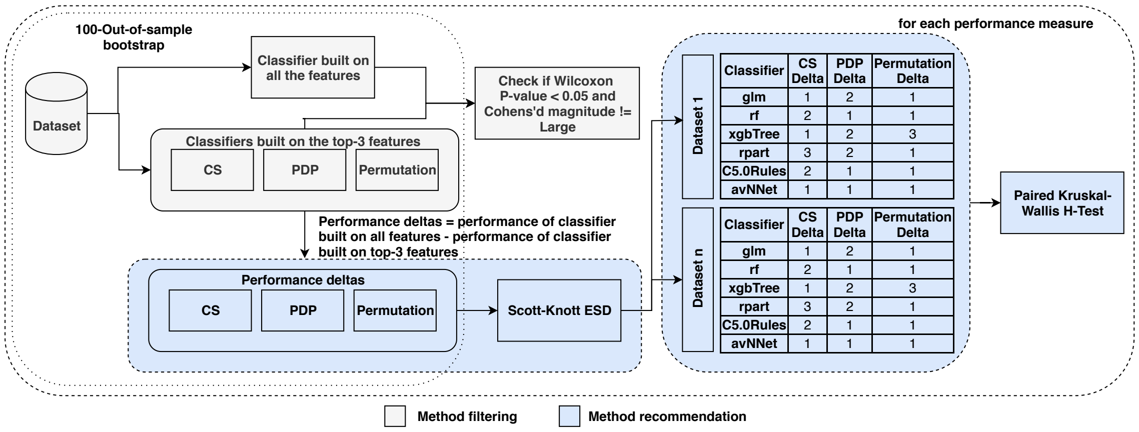
Approach: For each of the datasets, we obtain the computed feature importance ranks by the studied CS methods of each of the six studied classifiers. We then calculate the Kendall's W between the six feature importance rank lists that are computed by the studied CS method of each classifier. Unlike the previous RQs, we compute the Kendall's W instead of Kendall's Tau, as Kendall's W is able to measure agreement among multiple feature importance rank lists (Section 6.3.5). Furthermore, we also calculate the Top-3 overlap among all the six feature importance rank lists. We do so for all the studied datasets. A high Kendall's W and a high Top-3 overlap across all the studied datasets among the constructed classifiers would indicate high agreement between the computed feature importance ranks by different classifiers and the studied CS method of each classifier.

Results: The computed feature importance ranks by different CS methods vary extensively. Only for the `eclipse-3.0` dataset, the classifiers agree on the same most important feature. Furthermore, the maximum top-3 overlap is only small and it happens for only three out of 16 datasets. Finally, we also observe that, on a given dataset, none of the feature importance rank lists computed by the different CS methods strongly agree with each other. We summarize the Top 1 overlap, the Top-3 overlap and Kendall's W among the computed feature importance ranks for all the six studied classifiers across the studied datasets in Table 6.5.

From Table 6.5, we observe that both the Kendall's W and top-3 overlap among the feature importance ranks that are computed by studied CS methods associated with each of the classifier – which are widely used in the software engineering community–

Table 6.5: Top-1 overlap, Top-3 overlap, and Kendall's W among the computed feature importance ranks by the CS method of each classifier. Best results for each metric are shown in bold.

Dataset	**Top-1 overlap**	**Top-3 overlap**	**Kendall's W**
poi-3.0	Low (0)	Negligible (0)	Weak (0.15)
xalan-2.6	Low (0)	Negligible (0.17)	Weak (0.20)
eclipse34_debug	Low (0)	Negligible (0)	Weak (0.20)
eclipse34_swt	Low (0)	Negligible (0.22)	**Moderate (0.51)**
pde	Low (0)	**Small (0.4)**	**Moderate (0.46)**
PC5	Low (0)	Negligible (0)	Weak (0.07)
mylyn	Low (0)	**Small (0.33)**	Weak (0.08)
eclipse-2.0	Low (0)	Negligible (0)	Weak (0.30)
JM1	Low (0)	Negligible (0.20)	**Moderate (0.31)**
eclipse-2.1	Low (0)	Negligible (0.13)	Weak (0.26)
prop-5	Low (0)	Negligible (0)	Weak (0.18)
prop-4	Low (0)	Negligible (0.17)	Weak (0.26)
prop-3	Low (0)	Negligible (0)	Weak (0.27)
eclipse-3.0	**High (1)**	**Small (0.4)**	**Moderate (0.31)**
prop-1	Low (0)	Negligible (0.13)	Weak (0.27)
prop-2	Low (0)	Negligible (0)	Weak (0.23)


100-Out-of-sample bootstrap
Dataset
Classifier built on all the features
Check if Wilcoxon P-value < 0.05 and Cohens'd magnitude != Large
Classifiers built on the top-3 features
CS
PDP
Permutation
Performance deltas = performance of classifier built on all features - performance of classifier built on top-3 features
Performance deltas
CS
PDP
Permutation
Scott-Knott ESD
for each performance measure
Dataset 1
Dataset n
Paired Kruskal-Wallis H-Test
Method filtering
Method recommendation

Classifier	CS Delta	PDP Delta	Permutation Delta
glm	1	2	1
rf	2	1	1
xgbTree	1	2	3
rpart	3	2	1
C5.0Rules	2	1	1
avNNet	1	1	1

Classifier	CS Delta	PDP Delta	Permutation Delta
glm	1	2	1
rf	2	1	1
xgbTree	1	2	3
rpart	3	2	1
C5.0Rules	2	1	1
avNNet	1	1	1

Figure 6.4: Experiment setup of our discussion.

is very low. Such a small Top-3 overlap and Top-1 overlap for all the datasets indicates that computed feature importance ranks by CS methods differ substantially among themselves. Hence, different classifiers and their associated CS methods cannot be used interchangeably.

On a given dataset, even the commonly used CS methods yield vastly different feature importance ranks, including the top-3 and the top-1 most important feature(s).

6.5 Discussion

6.5.1 How discriminative are the top-3 reported features by the studied CS and CA methods?

Motivation: From RQ1 (Section 6.4.1), we observe that both CA and CS methods yield significantly different feature importance ranks on a given dataset for different classifiers. While such a result indicates that we cannot use one feature importance method in lieu of another, we do not know which of these feature importance methods yield the most discriminative (i.e., the features with the highest predictive capability) feature importance ranks. Such knowledge would help us with two things. First, to avoid the usage of a particular feature importance method that does not identify highly discriminative top-3 features (method filtering). Particularly, if any of the feature importance methods fail to identify highly discriminative features, we could discourage its usage. Second, to recommend the feature importance methods that could be more reliably used by researchers and practitioners (method recommendation). For instance, if one of the studied feature importance methods (amongst both CA and CS methods) consistently identifies the most discriminative top-3 feature set, we could recommend its usage over the other methods (even if all of these methods are highly discriminative).

Approach: Part 1 - Method filtering. We present the experiment setup of our Discussion in Figure 6.4. To estimate the discriminative capability of the top-3 computed features by each feature importance method, we first construct all of the studied classifiers using all the features of a given dataset as outlined in Section 6.3.4. Next, we construct classifiers with only using the top-3 features given by the two CA methods, as well as the CS method of each classifier (please see Table 6.4) for each studied dataset. We then measure the performance of the classifiers that are constructed using all the features against the performance of the classifiers that are constructed using the top-3 features. We measure the performance in terms of several performance measures (i.e., Accuracy, Precision, Recall, F1 Score, Brier Score and Mathew's Correlation Coefficient (MCC)) instead of using AUC as used in the earlier parts of our study. We do so in order to avoid the unfair advantage that the Permutation method would likely have, as such a method evaluates the importance of each feature based on their AUC performance.

Finally, we perform a Wilcoxon signed-rank test (Wilcoxon, 1945) and a Cohen's d effect size difference test (Cohen, 1992) to compare the performance in a statistically rigorous manner. In the negative case (i.e., if the magnitude ≤ 0.8 (Cohen, 1992)), we conclude that the performance of both classifiers is similar and thus the feature importance method is deemed as one that produces highly discriminative top-3 features. We consider a large Cohen's d effect size difference to quantify similarity over other cut-offs as we wanted to show that the differences in performances are large only if they differ by a considerable amount. If any of the studied feature importance methods fails to consistently produce highly discriminative top-3 features across the studied datasets, then we could discourage its use.

Part-2: Method recommendation. To identify which of the studied feature importance methods can be recommended, for each dataset, we find the difference in performance between the classifiers that are constructed using all the features of the dataset and the classifiers that constructed only using the top-3 features given by the CA and CS methods - performance deltas (the smaller the difference, the higher the discriminative capability). Then to observe if any of the feature importance method consistently produces more discriminative features than the others, we rank the performance deltas. We do so on all the studied datasets for each classifier using the Scott-Knott ESD test (as shown in the first blue part of Figure 6.4). Next, we aggregate the produced ranks on the performance deltas of the six studied classifiers across the three studied feature importance methods (2 CA methods and the studied CS method of the classifier) for all the 16 datasets (please refer the tables in Figure 6.4) for each of the studied performance measure. For instance, for the Accuracy performance measure, we will have three distribution of ranks pertaining to the CS, PDP and Permutation CA methods. Each distribution will have six values corresponding (as given by the rank of the performance differences of the six studied classifiers) for each dataset (a total of 96 values

Table 6.6: No. of dataset on which classifiers that are constructed using the top 3 features given by the studied feature importance method has similar discriminative capability (i.e., if the Cohen's d ≤ 0.8) as that of the classifier that is constructed using all the features of a dataset.

Classifier	**ACC**			**PRE**			**REC**			**B-S**			**F-M**			**MCC**		
	CS	**PD**	**PF**	**CS**	**PD**	**PF**	**CS**	**PD**	**PF**	**CS**	**PD**	**PF**	**CS**	**PD**	**PF**	**CS**	**PD**	**PF**
glm	16	14	16	14	14	14	10	9	10	9	7	8	10	8	10	14	13	14
rf	5	4	7	6	6	8	8	7	9	5	4	8	3	5	9	4	6	7
xgbTree	10	8	8	11	12	9	4	3	4	4	0	1	5	3	3	10	9	9
rpart	16	16	16	16	16	15	10	12	11	7	11	9	10	12	10	12	13	13
C5.0Rules	11	14	13	9	12	12	4	6	7	8	8	9	4	9	8	13	14	13
avNNet	14	15	15	15	12	13	9	12	9	7	8	7	10	11	10	14	15	14
K-W (P-Val)	**0.12**			**0.89**			**0.11**			**0.65**			**0.19**			**0.24**		

1. **Performance Measures:** ACC- Accuracy, PRE- Precision, REC- Recall, BS- Brier Score, F-M- F-Measure
2. **Feature importance methods:** CS- CS, PD- PDP, PF- Permutation
3. **K-W** - Kruskal-Wallis H-Test P-Value

per distribution). Each value in the distribution pertaining to a given feature importance method indicates its rank of discriminative capability in relation to the other feature importance methods on a given classifier and dataset. Finally, across each of the performance measure, we perform a paired Kruskal-Wallis H-test (Kruskal and Wallis, 1952). A p-value ≤ 0.05 on the Kruskall-Wallis H-test between the rank distributions of CS, PDP and Permutation CA methods across all the performance measures would indicate that one of the feature importance methods consistently yields more discriminative top-3 features than the other methods and we could recommend its usage.

Results: Method filtering: The top-3 most important features ascertained by all the studied feature importance methods are all highly discriminative. In Table 6.6, we present the number of datasets on which the classifiers built using the top-3 features from each of the studied feature importance methods exhibits similar performances to the classifiers built using all the features. From Table 6.6 we observe that, all the studied feature importance methods exhibit similar capabilities in identifying the top 3 discriminative features for each of the studied classifiers across all the computed performance measures. For instance, consider the discriminative capability of the top-3 features given by the different methods in terms of Accuracy from Table 6.6. Expect for the case of random forest and the xgbTree classifier, in at least 11 out of 16 datasets, the classifiers built using the top 3 features given by all the different feature importance methods, calculates Accuracy similar to that of the classifier built using all the features. Similar trends could be observed on all the threshold-dependent metrics like Precision, Recall, F-Measure and MCC. Each of classifier built using the top-3 features computed by the different feature importance methods exhibiting similar (high) discriminative capability indicates that each each feature importance method—though compute feature importances differently— are a valid way of computing feature importances in their own right.

The number of datasets on which the discriminative capability of the top-3 features being equivalent to that of all the features in a given dataset is lower in terms of the threshold-independent performance measure Brier Score. However, across the classifiers the different top-3 features computed by the different feature importance methods exhibit the similar characteristics even on threshold-independent measures.

Method recommendation: No method consistently produces top-3 features with higher discriminative capability than other methods. Hence, we cannot recommend the use a particular feature importance method. We present the result of Kruskal-Wallis H-Test between the performance difference rank distributions of CS, PDP and Permutation CA methods in Table 6.6. From the Table 6.6, we observe that all the p-values are greater than 0.05. Such a result indicates that, across different performance measures, there is no one method that is consistently ranked low. Lack of clear separation in the discriminative capability of the top 3 features computed by different feature importance methods indicates that we cannot recommend usage of one of the feature importance method over another.

6.5.2 Why do different feature importance methods produce different top-3 features on a given dataset?

Motivation: From the RQs explored in Section 6.4, we observe that different feature importance methods produce different feature importance ranks (including the top-3 ones) in spite of us having removed the correlated and redundant features from the datasets in a pre-processing step. While such a result could be attributed to different feature importance methods computing feature importances differently, from previous Section 6.5.1, we find that the different top-3 features produced by CA and CS methods are all similarly discriminative on a given classifier. In this discussion, we seek to find out why different feature importance methods produce different top-3 features that are equally discriminative.

We hypothesize that different feature importance methods produce different but equally discriminative top-3 feature importance ranks even on the same dataset and classifier because of feature interactions that are present in the studied datasets. Feature interactions can be defined as a phenomenon where the effect of the independent features on a dependent feature is not purely additive (Freeman, 1985; Molnar, 2018)). We arrive at such a hypothesis as many of the prior studies show that the presence of feature interactions in a given dataset can affect the different feature importance methods differently and make them assign different feature importance ranks to features with similar discriminative capability (de González et al., 2007; Freeman, 1985; Fisher et al., 2018; Devlin et al., 2019; Lundberg et al., 2018). Therefore, in this section, we seek to find out if different feature importance methods (in addition to being inherently different) yield different feature importance rankings that are similarly discriminative due to the presence of feature interactions in the dataset.

Approach: To test our hypothesis and detect if any of the features present in a given dataset interact with other features in that dataset, we compute the *Friedman H-Statistic* (Friedman et al., 2008) for each feature against all other features in a given dataset. The Friedman H-statistic works as follows. First, a classifier (any classifier - we use random forest as it captures interactions well (Wright et al., 2016)) is constructed using the given dataset. For instance, consider the Eclipse-2.0 dataset and that we wish to compute the Friedman H-Statistic between the feature `pre` and all the other features in Eclipse-2.0. We first compute the partial dependence between `pre` and the dependent variable with respect to random forest classifier (`PD_pre`) for all the data points. Following which, partial dependence between all the features (as a single block) in Eclipse-2.0 (except `pre`) is computed (`PD_rest`) for all the data points. Now if there is no feature interaction, between `pre` and the other features in Eclipse-2.0, the outcome probability of the constructed classifier can be expressed as a sum of `PD_pre` and `PD_rest`. Therefore, the difference between the outcome probability of the classifier and the sum of `PD_pre` and `PD_rest` is computed and the variance of this difference is reported as the Friedman H-Statistic. We compute the Friedman H-Statistic for all the studied datasets 10 times as Friedman H-Statistic is known to exhibit fluctuations because of the internal data sampling (Molnar, 2018). We then consider the median score of the Friedman H-Statistic for each dataset. We use the R package `iml`[3] to compute the Friedman H-Statistic.

The Friedman H-Statistic is a numeric score that ranges between 0 to 1 (However, it could sometimes exceed 1 when the higher order interactions are stronger than the individual features (Molnar, 2018)). A Friedman H-Statistic of 0 or closer to zero indicates that no interaction exists between the given feature and the rest of the features and a Friedman H-Statistic of 1 indicates extremely high levels of interaction. For a more theoretical and detailed explanation we refer the readers to (Friedman et al., 2008; Molnar, 2018). In this study, we consider a feature to exhibit interactions with other features if the Friedman H-Statistic is ≥ 0.3. We choose 0.3 as a cut-off, to only indicate the existence of a feature interaction, but not to qualify the strength of the interaction, because

[3]`https://cran.r-project.org/web/packages/iml/index.html`

the presence of feature interactions irrespective of the strength could potentially impact feature importance ranks (de González et al., 2007; Freeman, 1985; Fisher et al., 2018; Devlin et al., 2019; Lundberg et al., 2018). In addition we also report the results for the number of features that exhibit a Friedman H-statistic ≥ 0.5. We choose to report the results on multiple thresholds to present a comprehensive depiction of the feature interactions and due to the lack of prior empirical guidance on thresholds for the interpretation of Friedman H-statistic.

Finally, to assert if absence of feature interactions enables the different feature importance methods to compute the same top-3 features, we simulate a dataset with no feature interactions. We do so instead of using a real dataset as it is difficult to find a real-world defect dataset without any feature interactions. We generate a dataset 1,500 data points and 11 independent features of which five features carry the signal `signal = {x1,x2,x3,x4,x5}` and six features are just noise i.e., does not exhibit any relationship to the dependent feature `noise = {n1, n2, n3, n4, n5, n6}`. We add noise features to make our simulated dataset similar to that of a real-world defect datasets. Therefore the independent features of the dataset is comprised of `independent features = {signal,noise}`. All the signal features and `n1, n5, n6` are generated by randomly sampling the normal distribution with `mean = 0` and `standard deviation = 1`. Similarly, we sample the uniform distribution between 0 and 1 to generate the values for `n2, n3, n4`. We use both normal and the uniform distribution for the noise features to ensure the presence of different types of noise in our simulated dataset. Next, to construct our dependant feature for the dataset, we construct the y_{signal} with the signal variables as given in Equation 6.2. We assign different weights to the different signal features when constructing the y_{signal} to ensure that we know the true importance of each of the signal feature. We then convert the y_{signal} in to a probability vector y_{prob} with a sigmoid function as given in Equation 6.3. Finally, we generate the dependant feature by sampling the binomial distribution to generate the dependent feature $y_{dependant}$ with y_{prob} as given in Equation 6.4.

$$y_{signal} = 20x1 + 10x2 + 5x3 + 2.5x4 + 0.5x5 \tag{6.2}$$

$$y_{prob} = \frac{1}{1 + e^{-y_{signal}}} \tag{6.3}$$

$$y_{dependant} = Binomial(1500, y_{prob}) \tag{6.4}$$

We then construct all of the studied classifiers on the simulated dataset with $y_{dependent}$ as the dependent feature. We construct all the classifiers with 100-Out-of-sample bootstrap on the simulated dataset and compute the feature importance ranks computed by the CA and CS methods as outlined in Section 6.3.4. For each of the studied classifier, we calculate the top-1 and top-3 overlap between the feature importance ranks computed by the CA and CS methods for each of the classifier. We then check if they exhibit a top-1 and top-3 overlap close to 1 for all the classifiers among the CS and the CA methods. If they do so, we can then assert that, in addition to different feature importance methods computing importance differently, the feature interactions in the dataset affects the top-3 features computed by the different feature importance methods and vice versa.

Results: At least two features and as much as eight features interact with the rest of the features in all of the 16 studied datasets (i.e., Friedman H-Statistic ≥ 0.3). Furthermore, we find that 12 of the 16 datasets have at least two features with a Friedman H-Statistic $\geq$ 0.5. We present the number of features in each dataset with a Friedman H-statistic ≥ 0.3 and ≥ 0.5 in Table 6.7. From Table 6.7 we observe that all datasets contain more than two features that interact with the other features. Though Friedman H-Statistic only computes if a given feature interacts with the rest of the features and excludes other feature interactions like second-order interactions, pairwise interactions and higher-order interactions, it gives us a hint as to the presence or absence of feature interactions in a dataset.

Table 6.7: No. of features per dataset with Friedman H-Statistic ≥ 0.3 and ≥ 0.5

Dataset	**#F with H $\geq$ 0.3**	**#F with H $\geq$ 0.5**
Poi-3.0	3	0
Xalan-2.6	2	0
Eclipse34_debug	6	3
Eclipse34_swt	3	0
Pde	5	4
PC5	4	0
Mylyn	4	4
Eclipse-2.0	6	4
JM1	7	5
Eclipse-2.1	6	3
Prop-5	5	4
Prop-4	5	3
Prop-3	5	3
Eclipse-3.0	6	5
Prop-1	6	2
Prop-2	8	4

F - Features

The top-3 and the top-1 overlap between the feature importance ranks computed by the CS and CA methods on each of the classifier is 1 on our simulated dataset devoid of feature interactions. Such a result indicates that in the dataset without feature interactions, all the studied feature importance methods identify the same top-3 features. Furthermore, we also observe the all the studied feature importance identify `x1, x2, x3` as the top-3 features in the same order of importance. Thus, we assert that the different feature importance methods are able to assign the feature importance ranks correctly when independent features' contribution to the dependant feature is additive without any interactions.

Hence, we conclude that the presence of feature interactions in the studied defect datasets could be the reason why different top-3 features produced by different feature importance methods exhibit similar discriminative capability. In addition, we argue that alongside the fact that different feature importance methods compute feature importances differently, feature interactions in the datasets could also be one of the important confounders that affects the computed feature importance ranks by the studied feature importance methods. However, our inference is exploratory in nature and thus further research should be conducted to understand the exact impact of feature interaction on the computed feature importance ranks.

6.6 Implications

In this section, we outline the implications that one can derive from our results, including potential pitfalls to avoid and future research opportunities.

Implication 1) During replication or establishment of baselines from prior studies, one should employ the same feature importance method as in these prior studies to

compute the feature importance ranks. From Section 6.4, we observe that the computed feature importance ranks by CA and CS methods differ greatly even on the top-3 most important features across the studied classifiers and datasets. Therefore, it is essential that replication studies employ the same feature importance method as the original study. If not possible, the replication should acknowledge that the difference in feature importance ranks might be due to their choice of feature importance methods.

Implication 2) The lack of clear specification of the feature importance method employed in software engineering studies seriously threatens the reproducibility of these studies and the generalizability of their insights. Few of the prior work (17% of the studies specified in Table 6.1) do not specify their employed feature importance method to arrive at their insights. Such lack of specification of the feature importance method is mostly prevalent for random forest classifiers (3/11 studies) – the classifier that is widely used in software engineering. This poses a serious threat, as many random forest implementation across the popular data mining toolboxes come with many different ways of computing the feature importance. For instance, random forest implementation in the R package *randomForest* [4] has 3 feature importance methods available and the R package *partykit* [5] has 2 implementations of feature importance methods for random forest. Such a case is true for other classifiers (such as logistic regression's classifier specific feature importance method is different between the *caret* and *rms* R packages).

Implication 3) Future studies should use the results of feature importance methods with caution, especially in the presence of feature interactions. From Section 6.5.2

[4]https://cran.r-project.org/web/packages/randomForest/index.html

[5]https://cran.r-project.org/web/packages/partykit/index.html

we observe that all of the studied datasets have at least two features that interact with the other features in a dataset. Such interactions, in turn, impact the feature importance ranks computed by both CA and CS methods (de González et al., 2007; Freeman, 1985; Fisher et al., 2018; Devlin et al., 2019; Lundberg et al., 2018). Therefore, we suggest researchers and practitioners to analyze if the features in their dataset interact with each (e.g., by reusing the procedure that we outlined in Section 6.5.2). If features interact, we then recommend researchers and practitioners to report the feature interactions along with their feature importance ranks and interpret their results with caution.

Implication 4) Further research must be carried out to develop feature importance methods that account for feature interactions. In this Chapter, we highlight that different features importance methods generate feature importance ranks that differ greatly and therefore they cannot be interchangeably used. However, we cannot recommend the usage of anyone feature importance method over another, as generating consistent feature importance ranks across classifiers, especially in the presence of feature interactions, is still an open area of research. For instance, Kendler and Gardner (2010) suggest that we can only ever truly interpret main effects and tentative interpretation of interactions are justified whereas Freeman (1985) suggests modeling and interpreting them separately. Furthermore, Kendler and Gardner (2010) suggest using only additive models to avoid getting influenced by interactions. Therefore, we suggest that future studies should investigate the impact of feature interactions on the feature importance ranks much more deeply, especially in the context of software engineering datasets. Furthermore, there is also a need to develop robust feature importance methods that can compute consistent feature importance ranks across classifiers as the interpretation of machine learning classifiers has become pivotal in software engineering.

6.7 Threats to Validity

In the following, we discuss the threats to the validity of our study.

Internal validity. We choose classifier families who has a CS method. As previous studies show that different classifiers may have different performance on a given dataset (Rajbahadur et al., 2017; Ghotra et al., 2015), this could be a potential threat. However, we choose representative classifiers from 6 of the 8 commonly used classifier families as outlined by Ghotra et al. (2015).

Construct validity. In our study, we choose datasets where all the classifiers achieved an AUC above 0.70. According to Muller et al. (2005), an AUC score above 0.70 indicates the fair discriminative capability of a classifier. Furthermore, these datasets have been used in many of the studies as outlined in Table 6.1. Furthermore, in Section 6.5.1, we use Cohen's d effect size test, which is a parametric test that assumes the groups it tests to be normally distributed. However, we do not ensure if the obtained performance scores are normally distributed before using the Cohen's d effect size test. We acknowledge that this is a threat and future studies should revisit our findings by using a non-parametric effect size test.

External validity. In this study, we choose 18 datasets that represent software projects across several corpora (e.g., NASA and PROMISE) and domains (both proprietary and open-source). However, our results likely do not generalize to all software defect datasets. Nevertheless, the datasets that we use in our study are extensively used in the field of software defect prediction (Zimmermann and Nagappan, 2008; Zimmermann et al., 2009; Jiarpakdee et al., 2019; Tantithamthavorn et al., 2017; Rajbahadur et al., 2017; Premraj and Herzig, 2011) and is representative of several corpora and domains. Therefore we argue that our results will still hold. However, future replication across different datasets using our developed methodology might be fruitful.

Secondly, we only consider one defect prediction context in our study (i.e., within-project defect prediction). Yet, there are multiple defect prediction contexts such as Just-In-Time defect prediction (Hoang et al., 2019; Kamei et al., 2012) and cross-project defect prediction (Zimmermann et al., 2009). Hence future studies are needed to explore these richer contexts.

Finally, we study a limited number of CS and CA methods and therefore, our results might not readily generalize to other feature importance methods. For instance, there are recent developments like SHAP (Lundberg and Lee, 2017) and LIME (Ribeiro et al., 2016) that have been proposed in the machine learning community for generating feature importance ranks. Nevertheless, the approach and the metrics that we use in our study are applicable to any feature importance method. Therefore, we invite future studies to use our approach to re-examine our findings on other (current and future) feature importance methods.

6.8 Chapter Summary

Classifiers are increasingly used to derive insights from data. Typically, insights are generated from the feature importance ranks that are computed by either CS or CA methods. However, the choice between the CS and CA methods to derive those insights remains arbitrary, even for the same classifier. In addition, the choice of the exact feature important method is seldom justified. In other words, several prior studies use feature importance methods interchangeably without any specific rationale, even though different methods compute the feature importance ranks differently. Therefore, in this study, we set out to estimate the extent to which feature importance ranks that are computed by CS and CA methods differ.

By means of a case study on 18 defect datasets and 6 defect classifiers, we observe that the computed feature importance ranks by CA and CS methods do not always agree. Even the feature that is reported as the most important feature differs for many of the studied classifiers– raising concerns about the stability of conclusions across replicated studies. We end our study by providing several guidance for future studies. Therefore, we strongly recommend against the interchangeable usage of feature importance methods in the software analytics studies to compute feature importance ranks of a classifier.

Future research is needed to develop robust feature importance methods that account for feature interactions and that can compute ranks that are consistent across classifiers at least on the most important features.

CHAPTER 7

Conclusion and Future Work

Machine learning classifiers are extensively used in software analytics pipelines. Constructing these software analytics pipelines requires practitioners to make several experimental design choices. These experimental design choices critically impact the generated insights of a classifier. While the impact of a **few** experimental design choices on the generated insights of a classifier have been explored by the prior studies, the impact of **many** experimental design choices still remains unexplored. It is pivotal to gain a deeper understanding of how experimental design choices impact

In this thesis, we explore the impact of a number of experimental design choices on the generated insights of a classifier. More specifically we first explore the impact of discretization of the dependent feature. We find that discretizing the dependent features indeed impact the generated insights of a classifier. Therefore, we propose strategies to mitigate and avoid the discretization of the dependent feature. Second, we study the impact of interchangeably using the feature importance methods on the generated insights of a classifier in software analytics. We find that different feature importance methods generate significantly different feature importance ranks for a given classifier. Based on our findings we provide recommendations for future software analytics studies. We believe that deepening our understanding of how different experimental design choices impact the generated insights can help improve the reliability and trustworthiness of those insights.

7.1 Thesis Contributions

We highlight the main findings of our thesis and its implications as follows:

7.1.1 Avoiding the discretization of the dependent feature by using regression-based classifiers

Finding 1: We observe that in contrast to current practices in our field, the discretization of the dependent feature should be avoided in many cases (especially when the defective ratio is <15%).

Implication: Future studies should consider building regression-based classifiers and avoid discretizing the dependent feature (in particular when the defective ratio of the modelled dataset is low).

Finding 2: The most influential features vary between the different approaches to build classifiers.

Implication: Hence future studies should examine the influential factors using the best performing classifier (i.e., discretized or regression-based) instead of simply using discretized classifiers

7.1.2 Mitigating the impact of discretizing the dependent feature (Chapter 5)

Finding 3: Discretization noise impacts the different performance measures of classifiers differently across the different datasets. We observe that discretization noise leads to an up to 139% performance differences across various performance measures across all the studied classifiers.

Implication: Researchers and practitioners should use our framework to analyze the impact of discretization noise on a classifier for before either including or discarding discretization noise in their analysis.

Finding 4: Discretization noise impacts the overall computed feature importance ranks of a classifier. However, it does not impact the computed feature importance ranks of the top 3 ranks for our case studies.

Implication: Researchers and practitioners should use our framework to get a case by case guidance for their case. They should verify if the discretization noise impacts the derived feature importance ranks on their dataset, classifier and chosen discretization threshold.

7.1.3 The impact of interchangeably using feature importance methods

Finding 5: The computed feature importance ranks by CA and CS methods do not always strongly agree. Even the feature reported as the most important feature differs for many of the studied classifiers.

Implication: Same feature importance computation method must be used when replicating a prior study in order to avoid conclusion instabilities. In addition, we provide more guidelines for future software analytics studies to follow in Section 6.6 of Chapter 6.

Finding 6: The presence of feature interactions in the studied defect datasets impact the computed feature importance ranks of a classifier. That is to say, the presence of feature interactions in a dataset is one of the reasons why different feature importance methods produce vastly different feature importance ranks, even on the same dataset and classifier.

Implication: We then recommend researchers and practitioners to report the feature interactions along with their feature importance ranks and interpret their results with caution.

7.2 Future Research Directions

In light of the findings that we present in our thesis, we outline the following future research directions.

7.2.1 Developing new approaches to avoid the discretizing the dependent feature when the defective ratio is greater than 15%

We observe from Chapter 4 that the discretization of the dependent feature could be avoided only in cases where the defective ratio of the dataset is less than 15% (or in other words, when the class imbalance is high). However, from Chapter 5 we know that even when the class imbalance is low, the discretization of the dependent feature could impact the results of classifier. Therefore, future research needs to investigate more approaches that can avoid the discretization of the dependent feature.

7.2.2 Extending our proposed framework to support n-ary discretization of the dependent feature

Our proposed framework in Chapter 5 enables researchers and practitioners to investigate the impact of the generated discretization noise due to discretizing the dependent feature into two classes. But in many software analytics problems the continuous dependent feature maybe discretized into more than two classes. Therefore, future research should focus on extending our framework to support the investigation of the impact of the generated discretization noise due to discretizing the dependent feature into multiple classes.

7.2.3 Developing feature importance methods that can account for feature interactions in the dataset

In Chapter 6, we highlight that different features importance methods generate feature importance ranks that differ greatly and therefore they cannot be interchangeably used. However, we cannot recommend the usage of any one feature importance method over another, as none of the studied feature importance methods generate consistent feature importance ranks across classifiers. We observe that it is the presence feature interactions that cause the feature importance methods to compute vastly different feature importance ranks for a given classifier. We further observed that on a simulated dataset without any feature interactions, all the studied feature importance methods reported the same features in the top-3 ranks. Therefore there is an urgent need to develop robust feature importance methods that account for the feature interactions in the dataset automatically. Such methods should focus on computing consistent feature importance ranks across classifiers as the interpretation of machine learning classifiers has become pivotal in software analytics.

Bibliography

Abdelmoez, W., Kholief, M., and Elsalmy, F. M. (2012). Bug fix-time prediction model using naïve bayes classifier. In *Proceedings of the International Conference on Computer Theory and Applications (ICCTA)*, pages 167–172. IEEE.

Abdelwahab, M. and Busso, C. (2015). Supervised domain adaptation for emotion recognition from speech. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5058–5062. IEEE.

Abebe, S. L., Ali, N., and Hassan, A. E. (2016). An empirical study of software release notes. *Empirical Software Engineering (EMSE)*, 21(3):1107–1142.

Agrawal, A., Fu, W., Chen, D., Shen, X., and Menzies, T. (2019). How to “dodge" complex software analytics. *IEEE Transactions on Software Engineering (TSE)*, pp(99):1–1.

Agrawal, A., Menzies, T., Minku, L. L., Wagner, M., and Yu, Z. (2020). Better software analytics via" duo": Data mining algorithms using/used-by optimizers. *Empirical Software Engineering (EMSE)*, 25(3):2099–2136.

Aha, D. W. and Kibler, D. F. (1989). Noise-tolerant instance-based learning algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 794–799. Citeseer.

Akoglu, H. (2018). User's guide to correlation coefficients. *Turkish journal of emergency medicine*, 18(3):91–93.

Alm, C. O., Roth, D., and Sproat, R. (2005). Emotions from text: Machine learning for text-based emotion prediction. In *Proceedings of of the InteConference on Human Language Technology and Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 579–586. Association for Computational Linguistics.

Altman, D. G. and Royston, P. (2006). The cost of dichotomising continuous variables. *British medical journal (BMJ)*, 332(7549):1080.

Altmann, A., Toloşi, L., Sander, O., and Lengauer, T. (2010). Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347.

Angelis, L., Stamelos, I., and Morisio, M. (2001). Building a software cost estimation model based on categorical data. In *Proceedings Seventh International Software Metrics Symposium*, pages 4–15. IEEE.

Arar, Ö. F. and Ayan, K. (2015). Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, 33:263–277.

Arisholm, E., Briand, L. C., and Fuglerud, M. (2007). Data mining techniques for building fault-proneness models in telecom java software. In *Proceedings of the International Symposium on Software Reliability (ISSRE)*, pages 215–224. IEEE.

Austin, P. C. and Brunner, L. J. (2004). Inflation of the type i error rate when a continuous confounding variable is categorized in logistic regression analyses. *Statistics in medicine*, 23(7):1159–1178.

Avati, A., Jung, K., Harman, S., Downing, L., Ng, A., and Shah, N. H. (2018). Improving palliative care with deep learning. *BMC medical informatics and decision making*, 18(4):122.

Bachmann, A. and Bernstein, A. (2010). When process data quality affects the number of bugs: Correlations in software engineering datasets. In *Proceedings of Working Conference on Mining Software Repositories (MSR 2010)*, pages 62–71. IEEE.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The missing links: bugs and bug-fix commits. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 97–106. ACM.

Bao, L., Xia, X., Lo, D., and Murphy, G. C. (2019). A large scale study of long-time contributor prediction for github projects. *IEEE Transactions on Software Engineering (TSE)*, PP(99):1–1.

Bao, L., Xing, Z., Xia, X., Lo, D., and Li, S. (2017). Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In

Proceedings of the International Conference on Mining Software Repositories (MSR), pages 170–181. IEEE.

Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE)*, 41(4):384–407.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research (JMLR)*, 13(Feb):281–305.

Binkley, D., Lawrie, D., and Morrell, C. (2018). The need for software specific natural language techniques. *Empirical Software Engineering (EMSE)*, 23(4):2398–2425.

Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. (2009). Does distributed development affect software quality? an empirical case study of windows vista. In *Proceedings of the International conference on software engineering (ICSE)*, pages 518–528. ACM/IEEE.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 4–14. ACM.

Biswas, E., Vijay-Shanker, K., and Pollock, L. (2019). Exploring word embedding techniques to improve sentiment analysis of software engineering texts. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 68–78. IEEE.

Blincoe, K., Dehghan, A., Salaou, A.-D., Neal, A., Linaker, J., and Damian, D. (2019). High-level software requirements and iteration changes: a predictive model. *Empirical Software Engineering (EMSE)*, 24(3):1610–1648.

Bonferroni, C. (1936). Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, 8:3–62.

Boslaugh, S. and Watters, P. (2008). *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media.

Boughorbel, S., Jarray, F., and El-Anbari, M. (2017). Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):e0177678.

Breiman, L. (2017). *Classification and regression trees*. Routledge.

Briand, L. C., Daly, J., Porter, V., and Wust, J. (1998). Predicting fault-prone classes with design measures in object-oriented systems. In *Proceedings Ninth International Symposium on Software Reliability Engineering*, pages 334–343. IEEE.

Brier, G. W. (1950). Verification of forecasts expressed in terms of probability. *Monthey Weather Review*, 78(1):1–3.

Buse, R. P. and Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 987–996. ACM/IEEE.

Caglayan, B., Turhan, B., Bener, A., Habayeb, M., Miransky, A., and Cialini, E. (2015). Merits of organizational metrics in defect prediction: an industrial replication. In

Proceedings of the International Conference on Software Engineering (ICSE), pages 89–98. ACM/IEEE.

Calefato, F., Lanubile, F., and Novielli, N. (2019). An empirical assessment of best-answer prediction models in technical q&a sites. *Empirical Software Engineering (EMSE)*, 24(2):854–901.

Calle, M. L. and Urrea, V. (2011). Letter to the editor: stability of random forest importance measures. *Briefings in bioinformatics*, 12(1):86–89.

Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering (TSE)*, 35(6):864–878.

Ceylan, E., Kutlubay, F. O., and Bener, A. B. (2006). Software defect identification using machine learning techniques. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 240–247. IEEE.

Chakraborty, S., Tomsett, R., Raghavendra, R., Harborne, D., Alzantot, M., Cerutti, F., Srivastava, M., Preece, A., Julier, S., Rao, R. M., et al. (2017). Interpretability of deep learning models: a survey of results. In *Proceedings of the SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–6. IEEE.

Chen, C., Liaw, A., Breiman, L., et al. (2004). Using random forest to learn imbalanced data. *University of California, Berkeley*, 110(1-12):24.

Chen, D., Fu, W., Krishna, R., and Menzies, T. (2018). Applications of psychological science for actionable analytics. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 456–467. ACM.

Cohen, J. (1983). The cost of dichotomization. *Applied psychological measurement*, 7(3):249–253.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences . Hilsdale*, volume 2. Hillsdale, N.J. : L. Erlbaum Associates.

Cohen, J. (1992). A power primer. *Psychological bulletin*, 112(1):155.

Dam, H. K., Tran, T., and Ghose, A. (2018). Explainable software analytics. In *Proceedings of the International Conference on Software Engineering (ICSE): New Ideas and Emerging Results*, pages 53–56. ACM/IEEE.

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

Dawson, N. V. and Weiss, R. (2012). Dichotomizing continuous variables in statistical analysis a practice to avoid. *Medical Decision Making*, 32(2):225–226.

de Almeida, M. A., Lounis, H., and Melo, W. L. (1998). An investigation on the use of machine learned models for estimating correction costs. In *Proceedings of the International conference on Software engineering (ICSE)*, pages 473–476. ACM/IEEE.

de González, A. B., Cox, D. R., et al. (2007). Interpretation of interaction: A review. *The Annals of Applied Statistics*, 1(2):371–385.

DeCoster, J., Iselin, A.-M. R., and Gallucci, M. (2009). A conceptual and empirical examination of justifications for dichotomization. *Psychological methods*, 14(4):349.

Dehghan, A., Neal, A., Blincoe, K., Linaker, J., and Damian, D. (2017). Predicting likelihood of requirement implementation within the planned iteration: an empirical study at ibm. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 124–134. IEEE.

Devlin, S., Singh, C., Murdoch, W. J., and Yu, B. (2019). Disentangled attribution curves for interpreting random forests and boosted trees. *arXiv preprint arXiv:1905.07631*.

Dey, T. and Mockus, A. (2018). Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 66–69. ACM.

Efron, B. (1983). Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association*, 78(382):316–331.

El-Emam, K., Goldenson, D., McCurley, J., and Herbsleb, J. (2001). Modelling the likelihood of software process improvement: An exploratory study. *Empirical Software Engineering (EMSE)*, 6(3):207–229.

Fagan, M. E. (1999). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2-3):258–287.

Fan, Y., Xia, X., Lo, D., and Li, S. (2018). Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering (EMSE)*, 23(6):3346–3393.

Ferri, C., Hernández-Orallo, J., and Modroiu, R. (2009). An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38.

Fisher, A., Rudin, C., and Dominici, F. (2018). All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *arXiv preprint arXiv:1801.01489*.

Fisher, D., DeLine, R., Czerwinski, M., and Drucker, S. (2012). Interactions with big data analytics. *interactions*, 19(3):50–59.

Flach, P. A. (2003). The geometry of roc space: understanding machine learning metrics through roc isometrics. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pages 194–201. AAAI.

Folleco, A., Khoshgoftaar, T. M., Van Hulse, J., and Bullard, L. (2008). Software quality modeling: The impact of class noise on the random forest classifier. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, pages 3853–3859. IEEE.

Freeman, G. (1985). The analysis and interpretation of interactions. *Journal of Applied Statistics*, 12(1):3–10.

Friedman, J. H., Popescu, B. E., et al. (2008). Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954.

Fu, W., Menzies, T., and Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology (IST)*, 76:135–146.

Garcia, S., Luengo, J., Saez, J. A., Lopez, V., and Herrera, F. (2013). A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):734–750.

Gay, G., Menzies, T., Davies, M., and Gundy-Burlet, K. (2010). Automatically finding the control variables for complex system behavior. *Automated Software Engineering (ASE)*, 17(4):439–468.

Ghaleb, T. A., da Costa, D. A., and Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering (EMSE)*, 24(4):2102–2139.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 789–800. ACM/IEEE.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2017). A large-scale study of the impact of feature selection techniques on defect classification models. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 146–157. IEEE.

Goldstein, A., Kapelner, A., Bleich, J., and Pitkin, E. (2015). Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1):44–65.

Gousios, G., Pinzger, M., and Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 345–355. ACM/IEEE.

Greenwell, B. M. (2017). pdp: an r package for constructing partial dependence plots. *The R Journal*, 9(1):421–436.

Greenwell, B. M., Boehmke, B. C., and McCarthy, A. J. (2018). A simple and effective model-based variable importance measure. *arXiv preprint arXiv:1805.04755*.

Grömping, U. (2009). Variable importance assessment in regression: linear regression versus random forest. *The American Statistician*, 63(4):308–319.

Guo, L., Ma, Y., Cukic, B., and Singh, H. (2004). Robust prediction of fault-proneness by random forests. In *Proceedings of the International symposium on software reliability engineering*, pages 417–428. IEEE.

Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 495–504. ACM/IEEE.

Hall, M. A. and Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data engineering*, 15(6):1437–1447.

Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2011). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering (TSE)*, 38(6):1276–1304.

Haran, M., Karr, A., Last, M., Orso, A., Porter, A. A., Sanil, A., and Fouche, S. (2007). Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Transactions on Software Engineering (TSE)*, 33(5):287–304.

Harrell Jr, F. E. (2015). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis.* Springer.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 78–88. ACM/IEEE.

Hassan, A. E. and Holt, R. C. (2005). The top ten list: Dynamic fault prediction. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 263–272. IEEE.

Hassan, S., Bezemer, C.-P., and Hassan, A. E. (2018). Studying bad updates of top free-to-download apps in the google play store. *IEEE Transactions on Software Engineering (TSE)*, 46(7):773–793.

Herbsleb, J. D. and Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on software engineering (TSE)*, 29(6):481–494.

Herzig, K. (2014). Using pre-release test failures to build early post-release defect prediction models. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 300–311. IEEE.

Herzig, K., Just, S., and Zeller, A. (2016). The impact of tangled code changes on defect prediction models. *Empirical Software Engineering (EMSE)*, 21(2):303–336.

Hihn, J. and Menzies, T. (2015). Data mining methods and cost estimation models: Why is it so hard to infuse new ideas? In *Proceedings of the International Conference on Automated Software Engineering Workshop (ASEW)*, pages 5–9. IEEE.

Ho, T. K. and Basu, M. (2002). Complexity measures of supervised classification problems. *IEEE Transactions on pattern analysis and machine intelligence*, 24(3):289–300.

Hoang, T., Dam, H. K., Kamei, Y., Lo, D., and Ubayashi, N. (2019). Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE.

Hoekstra, A. and Duin, R. P. (1996). On the nonlinearity of pattern classifiers. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, pages 271–275. IEEE.

Hou, C., Nie, F., Yi, D., and Wu, Y. (2013). Efficient image classification via multiple rank regression. *IEEE Transactions on Image Processing*, 22(1):340–352.

Huang, J., Keung, J. W., Sarro, F., Li, Y.-F., Yu, Y.-T., Chan, W., and Sun, H. (2017). Cross-validation based k nearest neighbor imputation for software quality datasets: An empirical study. *Journal of Systems and Software (JSS)*, 132:226–252.

Huang, J. and Ling, C. X. (2005). Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579.

Jahanshahi, H., Jothimani, D., Başar, A., and Cevik, M. (2019). Does chronology matter in jit defect prediction?: A partial replication study. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 90–99. ACM.

Jalali, O., Menzies, T., and Feather, M. (2008). Optimizing requirements decisions with keys. In *Proceedings of the International workshop on Predictor models in software engineering*, pages 79–86. ACM.

Janitza, S., Strobl, C., and Boulesteix, A.-L. (2013). An auc-based permutation variable importance measure for random forests. *BMC bioinformatics*, 14(1):119.

Jiang, Y., Adams, B., and German, D. M. (2013). Will my patch make it? and how fast? case study on the linux kernel. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 101–110. IEEE.

Jiang, Y., Cukic, B., and Menzies, T. (2008). Can data transformation help in the detection of fault-prone modules? In *Proceedings of the Workshop on Defects in large software systems*, pages 16–20. ACM.

Jiang, Y., Lin, J., Cukic, B., and Menzies, T. (2009). Variance analysis in software fault prediction models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 99–108. IEEE.

Jiarpakdee, J., Tantithamthavorn, C., Dam, H. K., and Grundy, J. (2020). An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, PP(99):1–1.

Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2019). The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, PP(99):1–1.

Jiarpakdee, J., Tantithamthavorn, C., and Treude, C. (2018). Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 92–103. IEEE.

Jimenez, M., Maxime, C., Le Traon, Y., and Papadakis, M. (2018). On the impact of tokenizer and parameters on n-gram based code analysis. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 437–448. IEEE.

Jing, X.-Y., Wu, F., Dong, X., and Xu, B. (2016). An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering (TSE)*, 43(4):321–339.

Judd, T., Ehinger, K., Durand, F., and Torralba, A. (2009). Learning to predict where humans look. In *Proceedings of the International conference on Computer Vision*, pages 2106–2113. IEEE.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering (TSE)*, 39(6):757–773.

Kang, H. J., Bissyandé, T. F., and Lo, D. (2019). Assessing the generalizability of code2vec token embeddings. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE.

Kendall, M. G. (1948). *Rank correlation methods.* Griffin.

Kendler, K. S. and Gardner, C. O. (2010). Interpretation of interactions: guide for the perplexed. *The British Journal of Psychiatry*, 197(3):170–171.

Kim, S., Zhang, H., Wu, R., and Gong, L. (2011). Dealing with noise in defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 481–490. ACM/IEEE.

Kim, S., Zimmermann, T., Whitehead Jr, E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 489–498. ACM/IEEE.

Kitchenham, B., Pfleeger, S. L., and Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Transactions on software Engineering (TSE)*, 21(12):929–944.

Knab, P., Pinzger, M., and Bernstein, A. (2006). Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International workshop on Mining software repositories (MSR)*, pages 119–125. ACM.

Kondo, M., Bezemer, C.-P., Kamei, Y., Hassan, A. E., and Mizuno, O. (2019). The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering (EMSE)*, 24(4):1925–1963.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., and Godfrey, M. W. (2015). Investigating code review quality: Do people and participation matter? In *Proceedings of the International conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE.

Krishna, R., Menzies, T., and Layman, L. (2017). Less is more: Minimizing code reorganization using xtree. *Information and Software Technology (IST)*, 88:53–66.

Krstajic, D., Buturovic, L. J., Leahy, D. E., and Thomas, S. (2014). Cross-validation pitfalls when selecting and assessing regression and classification models. *Journal of cheminformatics*, 6(1):10.

Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621.

Kuhn, M. (2012). Variable importance using the caret package. *Journal of Statistical Software*.

Kuhn, M. et al. (2008). Building predictive models in r using the caret package. *Journal of statistical software*, 28(5):1–26.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering (TSE)*, 34(4):485–496.

Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Whitehead, E. J. (2013). Does bug prediction support human developers? findings from a google case study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 372–381. ACM/IEEE.

Li, H., Shang, W., Zou, Y., and Hassan, A. E. (2017). Towards just-in-time suggestions for log changes. *Empirical Software Engineering (EMSE)*, 22(4):1831–1865.

Li, Y., Jiang, Z. M., Li, H., Hassan, A. E., He, C., Huang, R., Zeng, Z., Wang, M., and Chen, P. (2020). Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–24.

Lipton, Z. C. (2016). The mythos of model interpretability. *arXiv preprint arXiv:1606.03490*.

Lundberg, S. M., Erion, G. G., and Lee, S.-I. (2018). Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774.

Ma, S., Liu, Y., Lee, W.-C., Zhang, X., and Grama, A. (2018). Mode: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 175–186. ACM.

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

MacCallum, R. C., Zhang, S., Preacher, K. J., and Rucker, D. D. (2002). On the practice of dichotomization of quantitative variables. *Psychological methods*, 7(1):19.

Malgonde, O. and Chari, K. (2019). An ensemble-based model for predicting agile software development effort. *Empirical Software Engineering (EMSE)*, 24(2):1017–1055.

Malhotra, R. and Khanna, M. (2017). An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering (EMSE)*, 22(6):2806–2851.

Marks, L., Zou, Y., and Hassan, A. E. (2011). Studying the fix-time for bugs in large open source projects. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, pages 1–8. ACM.

Martens, D. and Maalej, W. (2019). Towards understanding and detecting fake reviews in app stores. *Empirical Software Engineering (EMSE)*, 24(6):3316–3355.

McIntosh, S. and Kamei, Y. (2017). Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 44(5):412–428.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 192–201. ACM.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering (EMSE)*, 21(5):2146–2189.

Medin, D. L. and Schwanenflugel, P. J. (1981). Linear separability in classification learning. *Journal of Experimental Psychology: Human Learning and Memory*, 7(5):355.

Mende, T. (2010). Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the International Conference on Predictive Models in Software Engineering*, pages 1–10. ACM.

Menzies, T. (2019). The five laws of se for ai. *IEEE Software*, 37(1):81–85.

Menzies, T., Bird, C., Zimmermann, T., Schulte, W., and Kocaganeli, E. (2011). The inductive software engineering manifesto: principles for industrial data mining. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, pages 19–26. ACM.

Menzies, T., Dekhtyar, A., Distefano, J., and Greenwald, J. (2007a). Problems with precision: A response to "comments on'data mining static code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering (TSE)*, 33(9):637–640.

Menzies, T., Greenwald, J., and Frank, A. (2006). Data mining static code attributes to learn defect predictors. *IEEE Transactions on software engineering (TSE)*, 33(1):2–13.

Menzies, T., Owen, D., and Richardson, J. (2007b). The strangest thing about software. *Computer*, 40(1).

Menzies, T. and Zimmermann, T. (2013). Software analytics: so what? *IEEE Software*, 30(4):31–37.

Mockus, A. (2010). Organizational volatility and its effects on software defects. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 117–126. ACM.

Molnar, C. (2018). Interpretable machine learning. *A Guide for Making Black Box Models Explainable*, 7.

Morales, R., McIntosh, S., and Khomh, F. (2015). Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180. IEEE.

Mori, T. and Uchihira, N. (2019). Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering (EMSE)*, 24(2):779–825.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 181–190. ACM/IEEE.

Mossman, D. (1994). Assessing predictions of violence: being accurate about accuracy. *Journal of consulting and clinical psychology*, 62(4):783.

Mullen, R. E. and Gokhale, S. S. (2005). Software defect rediscoveries: a discrete lognormal model. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 10pp–212. IEEE.

Muller, M. P., Tomlinson, G., Marrie, T. J., Tang, P., McGeer, A., Low, D. E., Detsky, A. S., and Gold, W. L. (2005). Can routine laboratory tests discriminate between severe acute respiratory syndrome and other causes of community-acquired pneumonia? *Clinical infectious diseases*, 40(8):1079–1086.

Muthukumaran, K., Rallapalli, A., and Murthy, N. B. (2015). Impact of feature selection techniques on bug prediction models. In *Proceedings of the India Software Engineering Conference*, pages 120–129. ACM.

Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the International conference on Software engineering (ICSE)*, pages 284–292. ACM/IEEE.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the International conference on Software engineering (ICSE)*, pages 452–461. ACM/IEEE.

Nam, J., Fu, W., Kim, S., Menzies, T., and Tan, L. (2018). Heterogeneous defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 44(09):874–896.

Nam, J. and Kim, S. (2015). Clami: Defect prediction on unlabeled datasets (t). In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE.

Nicodemus, K. K. (2011). Letter to the editor: On the stability and ranking of predictors from random forest variable importance measures. *Briefings in bioinformatics*, 12(4):369–373.

Niedermayr, R. and Wagner, S. (2019). Is the stack distance between test case and method correlated with test effectiveness? In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 189–198. ACM.

Othmane, L. B., Chehrazi, G., Bodden, E., Tsalovski, P., and Brucker, A. D. (2017). Time for addressing software security issues: Prediction models and impacting factors. *Data Science and Engineering*, 2(2):107–124.

Panichella, A., Oliveto, R., and De Lucia, A. (2014). Cross-project defect prediction models: L'union fait la force. In *Proceedings of the Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 164–173. IEEE.

Peduzzi, P., Concato, J., Kemper, E., Holford, T. R., and Feinstein, A. R. (1996). A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379.

Pelayo, L. and Dick, S. (2012). Evaluating stratification alternatives to improve software defect prediction. *IEEE Transactions on reliability (TRE)*, 61(2):516–525.

Peters, F., Menzies, T., Gong, L., and Zhang, H. (2013). Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 39(8):1054–1068.

Peters, F., Tun, T., Yu, Y., and Nuseibeh, B. (2017). Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering (TSE)*, 45(6):615–631.

Premraj, R. and 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*, pages 215–224. IEEE.

Putin, E., Mamoshina, P., Aliper, A., Korzinkin, M., Moskalev, A., Kolosov, A., Ostrovskiy, A., Cantor, C., Vijg, J., and Zhavoronkov, A. (2016). Deep biomarkers of human aging: application of deep neural networks to biomarker development. *Aging (Albany NY)*, 8(5):1021.

Rajbahadur, G. K., Wang, S., Kamei, Y., and Hassan, A. E. (2017). The impact of using regression models to build defect classifiers. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 135–145. IEEE.

Rajbahadur, G. K., Wang, S., Kamei, Y., and Hassan, A. E. (2019). Impact of discretization noise of the dependent variable on machine learning classifiers in software engineering. *IEEE Transactions on Software Engineering (TSE)*, PP(99):1–1.

Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. (2016). On the “naturalness" of buggy code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 428–439. ACM/IEEE.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the international conference on knowledge discovery and data mining*, pages 1135–1144. ACM.

Royston, P., Altman, D. G., and Sauerbrei, W. (2006). Dichotomizing continuous predictors in multiple regression: a bad idea. *Statistics in medicine*, 25(1):127–141.

Rucker, D. D., McShane, B. B., and Preacher, K. J. (2015). A researcher’s guide to regression, discretization, and median splits of continuous variables. *Journal of Consumer Psychology*, 25(4):666–678.

Sakia, R. (1992). The box-cox transformation technique: a review. *The statistician*, 41(2):169–178.

Santos, J. and Belo, O. (2013). Estimating risk management in software engineering projects. In *Proceedings of the Industrial Conference on Data Mining*, pages 85–98. Springer.

Sayyad Shirabad, J. and Menzies, T. (2005). The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada.

Schumann, J., Gundy-Burlet, K., Pasareanu, C., Menzies, T., and Barrett, T. (2009). Software v&v support by parametric analysis of large software simulation systems. In *Proceedings of the IEEE Aerospace Conference*, pages 1–8. IEEE.

Seiffert, C., Khoshgoftaar, T. M., and 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.

Shepperd, M., Bowes, D., and Hall, T. (2014). Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 40(6):603–616.

Shihab, E., Hassan, A. E., Adams, B., and Jiang, Z. M. (2012). An industrial study on the risk of software changes. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, page 62. ACM.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering (EMSE)*, 18(5):1005–1042.

Shihab, E., Mockus, A., Kamei, Y., Adams, B., and Hassan, A. E. (2011). High-impact defects: a study of breakage and surprise defects. In *Proceedings of the Joint Meeting* *on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 300–310. ACM.

Shimagaki, J., Kamei, Y., McIntosh, S., Hassan, A. E., and Ubayashi, N. (2016). A study of the quality-impacting practices of modern code review at sony mobile. In *Proceedings of the International Conference on Software Engineering (ICSE) Companion*, pages 212–221. ACM/IEEE.

Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., and Zelkowitz, M. (2002). What we have learned about fighting defects. In *Proceedings of the International Symposium on Software Metrics (METRICS)*, pages 249–258.

Singh, R., Jaakkola, T., and Mohammad, A. (2006). 6.867 machine learning. fall 2006.

Sokolova, M., Japkowicz, N., and Szpakowicz, S. (2006). Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In *Proceedings of the Australasian joint conference on artificial intelligence*, pages 1015–1021. Springer.

Song, Q., Guo, Y., and Shepperd, M. (2018). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 45(12):1253–1269.

Song, Q., Jia, Z., Shepperd, M., Ying, S., and Liu, J. (2010). A general software defect-proneness prediction framework. *IEEE transactions on software engineering (TSE)*, 37(3):356–370.

Strobl, C., Boulesteix, A.-L., Kneib, T., Augustin, T., and Zeileis, A. (2008). Conditional variable importance for random forests. *BMC bioinformatics*, 9(1):307.

Strobl, C., Boulesteix, A.-L., Zeileis, A., and Hothorn, T. (2007). Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC bioinformatics*, 8(1):25.

Subramanyam, R. and Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering (TSE)*, 29(4):297–310.

Tan, S.-Y. and Chan, T. (2016). Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics. *arXiv preprint arXiv:1606.03510*.

Tantithamthavorn, C. (2016a). Scottknottesd: The scott-knott effect size difference (esd) test. *R package version*, 2.

Tantithamthavorn, C. (2016b). *Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Models*. PhD thesis, Nara Institute of Science and Technology.

Tantithamthavorn, C. and Hassan, A. E. (2018). An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the International Conference on Software Engineering (ICSE): Software Engineering in Practice*, pages 286–295. ACM/IEEE.

Tantithamthavorn, C., Hassan, A. E., and Matsumoto, K. (2018a). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, PP(99):1–1.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., Ihara, A., and Matsumoto, K. (2015). The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the International Conference on Software Engineering*, volume 1, pages 812–823. ACM/IEEE.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016). Comments on "researcher bias: the use of machine learning in software defect prediction". *IEEE Transactions on Software Engineering (TSE)*, 42(11):1092–1094.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2017). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 43(1):1–18.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2018b). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 45(7):683–711.

Thakkar, D., Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). Retrieving relevant reports from a customer engagement repository. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 117–126. IEEE.

Theisen, C., Herzig, K., Morrison, P., Murphy, B., and Williams, L. (2015). Approximating attack surfaces with stack traces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 199–208. ACM/IEEE.

Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1039–1050. ACM/IEEE.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? a case study on free android applications. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–310. IEEE.

Treude, C. and Wagner, M. (2019). Predicting good configurations for github and stack overflow topic models. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 84–95. IEEE.

Tu, H. and Nair, V. (2018). While tuning is good, no tuner is best. *FSE SWAN*.

Turhan, B. (2012). On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering (EMSE)*, 17(1-2):62–74.

Wang, H. and Song, M. (2011). Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming. *The R journal*, 3(2):29.

Wang, S., Chen, T.-H., and Hassan, A. E. (2018). Understanding the factors for fast answers in technical q&a websites. *Empirical Software Engineering (EMSE)*, 23(3):1552–1593.

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

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83.

Wright, M. N., Ziegler, A., and König, I. R. (2016). Do little interactions get lost in dark random forests? *BMC bioinformatics*, 17(1):145.

Xiang, S., Nie, F., and Zhang, C. (2010). Semi-supervised classification via local spline regression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(11):2039–2053.

Xu, Z., Liu, J., Yang, Z., An, G., and Jia, X. (2016). The impact of feature selection on defect prediction performance: An empirical comparison. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–320. IEEE.

Yu, L. and Liu, H. (2004). Efficient feature selection via analysis of relevance and redundancy. *Journal of machine learning research (JMLR)*, 5(Oct):1205–1224.

Yu, T., Wen, W., Han, X., and Hayes, J. (2019). Conpredictor: Concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering (TSE)*, 45(6):558–575.

Zhang, F., Hassan, A. E., McIntosh, S., and Zou, Y. (2016). The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 43(5):476–491.

Zhang, F., Keivanloo, I., and Zou, Y. (2017). Data transformation in cross-project defect prediction. *Empirical Software Engineering (EMSE)*, 22(6):3186–3218.

Zhang, F., Khomh, F., Zou, Y., and Hassan, A. E. (2012). An empirical study on factors impacting bug fixing time. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 225–234. IEEE.

Zhang, F., Mockus, A., Keivanloo, I., and Zou, Y. (2014). Towards building a universal defect prediction model. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 182–191. ACM.

Zimmermann, T. and Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 531–540. ACM/IEEE.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B. (2009). Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, pages 91–100. ACM.

Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1074–1083. ACM/IEEE.

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. ACM/IEEE.

Appendices

A A running example for our framework

We provide additional details for each step of our framework. In addition, we provide a running example outlining how each step could be applied for a dataset.

Step 1: Correlation and Redundancy Analysis

We take the Stack Overflow dataset and perform a correlation and redundancy analysis on the 65 independent features of the Stack Overflow dataset and arrive at 28 features. Redundant variables are detected by fitting preliminary models that explain each predictor using other predictors. The R^2 value of the preliminary models that are constructed is used as a measure to observe how well each predictor is explained using other predictors.

Step 2: Discretization

For the Stack Overflow dataset, when the median (21.38 mins) of the dependent variable is used as the discretization threshold to discretize the data points into "class 1" and "class 2" classes. The noisy area for the dataset is defined by choosing a *limit* value. We estimate the limit value with our Algorithm 1 that is presented in our paper. To estimate the limit value, we need to choose the value of *step_size* for our automated limit estimation algorithm. We choose *step_size = 2*. We input the chosen *step_size* and the threshold to our algorithm. Lets consider if our algorithm reports 0.50 as the limit, then we choose $limit = 0.50$ and use the formula $cutpoint \pm limit * cutpoint$ to calculate the noisy area, as $21.38 - (21.38 * 0.50)$ to $21.38 - (21.38 * 0.50)$ which is 10.70 to 32.10 mins.

Step 3: Classifier construction

In this step, we construct a classifier on the independent features that we obtained from step 1 and dependent variable which we obtained from step 2. Firstly, we construct a classifier on the whole dataset and next we construct a classifier on the dataset with the noisy area of the Stack Overflow dataset (i.e., data points within the range 10.70 to 32.10 mins) is removed incrementally as presented in the Section 5.

Step 4: Performance evaluation

We evaluate the performance of the RFCMs that are constructed on the Stack Overflow dataset (both on the whole dataset and the dataset without noisy area) by computing their performance measures (Accuracy, Precision, Recall, AUC, Brier score). These performance measures are then compared to analyse the impact of discretization noise on the performance of the classifiers. For instance, if a one decides to observe the performance impact of discretization noise in terms of AUC, then if the AUC if the of an RFCM that is constructed on the whole Stack Overflow dataset is 0.85 and the AUC of an RFCM classifier that is constructed on the Stack Overflow dataset without noisy area (i.e., data points within the range 10.70 to 32.10 mins) is 0.90. Then we estimate that the discretization noise impacts the performance. If the discretization noise does impact the performance as it is in our example, our framework reports the percentage of data from the noisy area, that when dropped, starts statistically impacting the given performance measure. In our example if the impact starts occuring from 10%, then we only need to drop the data points within the range of 19.24 mins to 23.51 mins and not the whole noisy area to avoid impacting the AUC of the constructed RFCM classifier.

Step 5: Feature importance calculation

Similar to step 4, we evaluate the feature importance of the RFCMs that are constructed on the Stack Overflow dataset (both on the whole dataset and on the dataset without the noisy area) by computing their derived feature importance ranks. For instance, if an RFCM that is constructed on the whole Stack Overflow dataset has *A_median_time_answer_sofar, total_upVote, title_length* features as its top features. Whereas an RFCM that is constructed on the Stack Overflow dataset without the noisy area (i.e., data points within the range 10.70 to 32.10 mins) also has the same features as its top features. Then, we say that the interpretation of the classifier is not impacted by the discretization noise. If the generated top features had changed, then we interpret that as a sign that the discretization noise impacts the feature importance of the studied classifiers.

Step 6: Inference validation

We repeat step 3 to step 5 100 times using the out-of-sample bootstrap method outlined above for the classifiers that are constructed on the Stack Overflow dataset (both on the whole dataset and the dataset without noisy area). The generated results for all 100 iterations are taken together for consideration to draw an inference.

B Supplementary figures and tables

B.1 Understanding the impact of discretization noise on the performance and interpretation of a classifier - experiment setup

Figure B.1 depicts the experimental setup for the incremental performance and feature analysis. We also present the results for impact of discretization noise that is generated by the **Median based discretization threshold (MT), univariate clustering based threshold (CT) Wang and Song (2011) and CART based discretization threshold (RTT)** on the performance and interpretation of the studied classifiers. More details about the thresholds are available in the Section 5.3.2 of our paper.

B.2 Studying the impact of discretization noise on the performance of a classifier - additional result tables

Table B.1 and B.2 present the impact of the discretization noise that is generated by the univariate **clustering based threshold (CT) and CART based discretization threshold (RTT)** on various performance measures for all the four classifiers. We observe performance impacts similar to the results presented in our paper.

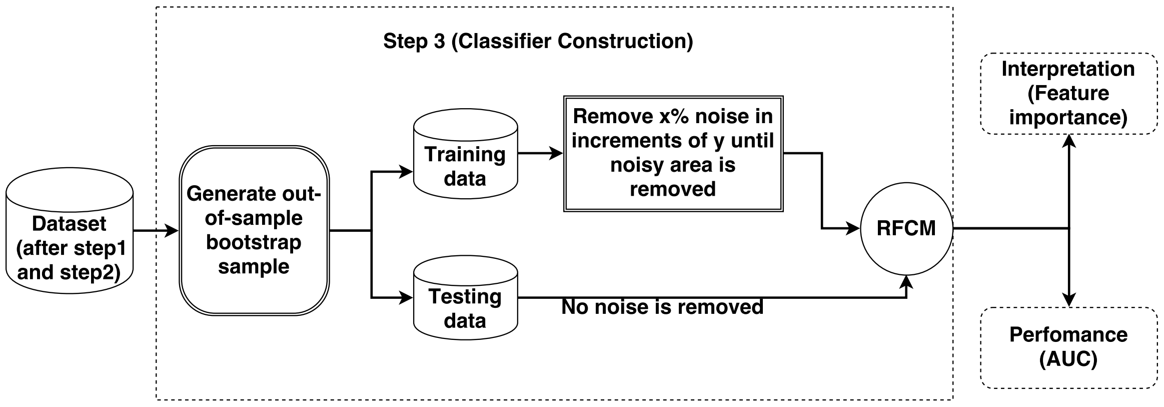

Step 3 (Classifier Construction)
Dataset (after step1 and step2)
Generate out-of-sample bootstrap sample
Training data
Testing data
Remove x% noise in increments of y until noisy area is removed
No noise is removed
RFCM
Interpretation (Feature importance)
Perfomance (AUC)

Figure B.1: Detailed overview of the classifier construction step for Section 4.

Table B.1: Percentage of improvement in median performance of various classifiers with noisy area (generated with CT) removed over classifiers with no data removed across various performance measures (The x value for which the performance impact first occurs for the given measure is also provided).

Classifier	**Dataset**	**ACC (%)**		**PRC (%)**		**RCL (%)**		**BS (%)**		**AUC (%)**		**F-M (%)**		**MCC (%)**	
		Mag	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x
RF	SO	**-2.77**	65	**4.51**	30	**-10.47**	30	**-1.65**	5	**1.2**	70	**-2.62**	45	**9.16**	25
	MA	**-4.2**	55	**5.49**	20	**-14.63**	20	**-2.21**	5	1.22$^{\#}$	70	**-4.01**	40	**7.01**	20
	AU	**-10.45**	75	**6.45**	60	**-26.97**	50	**7.09**	5	**1.37**	80	**-8.77**	70	**21.88**	45
	SU	**-18.9**	65	**6.86**	45	**-40.37**	30	**6.54**	5	**1.39**	65	**-14.65**	60	**15.69**	40
	PH	**-1.01**	10	**0.92**	30	**-10.51**	10	**-1.43**	5	**-1.04**	25	**-5.32**	10	**-5.22**	10
	BF	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	AR	-0.44§	0	**-4.07**	2	**10.17**	2	**-2.38**	0.5	0	3	**5.11**	7.5	-1.25	0
LR	SO	**-1.42**	75	**3.98**	40	**-8.44**	25	**0.8**	5	0	0	**-1.69**	55	**24.68**	40
	MA	**-2.81**	65	**4.46**	25	**-11.7**	15	**-0.29**	5	1.32$^{\#}$	85	**-2.91**	45	**16.51**	25
	AU	**-10.01**	75	**3.83**	80	**-21.75**	50	**10.57**	5	0.76$^{\#}$	95	**-7.57**	70	**40.06**	75
	SU	**-13.79**	75	**3.82**	70	**-26.15**	30	**13.61**	5	1.56$^{\#}$	90	**-9.93**	65	**32.47**	55
	PH	-0.24$^{\#}$	40	**3.9**	20	**-7**	15	**-1.9**	5	0§	0	**-2.07**	30	**-1.48**	35
	BF	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	AR	-0.68$^{\#}$	10	**-8.66**	7.5	**41.03**	5	**-5.19**	0.5	0	0	**27.95**	5	**10.69**	5
CART	SO	**-1.6**	20	**4.18**	30	**-8.35**	45	**-6.02**	45	1.3$^{\#}$	70	**-1.9**	20	**17.25**	20
	MA	**-2.81**	20	**4.98**	25	**-11.92**	35	**-8.8**	50	**1.32**	65	**-3.11**	25	**16.2**	20
	AU	**-10.31**	30	**4.34**	65	**-23.2**	75	-0.34	0	**4.48**	90	**-8.77**	80	**23.23**	40
	SU	**-17.04**	50	**5.61**	55	**-35.14**	65	-3.86$^{\#}$	50	**4.48**	95	**-13.93**	70	**31.1**	20
	PH	**-0.93**	10	0.97§	0	**-9.7**	10	-2.42$^{\#}$	25	**-6.98**	10	**-4.59**	10	**-4.96**	10
	BF	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	AR	-0.92$^{\#}$	10	-1.95$^{\#}$	10	**7.77**	8	-3.08§	0	0.82	0	4.21$^{\#}$	8	1.85	0
KNN	SO	**-1.83**	10	**2.75**	50	**-7.06**	5	**1.24**	5	**4.23**	35	**-1.88**	10	**22.56**	30
	MA	**-3.07**	10	**3.61**	30	**-10.66**	5	0.12$^{\#}$	5	**5.8**	15	**-3.3**	5	**22.94**	15
	AU	**-12.32**	20	**2.37**	80	**-22.94**	15	**7.4**	10	**5.08**	60	**-9.96**	15	**39.16**	65
	SU	**-17.17**	15	**3.05**	75	**-31.05**	10	**7.77**	5	**6.56**	60	**-13.36**	10	**28.07**	55
	PH	**0.68**	20	**3.39**	25	**-9.05**	10	**-1.4**	10	**-2.67**	15	**-3.65**	20	-2.35$^{\#}$	40
	BF	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	AR	0.02	1.5	0.34	4	-0.21	1.5	**-1.36**	1	0	0	-0.1	2	0.22	0

1. **Performance Measures:** ACC- Accuracy, PRC- Precision, RCL- Recall, BS- Brier Score, F-M- F-Measure
2. **Datasets:** SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BF- Bug-fix, AR- App-rating
3. Mag- Magnitude of the performance impact |x- % of data of the noisy area when dropped starts statistically impacting the given performance measure
4. **Cohen's d effect size:** Negligible - No formatting, Small -§, Medium -$^{\#}$, Large - **bold**
5. '−' indicates performance measure decreases due to removal of noisy area; '+' indicates performance measure increases due to removal of noisy area
6. All the values with small, medium or large effect size are statistically significant with $p \leq 0.05$
7. * Noisy area was not demarcated for Bug-fix dataset as given in Table 5.2 of our Chapter 5
8. '−'in cases of Brier score indicates an actual increase in the Brier score and '+' a decrease in Brier score (Lower the Brier score, the lesser the error)

Table B.2: Percentage of improvement in median performance of various classifiers with noisy area (generated with RTT) removed over classifiers with no data removed across various performance measures (The x value for which the performance impact first occurs for the given measure is also provided).

Classifier	**Dataset**	**ACC (%)**		**PRC (%)**		**RCL (%)**		**BS (%)**		**AUC (%)**		**F-M (%)**		**MCC (%)**	
		Mag	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x	**Mag**	x
RF	SO	**-10.76**	65	**6.9**	30	**-24.63**	20	**1.8**	5	0§	85	**-8.13**	55	**10.01**	25
	MA	**-6.46**	65	**4.52**	35	**-14.26**	30	**2.02**	5	**0**	85	**-4.36**	55	**18.86**	35
	AU	**-9.96**	80	**6.27**	60	**-25.31**	50	**7.28**	10	**1.37**	80	**-8.17**	70	**22.58**	50
	SU	**-12.15**	75	**5.07**	50	**-24.01**	35	**8.33**	5	**1.37**	70	**-8.46**	70	**20.81**	45
	PH	**-5.59**	10	0.2§	15	**-7.29**	5	**-3.36**	5	**-4.76**	10	**-3.38**	10	**-31.29**	10
	BF	0.17	95	0.15	95	0	70	-0.73$^{\#}$	10	0§	100	0.08	90	2.12	95
	AR	**-0.84**	2.5	**-6.09**	1	**4.08**	2.5	**-3.67**	0.5	**-1.47**	11.5	1.49§	3	**-5.9**	2
LR	SO	**-8.38**	70	**5.92**	45	**-20.68**	30	**5.62**	5	0	0	**-6.35**	60	**32.78**	45
	MA	**-3.86**	70	**2.92**	55	**-8.82**	40	**3.08**	5	0$^{\#}$	90	**-2.57**	60	**29.93**	45
	AU	**-9.35**	75	**3.69**	80	**-20.19**	50	**10.65**	10	1.52$^{\#}$	95	**-6.98**	70	**40.12**	75
	SU	**-6.19**	75	**2.46**	75	**-11.83**	45	**12.96**	5	0$^{\#}$	90	**-4.1**	75	**36.84**	60
	PH	**-2.95**	25	**0.87**	20	**-4.95**	20	**-3.77**	5	0§	20	**-1.85**	25	**-14.48**	30
	BF	**-6.99**	90	**2.44**	95	**-13.51**	60	**6.5**	10	0	0	**-4.87**	85	**31.14**	85
	AR	0.07	8.5	-2.99§	4	**75.2**	3.5	**-6.91**	0.5	0	0	**68.13**	3.5	**51.24**	3.5
CART	SO	**-8.94**	20	**5.99**	35	**-20.63**	55	**-4.06**	35	**2.56**	85	**-6.99**	20	**20.25**	20
	MA	**-5.9**	25	**3.81**	30	**-12.55**	65	-3.19$^{\#}$	35	1.32§	0	**-4.11**	30	**25.89**	15
	AU	**-9.81**	25	**4.42**	65	**-21.85**	75	-3.38§	0	**4.48**	90	**-8.26**	80	**22.67**	35
	SU	**-11.2**	75	**3.61**	65	**-20.65**	70	-3.32§	55	1.52$^{\#}$	95	**-8**	75	**27.22**	55
	PH	**-4.77**	15	0.4$^{\#}$	60	**-6.42**	20	-3.82$^{\#}$	45	**-7.43**	35	**-2.97**	15	**-18.89**	10
	BF	**-14.1**	90	**1.99**	100	**-21.65**	25	**5.5**	25	1.61§	0	**-9.58**	90	-1.48	0
	AR	-1.04§	5.5	-2.47§	5.5	4.53§	13.5	**-12.71**	7	-1.67§	0	2.43§	13	-2.61	2
KNN	SO	**-8.37**	20	**4.46**	60	**-18.22**	10	**5.4**	5	**5.56**	45	**-6.3**	15	**32.63**	45
	MA	**-4.85**	10	**2.66**	55	**-9.54**	5	**2.31**	5	**7.25**	50	**-3.33**	10	**37.1**	15
	AU	**-11.98**	20	**2.23**	80	**-22.12**	15	**7.43**	10	**5.08**	60	**-9.45**	20	**36.63**	65
	SU	**-10.49**	10	**1.91**	70	**-17.05**	10	**7.16**	5	**6.56**	45	**-7.19**	10	**32.1**	25
	PH	**-2.41**	30	0.06§	0	**-3.16**	30	**-3.2**	5	**-1.41**	10	**-1.48**	30	**-10.53**	30
	BF	**-8.37**	35	**1.14**	100	**-13.16**	20	**7.03**	10	**5.17**	100	**-5.64**	35	**38.8**	100
	AR	0.23	2.5	-0.28§	0	**-7.64**	1.5	**-2.01**	1	-0.88$^{\#}$	13.5	**-4.64**	3	-10.65$^{\#}$	15

1. **Performance Measures:** ACC- Accuracy, PRC- Precision, RCL- Recall, BS- Brier Score, F-M- F-Measure
2. **Datasets:** SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BF- Bug-fix, AR- App-rating
3. Mag- Magnitude of the performance impact |x- % of data of the noisy area when dropped starts statistically impacting the given performance measure
4. **Cohen's d effect size:** Negligible - No formatting, Small -§, Medium -$^{\#}$, Large - **bold**
5. '−' indicates performance measure decreases due to removal of noisy area; '+' indicates performance measure increases due to removal of noisy area
6. All the values with small, medium or large effect size are statistically significant with $p \leq 0.05$
7. '−'in cases of Brier score indicates an actual increase in the Brier score and '+' a decrease in Brier score (Lower the Brier score, the lesser the error)

B.3 Studying the impact of discretization noise on the interpretation of a classifier - additional result tables

Table B.3, B.4 and B.5 presents the impact of discretization noise that is generated by the **Median based discretization threshold (MT), univariate clustering based threshold (CT) and CART based discretization threshold (RTT)** on the interpretation of all the four classifiers. We observe that the impact on interpretation for all the studied classifiers on all the discretization thresholds follow the same trend. Furthermore, the presented results are similar to the results and inferences presented in our paper.

B.4 Experiment setup of Section 5.5.1

Figure B.2 shows the overview of the experiment setup for answering why are the classifiers that are trained on the noisy area able to perform well on extremes.

B.5 An overview of the complexity metrics that are used in Section 5.5.2

Table B.6 gives a brief description of the complexity metrics used in Section 5.5.2

B.6 Experimental setup of Section 5.5.2

Figure B.3 gives a detailed overview of the experiment setup for answering why are the classifiers that are trained on the noisy area are unable to perform well on the noisy area.

Table B.3: The likelihood of rank shifts in the top 3 most important ranks (column A) and the comparison of the derived feature importance ranks of a classifier that is trained on the whole dataset (Rank$_W$) and a classifier that is trained on the dataset with the noisy area (with MT) removed (Rank$_{NR}$) (column B).

Classifier	**Dataset**	**Rank shift likelihood (A)**			**Rank$_W$ vs. Rank$_{NR}$ (B)**	
		Rank 1	**Rank 2**	**Rank 3**	**p-value**	**Cohen's d**
RFCM	SO	0	0	0	0	-1.29 (L)
	MA	0	0	0	0	-2.15 (L)
	AU	0	0	0	0	-2.28 (L)
	SU	0	0	0	0.02	-0.74 (M)
	PH	0	0	0	0.04	-0.62 (M)
	BF	0	0	0	0	-1.4 (L)
	AR	0	0	0	1	-0.31 (S)
LR	SO	0	0	0	0	-1.95 (L)
	MA	0	0	0	0	-1.19 (L)
	AU	0	0	0.02	0	-3.04 (L)
	SU	0	0	0	0	-1.82 (L)
	PH	0	0	0	0	-1.85 (L)
	BF	0	0	0	0	-1.5 (L)
	AR	0	0	0	0	-2.13 (L)
CART	SO	0	0	0	0	-2.29 (L)
	MA	0	0	0	0	-2.08 (L)
	AU	0	0	0	0	-1.43 (L)
	SU	0	0	0	0	-1.06 (L)
	PH	0	0	0	0	-1.07 (L)
	BF	0	0	0	0	-3.03 (L)
	AR	0	0	0	0.01	-1.01 (L)
KNN	SO	0	0	0	0	-1.68 (L)
	MA	0	0	0	0	-1.72 (L)
	AU	0	0	0	0	-3.12 (L)
	SU	0	0	0	0.01	-0.92 (L)
	PH	0	0	0.16	0	-2.14 (L)
	BF	0	0	0	0	-2.39 (L)
	AR	0	0	0	0	-1.41 (L)

Datasets: SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BF- Bug-fix, AR- App-rating

Table B.4: The likelihood of rank shifts in the top 3 most important ranks (column A) and the comparison of the derived feature importance ranks of a classifier that is trained on the whole dataset (Rank$_W$) and a classifier that is trained on the dataset with the noisy area (with CT) removed (Rank$_{NR}$) (column B).

Classifier	**Dataset**	**Rank shift likelihood (A)**			**Rank$_W$ vs. Rank$_{NR}$ (B)**	
		Rank 1	**Rank 2**	**Rank 3**	**p-value**	**Cohen's d**
RFCM	SO	0	0	0	0	-2.7 (L)
	MA	0	0	0	0	-2.56 (L)
	AU	0	0	0	0	-2.44 (L)
	SU	0	0	0	0	-1.99 (L)
	PH	0	0	0	0	-1.81 (L)
	BF	*	*	*	*	*
	AR	0	0	0	0	-2.21 (L)
LR	SO	0	0	0	0	-1.14 (L)
	MA	0	0	0	0	-2.01 (L)
	AU	0	0	0	0	-2 (L)
	SU	0	0	0	0	-2.25 (L)
	PH	0	0	0	0	-1.41 (L)
	BF	*	*	*	*	*
	AR	0	0	0	0	-1.41 (L)
CART	SO	0	0	0	0	-1.49 (L)
	MA	0	0	0	0	-1.15 (L)
	AU	0	0	0.01	0	-3.16 (L)
	SU	0	0	0	0	-3.12 (L)
	PH	0	0	0	0	-1.07 (L)
	BF	*	*	*	*	*
	AR	0	0	0	0	-1.85 (L)
KNN	SO	0	0	0	0	-1.80 (L)
	MA	0	0	0	0	-1.00 (L)
	AU	0	0	0	0	-1.38 (L)
	SU	0	0	0	0	-1.53 (L)
	PH	0	0	0	0	-1.99 (L)
	BF	*	*	*	*	*
	AR	0	0	0	0	-2.32 (L)

1. **Datasets:** SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BF- Bug-fix, AR- App-rating

2. * Noisy area was not demarcated for Bug-fix dataset as given in Table 5.2 of our Chapter 5

Table B.5: The likelihood of rank shifts in the top 3 most important ranks (column A) and the comparison of the derived feature importance ranks of a classifier that is trained on the whole dataset (Rank_W) and a classifier that is trained on the dataset with the noisy area (with RTT) removed (Rank_{NR}) (column B).

Classifier	**Dataset**	**Rank shift likelihood (A)**			**Rank_W vs. Rank_{NR} (B)**	
		Rank 1	**Rank 2**	**Rank 3**	**p-value**	**Cohen's d**
RFCM	SO	0	0	0	0	-2.31 (L)
	MA	0	0	0	0	-2.72 (L)
	AU	0	0	0	0	-1.64 (L)
	SU	0	0	0	0	-2.68 (L)
	PH	0	0	0	0	-1.1 (L)
	BF	0	0	0	0	-1.81 (L)
	AR	0	0	0	0	-2.45 (L)
LR	SO	0	0	0	0	-1.36 (L)
	MA	0	0	0	0	-1.75 (L)
	AU	0	0	0	0	-1.59 (L)
	SU	0	0	0	0	-2.63 (L)
	PH	0	0	0	0	-1.29 (L)
	BF	0	0	0	0	-1.84 (L)
	AR	0	0	0	0	-1.43 (L)
CART	SO	0	0	0	0	-1.73 (L)
	MA	0	0	0	0	-4.06 (L)
	AU	0	0	0.01	0	-1.75 (L)
	SU	0	0	0	0	-2.76 (L)
	PH	0	0	0	0.15	-0.52 (M)
	BF	0	0	0.01	0	-1.6 (L)
	AR	0	0	0	0	-1.57 (L)
KNN	SO	0	0	0	0	-1.76 (L)
	MA	0	0	0	0.14	-0.50 (M)
	AU	0	0	0	0	-1.70 (L)
	SU	0	0	0	0	-1.39 (L)
	PH	0	0	0	0	-2.13 (M)
	BF	0	0	0	0	-2.45(L)
	AR	0	0	0	0	-1.87 (L)

Datasets: SO- Stack Overflow, MA- Mathematics, AU- Ask Ubuntu, SU- Super User, PH- Patch, BF- Bug-fix, AR- App-rating

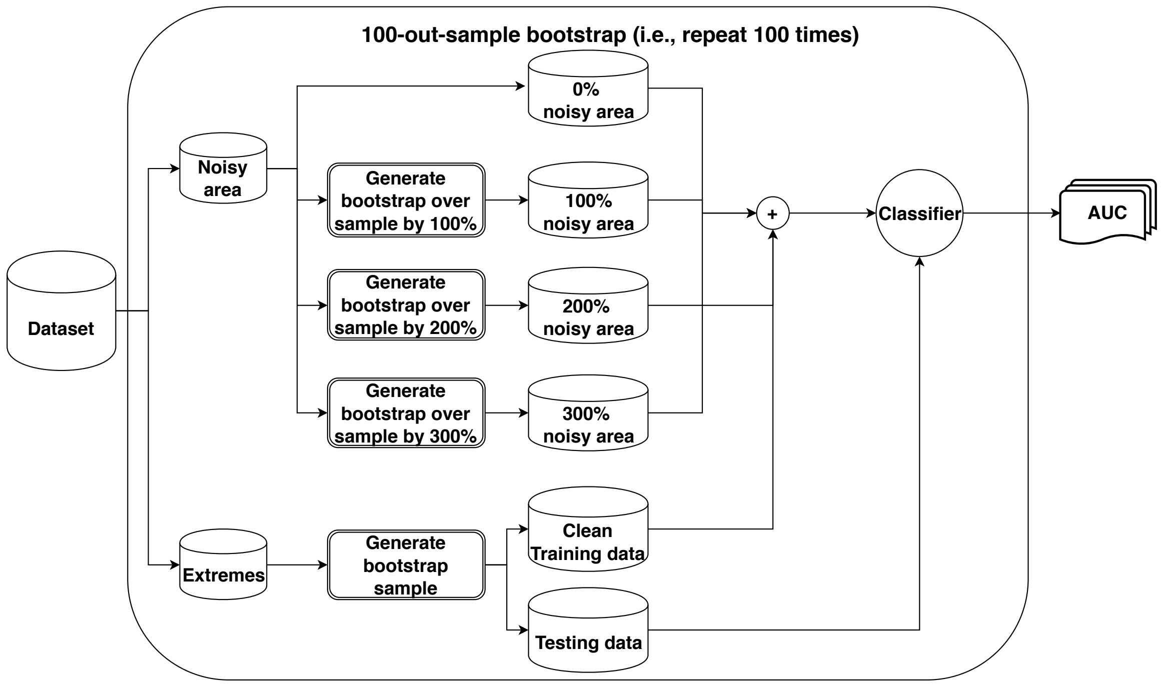

100-out-sample bootstrap (i.e., repeat 100 times)
Dataset
Noisy area
Extremes
0% noisy area
Generate bootstrap over sample by 100%
100% noisy area
Generate bootstrap over sample by 200%
200% noisy area
Generate bootstrap over sample by 300%
300% noisy area
Generate bootstrap sample
Clean Training data
Testing data
+
Classifier
AUC

Figure B.2: Overall experimental setup for Discussion 1 explaining why classifiers that are trained on the noisy area performs well on extremes.

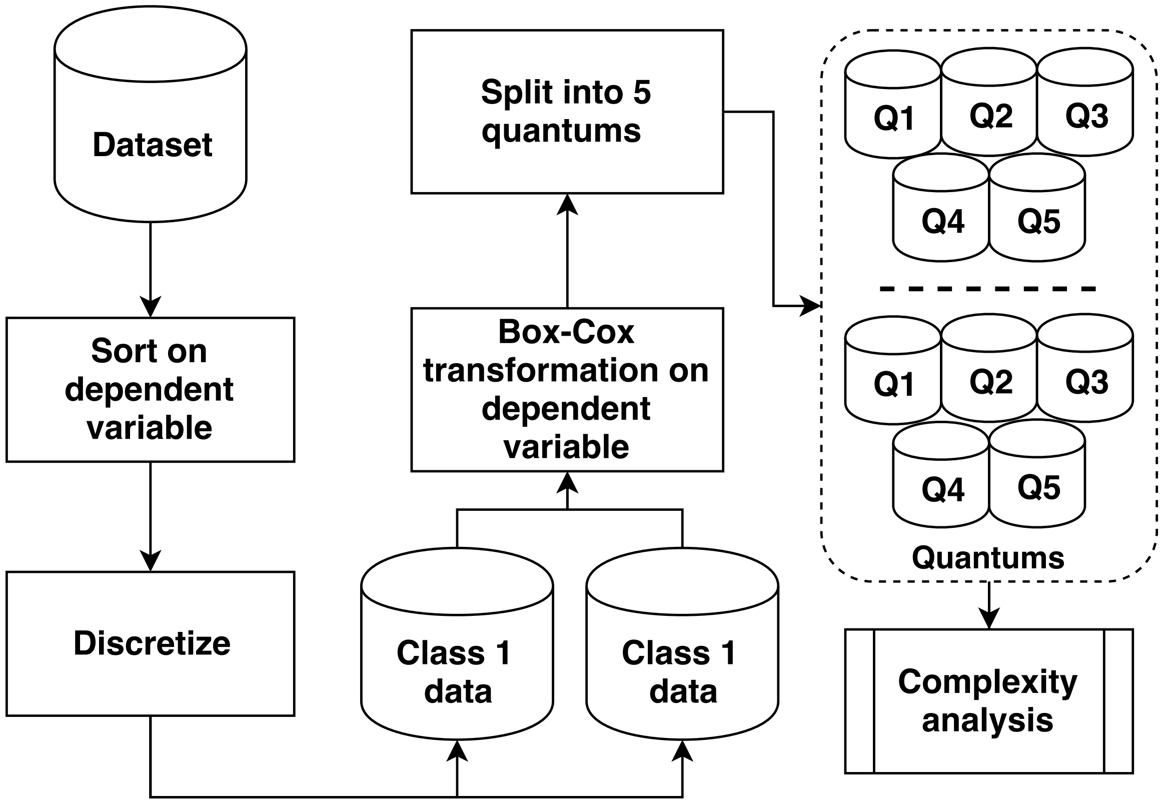


Figure B.3: Transformation of the data for the complexity analysis.

Table B.6: Data complexity metrics that are used in our analysis.

Metric	**Explanation**	**Interpretation**
Fisher's Discriminant Ratio (F1)	It is an overlap measure that calculates the amount of overlap that exists between the independent values of the data points that belong to "class 1" and "class 2". The amount of overlap between each feature of the data points that belongs to both the classes is computed and the maximum score over all the features is used as F1.	A higher F1 score means that there is less overlap and it is easy for a classifier to discriminate between data points in different classes and vice versa.
Linear Separability (L2)	It is the degree of separability between data points belonging to two classes. In our case, it is a measure of how easy it is for a classifier to discriminate between a data point belonging to "class 1" and "class2" Medin and Schwanenflugel (1981) amounts to linear separability. We measure this separability by taking into account the outliers and the error.	A high L2 score indicates that the separability is low and harder for a classifier to classify data points, whereas a low L2 score means the classifier can classify data points easily.
Mixture identifiability (N2)	It aims at capturing how identifiable is one class from another with respect to the independent feature. The ratio between the average Euclidean distances between all intra-class nearest neighbors and all inter-class nearest neighbors is used as the measure.	A higher N2 score indicates that it is hard to identify the classes correctly, whereas a lower score means that class loyalty of the data points is easy to identify.
Nonlinearity (N4)	It measures the nonlinearity of the data, we measure it using the technique that was proposed by Hoekstra and Duin Hoekstra and Duin (1996).	The higher the nonlinearity, the harder it is for a classifier to perform well.