Peaks and Valleys: A Journey Through Predictive Modelling for Software Engineering

Rebecca Moussa

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

November 3, 2023

I, Rebecca Moussa, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. The scientific contributions made in this thesis have been published in five peer-reviewed articles as follows:

Chapter 3: Sarro, F., Moussa, R., Petrozziello, A., Harman, M. (2020) Learning From Mistakes: Machine Learning Enhanced Human Expert Effort Estimates. IEEE Transactions on Software Engineering.

Chapter 4: Moussa, R., Sarro, F. and Harman, M. (2023) The Role of ML Libraries in Effort Estimation Studies. Transactions on Software Engineering (TSE), under review.

Chapter 5: Moussa, R., Sarro, F. (2022) On the use of evaluation measures for defect prediction studies. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).

Chapter 6: Moussa, R., Guizzo, G., Sarro, F. (2022) MEG: Multi-objective ensemble generation for software defect prediction. Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).

Chapter 6: Moussa, R., Guizzo, G., Sarro, F. (2023). MEG: Multi-objective Ensemble Generation for Software Defect Prediction (HOP GECCO'23). In Proceedings of the Companion Conference on Genetic and Evolutionary Computation (GECCO).

The following manuscripts are not directly related to the material in this thesis, but were produced in parallel to the research undertaken in this thesis:

1 Hort, M., Moussa, R., Sarro, F. (2023). Multi-Objective Search for Gender-Fair and Semantically Correct Word Embeddings. Applied Soft Computing.

2 Tawosi, V., Moussa, R., Sarro, F. (2022). Agile Effort Estimation: Have We Solved the Problem Yet? Insights From a Replication Study. IEEE Transactions on Software Engineering.

3 Tawosi, V., Moussa, R., Sarro, F. (2022). On the Relationship between Story Points and Development Effort in Agile Open-source Software. In Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.

5 Tawosi, V., Al-Subaihin, A., Moussa, R., Sarro, F. (2022). A versatile dataset of agile open source software projects. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR).

6 Hort, M., Moussa, R., Sarro, F. (2023). Multi-objective Search for Gender-fair and Semantically Correct Word Embeddings (HOP GECCO'23). In Proceedings of the Companion Conference on Genetic and Evolutionary Computation (GECCO).

7 Dakhama, A., Ding, Z., Mendoza, K., Menendez, H., Moussa, R., Sarro, F. (2023). EPICCap: Optimising Automated Generation of Figure Captions via SBSE+LLM, under-review.

8 Dakhama, A., Ding, Z., Mendoza, K., Menendez, H., Kelly, D., Moussa, R., Sarro, F. (2023). StableYolo: Optimizing Image Generation for Large Language Models, under-review.

9 Moussa, R., Azar, A., Sarro, F. (2020), Investigating the Use of One-Class Support Vector Machine for Software Defect Prediction (arXiv).

Impact Statement

The work presented in this thesis advances software analytics research, precisely predictive modelling for Software Effort Estimation (SEE) and Software Defect Prediction (SDP).

We propose an entirely orthogonal deployment route for estimation technology, one that advocates for techniques that facilitate human-machine cooperation to address the necessary tasks of software development effort estimation and software defect prediction. We show that human estimation errors are predictable by machine and such predictions can then be used to adjust and improve human expert effort estimates. We also show that machines can relieve engineers from the burden of manual design and experimentation in the quest for an optimal defect prediction models while still giving human experts the freedom to select optimal solutions based on the property of interest given the software application domain.

The work in this thesis also advances the state-of-the-art practices for prediction model evaluation to achieve more robust empirical evaluations in SEE and SDP research studies. Our work unveils crucial weakness in the methodology used in the empirical evaluation of prediction models for SEE and SDP, which impact the scientific conclusion stability and reproducibility of the results. Our findings draw on the community to reflect further on the selection of appropriate evaluation measures and machine learning libraries to support the scientific conclusions on the effectiveness of defect prediction and effort estimation models, respectively.

Abstract

Over the last decade automated predictive models have become very popular in a wide range of software engineering research areas, including software requirements, software design and development, testing and debugging and software maintenance.

Despite the huge rise in these automated approaches and the investigation of their use in a wide range of areas, optimal results have not yet been reached, experimental and evaluation pitfalls still exist, and few studies have sought how they can be applied in industry. As a result, both researchers and practitioners still seek ways to achieve more accurate estimates, as well as increase the adoption of automated predictive models in practice. Therefore, enhancing the design, use and evaluation of predictive models is of great need.

The work in this thesis seeks new ways to achieve machine-human cooperation to help ameliorate the performance and real-world applicability of automated prediction models. It also investigates current prediction evaluation measures and the use of different machine learning APIs as possible sources of conclusion instability (i.e., inability to consistently and uniformly present the results of empirical software engineering models), in order to increase the robustness of the empirical studies. The approaches presented herein target two of the main areas for software engineering predictive models, specifically the areas of software effort estimation and software defect prediction, and advances them with both algorithmic and methodological contributions.

Acknowledgements

I would like to express my heartfelt gratitude to all those who have contributed to the completion of this degree. This journey would not have been possible without the support, encouragement, and guidance of many individuals and loved ones.

First and foremost, I am deeply thankful to my advisor, Professor Federica Sarro, for her exceptional mentorship, unwavering support, and invaluable guidance throughout the entirety of my doctoral research. Her expertise, patience, and commitment to excellence have been instrumental in shaping this work. You set the bar high yet made sure I kept *swimming*!

I would also like to express my gratitude to my second advisor, Professor Mark Harman. His invaluable insights, experience and humility have been inspirational and have taught me a tremendous amount. "A paper rejected is just one not yet accepted" will stick with me throughout my career.

Federica and Mark, your unwavering belief in my abilities has played a crucial role in my journey and for that I will be forever grateful!

I extend my heartfelt appreciation to the members of the SOLAR group at UCL. They have been colleagues, friends and great supporters. Vali, Max and Giovani, it has been great working with you, thank you for the great collaborations!

This dissertation stands as a testament to the collective input of the countless individuals who have played a role, in the past and present, in my academic pursuits. I am deeply thankful for your support and encouragement on this transformative path of knowledge.

I also wish to acknowledge the members of the professional service at UCL. Dawn and Wendy, thank you for all your help and for bearing with me during the

past few years. Our chats have always been endearing.

To the close friends and those scattered around this globe, I am forever thankful for your existence in my life, your support means more than you can imagine. Our chats have kept my spirits high!

And last but certainly not least, I dedicate this work to my Mom, Rima, Robert, Roy and their wonderful families! You have been my pillars and a constant source of love, encouragement, motivation and support. Your belief in my abilities sustained me through the highs and lows of this journey and your sacrifices and understanding have always been immeasurable. Thank you is too little of an expression...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the rapidly evolving world of software development, data has become a core factor and a crucial aspect in the software development process. With a wealth of data found in a software's life-cycle including feature specifications, source code, test cases and user-feedback, we can draw a vast amount of information about the quality of the software and its services. And the recent emergence of public repositories, for example, allows for such data to be accessible. However, the massive amount of data makes it extremely hard, if not impossible, for practitioners to manually analyse it and draw insights from it. To this end, software engineering researchers and practitioners have investigated the use of data analytic to understand software-related data, thus leading to the rise of the now very well-known area of Software Analytics (SA). By leveraging analytical techniques, SA helps developers, project managers, and other stakeholders make informed decisions, improve software quality, optimize resource allocation, and enhance the overall user experience. It plays a vital role in modern software development and maintenance processes, allowing organizations to efficiently manage and evolve their software systems [1].

Predictive modeling is a key technology in SA which has been adopted to tackle different software engineering research areas, ranging from project management [2, 3, 4, 5, 6, 7, 8], and requirements engineering [9, 10], to software testing [11, 12, 13, 14, 15] and maintenance [16, 17, 18, 19, 18]. It exploits information from past events in order to determine patterns and predict future outcomes and trends. To this end data mining, statistics, machine learning, and artificial in-

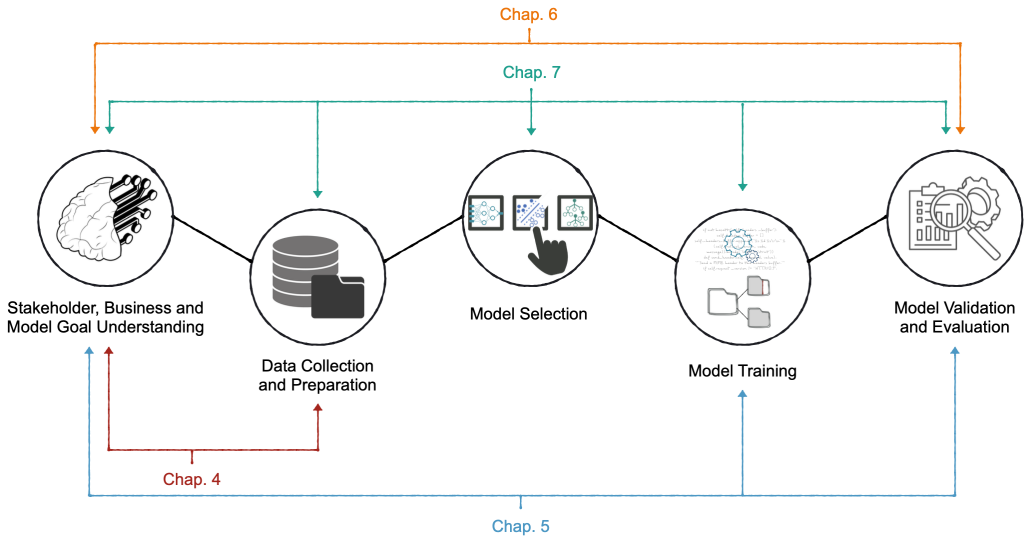


Figure 1.1: Predictive Modeling Process.

telligence techniques could be applied. Figure 1.1 presents a general predictive modeling process, which involves five main stages, namely Business Objective and Model Goal Understanding, Data Collection and Preparation, Model Selection, Model Training and Model Validation and Evaluation. The work in this thesis addresses different stages within the predictive modeling process, in an attempt to further bridge the gap and find common grounds between software engineers and automated predictive models for Software Effort Estimation (SEE) and Software Defect Prediction (SDP). Figure 1.1 highlights our work with respect to those stages.

SEE aims at predicting the amount of time required to realise a software project. It is a crucial activity for project planning and monitoring, specifically for ensuring that the product is delivered on time and within budget [20, 21]. Studies have shown that inaccurate estimates can negatively affect the outcome of software projects leading to great losses [22, 21]. SDP aims at assisting software engineers in the early identification of software defects during the development process, and ideally, before it is shipped to its customers. This is a critical task as it helps software developers and quality assurance teams prioritize their efforts, allocate resources efficiently, and improve the overall quality and reliability of the software by ad-

dressing vulnerabilities and weaknesses early in the development process [23].

While both SEE and SDP are among the earliest and most popular areas of application of automated prediction models with a copious amount of work proposed, both areas still face shortcomings when it comes to the adoption of automated techniques, with the uptake of such solutions still being relatively limited [24, 25, 26].

One reason for this is that previous work has mainly focused on the data and model selection stages, overlooking aspects which could facilitate the model adoption in practice. This is also due to the practitioners' low trust in the accuracy of automated models given that the SE predictive model literature suffer from a well-known conclusion instability problem (i.e., different sets of best predictors exist under various situations) [27, 28, 29]. For example, SEE still relies on human expert judgement to produce an estimate of the effort required to realise a software project [30]. The use of predictive models as a black-box has also left practitioners sceptical about their usage. However, previous studies have shown that there tends to be inaccuracies in human expert estimations, which can negatively affect the outcome of software projects leading to great losses [31, 30, 32, 33, 22, 21].

The ultimate aim of this thesis is to investigate and help build solutions to allow software engineers to confidently utilise automated predictive modeling by facilitating a closer cooperation between human and machine. To this end, we address two main research objectives: (1) investigating new technologies for human-machine cooperation for SEE and SDP; (2) advancing the state-of-the-art practices for prediction model evaluation to achieve more robust empirical evaluations in SEE and SDP research studies.

With respect to the first objective, this thesis presents

- a novel approach for SEE, dubbed **L**EARNING **F**ROM **M**ISTAKES (LFM), that supports human experts in estimating development effort (see Chapter 4) by providing automatic feedback on the expert's own judgements, rather than seeking to second-guess them.

- a novel approach for SDP, dubbed **M**ULTI-OBJECTIVE **E**NSEMBLE **G**ENERATION (MEG) that allows software engineers to encode model goals

and constraints based on the business objectives set by the decision-maker, and, at the same time, allows them to automatically find, among a large search space of possible solutions, a prediction model optimised with respect to these properties of interest (see Chapter 7).

With respect to the second objective the work in this thesis contributes to important methodological aspects by unveiling factors that can lead to conclusion instability in SEE and SDP, as follows:

- We investigate for the first time in the SEE literature, the role of ML libraries within the prediction process, which we found to be yet another factor that plays a role in SEE conclusion instability. Moreover, our study unveils the existence of a replicability problem resulting from the lack of reporting of ML libraries in the heart of the SEE literature which can no longer be neglected (see Chapter 5).

- We investigate a crucial factor which can lead to conclusion instability in DP studies. Specifically, we study the impact of the choice of evaluation measures. While previous work has raised alarms about the way researchers have employed evaluation measures to assess the effectiveness of the prediction models proposed for defect prediction [23, 34, 35, 36, 37, 38], in this thesis (see Chapter 6) we unveil that the importance of adopting suitable measures is still overlooked and that this has a significant impact on the results of a study and on the scientific conclusions made (see Chapter 6).

In the following subsection we summarise, for each of the contributions, the problem, the novelty of the proposed solution, and the main findings.

1.1 Contributions

Chapter 4. While a wide range of automated predictive models have been proposed for software effort estimations, the estimates, in practice, are still mainly based on human expert judgement. This is due to the fact that these models usually provide

point effort estimates without additional explanations or guidance to the practitioners, resulting in little confidence in using predictive modeling. The work presented in Chapter 4 proposes a different approach to using predictive modeling: We use predictive modeling as an aiding tool which attempts to learn the errors committed by human experts when estimating software effort and automatically adjusting the expert's estimates accordingly. **L**EARNING **F**ROM **M**ISTAKES (LFM), is a novel approach to predictive modeling where the core idea is that the error resulting from expert judgement when estimating the effort of a software project is predictable and can be used to improve the expert's final estimate. We demonstrate the effectiveness of LFM with an empirical study involving human expert effort estimates (and related errors) from 402 industrial software projects developed by different software companies. The results show that human misestimates are indeed predictable and that exploiting these predictions yield significantly better estimates than those provided by random guessing and traditional machine learners. Our results also highlight that providing this adjustment to the human expert estimate enhances the final predictions. This empirical evidence opens the door to the development of techniques that use the power of machine learning, coupled with the observation that human errors are predictable, to support engineers in estimation tasks rather than replacing them with machine-provided estimates.

Chapter 5. In the past two decades, several ML libraries have become freely available for the use of such models. Many studies have used these libraries to carry out empirical investigations on software development effort estimation. However, the differences stemming from using one library over another have been overlooked, implicitly assuming that using any of these libraries to build a certain machine learner would provide the user with the same or at least very similar results. The work presented in Chapter 5 precisely aims at investigating this overlooked aspect. We explore, for the first time, the accuracy achieved by the effort estimations produced by a same prediction model provided by different Machine Learning (ML) libraries. To this end we empirically investigate four deterministic machine learners, widely used in SEE, as provided by three very popular ML open-source ML libraries

written in different languages (namely, SCIKIT-LEARN, CARET and WEKA). Our findings highlight that the ML library is an important design choice for SEE studies, which can lead to large differences in performance. However, such a difference is under-documented and it leads to a pivotal replicability problem at the heart of the scientific literature. This suggests that end-users need to take such differences into account when choosing an ML library, and developers need to improve the documentation of their ML libraries to enable the end user to make a more informed choice.

Chapter 6. Another factor that can influence the results of a study, is the choice of the measure used to evaluate the results. While in software effort estimation the use of problematic measures has been addressed and current literature mostly follow best practices, the use of different evaluation measures is still a concern for software defect prediction, despite the various warnings which have been raised [23, 39]. The work presented in Chapter 6 reviews a total of 111 studies published between 2010 and 2020 to understand the way the community has handled this critical threat following previous warnings. Moreover, it presents the first empirical investigation exploring a comprehensive set of measures based on statistical significance and effect size analysis. The results shows that the level of disagreement among the evaluation measure is very high and can have a dramatic impact on the conclusion validity resulting from the assessment of current binary classifiers in defect prediction. Therefore, future studies need to carefully select and justify the evaluation measures used.

Chapter 7. Recent studies have found that ensemble prediction models (i.e., aggregation of multiple base classifiers) can achieve more accurate results than those that would have been obtained by relying on a single classifier when used for the early detection of software defects. However, designing an ensemble requires a non-trivial amount of effort and expertise with respect to the choice of the set of base classifiers, their hyper-parameter tuning, and the choice of the strategy used to aggregate the predictions. An inappropriate choice of any of these aspects can lead to over- or under-fitting, thereby heavily worsening the performance of the

prediction model. Examining all possible combinations is not computationally affordable, as the search space is too large, and there is a strong interaction among these aspects, which cannot be optimized separately. The work presented in Chapter 7 leverages the power of multi-objective evolutionary search to automatically generate defect prediction ensembles. We dubbed our proposal as **M**ULTI-OBJECTIVE **E**NSEMBLE **G**ENERATION (MEG). The main purpose of MEG is to automatically generate ensemble classifiers by choosing a set of base classifiers, tuning them, and then selecting an aggregation strategy to produce the final ensemble. MEG is not only novel with respect to the more general area of evolutionary generation of ensembles, but it also advances the state-of-the-art in the use of ensemble in defect prediction. We assess the effectiveness of MEG by empirically benchmarking it with respect to the most related proposals we found in the literature on defect prediction ensembles and on multi-objective evolutionary ensembles. Our results show that MEG is able to advance the state-of-the-art by producing similar or more accurate predictions in 73% of the cases, with favourable large effect sizes in 80% of them. MEG achieves such a high performance by automatically selecting the classifiers, tuning them, and identifying the most suitable aggregation technique, thus also relieving the engineers from the challenge and burden of carrying out each of these tasks manually.

1.2 Thesis Organisation

The remaining of this thesis is organiseed as follows. Chapters 2 and 3 discuss the background and previous work related to SSE and SDP, respectively. Chapters 4 to 7 present each of the contributions in greater detail. Chapter 8 concludes the thesis by summarising the main achievements and discussing open challenges in the field.

Chapter 2

Background

In this chapter, we provide an overview of the features, datasets and validation processes found in the literature in both, the effort estimation and defect prediction areas.

2.1 Software Development Effort Estimation

Software Development Effort Estimation is a crucial activity for project planning and monitoring, specifically for ensuring that the product is delivered on time and within budget. Studies have shown that inaccuracies in the estimations can lead to great losses [40]. In the literature, there are two types of erroneous predictions: under- and over-estimation. Overestimation occurs when the predicted effort exceeds the actual effort, whereas underestimation occurs when the the effort predicted is lower than the actual effort needed to complete a project. Both types of errors have detrimental effects on a company's reputation, performance and productivity [20, 41, 42]. For example, overestimated projects can result in the team stretching their work hours to cover the unnecessary gap leading to a reduced productivity rate [43] and missed opportunities of promising ideas due to poor resource allocation. On the other hand underestimating the effort a software project requires to be developed leads to schedule and budget overruns, possibly resulting in project cancellation which effects the reputation and performance of the company [41, 42]. As a result, practitioners have highlighted the importance and need of proposing methods that support better estimates and improve estimation accuracy. To this end,

different types of data-driven approaches have been explored in the literature. These approaches take certain cost-drivers as input in order to predict an informative estimate of the effort required to develop or complete a project.

2.1.1 Data and Cost Drivers

In order to estimate the effort required to develop a software project, researchers have used several cost drivers related to the software which are known at the time of estimation and hence at an early stage of a project's life cycle. One of the most common factors considered is the estimated size of the software. Researchers have identified two main approaches to measure the size of a software system. One is solution-oriented; expressing the software in terms of the size of the code written by counting the source lines of code, while the other is problem-oriented; measuring the size of the software based on its requirements. However, the former showed to be problematic and received a lot of criticism for reasons regarding the lack of accepted standardised guidelines [44, 45], regarding the fact that it is highly dependent on the programming language [46] and that it would be difficult to estimate at an early stage in the project life cycle [47]. On the other hand, Functional Size Measurement (FSM) were introduced and standardised as a mean to measure software systems based on their functional user requirements which can be derived and used at the project planning stage. Another advantage of FSM is that it is not dependent on the technology and programming language used to develop the software rendering it more adaptable and applicable to a wide range of software types.

Several procedures were proposed to achieve this. Two very well known and widely used FSM methods are the IFPUG Function Point Analysis (FPA) and the Common Software Measurement International Consortium (COSMIC). FPA was the 1st FSM method proposed [48] to evaluate the functionality of a system by quantifying the functions contained within the software in terms that can be expressed as requirements to the users. It defines five function types that are based on, internal and external, data and transaction functions and that can be extracted from early life cycle documents. These describe the internal and external resources that affect the system and the processes that are exchanged between the user, any external applica-

tion and the system being measured. FPA was mainly developed to measure Management Information Systems and despite its popularity, it has driven researchers to propose improvements in order to broaden the types of systems for which it can be used. To this end, several other 1st generation variants were proposed with the aim of improving FSM and its applicability for various types of software [49]. COSMIC was established for the same reasons and was considered a 2nd generation method mainly due to its compliance to the standard ISO/IEC14143/1 [50] and its distinguishing characteristics, including its applicability to business, real-time, and infrastructure software (or their hybrids) [51], and possibility to extend its usage to other kinds of software such as Web and Mobile applications[52, 53, 54, 55, 56, 57]. Unlike FPA, instead of describing internal and external functions, COSMIC focuses on four main data movements (read, write, entry, exit) of each functional process among the users, external applications and the system being measured. Other less popular methods of measuring software size include use case points and object points. These methods have not been internationally standardised and as a result are not widely supported.

Different types of software management and development processes call for different units of measurements. For example, in agile development, one of the most widely used methods includes the count of story points. Story points describe the overall effort and reflect the difficulty required to implement the requirements of a system by assigning point values to its user stories.

There are also measures that do not relate to the size and the requirements of the system. Instead, they focus more on factors such as the manager and team's experience, developer training, clarity of the system's manual, etc. Unfortunately, while these measurements can provide information that might improve the predictive model's performance and, as a result, provide more accurate estimates, this data is considered sensitive and it is usually only available within the company (i.e., internally). While large companies have the resources to collect this data, they refrain from sharing it publicly. On the other hand, small sized companies which are the ones who will likely to benefit the most from the collection of this data, usually

do not have the required resources and cannot afford to do so, making it extremely difficult for researchers to investigate and adopt them within their proposed models given that they cannot be accessed/obtained publicly.

Several datasets, composed of projects mainly described in terms of functional size cost drivers, have been made publicly available over the years. For example, PROMISE [58] is a repository which currently host a number of datasets, such as such as China, Desharnais, Finnish, Maxwell, Miyazaki, etc., which have been widely used by practitioners as well as researchers for software project effort estimation studies. These datasets represent a varied sample of industrial software projects collected from a single company or several software companies. The datasets cover a diversity of application domains and project characteristics. Another widely known repository is the International Software Benchmarking Standards Group (ISBSG) repository [59], which includes data that describe software projects submitted by leading IT and metrics companies from around the world. While the ISBSG is a global and independent source of software metrics data for IT projects, access to its resources needs to be purchased for either commercial or academic purposes.

The datasets used in each of the empirical studies presented in this thesis, will be described in detail in their own chapter along with the motivation for their choice.

2.1.2 Validation Processes

When comparing or proposing new prediction methods, one very important factor is the way in which they are assessed. However, in an area, like SEE, where data is scarce and the number of projects in a dataset is usually very small, the validation process can be very challenging. Studies are often left with a handful of projects for training and evaluating their proposed models [60, 61, 62, 63].

To this end, different validation processes have been applied and evaluated in order to test a proposed model's performance when estimating software effort on unseen data. One of the most widely used validation techniques is Cross-Validation (CV). The latter is described as a resampling procedure which has a single parameter, k, referring to the number of groups that a given data sample is to be split

into. The model is then trained on k-1 folds, and tested on the remaining fold. This process is done for each fold and the entire procedure is repeated multiple times to minimise bias and account for any randomness that might be present in the model's algorithm/behaviour. For example, 5-fold CV indicates that the data is split into five folds of approximately equal size. The model is then trained on four folds and tested on the remaining one, repeating the process for a total of five times.

Multiple values for k have been previously explored in the literature. Among the most common ones are the 3-fold CV [64, 65, 62], 10-fold CV [66, 62, 67] and the widely used Leave One Out (LOO) CV where the k is equal to one [68, 69, 70, 71, 63, 72]. In this case, given a dataset of size *n*, *n-1* instances are used to train the model and the remaining single instance is used as the target. This results in n pairs of training and testings sets. In their work, Kocaguneli and Menzies [73] investigate the use of the aforementioned evaluation procedures in terms of stability and execution time. They highlight that future work should consider assessments via LOO given the technique's deterministic nature which eliminates conclusion instability. The authors also report that LOO is not necessarily slower than 3-fold or 10-fold CV and that it does not generate different biases and variances. However in a more recent study, Sigweni et al. [74] propose a more realistic time-based approach, when chronological data is available, based on a grow-one-at-a-time (GOAT) validation principle. This better reflects a real life setting given that projects can only be used for prediction after they have been completed.

2.2 Software Defect Prediction

Software bugs can be very costly to both, the users and the company providing the software. Among several software failures experienced by international airlines, was that of British Airways in 2019. This glitch caused chaos as more than a 100 flights were cancelled and more than 300 were delayed [75]. As the world becomes more dependent on software, tasks such as detecting defects become more essential and critical. In a severe case scenario, their cost can be life-threatening as in the case of the IT glitch of the UK National Health Service which put 10,000 patients at risk

of being given the wrong medication in 2018 [76]. The earlier, in the process, a bug is found and fixed, the less it costs. According to Capers [77], addressing bugs post-release costs $16,000, but a bug found at the design phase costs $25. In his study, Boehm [78] also revealed that, at least for small non-critical systems, the ratio between finding a bug in the development stage as opposed to finding it while the system is already in production is 5:1. Quality assurance budget is often spent on fixing post-release bugs which could have been fixed earlier on for much less. As a result, software defect prediction aims to reduce testing cost and effort by flagging possible problematic software components, before their deployment. This allows engineers to better focus their testing efforts, and hence allocate resources effectively.

In order to achieve this, researchers have explored different automated techniques that can generate a prediction system from known training examples. These techniques usually use a wide range of metrics as inputs as described in the following section.

2.2.1 Data and Metrics

One crucial component in the process of defect prediction is the identification of measurable properties about the software modules, known as metrics. Similar to the area of Software Effort Estimation, the earliest metric proposed was also based on the size of the system under study [79] by capturing information such as Lines of Code (LOC). However researchers also found the complexity of a software to be a good indicator to whether a module could be defective or not. To this end, McCabe's cyclomatic complexity metric [80] and Halstead's metrics [81] were proposed. The former is a measure of the maximum number of linearly independent circuits in a program control graph, while the latter provides information about the number of operators, operands and the total use of each. However, the complexity of a software could not entirely be described by these metrics alone. As a result, Chidamber and Kemerer [82] proposed, what is now, one of the most popular metric suites describing software complexity by capturing object-oriented concepts such as cohesion and coupling.

Since then, a wide range of metrics treated as features have been used for defect prediction to improve software quality. They can be divided into three classes [83]: (i) Object-Oriented Code Measures [84, 85, 86, 87, 88, 89, 90]; (ii) Delta Measures [91, 92, 93] and (iii) Process Measures [94, 95, 96].

The aforementioned metrics have been widely explored and used in the literature. In fact, in a systematic literature review, Radjenovic et al. [97] show that 49% of the studies have used object-oriented code measures, while 24% of them referred to process measures for their work.

Object-oriented code measures, also known as structural measures, are based on the Chidamber and Kemerer metrics suite [82], derived from the source code and are based on aspects of the code such as the size and complexity. Some of these measures include the lines of code, the number of methods in the class, lack of cohesion of methods and the cyclomatic complexity in the class, etc. Delta measures refer to the change that has occurred on the files between two successive releases. They include information about the way the structure of code has changed over versions of a code unit. For example, the number of lines of code deleted and added to a specific class. On the other hand process measures focus on the way the code had been developed. They include features that describe a surrogate measure of the experience of each developer and the number of developers that have made changes to a file. More recently measures drawn from software testing [14] and networking [98] have been found to carry out useful information about defect-prone software components.

There are multiple repositories providing datasets, some of which are publicly available, based on the metrics described above. According to a study done in 2013, Shepperd et al. [99] identified 24 different dataset families. Among them, the public NASA software defect repository [58] was found to be the most popular. The NASA Metrics Data Program datasets describe a variety of systems, mainly written in C and C++, ranging from spacecraft instruments and flight software for earth orbiting satellite to storage management for receiving and processing ground data and real-time predictive ground systems. While the adoption of the NASA datasets

became very popular among researchers due to their original format, which made it very easy to apply in defect prediction studies, these datasets were not released with a lot of contextual information, making it challenging to verify the consistency of the metrics extracted. This led previous work to question and investigate the quality of the NASA datasets. Specifically, Gray et al. [100] identified several erroneous factors such as the presence of repeated or inconsistent instances, constant and repeated attributes, missing values and invalid data values. Shepperd et al. [101] and later Petric et al. [102] further presented a set of cleaning steps for removing erroneous data from the NASA Metrics Data Program datasets. Other widely known repositories, such as the Tera-PROMISE [103], host Apache datasets describing Java libraries and tools of multiple version releases.

2.2.2 Validation Processes

The work in defect prediction has been mainly evaluated in three different scenarios: (i) Within project [104, 105, 106, 107]; (ii) Cross-version (CVDP) [108, 109, 110, 111] and (iii) Cross-project (CPDP) defect prediction [112, 113, 114, 115].

In the within-project scenario, practitioners usually aim to predict defect within a project by training the techniques on the modules within that project. K-fold Cross-Validation, is usually applied to validate the performance of the predictive models being used. This scenario has proven to work well as long as there is sufficient data on which the model can be trained [90]. In practice, however, this is not always the case as small companies might have little data to work with, or even no past data for the first release of a product. The other two scenarios, namely cross-version and cross-project defect prediction are alternative solutions for small sized releases of a given project or for a company not having any previous releases, respectively.

The former scenario is applicable to companies who have released multiple versions of their system. In this scenario, predictive models are usually trained on one or multiple previously released versions in order to predict defects in the latest version.

While this solves the problem for companies who release small sized versions

or updates of their systems, it does not help those who would like to predict defects for their first release. To this end, researchers and practitioners have proposed the use of cross-project defect prediction, where the model is trained on data collected from a different project than the one to be predicted for. Due to the heterogeneous nature of the training and testing data, this task has been shown to be more difficult, on which predictive models have achieved results relatively lower than the other two scenarios [113].

Chapter 3

Literature Review

In this chapter we review the main work that has tackled the problems of effort estimation and defect prediction relevant to the work presented in this thesis.

3.1 Effort Estimation

Previous research has been carried out to support engineers in estimating software development effort, focusing on the following aspects:

- Improving the accuracy of software effort estimates by searching for a single best approach, i.e. proposing and comparing a large number of techniques such as regression and analogy-based [116, 117], machine learning [118], ensemble [119], search-based [120].
- Experimenting with, and comparing, different size measures as cost drivers (e.g. [121, 122, 123, 124, 125]).
- Experimenting with, and comparing, within- vs. -cross company data (e.g [126, 127, 128, 129]).
- Investigating estimate uncertainty (e.g. [31, 130, 131, 132]) and prediction intervals (e.g. [133, 134, 135, 136, 137, 138, 139]).
- Studying human bias in effort estimates (e.g. [140, 141, 142, 143, 144]).

A comprehensive review on the use of expert judgement, formal models and their combination can be found in the work conducted by Jørgensen et al. [145].

In the following, we focus on those studies that have investigated human bias in predicting task duration, and specifically in predicting software development effort, which are relevant to the work we present in Chapters 4 and 5.

3.1.1 The Role of Humans and Bias in Software Effort Estimations

Predicting task duration has been the focus of a lot of research in different fields. Despite the different nature of tasks under examination, previous studies almost universally show that human predictions tend to be biased. For example, studies in psychology and human cognition show that humans might be subject to the phenomenon of the central tendency of judgement, which describes the tendency for humans to over-estimate small tasks and under-estimate large ones [146], as well as to the phenomenon of planning fallacy, which is the human tendency to underestimate future task duration despite knowing that previous similar tasks could not be completed on time [147]. This kind of bias has been later attributed to misremembering previous task duration (i.e. memory-bias) and using such a duration as a basis for future predictions. For example, Roy and Christenfeld [148] studied whether a systematic memory-bias has an effect, or could explain, a similar systematic bias in prediction, and showed that people tend to underestimate the duration of future events because they based their estimation on the perceived duration rather than actual duration of similar events that had occurred in the past. Subsequent studies have shown that this prediction bias can be reduced when feedback about previous task duration is provided, thus refreshing and ultimately correcting the memory (see e.g. [149]).

The principle of human bias in predictions has also been studied, from different perspectives, for the task of software effort estimation, as detailed below.

Surveys on estimation practice in the software industry found that human effort estimates are over-optimistic [**?**, 31] and there is a strong over-confidence in their accuracy [32]. A recent survey on agile practice also revealed that half of the respondents believe that their effort estimates on average are under/over estimated by an error of 25% or more [33].

Other studies have looked into possible reasons for bias mainly basing their investigation on statistical analysis of project characteristics and questionnaires posed to project managers [140, 142, 143, 150, 144]. Lederer and Prasad [140] found that the main cause of misestimates was from the management control side. Specifically, the lack of tasks like consideration in performance reviews as to whether estimates were met, project control comparing estimates and actual performance, and careful examination of the estimate by the management of the information system department resulted in inaccurate estimates. Whereas Gray et al. [142] used contingency table analysis, logistic regression and log-linear modeling to prove that the expert-derived effort prediction used to develop a collection of modules from a large health-care system showed systematic biases involving the size and type of the modules understudy. Jørgensen [143] investigated the accuracy and bias variation of effort estimates through the use of a regression analysis-based model. The study analyses 49 software tasks from a single organisation using collected information about variables that were believed to have an effect on accuracy or the bias of the estimates. The results highlight that several factors influence the increase of error in estimates, such as the estimates being provided by a software developer rather than a project leader or the customer prioritizing time-to-delivery as opposed to quality or cost. Jørgensen and Moløkken-Østvold [150] also studied differences in types of reasons for estimation error depending on the role of the estimators, data collection approach, and analysis technique with results showing that all three types of reasons play a major role in estimate inaccuracies. In a more recent study, Jørgensen and Grimstad [144] examined the relationship between biases resulting from effort estimates produced by software developers, and developer cultural dimensions such as the way one sees oneself, the thinking style, nationality, experience, skill, education, sex, and organizational role. Results showed that estimation bias was present along most of the studied dimensions and that there was a strong correlation between estimation bias and the developer level of interdependence.

While the aforementioned work has focused on reasons for human bias to support experts in making more accurate and realistic estimates, our study uses ma-

chine learning predictions of human bias to automatically adjust and enhance the expert's final estimate of the overall effort which, to the best of our knowledge, has not been explored yet. This concept of using error to adjust future estimates has been used by Kultur et al. [151], however it has been applied to adjust for errors resulting from machine estimates of effort. Their work proposes an ensemble of neural networks with associative memory (ENNA) and takes the machine learner's estimation bias into account by using KNN to retrieve past projects that are similar to the new one. The estimated bias of the nearest neighbors ensemble is calculated as the average of the differences between the actual and the estimated values for those projects. This bias is then added to the estimated effort of the new project. Results show that ENNA provides estimates that are significantly better than neural networks and regression trees. On the other hand, we propose the use of automated models in order to predict the errors made by human experts (rather than predicting the effort) when estimating effort and exploiting this information to adjust their final estimates. Rather than seeking solely to compete with (or even replace) human experts, our approach aims to use machine learners to learn, from both traditional past projects cost drivers and from past expert judgements, essentially building into the predictive model the ability to learn from their past estimation errors. Although a part of our work also proposes to adjust the final estimates by taking into account or adding an error/bias (Phase 3), the main idea behind our proposed approach is to automatically learn from and predict human expert error, and to show that these predictions can be exploited to help experts improve their final estimates.

3.1.2 The Role of Machine Learning Libraries in Software Effort Estimations

The increasing growth and success of ML has attracted researchers and practitioners of various skill levels to use popular and publicly available ML libraries to carry out Software Engineering (SE) tasks, including critical prediction tasks such as software development effort estimation [116, 5, 152], defect prediction [23], bug fixing time estimation [153]. The number of such ML libraries has been rapidly growing and their development is fast paced, leaving the engineer with a wide range of tools to

choose from. Previous work has provided reviews of some of these popular libraries [154, 155, 156].

With the rise in the use of these libraries, the question that emerges is: *Would it matter if one uses one library over another?* If the results reported by these tools are largely consistent, then one can be confident in using any of them. While previous studies in effort estimation have shown that differences in training data, learning techniques, hyper-parameter tuning and evaluation procedures can lead to variance in the prediction result causing conclusion instability [157, 27, 28, 158, 159], no study has investigated the impact that the use of different ML libraries to build the prediction model can have on the variance in the estimates across different libraries, and unfortunately, our work, found in Chapter 5, proves that similarly to the other design aspects (i.e., validation approach, machine learning models, etc.), the choice of ML libraries does have an important impact on a study's conclusion stability.

Moreover, recent studies have highlighted that developers fail to grasp how to make ML, and more generally AI, libraries work properly, and very often seek further documentation or support from their peers on forums such as Stack Overflow [160, 161]. Therefore, researchers have started focusing their attention on understanding potential problems faced by engineers when using these libraries [162, 163, 164] and to ultimately improve, both the software documentation [165, 161] and testing practices [166, 167] for ML/AI software. Moreover, some previous work has looked into the variance due to the stochastic nature of most of the approaches used for deep learning [168, 169], sentiment analysis [170, 171, 172], or classification models [173, 174, 175, 176]. Differently from these studies, which mainly investigated the variance of non-deterministic approaches [169, 177] in other application domains, the work we present in Chapter 5 intentionally designs an empirical study in a way that reduces, as much as possible, all those factors that would introduce stochasticity in the machine learner output. This allows us to analyse the variances in performance that are primarily due to the use of different ML libraries. To the best of our knowledge, no previous work investigates differences arising from the use of ML libraries for SEE.

3.2 Defect Prediction

A great deal of research has been conducted to improve the quality of software systems. One important component of that is the prediction of software defects and the module in which these defects might be located. As a result, researchers have mainly focused on the following aspects:

- Improving the accuracy of defect predictions by searching for a single best approach, i.e. proposing and comparing a number of machine learning techniques [178, 179, 105], statistical methods [180], search-based approaches [181, 182].

- Experimenting with, and comparing, the use of different software defect metrics [183, 184, 91].

- Investigating the class-imbalance problem [107, 185, 186].

- Experimenting with, and comparing, within-project, cross-version and cross-project defect prediction [104, 108, 113, 187].

In the following, we focus on previous studies that are considered relevant to the work in this thesis. Specifically, we first look into the literature of work aiming at improving the accuracy of defect predictions by proposing and comparing new approaches such as machine learning and search-based techniques, and those aiming at improving defect prediction research by strengthening the methodology used in empirical studies.

3.2.1 The Role of Search-Based and Ensemble Models in Defect Prediction

Search-Based Software Engineering (SBSE) has been shown to be a powerful tool to address Software Engineering prediction tasks [188, 189, 190], such as software effort estimation, change prediction, defect prediction and maintainability prediction [191].

SBSE is an approach to software engineering in which search based optimisation algorithms are used to identify optimal or near optimal solutions and to get

insight into SE problems characterised by a large space of possible solutions, which cannot be exhaustively explored in a reasonable time.

In the context of defect prediction, previous studies have investigated the use of both single- and multi-objective search-based approaches [192] to either build [182] or fine-tune [193, 194, 195] learning models. Ensemble learning models have also been explored to build defect prediction models [196, 197, 198, 199, 200, 201].

Ensemble Learning is a technique used for building more robust machine learning models achieving better performance by the means of combining multiple classifiers trained to solve the same problem.

However, no previous defect prediction study has explored the product of combining both types of approaches together, specifically the use of search-based approaches to guide the construction of ensemble models, which have instead been exploited to solve general-purpose classification tasks. Moreover, to the best of our knowledge, the work by Petrić et al. [202] is the only to contemplate diversity, together with accuracy, in order to build more robust ensembles.

Previous work which provides a comprehensive view on machine learning models for defect prediction is that conducted by Hall et al. [23]; for search-based approaches in defect prediction is the one by Malhotra et al., [191]; for ensemble models in defect prediction are those published by Afzal et al. [203] and Matloob et al. [196]; and for general-purpose evolutionary ensembles is the one conducted by Ren et al. [204]. The rest of this section focuses on the work most related to the one proposed in this thesis.

While the use of Multi-Objective Evolutionary Algorithms (MOEAs) [205] in the context of ensemble generation by considering both accuracy and diversity as target objectives for different domains has been explored previously [206, 207, 208, 209], none had been previously conducted in the context of defect prediction. In addition, all this work has relied on the Pareto approach to generate the ensembles (more details are provided in Chapter 7). Among these, the closest work to ours is that by Chandra and Yao [207], who used multi-objective optimisation based on NSGA-II in their algorithm called DIVACE, to evolve and train the set of weights

of 3-layer neural networks by simultaneously optimising accuracy and diversity.

Studies in the AutoML literature have also investigated EAs for such optimisations, such as Autostacker [210]. However, work in this area generally optimises the parameters of the classifier composing the multi-layered ensemble, but does not encompass their selection or the aggregation strategy choice. Other work used MOEAs to make an optimal selection of pre-defined (or pre-trained) classifiers with feature selection for other classification tasks [211, 212, 213, 214, 210]. None of them leverage diversity as an objective.

In this thesis (see Chapter 7), we instead propose an alternative and novel way, named MEG, to automatically generate ensemble models, based on *whole-ensemble generation* which has been inspired by the idea proposed in the work of Petrić et al. [202] where both diversity and accuracy are optimised.

3.2.2 The Role of Evaluation Measures in Defect Prediction

Software defect prediction research has adopted various evaluation measures to assess the performance of prediction models. The use of appropriate evaluation measures is crucial given that it guides practitioners and researchers to understand whether a given prediction model is fit for their purpose [23].

The first comprehensive study to present a survey of commonly used model evaluation measures as well as a comparison of the results obtained by the use of different measures on NASA datasets was carried out in 2008 by Jiang et al. [39].

In their investigation of methods to build and evaluate defect prediction models, previous work proposed by Arisholm et al. [83] also acknowledges the impact of using different evaluation measures and compares several alternative ways of assessing the performance of the models under study. The authors conclude that what is considered the best model is highly dependent on the criteria used to evaluate and compare predictive models.

Subsequently, Jingxiu and Shepperd [215] performed a meta-analysis of eight papers on defect prediction in order to understand the differences resulting from the use of F-measure as opposed to MCC. They illustrated potential biases by using confusion matrices that portray different scenarios and found that the use of F-

measure is problematic. However they did not quantify the differences resulting from the comparison, in fact as they state, their study "captures a change in direction of the effect, it does not, however, capture the magnitude of the effect"[215]). In Chapter 6, we specifically study the magnitude of the effect of using six different evaluation measures (including F-measure and MCC) based on both statistical and effect size analyses. This analysis is crucial to provide solid empirical evidence on whether the use of a measure over another significantly changes the way model performance is interpreted with respect to the business needs.

Other studies highlighting that the problem exists, is that of Xuan et al. [216] and Hall et al. [23]. While Xuan et al. investigate the performance of defect prediction approaches on a large number of evaluation metrics in order to find the best performing technique, the comprehensive survey by Hall et al. [23] reviews defect prediction studies published up to 2010. Despite the fact that the primary goal of the survey was not to investigate bias resulting from the use of different evaluation measures, they do observe that this is an issue. As this study was published in 2012, we expected subsequent research to adopt/follow these guidelines, however the work we present in Chapter 6 shows that this has not been the case.

Chapter 4

Learning From Mistakes: Machine Learning Enhanced Human Expert Effort Estimates

In this chapter, we introduce a novel approach to predictive modeling for software engineering, named Learning From Mistakes (LFM). The core idea underlying our proposal is to automatically learn from past estimation errors made by human experts, in order to predict the characteristics of their future misestimates, therefore resulting in improved future estimates. We show the feasibility of LFM by investigating whether it is possible to predict the type, severity and magnitude of errors made by human experts when estimating the development effort of software projects, and whether it is possible to use these predictions to enhance future estimations. To this end we conduct an empirical study investigating 402 maintenance and new development industrial software projects. The results of our study reveal that the type, severity and magnitude of errors are all, indeed, predictable. Moreover, we find that by exploiting these predictions, we can obtain significantly better estimates than those provided by random guessing, human experts and traditional machine learners in 31 out of the 36 cases considered (86%), with large and very large effect sizes in the majority of these cases (81%).

4.1 Introduction

Software development effort estimation is a crucial activity for project planning and monitoring, specifically for ensuring that the product is delivered on time and within budget [20, 21].

Studies have shown that engineers make inaccurate effort estimations [30, 31, 32, 33], which can negatively affect the outcome of software projects leading to great losses [22, 21].

To support engineers in obtaining more accurate estimates, researchers and practitioners have attempted to devise various automated methods over the last three decades [20, 120]. However, despite the rise of automated predictive modeling, human expert judgement is still the most commonly applied strategy for software effort estimation [217, 33] and their expertise has not been fully exploited in combination with automated approaches [145, 130].

This observation motivates us to depart from received wisdom and current research practice in the predictive modeling community. In this work, we shift the focus from creating automated models able to predict software effort to creating automated models able to predict the errors made by human experts when estimating effort and using this to adjust their estimates. Rather than seeking solely to compete with (or even replace) human experts, our approach learns, not only from traditional past projects cost drivers, but also from past expert judgements, essentially building into the predictive model the ability to learn from their past estimation errors (i.e. misestimates). We name this approach *Learning From Mistakes* (LFM) as it argues that:

1. it is possible to predict the type, severity and magnitude of human experts misestimates by learning from the estimation errors they have made in the past;

2. these predictions can be usefully exploited in order to enhance future effort estimates.

In order to evaluate the feasibility and effectiveness of LFM, we carry out an

empirical study following best practice for the evaluation of prediction models in Software Engineering [218, 219, 73].

To address the first claim, we study the predictability of 402 human expert misestimates in terms of their type (i.e. under-/over- estimates), severity (i.e. low, medium, high), and magnitude (i.e. estimation error relative to the true effort value).

If we can show that these misestimate characteristics are indeed predictable, then we can investigate whether their prediction can be used to improve future effort estimates. In particular, to address the second claim, we adjust the original human expert estimates with the predicted magnitude errors and compare the two (i.e. $originalestimate$ vs. $originalestimate - predictedmisestimate$).

The results of our empirical study show that:

1. **Human expert misestimates are predictable**. The average classification accuracy, measured in Area Under the ROC Curve, for both the type of misestimation and its level of severity of all techniques over all datasets, is 71% and 70%, respectively. Also, the prediction of the amount of misestimation made by human experts is very close to the true amount of misestimation (i.e. the average median absolute error of all classifiers across all datasets is 0.28).

2. **This predictability can be usefully exploited**. That is, LFM enhances human experts' effort predictions obtaining estimates that are significantly better than those provided by random guessing, human experts and traditional automated learners in 32 out of 36 cases (89%) (with large and very large effect sizes observed in 81% of these cases), and never worse in the remaining 11% cases.

The scientific contribution of these findings is the empirical evidence to support the claim that human estimation errors are, indeed, predictable and can be used to improve human experts' effort estimates. This is the first time this question has been investigated in the software engineering literature. The finding is important because it provides an entirely orthogonal deployment route for estimation technology: Instead of replacing human estimators with machine learnt estimations,

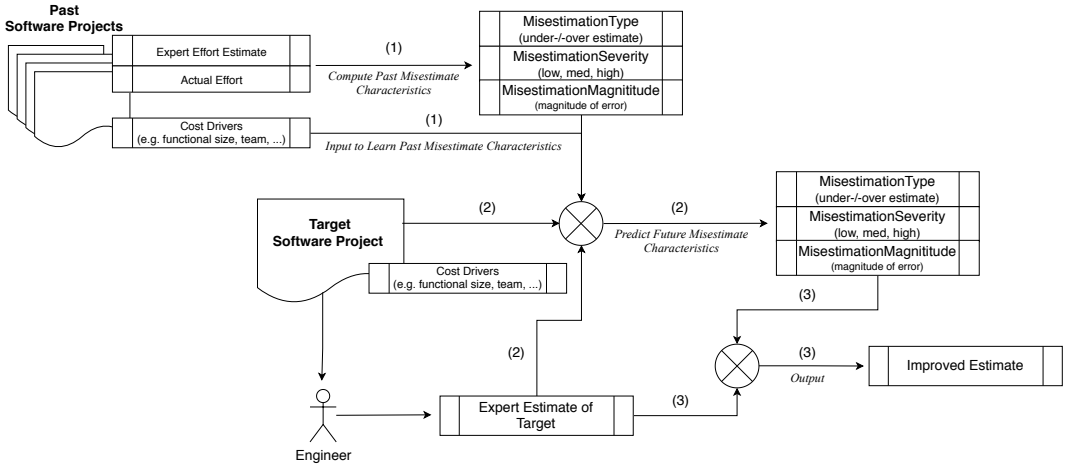


Figure 4.1: The Learning From Mistakes Approach (LFM). The numbers on the arrows correspond to each of the phases described in Section 4.2.

it advocates for techniques that support humans in the necessary task of software development effort estimation.

In the remainder of the chapter we first explain the details of our LFM approach for software effort estimation (Section 4.2) and then the empirical study we carried out to assess its feasibility and effectiveness (Section 4.3). The results of the study are discussed (Section 4.4) and future studies (Section 4.5).

4.2 LFM for Software Effort Estimation

In this section we explain how LFM can be used in practice, by software companies, to estimate/predict the effort needed for realising software projects. LFM is useful in any scenario where the target project for which the effort is unknown but has been estimated by the expert. The aim of LFM is to support the human expert in acknowledging possible errors in their estimations and use this information to improve their estimates. In order to do that, LFM looks at past software projects and learns the errors that experts had committed when estimating the effort. This is divided into three phases that are shown in Figure 4.1 and are described in details below.

Phase 1. Deriving Type, Severity and Magnitude of Past Estimate Errors

This step gathers information about historical software projects realised by the same company or by different ones (usually referred to as single-company data or

cross-company data, respectively). Each of these projects is described by a set of cost drivers, such as functional size, team experience, programming languages and the actual effort (e.g. person-hours) required to realise the project, as recorded by the company employers[1]. Part of the novelty of LFM is to augment these cost drivers by adding information about human expert past misestimates. In particular, LFM computes the type, the severity and the magnitude of past misestimates by using both the human expert effort estimates and the actual effort.

More formally, given a set of past software projects Π, each project $p \in \Pi$, is characterised by the actual effort, ActualEff_p, which was required to complete p and by the estimated effort, EstEff_p, which was originally estimated by the expert[2]. Based on this, the type of error estimate ($\text{MisestimationType}_p$) of a project p is given by

$$\text{MisestimationType}_p = \begin{cases} \text{over-estimate, if } \text{EstEff}_p > \text{ActualEff}_p \\ \text{under-estimate,if } \text{EstEff}_p < \text{ActualEff}_p \end{cases}$$

The severity of the error estimate of a project p ($\text{MisestimationSeverity}_p$) is computed by ranking the past projects with respect to the Magnitude of Relative Error (MRE) and grouping these MRE calculations into different severity levels (i.e. low, medium, high), according to given thresholds as follows:

$$\text{MisestimationSeverity}_p = \begin{cases} \text{low}, & \text{if } \text{MRE}_p < \alpha \\ \text{med}, & \text{if } \alpha \leq \text{MRE}_p < \beta \\ \text{high}, & \text{if } \text{MRE}_p \geq \beta \end{cases}$$

where MRE_p measures, for a given project p, the absolute difference between the actual effort and the estimated effort (i.e. absolute residual error) relative to the actual effort:

$$\text{MRE}_p = \frac{|\text{EstEff}_p - \text{ActualEff}_p|}{\text{ActualEff}_p}$$

[1] More details about the cost drivers used in our experiments are provided in Section 4.3.2.

[2] Such an effort in our study is provided for each project by the software company and it is measured as person hours.

The number of severity levels and the associated thresholds are parameters to our approach determined by the procedures in place in a given company. In this work we experimented with three levels of severity (i.e. low, med, high) according to the following thresholds: the first 33^{th} percentile for the low level (i.e. 33% of projects with the lowest MRE), the 34^{th} to 66^{th} percentile for the med severity level, and the remaining projects for the high one (i.e. 33% of projects with highest MRE). Of course, different settings can be used without altering the formulation of LFM.

The magnitude of the misestimation (MisestimationMagnitude) for a given project p is computed as the relative error:

$$\text{MisestimationMagnitude}_p = \frac{\text{EstEff}_p - \text{ActualEff}_p}{\text{ActualEff}_p}$$

These misestimation characteristics, together with company-specific cost drivers, will be exploited in Phase 2.

Phase 2. Predicting The Characteristics of Future Estimation Errors

The second phase, starts with a software project for which the development effort is unknown and needs to be estimated (i.e. *Target Software Project*). LFM exploits the information gathered in Phase 1, in order to find similarities between the target project and past projects. It uses this knowledge in order to predict the characteristic of the estimate error for the target project.

Specifically, from the information gathered in Phase 1 we are able to predict the type, the severity and the magnitude of the errors that might occur when predicting the effort of a target project (tp) (i.e. $\text{MisestimationType}_{tp}$, $\text{MisestimationSeverity}_{tp}$, $\text{MisestimationMagnitude}_{tp}$).

In order to predict $\text{MisestimationType}_{tp}$, we can use a two-class (i.e. binary) classifier since MisestimationType can assume only two values (i.e. under-/over-estimates), while to predict the severity of the error (i.e. $\text{MisestimationSeverity}_{tp}$), a multiclass (i.e. multinomial) classifier[3] is necessary, given the severity levels de-

[3]Multiclass classification is the problem of classifying instances into one of the more than two classes [220]. Classifying instances into one of the two classes is called binary classification. Multi-class classification should not be confused with multi-label classification, where multiple labels are to be predicted for each instance.

fined in Phase 1. In order to predict the magnitude of the misestimates for the target project (i.e. MisestimationMagnitude$_{tp}$), and given the regression nature of the problem, any automated estimator ranging from simple regression- or analogy-based learners [221, 62] to more sophisticated ones such as those based on evolutionary approaches [120, 132] or deep-learning [222] can be applied. Obviously, different learners may exhibit different performance for different scenarios. Ultimately, the choice of the learner is a parameter of our approach.

Dataset	Misestimation Type (RQ1.1)		Misestimation Severity (RQ1.2)			Misestimation Magnitude (RQ2)				Actual Effort (RQ3)			
	Under	Over	Low	Med	High	Min	Max	Mean	Std. Dev.	Min	Max	Mean	Std. Dev
ISBSG-C	63%	37%	33%	33%	34%	-0.93	29.49	0.54	4.26	207	46787	6574.22	10641.24
ISBSG-FP	65%	35%	33%	33%	34%	-0.96	10.83	-0.02	0.88	207	63732	5827.29	7550.27
KD	41%	59%	34%	31%	35%	-0.40	3.59	0.23	0.72	286	113930	5450.00	17723.16
KP	35%	65%	33%	33%	34%	-0.65	1.31	0.15	0.35	219	8656	2046.30	1927.96
Medical	52%	48%	32%	34%	34%	-0.83	0.07	0.46	1.42	60	10060	1530.00	1785.98
Telecom	59%	41%	35%	30%	35%	-0.49	0.61	-0.09	0.26	279	10244	2403.29	2707.80

Table 4.1: Descriptive statistics of the target variables used for each research question.

Phase 3. Exploiting Predicted Misestimations:

The third and final phase of our approach involves exploiting the misestimations predicted for the target project. The early identification of potential under-/over- estimates and their severity can better guide human experts in understanding their predictions. On the other hand, the predicted misestimation magnitude can be used to automatically adjust human expert estimates. In the empirical study presented in this chapter, we show that we can enhance the effort estimate produced by a human expert for a target project by adjusting it using the misestimation magnitude predicted during Phase 2 ($EstEff_{tp} - MisestimationMagnitude_{tp}$).

4.3 Empirical Study Design

In this section we explain the design of the empirical study we carried out to asses the feasibility and effectiveness of LFM. Our study follows the most recent best practices for the evaluation of prediction models in software engineering [218, 223, 224, 225, 226].

4.3.1 Research Questions

Our first two research questions investigate the predictability of the error made by human experts when estimating software project effort. The first research question tackles this as a classification problem, whereas the second research question treats it as a regression task. Specifically, RQ1 investigates whether we can classify the misestimation type and severity (i.e. MisestimationType and MisestimationSeverity as defined in Section 4.2 Phase 2), while RQ2 investigates whether we can estimate the magnitude of the error (MisestimationMagnitude as defined in Section 4.2 Phase 2). Our third and final research question focuses on the use of the predicted misestimations in order to adjust future human expert estimates as explained in Section 4.2 Phase 3. In the following we describe the way they are addressed.

RQ1. Predicting Type/Severity of Human Expert Misestimations: Can we predict the type and severity of the errors made by human experts when estimating software effort?

To address this question, we use four different machine learning techniques, namely CART, KNN, NB and RF (for more details, see Section 4.3.3) to classify the type and severity of human expert misestimates. In particular, we answer the following sub-questions:

RQ1.1 To what extent is the type of human expert misestimation predictable?

RQ1.2 To what extent is the severity of human expert misestimation predictable?

RQ2. Predicting the Magnitude of Human Expert Misestimations: Can we predict the magnitude of the misestimations made by human experts?
To answer this question we assess the effectiveness of four machine learners (i.e. CART, KNN, LP, RF). As a sanity check, we compare them with Random Guessing (RG).

RQ3. Enhancing Software Effort Estimates via LFM: Can software effort predictions be improved by learning from previous misestimations?

In order to address this question, we compare the prediction produced by LFM against human estimations of software development effort. For completeness, we

also compare LFM with estimations obtained using traditional machine learning approaches alone. As a sanity check, we also compare LFM against RG. Therefore, we answer the following sub-questions:

RQ3.1: Does LFM provide better effort estimates than RG?

RQ3.2: Does LFM provide better effort estimates than traditional machine learners?

RQ3.3: Does LFM provide better effort estimates than human experts?

In the following we describe in detail the data (Section 4.3.2), the techniques (Section 4.3.3), the validation approach (Section 4.3.4), and the evaluation criteria and statistical tests (Section 4.3.5) used to address the above RQs. We also discuss possible threats to the validity of our empirical study (Section 4.3.6).

4.3.2 Datasets

To answer the RQs outlined in Section 4.3.1 we carry out an empirical study using real-world industrial datasets containing a total of 402 software projects developed by different software companies world-wide and collected up to 2018.

These datasets cover a variety of application domains (ranging from telecommunications to medical information systems), exhibit different project characteristics (e.g. technologies, tools and programming languages) and also vary in size (17 to 190 projects, 4 to 17 cost drivers depending on the dataset).

Table 4.2 summarises the descriptive statistics of the features of each of the datasets. We can observe that five datasets contain cost drivers based on the Function Point Analysis (FPA) [227] or COSMIC [51] functional size measurement (FSM) methods[4], which are widely used as independent variables to derive effort estimation models [124]. The `Medical` dataset instead contains features computed based on data models (e.g. number of entities), which have been shown to be useful

[4]FSM methods have obtained world-wide acceptance [20] and allow software size measurement in terms of the functionality with which users are provided. The first FSM method was FPA [227], and several variants have since been defined (e.g. MarkII and NESMA) with the aim of improving size measurement or extending the applicability domain [228]. These are all referred to as the first generation of FSM methods. COSMIC is instead a second generation FSM method having distinguishing characteristics, including its applicability to business, real-time, and infrastructure software (or their hybrids) [51], and possibility to extend its usage to other kinds of software such as Web and Mobile applications[52, 53, 54, 55, 56, 57].

cost drivers in previous work [57]. Moreover, all datasets contain two more features: the hours to complete a software project as estimated by a human expert (i.e. Expert Estimated Effort) and the number of hours actually required to complete it as recorded at the end of the project by the company (i.e. Actual Effort). These two features are used to compute the MisestimationType, MisestimationSeverity and the MisestimationMagnitude which are used as prediction target (i.e. dependent variable) for RQ1 and RQ2 as explained in Section 4.2, while the Actual Effort is used as a prediction target for RQ3. In our experiments we use both datasets consisting of projects that have been estimated by one expert (e.g., Medical) and others where the effort of different projects has been estimated by different experts (e.g., ISBSG). Descriptive statistics of the targets of each RQ are provided in Table 4.1.

Further details for each of the datasets are provided below to allow readers to assess whether the results we have gathered may apply to their own context.

The `ISBSG-C` and `ISBSG-FP` are private datasets which have been collected from the International Software Benchmarking Standards Group (ISBSG) repository release June 2018 R2 [59]. This repository contains software projects submitted by leading IT and metrics companies from around the world and has been widely used by practitioners as well as researchers for software project effort estimation studies [229]. The `ISBSG-C` dataset contains 49 projects characterised by four independent variables based on COSMIC [51] (i.e. Cosmic Entry, Cosmic Exit, Cosmic Read, Cosmic Write), Expert Estimated Effort and Actual Effort. While the `ISBSG-FP` dataset contains 190 projects characterised by six independent variables based on FP [227] (i.e. Input Count, Output Count, Enquiry Count, File Count, Interface Count, Added Count), Expert Estimated Effort and Actual Effort. We cannot disclose more details about this dataset due to a non-disclosure agreement (NDA).

The `Kitchenham` dataset consists of public data from both maintenance and new development software projects curated by the Computer Sciences Corporation on behalf of several client organisations [230]. This dataset contains projects spanning different products from different sources. All the projects are characterised

by five independent variables based on the Function Point counts (i.e. External Input, External Output, Logical Internal, External Interface, External Inquiry), the Expert Estimation and the Actual Effort. The effort estimates were made as part of the company's standard estimating process. In our study we considered only those software projects for which the effort estimate was made solely based on human experts' judgement for a total of 69 projects. Since these projects include both perfective maintenance and development projects, we analyse them separately, thereby obtaining two disjoint sets, namely `KD` and `KP` which contain 29 and 40 projects, respectively. More details about this dataset and its raw data can be found elsewhere [230].

The `Medical` dataset provides a set of 24 cost drivers (see Table 4.2 for the the full list) and effort data privately recorded for 77 modules of a single software system (i.e. a medical records database system built and implemented over a period of five months) [130]. Each of the modules implements a data entry/edit or reporting functionality and has associated 24 cost drivers. Since all this data was available during the module specification phase it can be used as input to a prediction system [130]. For each of the modules a single project manager's estimate of the effort, and the actual effort (both expressed in person-hours) are available.

`Telecom` is a privately curated dataset which contains 17 software projects developing typical administrative software, internal software development for a telecommunication company.

All projects are characterised by four independent variables (i.e. Input Types, Entities, Output Types, Transactions), each representing an FP basic component [227], Expert Estimated Effort and Actual Effort. We cannot disclose more details about this dataset due to an NDA.

Dataset	Feature	Min	Max	Mean	Std. Dev.
ISBSG-C (49 projects)	Cosmic Entry	4	447	86.51	107.63
	Cosmic Exit	2	594	107.31	130.75
	Cosmic Read	0	545	83.88	109.24
	Cosmic Write	0	542	55.49	93.62
	Expert Estimated Effort	90	53774	7349.02	10641.24
	Actual Effort	207	46787	6574.22	9338.89
ISBSG-FP (190 projects)	Input count	0	2014	128.34	196.70
	Output Count	0	2760	77.34	213.85
	Enquiry Count	0	2356	105.85	189.40
	File Count	0	3196	93.69	254.20
	Interface Count	0	261	18.25	38.39
	Added Count	0	10571	331.92	815.44
	Expert Estimated Effort	80	58800	5082.94	7407.87
	Actual Effort	207	63732	5827.29	7550.27
KD (29 projects)	External Input	0	4701	263	731.69
	External Output	6	5265	241.40	812.53
	Logical Internal	0	1724	113.90	276.95
	External Interface	0	92	6.73	15.51
	External Inquiry	0	2925	152.40	454.82
	Expert Estimated Effort	337	79870	4586.00	12417.53
	Actual Effort	286	113930	5450.00	17723.16
KP (40 projects)	External Input	0	789	125.82	147.63
	External Output	0	360	81.00	95.71
	Logical Internal	0	402	61.02	89.45
	External Interface	0	614	25.52	89.72
	External Inquiry	0	618	80.50	125.81
	Expert Estimated Effort	200	8690	2038.70	1736.68
	Actual Effort	219	8656	2046.30	1927.96
Medical (77 projects)	Create Transactions	0	3	0.85	0.74
	Read Transactions	0	25	5.19	4.57
	Update Transactions	0	16	1.47	2.20
	Delete Transactions	0	2	0.26	0.59
	Reports Called	0	2	0.23	0.60
	Reports Produced	0	2	0.27	0.50
	Elements Reported	0	24	3.26	6.46
	Fields Calculated	0	14	0.77	2.45
	Fields Entered	0	19	5.26	3.82
	Screens Called	0	10	0.79	1.89
	Screens Displayed	0	6	1.04	0.90
	Elements Displayed	0	78	11.64	14.04
	Entities	0	22	4.27	3.72
	Entities Providing Data	0	14	4.13	3.25
	Entities Consuming Data	0	16	1.20	1.90
	Attributes	0	19	5.26	3.82
	Attributes Updated	0	69	9.42	13.26
	Attributes Consumed	0	83.00	19.81	18.69
	Links (1.1)	0	2	0.30	0.49
	Links (1.m)	0	13	3.09	3.32
	Optional Links	0	12	3.13	3.30
	Mandatory Links	0	3	0.25	0.54
	Entity Provisions	0	25	5.20	4.57
	Entity Consumptions	0	16	1.52	2.20
	Expert Estimated Effort	228	9450	1120.00	1278.82
	Actual Effort	60	10060	1530.00	1785.98
Telecom (17 projects)	Input types	4	858	201.71	242.15
	Entities	15	444	124.53	110.53
	Output Types	10	2322	484.88	640.69
	Transactions	7	265	51.18	59.54
	Expert Estimated Effort	450	9595	1967.29	2284.26
	Actual Effort	279	10244	2403.29	2707.80

Table 4.2: Descriptive statistics of the datasets used.

4.3.3 Classification and Regression Techniques

The concept of LFM is not defined by the machine learning approach used to classify the estimate errors. Therefore any technique can be used to this end and the choice is left to the practitioner.

In our empirical study, we experiment with five publicly available machine learners, namely Classification and Regression Trees (CART) [231], k-Nearest Neighhbours (KNN) [232], Naïve Bayes (NB) [233], Linear Programming (LP) [234] and Random Forest (RF) [235], all of which are well-known and widely-used by software engineering researchers and practitioners. Using such approaches avoids the risk that LFM benefits from some special or sophisticated ML technique. The results achieved with these traditional techniques can be considered as a lower bound to any more advanced technique, while their public availability supports and promotes replicability and extension of our work.

In order to address RQ1 (which involves a classification task), we use four machine learning approaches, namely CART, KNN, NB, RF, which are able to handle both binary and multiclass problems [220]. To address RQ2 and RQ3, which involve a regression task, instead, we use LP4EE (as it has been recently proposed as a robust baseline approach for prediction studies [219]) and three traditional and widely used estimation methods, namely CART, RF and KNN, which are representative of regression-based and analogy-based estimators, respectively. Moreover, as a sanity check we always compare all the approaches to RG for all RQs. For each of these techniques we use the `R` tool[5] version 3.4.1. For CART, KNN, NB and RF, we build and tune a model for each LOO training set within each dataset, and use it to predict the effort of the target observation. We use the function `trainControl` available from the `R` package `Caret`[6] version 6.0.84, which performs a search to identify machine learning settings that generalise best on the training set[7], as recommended in recent work [238, 225]. In the following we briefly describe each of

[5] `https://www.r-project.org`

[6] `http://topepo.github.io/caret/index.html`

[7] Specifically, we used the setting `method=repeatedcv, repeats=30` (and `tuneLength = 10` for KNN). Since more advanced tuning techniques can be used [236, 237], the results provided herein can be considered as a lower bound.

the techniques.

Random Guessing (RG) is a worst case lower bound benchmark suggested to assess the usefulness of a prediction system [218]. It randomly assigns the y value of another case to the target case. More formally, it is defined as: predict a y for the target case t by randomly sampling (with equal probability) over all the remaining $n-1$ cases and take $y = r$ where r is drawn randomly from $1...n^r = t$ [218]. Any prediction system should outperform RG since an inability to predict better than random implies that the prediction system is not using any target case information.

Classification and Regression Trees (CART) are machine learning methods used to build prediction models by recursively partitioning the data and fitting a simple prediction model within each partition [239]. The partitioning can be represented graphically with a decision tree. Decision trees where the dependent variable takes a finite set of values are called classification trees, while those where the dependent variable takes continuous values are called regression trees.

K-Nearest Neighbor (KNN) is an analogy-based approach that, given a target instance (i.e. a software project characterized by a vector of n features), retrieves from a case base of past projects, those instances which are relevant to the target one [240]. These relevant cases are identified by using the Euclidean distance as a similarity function, which measures the distance between the target case and the other cases based on the values for the n features of these projects. The average of the effort values of the k most similar past projects is then used as the effort predicted for the target project. If there are ties for the k-th nearest vectors, all candidates are used to compute the average. The choice of k is left to the user, in this work we experiment with different values of $k = 1, ..., 10$.

Linear Programming for Effort Estimation (LP4EE) is a baseline prediction model recently proposed to provide a robust yet easy-to-use approach for effort estimation[8][219]. The model takes advantage of the Simplex algorithm, which deterministically minimizes an error function on a training set, and applies the learnt weights on a test set to make predictions [219]. In this work we extend the original

[8]The source code is available at `https://github.com/fedsar/LP4EE`

formulation of LP4EE [219] to handle negative-values predictions (see Appendix A). The original version is used in RQ3 (to predict the effort) whereas, the modified version is used in RQ2 (to predict the MisestimationMagnitude which can take the form of both positive and negative values).

Naïve Bayes (NB) is a statistical technique that uses the combined probabilities of the different attributes to predict the target variable, based on the principle of Maximum A Posteriori [241]. This approach is naturally extensible to the case of having more than two classes, and was shown to perform well in spite of the underlying simplifying assumption of conditional independence.

Random Forest (RF) is an ensemble technique which aggregates the predictions made by a collection of decision trees (each with a subset of the original set of attributes) [242]. Each tree infers a split of the training data based on feature values to produce a good generalization. RF can naturally handle binary or multiclass classification problems. The leaf nodes refer to either of the classes concerned.

4.3.4 Validation Approach

For each of the datasets in our study, we perform a Leave-One-Out (LOO) cross-validation for all RQs. Given a dataset containing n observation, one observation at a time is used as target and the remaining $n-1$ instances are used to train the model; the process is repeated n times. Thus, for each dataset, we obtained n pairs of training data and test data, and we report the results obtained on the test data by using boxplots, summary statistics, and statistical tests as detailed in Section 4.3.5. LOO is a deterministic approach that, unlike other cross validation techniques, does not rely on any random selection to create the training and testing sets. According to recent work [73], assessment via LOO eliminates conclusion instability caused by random sampling, making evaluations that use it more easily reproducible. However, if chronological information about the projects is available it would be preferable to adopt a time-based validation approach, because LOO may give more optimistic results than those that might realistically be achieved in practice [74]. In our study we use LOO because start and completion dates are not available for all projects.

4.3.5 Evaluation Criteria and Statistical Tests

In order to evaluate the performance of the techniques considered in RQ1 (i.e. CART, KNN, NB and RF) to classify misestimation types and severity we used the Area Under the ROC Curve (AUC-ROC) [243], which value ranges between 0 and 1. For a two-class problem (such as classifying error types) an AUC-ROC value of 1 represents a perfect classifier, while an area of 0.5 represents a Random (i.e. worthless) one. To evaluate the performance for a multi-class problem (such as classifying the error severity) we exploited the generalization to n-class classification proposed by Hand and Till [244] which extends the AUC definition to the case of more than two classes by averaging pairwise comparisons. In this case a value of 1 still represents a perfect classifier, while an area of $1/n$ represents a Random classifier, where n is the number of classes considered.

In order to compare the performance of the estimation methods analysed for RQ2 and RQ3 (i.e. CART, KNN, LP, LFM and Expert), we measured the Absolute Error (i.e. $|PredictedValue - RealValue|$), where the *RealValue* is our target prediction variable. This target prediction variable is equal to MisestimationMagnitude for RQ2 and it is equal to ActualEffort for RQ3. Thus, for RQ2 we measure the Absolute Error between the human expert MisestimationMagnitude and the one predicted by LFM; while to answer RQ3 we compute the absolute error between the actual effort and the effort predicted by LFM, human expert, and the considered ML. We use boxplots to visualise the difference in performance among different prediction methods and also use significance statistical tests. Both the boxplots and the statistical tests are based on these distributions.

In order to evaluate whether the differences in performance of the classifiers used in RQ1 are significant, we use the Friedman Test[9] [245]. This is a non-parametric test which works with the ranks of the techniques rather than their actual performance values, making it less susceptible to the distribution of the performance of these parametric values. The null hypothesis that is tested in our work is the following: "There is no significant difference in the AUC-ROC values obtained by the

[9]The `R` package `stats` (version 3.6.1) was used for the Friedman and the Wilcoxon Signed-Rank tests, the `R` package `PMCMRplus` (version 1.4.2) was used for the Nemenyi Test.

approaches compared", at a confidence limit, α, of 0.05. If the null-hypothesis is rejected, then it can be concluded that at least two of the techniques are significantly different from each other. When a significant difference is found, the Nemenyi test [246] is often recommended as a post-hoc test to identify the techniques with a statistically significant difference[9] [247]. The performance of two classifiers is thought to be significantly different if the corresponding average ranks differ by at least the critical distance (CD) [247]. The results of this test are presented in a diagram which is used to compare the performance of multiple techniques by ranking them. It consists of an axis, on which the average ranks of the methods are plotted and of the CD bar. The groups of classifiers whose values are significantly different, are not connected by a line.

To establish if the estimates of one method are statistically significantly better than the estimates provided by another method (RQ2 and RQ3), we compare the absolute errors they achieved for each of the datasets. In particular, to check for statistical significance, we use the Wilcoxon Signed-Rank Test [248][9], which is a safer test to apply than parametric tests, since it raises the bar for significance, by making no assumptions about underlying data distributions. In particular, we test the following Null Hypothesis: "The absolute errors provided by the prediction model P_i are not significantly lower than those provided by the prediction model P_j.", set the confidence limit, α, at 0.05 and applied the Bonferroni correction (α/K, where K is the number of hypotheses) when multiple hypotheses were tested. The Bonferroni correction is the most conservatively cautious of all corrections and its usage allows us to avoid the risk of Type I errors (i.e. incorrectly rejecting the Null Hypothesis and claiming predictability without strong evidence). In order to investigate the effect size of the Wilcoxon Signed-Rank Test results, we compute the correlation coefficient $r = \frac{Z}{\sqrt{N}}$, where Z is the standard score of the Wilcoxon test and N is the number of pair observations. Indeed, r is recommended as an effect size measure for paired non-parametric statistical significance tests [249]. The r effect is considered small $\geq$0.10, medium $\geq$0.30, large $\geq$ 0.50 and very large $\geq$ 0.70 [250, 251].

4.3.6 Threats to Validity

In this section we discuss the construct, conclusion, and external threats to the validity of our empirical study.

To satisfy construct validity, a study has "to establish correct operational measures for the concepts being studied" [252]. This means that the study should represent to what extent the predictor and response variables precisely measure the concepts they claim to measure [253]. Thus, the choice of the features and how to collect them represents a crucial aspect. We tried to mitigate such a threat by using real-world data previously used to empirically evaluate effort estimation methods. We mitigate threats arising from unrealistic or incorrect data usage by only considering software projects for which the cost-drivers were collected and measured before human experts made the predictions, and were never modified afterwards, so that they can be correctly used as independent variables in machine learning prediction systems. Moreover, we considered only those projects for which the human-estimated efforts were made solely based on human expert judgement (i.e. no other technique was used to support experts in their estimation). Given that our approach aims to adjust and enhance the expert's final estimate, we need to use projects where the expert's estimation is provided. However, not all projects contain also chronological information and we had to use the LOO validation, which may lead to more optimistic results with respect to a time-based validation [74].

With regards to the conclusion validity, we carefully applied the statistical tests, verifying all the required assumptions and correcting for multiple hypotheses statistical testing. We also followed recent best practice to assess prediction systems [218, 254, 219]. Moreover, we used datasets of different sizes to mitigate the threats related to the number of observations. We also used traditional ML techniques implemented in publicly available tools to allow for replications and comparisons.

To mitigate threats to external validity we used six real-world industrial datasets containing software projects related to different application domains and companies, which are thus characterised by various project and human factors such as development process, developer experience, tools and technologies used, cost

drivers, time and budget constraints [255]. Although we used a set of subjects that has such a degree of diversity, we cannot claim that our results generalise beyond the subjects studied. It is worth noting that the formulation of the approach is independent from the nature of the projects. That is, the approach could potentially work with any kind of project as long as they can be characterised in terms of the same (or a subset of) cost drivers used for describing the past projects stored in the database. In our empirical study we experimented with both new development and maintenance projects, and we study both the effort of realizing entire projects and specific software modules, in order to assess the feasibility of LFM for a wide range of projects type.

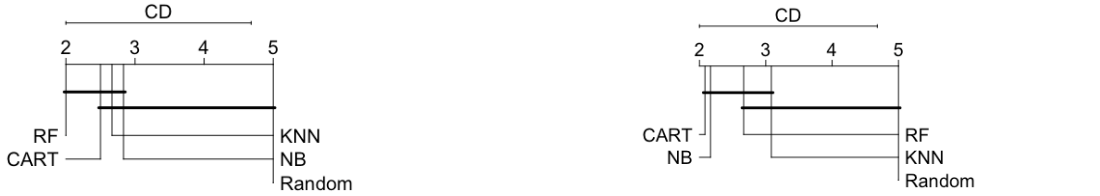
4.4 Empirical Study Results

In this section we report and discuss the results we obtained carrying out the empirical study described in Section 4.3.

4.4.1 RQ1. Predicting Type/Severity of Human Expert Misestimations

To address RQ1 we compare the performance of CART, KNN, NB and RF for predicting the type and the severity of human expert estimate errors. The accuracy results, measured by the AUC-ROC, are summarised in Table 4.3.

RQ1.1-Predicting human expert misestimation type: From Table 4.3 we can observe that all techniques outperform the random classifier (i.e. AUC-ROC > 0.5) in all of the cases studied with an average AUC-ROC across all techniques and datasets equal to 0.71. Moreover, results show that RF obtains the highest AUC-ROC values on three out of the six datasets under study (i.e. `ISBSG-FP`, `KP`, `Medical`) with AUC-ROC values ranging from 0.65 to 0.92. This conclusion is reinforced by the Friedman Test as it shows statistically significant difference (p-value < 0.001) between the performance of the techniques studied (CART, KNN, NB, RF, RG). Nemenyi's Critical-Difference (shown in Figure 4.2a) also supports this, ranking RF first and Random last with a statistically significant difference (p-value < 0.001).



(a) MisestimationType **(b)** MisestimationSeverity

Figure 4.2: RQ1: Critical Difference (CD) diagram of the post-hoc Nemenyi test with α = 0.05. The difference between two methods is significant if the gap between their ranks is larger than the critical distance. There is a line between two methods if the rank gap between them is smaller than the critical distance.

Target Variable	Dataset	CART	KNN	NB	RF
MisestimationType (binary class)	ISBSG-C	**0.75**	0.63	0.62	0.56
	ISBSG-FP	0.61	0.59	0.61	**0.65**
	KD	0.71	**0.85**	0.62	0.63
	KP	0.59	0.68	0.71	**0.82**
	Medical	0.86	0.88	0.91	**0.92**
	Telecom	**1.00**	0.64	0.57	0.65
MisestimationSeverity (multi-class)	ISBSG-C	**0.95**	0.54	0.66	0.65
	ISBSG-FP	0.53	**0.57**	0.53	0.56
	KD	**0.86**	0.69	0.79	0.72
	KP	0.64	**0.68**	0.62	0.60
	Medical	0.64	0.65	**0.73**	0.67
	Telecom	**1.00**	0.93	0.83	0.83

Table 4.3: RQ1: AUC-ROC values obtained by CART, KNN, NB and RF when predicting the type and the severity of human expert misestimations.

RQ1.2-Predicting human expert misestimation severity: Similar observations hold when we consider the prediction of MisestimationSeverity. Results show that all techniques always provide better AUC-ROC values than random classification (i.e. AUC-ROC $>$ 0.33), with an average AUC-ROC value of 0.70 of all techniques across all datasets. Results also show that CART obtains the highest AUC-ROC values on three out of six datasets (ISBSG-C, KD and Telecom), whereas KNN performs best on two of the remaining datasets (ISBSG-FP and KP) and NB performs best on the remaining dataset. The Friedman Test also concludes a difference in the predictors' performance (p-value $<$0.001) with Nemenyi's Critical-Difference Diagram (shown in Figure 4.2b) ranking CART first and Random last with a statistically significant difference when comparing the two. On the other end, KNN, NB and RF rank second, third and fourth, respectively, with the gap between their ranks and Random not being larger than CD.

Therefore, in answering to RQ1 we can state that:

> **Answer to RQ1:** The type and severity of human expert misestimations are predictable with an average AUC-ROC value of all techniques (over all datasets) being equal to 0.71 for type and 0.70 for severity.

4.4.2 RQ2. Predicting the Magnitude of Human Expert Misestimations

To answer RQ2, we investigate the capability of traditional regression- and analogy-based estimation approaches (i.e. CART, KNN, LP and RF) to predict the MisestimationMagnitude of human expert misestimates.

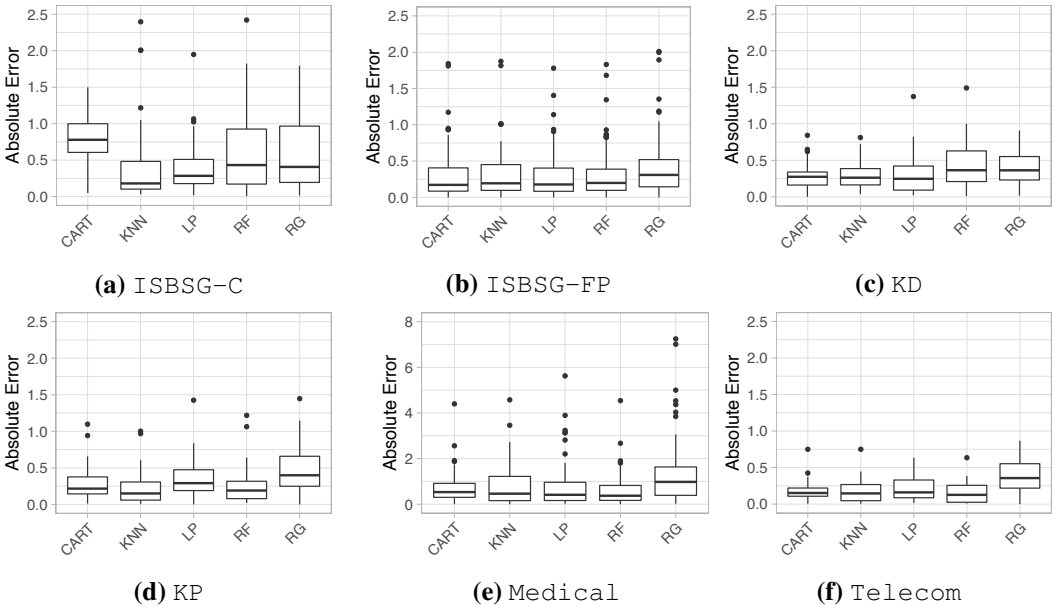
Figure 4.3 shows the boxplots of the distribution of the absolute prediction errors produced by CART, KNN, LP and RF as well as the sanity check, RG.

We can observe that all techniques are able to predict, with a low absolute error, the magnitude of the error committed by human experts when estimating software effort. This can be seen from the boxplots of each dataset where the median of the best technique does not exceed an absolute error of 0.25 on all datasets. Results also show that the median absolute error of all ML techniques over all datasets is also low, with an average equal to 0.28.

Figure 4.3 shows that all techniques outperform RG on four (out of six) datasets. Whereas, on the remaining two datasets (`ISBSG-C` and `KD`), at least two of the machine learners (namely KNN and LP) achieve better results than RG, with no technique being worse. The Wilcoxon test results (reported in Table 4.4) also support this conclusion as they show that all techniques are statistically significantly better than RG on four out of six datasets (i.e. `ISBSG-FP`, `KP`, `Medical`, `Telecom`), with 13% of these cases having a very large effect size, 31% having a large effect size, 50% having a medium and 6% small effect sizes. Whereas, on the `ISBSG-C` dataset KNN and LP significantly outperform RG, and for the other two techniques, the null hypothesis cannot be rejected as well as for the `KD` dataset.

These results highlight that the magnitude of misestimations is indeed predictable for all datasets considered. Moreover, for each of the datasets we can identify the best performing approach based on its median absolute errors (as shown by

the bar in the boxplots - Figure 4.3) and the number of times it is statistically significantly better than the other ones according to the Wilcoxon test and effect size (Table 4.4). In RQ3 we will refer to this approach as LFM.



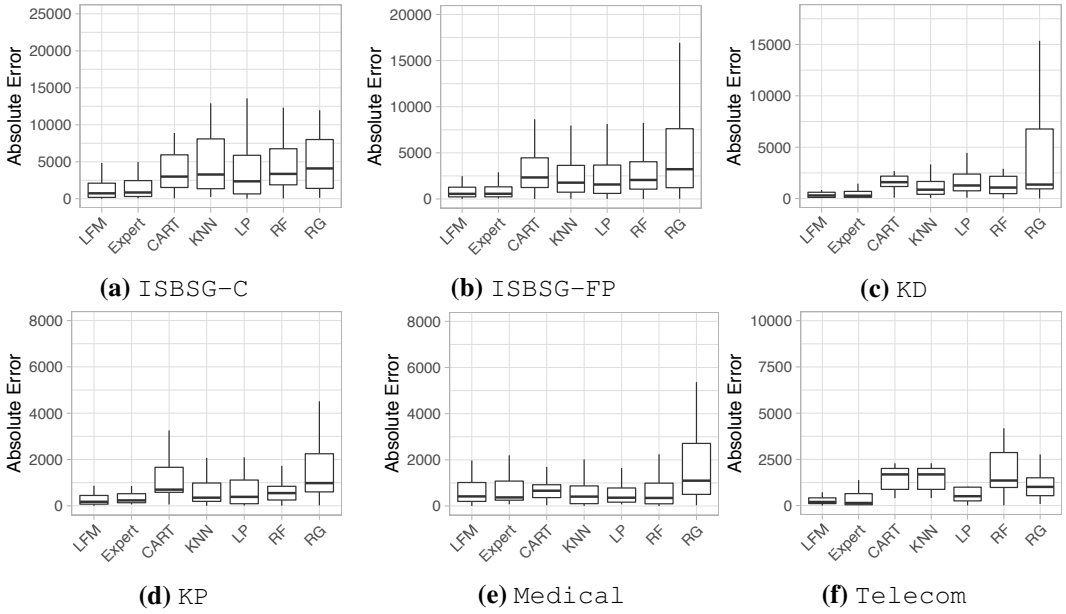
(a) ISBSG-C **(b)** ISBSG-FP **(c)** KD

(d) KP **(e)** Medical **(f)** Telecom

Figure 4.3: RQ2. Boxplots of the absolute errors achieved by CART, KNN, LP, RF and RG when predicting the magnitude of human expert misestimations.

Dataset	Technique	vs. CART	vs. KNN	vs. LP	vs. RF	vs. RG
ISBSG-C	CART	-	1.000 (0.00)	1.000 (0.00)	0.862 (0.02)	0.905 (0.02)
	KNN	<0.001 (0.51)	-	0.201 (0.18)	0.001 (0.47)	0.001 (0.49)
	LP	<0.001 (0.73)	0.802 (0.04)	-	0.003 (0.43)	0.036 (0.30)
	RF	0.140 (0.21)	0.999 (0.00)	0.997 (0.00)	-	0.547 (0.09)
ISBSG-FP	CART	-	0.126 (0.11)	0.996 (0.00)	0.755 (0.02)	<0.001 (0.37)
	KNN	0.874 (0.01)	-	0.964 (0.00)	0.951 (0.00)	<0.001 (0.29)
	LP	0.004 (0.21)	0.036 (0.15)	-	0.382 (0.06)	<0.001 (0.39)
	RF	0.245 (0.08)	0.049 (0.14)	0.619 (0.04)	-	<0.001 (0.38)
KD	CART	-	<0.001 (0.74)	0.517 (0.12)	0.007 (0.50)	0.111 (0.30)
	KNN	0.999 (0.00)	-	0.710 (0.07)	0.088 (0.32)	0.221 (0.23)
	LP	0.492 (0.13)	0.297 (0.19)	-	0.008 (0.49)	0.078 (0.33)
	RF	0.994 (0.00)	0.916 (0.02)	0.992 (0.00)	-	0.703 (0.07)
KP	CART	-	0.999 (0.00)	0.032 (0.34)	0.969 (0.01)	0.005 (0.44)
	KNN	0.001 (0.53)	-	0.001 (0.54)	0.049 (0.31)	<0.001 (0.70)
	LP	0.969 (0.01)	0.999 (0.00)	-	0.999 (0.00)	0.020 (0.37)
	RF	0.032 (0.34)	0.953 (0.01)	0.001 (0.51)	-	<0.001(0.62)
Medical	CART	-	0.790 (0.03)	0.320 (0.12)	0.999 (0.00)	<0.001 (0.46)
	KNN	0.211 (0.15)	-	0.461 (0.09)	0.987 (0.00)	<0.001 (0.45)
	LP	0.682 (0.05)	0.542 (0.07)	-	0.987 (0.00)	<0.001 (0.47)
	RF	0.001 (0.39)	0.014 (0.29)	0.013 (0.29)	-	<0.001 (0.56)
Telecom	CART	-	0.663 (0.11)	0.482 (0.17)	0.897 (0.03)	0.010 (0.62)
	KNN	0.363 (0.22)	-	0.373 (0.22)	0.824 (0.05)	0.010 (0.62)
	LP	0.537 (0.15)	0.644 (0.11)	-	0.785 (0.07)	0.005 (0.67)
	RF	0.112 (0.39)	0.189 (0.32)	0.229 (0.29)	-	<0.001 (0.96)

Table 4.4: RQ2: Results of the Wilcoxon test (p-value and *r* effect size) comparing the absolute errors provided by CART, KNN, LP and RF vs. each other and vs. RG when predicting the human expert MisestimationMagnitude.



(a) ISBSG-C **(b)** ISBSG-FP **(c)** KD

(d) KP **(e)** Medical **(f)** Telecom

Figure 4.4: RQ3. Boxplots of absolute errors obtained by LFM, Human Experts and traditional automatic estimators (i.e. CART, KNN, LP, RF) when predicting software projects' effort.

Based on the results above we can conclude that:

> **Answer to RQ2:** The misestimate magnitude is highly predictable with an average value of the median absolute errors (MedAE) obtained by all techniques, across all datasets, being equal to 0.28.

4.4.3 RQ3. Enhancing Software Effort Estimates via LFM

To address RQ3, we investigate the capability of LFM to improve human expert estimates. Figure 4.4 shows the boxplots of the distributions of absolute error values obtained by LFM, human experts and the automated estimators (i.e. CART, KNN, LP and RF) when predicting the effort of software projects.

RQ3.1 LFM vs. Random Guessing (RG): The improvements achieved by LFM over RG (as shown in Figure 4.4) are always statistically significant ($p < 0.001$) with five very large ($r \geq 0.7$) effect sizes and a large one ($r \geq 0.5$) (see Table 4.5). Thereby LFM successfully passes our sanity check of beating Random Guessing.

RQ3.2 LFM vs. Traditional ML Estimators: The analysis of the boxplots of

	vs. Expert	vs. CART	vs. KNN	vs. LP	vs. RF	vs. RG
ISBSG-C	0.002 (0.44)	<0.001 (0.66)	<0.001 (0.66)	<0.001 (0.58)	<0.001 (0.62)	<0.001 (0.75)
ISBSG-FP	0.011 (0.19)	<0.001 (0.65)	<0.001 (0.62)	<0.001 (0.55)	<0.001 (0.61)	<0.001 (0.71)
KD	0.449 (0.14)	<0.001 (0.72)	0.001 (0.60)	0.002 (0.57)	0.003 (0.55)	<0.001 (0.97)
KP	0.035 (0.33)	<0.001 (0.82)	0.001 (0.55)	0.005 (0.44)	<0.001 (0.65)	<0.001 (0.91)
Medical	0.013 (0.29)	0.014 (0.29)	0.963 (0.01)	0.971 (0.00)	0.988 (0.00)	<0.001 (0.60)
Telecom	0.132 (0.37)	<0.001 (0.96)	<0.001 (0.96)	0.002 (0.74)	<0.001 (0.96)	<0.001 (0.94)

Table 4.5: RQ3: Results of the Wilcoxon test (p-value and r effect size) comparing the absolute errors obtained by LFM vs. those obtained by human experts and traditional automatic learners (i.e. CART, KNN, LP, RF) when predicting software projects' effort.

the absolute errors (see Figure 4.4) reveals that our proposed algorithm, LFM, not only outperforms RG, but it also performs better than all the other machine learning methods against which we compare it, on five (out of the six) datasets. That is, the absolute error values provided by LFM are lower than those provided by CART, KNN, LP and RF in 21 out of the 24 cases considered. These observations are confirmed by inferential statistical analysis, the results of which are presented in Table 4.5; the improvement of our algorithm over these four techniques is statistically significant and the effect size is very large ($r \geq 0.7$) in six of these comparisons, large ($r \geq 0.5$) in 13, medium ($r \geq 0.3$) in one and small ($r \geq 0.1$) in two of them. As for the remaining dataset (Medical), LFM performs statistically significantly better than CART (as shown by the statistical test), with effect size equal to 0.29.

Therefore, we can positively answer RQ3.2: The use of LBM allows us to obtain significantly more accurate estimates than traditional ML techniques in 88% of the cases.

RQ3.3 LFM vs. Human Expert Judgement: From the boxplots in Figure 4.4 we observe that LFM enhances the original human expert estimates for all the datasets by providing the lowest absolute errors. Results of the Wilcoxon test (see Table 4.5) reveal that it achieves statistically significant better estimations (all p-values being less than 0.035) on four out of the six datasets with two of them having a medium effect size ($r \geq 0.3$) and the other two having a small one ($r \geq 0.1$). For the other two datasets (i.e. KD and Telecom), LFM obtains lower absolute error values than human expert estimates (see Figure 4.4), however based on the results of the Wilcoxon tests we cannot reject the null hypothesis. This may be due to the

size of these datasets; both KD and Telecom are the two smallest datasets with 29 and 17 instances, respectively. The small number of instances makes it difficult for ML models to learn the characteristics and trends of the errors committed by human experts. Finally, we observe that the average relative errors across all datasets obtained by LFM are up to 33% lower than those resulting from the estimations made by the human expert[10].

These results show that LFM is not only better than alternative automated techniques, but that it also has an edge over purely human expertise alone. Therefore, we can positively answer RQ3.3: The use of LFM allows us to improve human expert estimates.

Based on the results above, we can conclude that:

> **Answer to RQ3:** LFM improves expert judgement on all datasets with improvements that are statistically significant on four out of the six datasets studied.

4.5 Conclusions and Future Work

In this chapter we showed that we can learn from past misestimations in order to improve future estimates; the Learning From Mistakes (LFM) approach.

We demonstrated the effectiveness of LFM with an empirical study involving human expert effort estimates (and related errors) from 402 industrial software projects developed by several different companies. Our results reveal that it is possible to predict the type, the severity, and the magnitude of the mistakes made by human experts when estimating software effort, and that we can successfully exploit this information to significantly improve future human expert-based estimates.

The LFM approach proposed in this thesis can be applied by following these three steps:

1. Maintain a database of projects for which both the effort estimated and the

[10]MMRE should not be used as the only indicator to compare prediction models as it can be misleading (see e.g. [256, 257]), we use it only to provide a notion of the error with respect to the actual effort.

one actually needed to realise the project have been recorded, together with the cost drivers characterising the project (e.g. functional size measures);

2. For any target project: use LFM to classify the type and severity of the errors human experts are likely to make when estimating the effort needed for target projects based on the information stored in the database (our findings from RQ1 show that this is possible); use LFM to quantify the estimate error for target projects based on the magnitude of human expert past misestimations (our findings from RQ2 show that this is possible);

3. Enhance the human expert effort estimates based on the past errors learned from Step 2 (our findings from RQ3 show that such an enhancement can be statistically significant as in the case of the projects we considered herein).

Our LFM approach provides two routes to support human experts. One of these is the more traditional, and widely-studied route of improved effort estimation, but the other is the less explored one of providing automatic feedback on the experts' own judgements, rather than seeking to second-guess them. For example, one can investigate whether using information gathered from past misestimations as a cost driver can enhance the accuracy of traditional automated predictive models. While, following the second route, one could investigate the way in which machine learners model human expert misestimates (e.g., do bias models exhibit similar bias trends as those observed in human predictions when estimating effort?) to further provide useful insights to the experts. It would also be interesting to investigate whether factors that relate to the team such as the number of developers involved in producing the system and their level of expertise are good proxies for understanding and predicting human expert errors. Another aspect to explore and which might strengthen the human-machine collaboration and increase the trust of companies in using predictive models would be to provide explanations of the estimates rather than using them as a black-box models. This can be achieved by applying Explainable AI (xAI) techniques.

Chapter 5

The Role of Machine Learning Libraries in Effort Estimation Studies

In the previous chapter, we presented a novel approach to support practitioners estimating the effort needed to realise a software project by relying on the use of different machine learners. While undertaking this study, just like many researchers, we had to decide on a third party open-source machine learning API to realise our empirical study. The choice was based on the fact that one of the algorithms required for an approach (namely, LP) was publicly available in R. However, several Machine Learning libraries have, since then, become freely available, and the choice of choosing one library over another to implement a study became less obvious. Given that there might be no preference in the use of one library over another when designing an empirical study, we became curious to check whether a *no preference of library use* would also signify a *no difference in results*. We questioned whether the differences stemming from using one library over another have been overlooked, implicitly assuming that using any of these libraries to run a certain machine learner would provide the user with the same or at least very similar results. The work presented in this chapter aims at raising awareness of the differences incurred when using different Machine Learning libraries for Software Development Effort Estimation, which is one of most widely studied and well-known SE prediction tasks

and for which differences in estimations can lead to great losses.

5.1 Introduction

There are several Machine Learning ML libraries that can be used for Software Effort Estimation studies. If it should turn out that library choice impacts study outcomes, then that choice would constitute a threat to scientific validity. However, if researchers follow rigorous reporting, the gravity of such threats might not pose long term problems. Reporting the library used facilitates replication, thereby tackling potential propagation of unsound scientific conclusions.

However, in cases where library choice has an impact and yet there is *no* such rigorous reporting, there is a potent threat to our body of knowledge. As this chapter reveals, there is such a fundamental threat in the case of software effort estimation: Library choice is highly impactful and not only is there a general lack of reporting, but a worrying declining trend in such reporting.

We first analyse the way in which the ML libraries have been used in previous SEE work. We conduct a manual examination of the literature, examining 256 articles on ML for SEE, which were included in two previous surveys [5, 152]. We then carry out an empirical study that aims to compare the use of the same set of ML techniques provided by three popular ML libraries (i.e., SCIKIT-LEARN, CARET and WEKA) in the most common usage scenarios in SEE, as identified by our literature review.

The first scenario depicts the use of machine learners out-of-the-box, where we investigate the extent to which building predictive models using the library API as-is (i.e., with default settings) would yield different results when using different libraries (i.e., *out-of-the-box-ml* scenario). Whereas the second scenario analyses the use of machine learners with hyper-parameter tuning (i.e., *tuned-ml* scenario) when using different libraries. Finally, we manually analyse the API documentation and the source code of the three libraries under study in order to unveil further characteristics. Therefore, overall we study the current body of knowledge on software effort estimation from three complimentary angles: literature analysis, empirical

study, and API documentation. Each of these three angles combines to reinforce the conclusion that there is a replicability problem in the heart of the software effort estimation literature that can no longer be ignored:

Our *literature analysis* revealed that more than half of the studies do not report the ML library used (55%), and among those that do report it, the majority (34%) do not provide the complete description (i.e., no mention of the version used). Thus, suggesting that the choice of the ML library has not been regarded as an important design choice on par with other design choices such as the evaluation measure or the validation approach, which have been shown to affect the results of a study [258, 259].

Our *empirical study* revealed that the estimates provided by a given machine learner in each of the libraries under study largely differ (i.e., on average we observed differences in 95% of the cases across a total of 105 cases studied). This, coupled with the fact that most of previously published work do not state the ML library used in their study presents a worrying barrier to replicability. It also highlights the way in which the choice of the ML library may have contributed to the conclusion instability observed in previous studies [29, 260].

Our *analysis of the API documentation and code* reveals a lack of consistency among different libraries, which might not only lead to variance in the accuracy observed in our empirical study, but it might also induce users to misuse these APIs.

To summarise, the contributions of the work presented in this chapter are:

❶ Raising awareness on the importance of the choice of ML libraries, including analysing how this problem has been tackled so far.

❷ Carrying out an empirical study comparing four widely used SEE deterministic machine learners as provided by three popular open-source ML libraries on the five largest SEE datasets.

❸ Providing an analysis of the APIs of these libraries aiming at investigating possible reasons for/sources of the large differences in accuracy observed in the empirical study.

❹ Identifying initial suggestions for both developers and users of ML libraries and

highlighting open-challenges to be addressed in future work.

5.2 Research Questions

We first need to establish whether the literature has a reporting problem. If it should turn out that the ML library used by previous studies is typically well reported, then at least we could rest assured that replicability is not a problem.

RQ1. Current Literature: *How much importance has the current SEE literature given to the selection and reporting of the* ML *libraries in their work?*

If it should turn out that a significant fraction of the literature fails to report the ML library used, then there is a potential replicability problem. However, this problem will only have significant impact, should it also turn out that library choice is itself impactful. This observation motivates the next three research questions: We need to assess the degree to which library choice impacts the scientific conclusions drawn from studies of software effort estimation.

The first step towards identifying whether there are any discrepancies in prediction accuracy performance when using different ML libraries for SEE lies in recognizing the frequency of its occurrence, which motivates our second research question:

RQ2. Prediction Accuracy: *How often does a given machine learner provide different SEE results when built with different* ML *libraries?*

If we find that, in fact, the results differ in many cases, then it is important to verify the way in which the SEE prediction performance of a machine learner is affected depending on the ML library used. Thus our third research question assesses:

RQ3. Change in Prediction Performance: *How does the SEE performance of a given machine learner change when built using different* ML *libraries?*

In other words, while RQ2 aims to reveal whether the number of cases showing inconsistency in results due to the ML library used is high, RQ3 further analyses these results by looking into the magnitude of these differences to understand if they are relevant.

We aim to shed light on questions like "*Would my conclusion have been different had I used* SCIKIT-LEARN *instead of* CARET*?*", *"Would the results of my proposed approach have been better or worse, compared to others, had another* ML *library been used?"*. As previous studies usually rank machine learners based on their estimation performance for comparison purposes, our fourth research question asks:

RQ4. Change in Ranking: *How does the ranking of SEE machine learners change when considering different* ML *libraries?*

If we find that different rankings are provided by different ML libraries, this would point out possible conclusion instability threats in previous (and future) studies given that, for example, a technique that would rank first in a study using a given library, might not have the same rank (i.e., first) when another library is used instead.

Lastly, should it turn out that library choice impacts scientific conclusions (RQs 2–4), then we also wish to understand potential reasons for such differences by manually analysing their APIs:

RQ5. API Analysis: *What can we learn by analysing the API documentation and code of the* ML *libraries?*

5.3 Methodology

In this section, we describe the methodology followed to answer the research questions described in the previous section.

5.3.1 Collection of SEE Research Papers

We identified 256 relevant studies through a two step literature search. We elaborate on each of these phases in the rest of this section.

In order to answer RQ1 and understand the way in which the ML libraries have been used in previous empirical studies addressing SEE, we collected 159 studies surveyed in two literature reviews on SEE [5, 152], as well as additional 602 papers found by using one level forward snowballing of all papers citing these two literature reviews up until 2023. We carried out this additional literature search to augment the initial set of papers gathered from the two surveys as they included

work published up until 2017, thus studying the trend of more recent work and reducing the risk of missing relevant articles.

Wen et al. [5] investigated 84 primary studies of ML techniques in SEE published between 1991 and 2010, focusing their analysis on the type of ML technique, estimation accuracy, model comparison, and estimation context. Ali and Gravino [152] collected 75 primary studies of ML for SEE published between 1991 to 2017 and investigated the most common ML techniques, datasets and accuracy metrics used. Any manuscript that is listed in both surveys was removed to avoid any duplicates. This resulted in a total of 117 papers to be examined for our study.

The forward snowballing based on Google Scholar search conducted in August 2023 resulted in a total of 602 papers. To filter out irrelevant publications found during our search, we manually examined every publication using the process suggested by Martin et al. [261]. This process assesses whether publications satisfy the inclusion criteria in the following three stages:

1. **Title:** First, all those publications whose title clearly does not match our inclusion criteria are excluded;
2. **Abstract:** Second, the abstract of every remaining publications is checked. Publications whose abstract does not meet our inclusion criteria are excluded at this step;
3. **Body:** Publications that passed the previous two steps are then read in full, and excluded if their content does neither satisfy the inclusion criteria nor contribute to this survey.

To ensure that the articles included in this survey are relevant to the context of machine learning for effort estimation, at each of the steps we applied the following inclusion criteria:

Inclusion criteria

- The paper presents an approach, study, framework, or tool on the use of machine learning for software effort estimation.
- The paper carries out an empirical investigation which includes machine learning approaches applied to software development estimation.

Exclusion criteria

- Studies focusing on techniques other than machine learning to address software development estimation tasks.
- Articles that are not peer-reviewed (such as articles available only on arXiv.org) or articles submitted for the fulfilment of graduate or doctoral studies (i.e., theses).
- Studies whose full text is not available, or is written in any other language than English.

Based on the above three-stage process and inclusion criteria, we identified a total of 139 relevant publications among the 602 initially obtained from snowballing, as detailed below. After verifying the title and information provided in the abstract of the studies, we discarded 232 papers which did not satisfy the inclusion criteria. This resulted in a total of 370 articles which were deemed relevant for our work. 32 additional articles were excluded from our study because they were not peer-reviewed or published work, they were not written in English. We also found duplicate papers across both literature reviews, and articles that were not accessible or had been retracted, which accustomed for 12 studies. This resulted in a total of 326 articles which were inspected in full. After further inspection of the body of the articles, we excluded an additional 187 articles which had passed the first two stages of the analysis, however their content did not satisfy the inclusion criteria. We, thus, retained a final number of 139 relevant papers after the snowballing process.

A total of 256 relevant articles (117 from the two previous surveys and 139 from the forward snowballing) were manually examined in order to extract the following information: the year of publication, the venue, the validation approach used and whether the ML library and version used were reported. The list of papers and the details on the data extracted can be found in our online repository [262]. Our findings are discussed in Section 5.4.1.

5.3.2 Empirical Study Design

In order to answer RQs 2–4, we carried out an empirical study to assess the extent to which using machine learners provided by different ML libraries might yield

different results. We describe the design of this study in the following subsections, while its results are discussed in Sections 5.4.2, 5.4.3, 5.4.4.

5.3.2.1 Machine Learning Libraries

We compare three popular open-source ML libraries written in different languages: SCIKIT-LEARN [263] in PYTHON, CARET [264] in R, and WEKA [265] in Java.

SCIKIT-LEARN is a Python library, distributed under BSD license, which includes a wide range of state-of-the-art supervised and unsupervised ML techniques. The aim of SCIKIT-LEARN is to provide efficient and well-established ML libraries within a programming environment that is accessible to non-ML experts and reusable in various scientific areas [266].

SCIKIT-LEARN is designed to adhere to a number of engineering principles, including the use of sensible defaults stating "Whenever an operation requires a user-defined parameter, an appropriate default value is defined by the library. The default value should cause the operation to be performed in a sensible way."

CARET, acronym for Classification and Regression Training, is an `R` package, distributed under the GNU General Public Licence, containing functions to streamline model training processes for complex regression and classification problems [264]. It was made publicly available on CRAN in 2007 and it relies on several other `R` packages that can be loaded as needed.

The aim of CARET is to provide the user with an easy interface for the execution of several classifiers, allowing automatic parameter tuning and reducing the requirements on the researcher's knowledge about the tunable parameter values [267].

WEKA, acronym for Waikato Environment for Knowledge Analysis, is licensed under the GNU General Public Licence and was first released in 1997 [265]. WEKA provides implementations of ML techniques that can be easily used through a simplified and interactive graphical interface by users who cannot or do not need to write code, or through an API that allows users to access the libraries from their own Java programs. The online documentation is automatically generated from the source code and concisely reflects its structure [268]. WEKA provides the main

Table 5.1: Machine learners investigated and corresponding class/method name in SCIKIT-LEARN, CARET and WEKA.

Machine Learner	Library	Class/Method Name
CART	Caret	rpart
	SkLearn	DecisionTreeRegressor
	Weka	REPTree
KNN	Caret	knn
	SkLearn	KNeighborsRegressor
	Weka	IBk
LR	Caret	lm
	SkLearn	LinearRegression
	Weka	SimpleLinear
SVR	Caret	svmRadial
	SkLearn	SVR
	Weka	SMOReg

methods for supervised and unsupervised ML techniques, as well as methods for data pre-processing and visualization.

We used the latest stable version for each library at the time the study was done: SCIKIT-LEARN 0.23.1, CARET 6.0.85, WEKA 3.8 with `Python` 3.7.5, `R 3.6.3`, and `Java 11`, respectively.

5.3.2.2 Machine Learners and Settings

In this section we briefly introduce the machine learners we investigated to compare the performance of the three ML libraries CARET, SCIKIT-LEARN and WEKA. We focus on a set of deterministic ML techniques to eliminate any randomness that could arise from the use of other learners of a stochastic nature, and thereby to analyse differences stemming *only* from the library usage. These techniques are Classification and Regression Tree (CART) [239], K-Nearest Neighbours (KNN) [240], Linear Regression (LR) [269] and Support Vector Regression (SVR) [270]. All of them have been widely used in the SEE literature. In Table 5.1, we list the machine learners together with their corresponding implementation per ML library.

To investigate the *out-of-the-box-ml* scenario, we use all the techniques with the default settings provided by each of the ML libraries, without any further modifications.

To investigate the *tuned-ml* scenario, we apply Grid Search as a hyper-

parameter tuning technique. We chose this technique over others because it is deterministic and thus it eliminates any difference in performance resulting for example from using techniques having stochastic behaviour (e.g., Search-Based ones). This allows us to accurately verify whether variances in performance are due in fact to the choice of ML libraries used, rather than being jeopardised by other design choices such as using Random Search for hyper-parameter tuning.

We run each library's corresponding grid search method (i.e., `GridSearchCV` in SCIKIT-LEARN, `tuneGrid` along with `trControl` in CARET and `GridSearch` in WEKA) 30 times using the same inner cross-validation across all libraries to eliminate any possible stochastic behaviour deriving from the data splits, and using the same set of values for the hyper-parameters across all three libraries. In order to maintain a fair comparison, we tune the parameters that are found in common across all three libraries. The settings used for both the *out-of-the-box-ml* and *tuned-ml* scenarios can be found in our online repository [262].

5.3.2.3 Datasets

To empirically investigate our RQs we used the largest SEE publicly available datasets (namely, China, Desharnais, Kitchenam, Maxwell, Miyazaki) containing a diverse sample of industrial software projects developed by a single company or several software companies [271]. These datasets exhibit a high degree of diversity both in terms of number of observations (from 48 to 499), number and type of features (from 3 to 17), technical characteristics (e.g., software projects developed in different programming languages and for different application domains), number of companies involved and their geographical locations. Furthermore, all these datasets have been widely used in numerous SEE studies (see e.g., [2, 272, 273, 274, 74, 219, 117, 275]). A comprehensive description of these datasets, together with the actual data is available in our online repository [262].

5.3.2.4 Validation

In order to eliminate any possible variance in performance caused by non-deterministic influencing factors such as the validation approach employed or the sampled data used as denoted by Rahman et al. [276], we perform a Leave-One-Out

cross-validation (LOO), where, as previously described in Chapter 2, each instance (i.e., a software project in our case) of a dataset of n observations is considered to be a fold.

LOO is a deterministic approach that, unlike other cross validation techniques, does not rely on any random selection to create the training and testing sets. Therefore, this validation process is fully reproducible and eliminates conclusion instability caused by random sampling [73, 275].

5.3.2.5 Evaluation Criteria

Several measures have been proposed to evaluate the accuracy of effort estimation prediction models. They are generally based on the *absolute error*, i.e., $|ActualEffort - EstimatedEffort|$. We use the *Mean Absolute Error* (MAE) as it is unbiased towards both over- and under-estimation [218, 277]. Given a set of N projects and the measured ($ActualEffort_i$) and estimated ($EstimatedEffort_i$) effort for each of the the project i in this set, MAE is defined as follows: $MAE = \frac{1}{N}\sum_{i=1}^{N}|ActualEffort_i - EstimatedEffort_i|$.

We also use statistical significance tests to assess differences in the performance of the ML libraries (RQ3) and the way they rank machine learners (RQ4). In order to evaluate whether the differences in MAE values resulting from the use of various ML libraries are statistically significant (RQ3), we perform the Wilcoxon Signed-Rank Test [278] at a confidence limit, α, of 0.05, with Bonferroni correction (α/K, where K is the number of hypotheses) when multiple hypothesis are tested. Unlike parametric tests, this test does not make any assumptions about underlying data distributions. The null hypothesis that is tested in our work follows: *"There is no significant difference in the MAE values obtained by the approaches when built using a different* ML *library"*.

We also compute the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure to assess whether any statistically significant difference is worthy of practical interest [223]. $\hat{A}_{12}$ is computed based on the following formula $\hat{A}_{12} = (R_1/m - (m+1)/2)/n$, where R_1 is the rank sum of the first data group we are comparing, and m and n are the number of observations in the first and second data

group, respectively. When the two compared groups are equivalent: $\hat{A}_{12} = 0.5$. An $\hat{A}_{12}$ higher than 0.5 denotes that the first data group is more likely to produce better results. The effect size is considered small when $0.5 < \hat{A}_{12} \leq 0.66$, medium when $0.66 < \hat{A}_{12} \leq 0.75$ and large when $\hat{A}_{12} > 0.75$, although these thresholds are not definitive [2].

In order to verify statistical significance in the ranking provided by the SCIKIT-LEARN, CARET and WEKA APIs (RQ4), we use the Friedman Test [245]. This is a non-parametric test which works with the ranks of the data groups rather than their actual performance values, making it less susceptible to the distribution of the performance of these parametric values. When a significant difference is found, the Nemenyi test [246] is often recommended as a post-hoc test to identify the data groups with a statistically significant difference [247]. The performance of two data groups is thought to be statistically significantly different if the corresponding average ranks differ by at least the Critical Distance (CD) [247]. The results of this test are presented in a diagram which is used to compare the performance of multiple techniques by ranking them based on each ML library. It consists of an axis, on which the average ranks of the methods are plotted and of the CD bar. The CD bars of the groups of classifiers whose values are significantly different do not overlap. The visualisation used herein is a more recent version of the Demsar's one [247], aiming at providing an easier interpretation. It can be obtained by using the `Nemenyi` function in R.

5.3.3 API Analysis

In order to answer RQ5, two of the authors independently investigated the API documentation and source code (function body) of CARET[1], SCIKIT-LEARN[2] and WEKA[3] for each of the techniques listed in Table 5.1 in order to find similarities and differences in (1) the algorithm reference for each machine learner; (2) the signature of the main method offered by these libraries to build each machine learner (i.e., the number and type of input parameters that can be set by the user through the use

[1] https://topepo.github.io/caret/
[2] https://scikit-learn.org/stable/modules/classes.html
[3] https://waikato.github.io/weka-wiki/documentation/

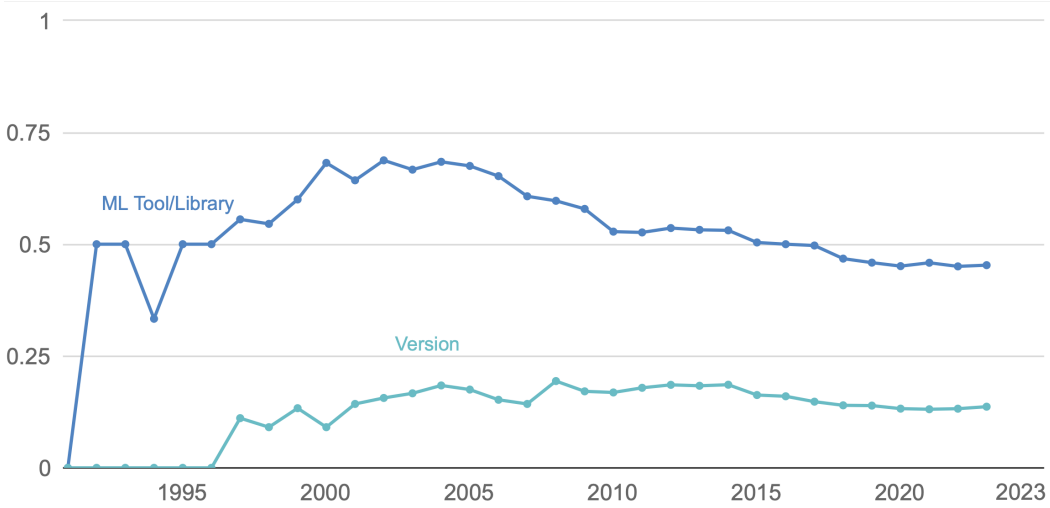


Figure 5.1: The ratio of studies reporting the ML libraries and versions over the years.

of their API) and the naming convention used; (3) the parameters' default values and the values that can be assigned for categorical parameters within each library; (4) the parameters only available in the source code (i.e., *hidden* parameters). We discuss our findings in Section 5.4.5.

5.4 Results

5.4.1 RQ1: Current Literature

Our analysis of 256 previous SEE work revealed that more than half of the studies (i.e., 55%) do not state the ML library used. Among the articles that report information about the ML libraries used in their study, we also analysed how many mention the version of the library, and found that only 35 (31%) articles report it.

We also investigated the ratio of studies reporting the ML libraries as well as the versions used over the years. As shown in Figure 5.1, the trend of such studies tends to decline, highlighting the alarming fact that not only is there a worryingly low level in reporting of the library and version used, but this practice has not become more popular, on the contrary it seems that it continues to be overlooked over the years.

> **Finding 1:** Details about an important factor in the empirical design of a study (i.e., ML libraries) are not provided in 55% of the studies inspected, creating additional barriers for the advancement of reproducibility in science. This trend does not seem to improve over the years.

5.4.2 RQ2: Prediction Results

In RQ2 we investigate how often a certain prediction model built using different ML libraries for SEE provides the same results. To answer this question, we look at the cases where ML libraries result in different predictions. As described in Section 5.3.2.2, we explore two common scenarios observed in the SEE literature, the use of out-of-the-box prediction models (i.e., *out-of-the-box-ml*) and their use when hyper-parameter tuning is performed (i.e., *tuned-ml*).

Tables 5.2a and 5.2b show the results for each scenario, respectively. As one can observe from Table 5.2a (i.e., *out-of-the-box-ml* scenario), there are only five out of the 60 cases under study (8%) where two out of the three ML libraries provide the same predictions. Specifically, LR achieves the same outcome when built using SCIKIT-LEARN and CARET, whereas WEKA obtains different results. Differences in prediction are observed for all other techniques when built using any of the three ML libraries.

As for the second scenario, we explore prediction models which are tune-able (i.e., they consist of parameters for which hyper-parameter tuning can be applied): CART, KNN and SVM.[4] Table 5.2b shows that there is a difference in prediction in all of the cases studied (i.e., there is no case where all three libraries agree on the same prediction).

> **Finding 2:** SCIKIT-LEARN, CARET and WEKA do not output matching results in 92% of the cases when the machine learners are used out-of-the-box. When tuning the models, each tool outputs a different result in each case considered.

[4]We exclude LR as there are no tune-able parameters common to all libraries

Table 5.2: RQ2-RQ3: Differences in MAE values obtained when building prediction models using SCIKIT-LEARN (Sk), CARET (C) and WEKA (W) in the (a) *out-of-the-box-ml* scenario and the (b) *tuned-ml* scenario. The ↑ indicates that the 1^{st} tool produces worse predictions than the 2^{nd}, whereas the ↓ indicates that 1^{st} tool produces better predictions than the 2^{nd}.

	CART			KNN			LR			SVM		
	Sk vs. C	Sk vs. W	C vs. W	Sk vs. C	Sk vs. W	C vs. W	Sk vs. C	Sk vs. W	C vs. W	Sk vs. C	Sk vs. W	C vs. W
China	↓643	↓477	↑166	↓99	↑734	↑833	0	↑420	↑420	↓359	↓637	↓278
Desharnais	↓396	↑172	↑568	↑111	↑972	↑861	0	↓226	↓226	↓440	↓577	↓137
Kitchenham	↓46	↓227	↓181	↓36	↑125	↑161	0	↑ 177	↑ 178	↓134	↓683	↓550
Maxwell	↓3079	↓1095	↑1983	↑193	↑2435	↑2242	0	↓450	↓450	↓1591	↓1319	↑272
Miyazaki	↑1652	↓578	↓2231	↑75	↑518	↑443	0	↑1229	↑1229	↓714	↓3188	↓2474

(a) *out-of-the-box-ml* scenario.

	CART			KNN			SVM		
	Sk vs. C	Sk vs. W	C vs. W	Sk vs. C	Sk vs. W	C vs. W	Sk vs. C	Sk vs. W	C vs. W
China	↓809	↓560	↑249	↑10	↓6	↓16	↓421	↓598	↓177
Desharnais	↓1,113	↓457	↑656	↓29	↑160	↑189	↓336	↓419	↓83
Kitchenham	↑185	↑247	↑62	↑14	↓25	↓39	↓250	↓237	↑14
Maxwell	↓2,515	↓652	↑1,864	↓421	↑668	↑1,089	↓1,044	↓1,738	↓694
Miyazaki	↑3,053	↑1,872	↓1,181	↓1,104	↓401	↑703	↓966	↓2,232	↓1,266

(b) *tuned-ml* scenario.

5.4.3 RQ3: Change in Prediction Performance

RQ3 investigates how the performance of a given SEE technique changes when the prediction model is built using different libraries.

Tables 5.2a and 5.2b show the difference in the MAE results for each technique obtained by comparing each pair of the ML libraries under study for the two scenarios analysed. We denote any positive difference by the up arrow, whereas any negative difference is represented by the down arrow. For each pair of tools, a positive difference signifies that the ML library listed as first achieves a higher MAE value (i.e., a less accurate prediction) than the other library. On the other hand, a negative difference denotes that the first library provides an MAE value lower than the one provided by the second library (i.e., it provides a better prediction). For example, when considering CART, specifically comparing its MAE when it is built using SCIKIT-LEARN and CARET (i.e., denoted as *Sk vs. C*), the difference in MAE is ↑ 809 meaning that when CART is built using SCIKIT-LEARN, it achieves a higher MAE (i.e., worse prediction performance) than that obtained when CART is built using CARET with a difference of 809 man-hours.

To study the magnitude of the error between the ML libraries, we analyse the results using three error ranges: a difference of 100, 500 and 1,000 hours. By doing

so we relax our constraint on what we consider the difference to be *impactful* by accepting a larger error between the results obtained by the libraries.

Table 5.3 presents the number of times the difference in results falls within these three ranges. When considering the *out-of-the-box-ml* prediction scenario, we can observe that while SCIKIT-LEARN and CARET provide the same MAE for LR only, they, along with WEKA achieve different results in all other cases (i.e., 51 out of the 60) with a difference of at least 100 hours.

When analysing the MAEs using a threshold $\geq$ 100 hours, results show that SCIKIT-LEARN and CARET are the least discordant pair with still more than half the cases (55%) having a difference of at least 100 hours. When comparing SCIKIT-LEARN with WEKA and CARET with WEKA, a difference $\geq$ 100 is seen in all the cases considered.

We also found cases with a difference of at least 500 hours. Specifically, between SCIKIT-LEARN and CARET there is a difference $\geq$ 500 hours in 25% of the cases (5 out of 20), whereas between CARET and WEKA it exists in 45% of the cases and between SCIKIT-LEARN and WEKA, it is in 60% of the cases (12 out of 20).

One would not expect to obtain a difference of such a high magnitude resulting from the use of one ML library over another, however these cases exist even with a difference $\geq$ 1,000 hours. As we can observe from Table 5.3, when comparing the results obtained by SCIKIT-LEARN with those by CARET, there is a difference of at least 1,000 hours in 15% of the cases under study. As for the comparison between the other two pairs of libraries (i.e., SCIKIT-LEARN vs. WEKA and CARET vs. WEKA), the difference is seen in 25% of the cases.

> **Finding 3:** In the out-of-the-box scenario, on average, across each pair of libraries, we observe a difference of at least 100 hours in 85% of the cases, at least 500 hours in 43%, and at least 1,000 hours in 22% cases.

When considering the *tuned-ml* scenario, we observe that differences occur more frequently than in the *out-of-the-box-ml* scenario. We found a difference of at

Table 5.3: RQ3: Number of cases the results provided by a given ML library differ from another of at least 100 hours, 500 hours and 1,000 hours for the *out-of-the-box-ml* scenario and the *tuned-ml* scenario.

	out-of-the-box-ml					*tuned–ml*			
Prediction Difference	CART	KNN	LR	SVM	Overall	CART	KNN	LR	Overall
	Sklearn vs. Caret					Sklearn vs. Caret			
$\geq$ *100 h*	4	2	0	5	11	5	2	5	12
$\geq$ *500 h*	3	0	0	2	5	4	1	2	7
$\geq$ *1,000 h*	2	0	0	1	3	3	1	1	5
	Sklearn vs. Weka					Sklearn vs. Weka			
$\geq$ *100 h*	5	5	5	5	20	5	3	5	13
$\geq$ *500 h*	2	4	1	5	12	3	1	3	7
$\geq$ *1,000 h*	1	1	1	2	5	1	0	2	3
	Caret vs. Weka					Caret vs. Weka			
$\geq$ *100 h*	5	5	5	5	20	4	3	3	10
$\geq$ *500 h*	3	3	1	2	9	3	2	2	7
$\geq$ *1,000 h*	2	1	1	1	5	2	1	1	4

least 100 hours in 80% of the cases (12 out of 15) for SCIKIT-LEARN vs. CARET, in 87% of the cases (13 out of 15) for SCIKIT-LEARN vs. WEKA and in 67% of the cases (10 out of 15) for CARET vs. WEKA (Table 5.3). Several differences still persist when we consider a difference of at least 500 hours: specifically in 47% of the cases (7 out of 15) when comparing each pair of libraries (i.e., SCIKIT-LEARN vs. CARET, SCIKIT-LEARN vs. WEKA and CARET vs. WEKA). By considering any difference less than a 1,000 hours to be negligible, we observe that differences $\geq$ 1,000 still exist in 33% of the cases for SCIKIT-LEARN vs. CARET, in 20% of the cases for SCIKIT-LEARN vs. WEKA, and in 27% for CARET vs. WEKA. Such magnitude in difference would not only have an impact on a project's feasibility and completeness but can also change the conclusion of any study proposing or comparing a new or existing techniques as we further investigate in RQ4.

> **Finding 4:** In the tuned-ml scenario, on average, across each pair of libraries, we observe a difference of at least 100 hours in 78% of the cases, at least 500 hours in 47%, and at least 1,000 hours in 27% of the cases.

The statistical significance test results show that when comparing SCIKIT-LEARN and CARET, the differences in MAE values still prove to be statistically

significantly different (p-value < 0.01 and A12 = 0.55) despite the low statistical power due to the small sample size being tested. The same observation could not be made when comparing SCIKIT-LEARN and WEKA (p-value = 0.164), and CARET and WEKA (p-value = 0.310) even though the MAE results reported in Table 5.5 show a large variance between the three ML libraries (for three out of the four techniques investigated). This could be due to the small sample size which can result in a *Type II error*, where the test fails to reject the null hypothesis, even though it should not be the case.

In order to further understand the severity of the error, we also compute the relative deviation in effort estimation with respect to the median actual effort of the projects in a given dataset. For example, if the median actual effort spent for realising projects in a company is 10,000 hours, then a deviation $\leq$ 1,000 hours will not be as severe as the same deviation when dealing with projects with a median actual effort of 2,000 hours. We observe that for the *out-of-the-box-ml* scenario, the average deviation ranges from 11% to 24% depending on the dataset, with three out of the five datasets having a deviation above 20% and with the largest differences between ML libraries ranging between 27% and 59% of projects' allocated hours (i.e., almost two thirds of the projects' actual effort). For example, we can observe a difference of 3,188 hours in the results achieved by SCIKIT-LEARN and WEKA on the Miyazaki dataset, which corresponds to a 52% deviation in effort estimation (i.e., more than half of the project's actual allocated time) with respect to the median actual effort of the projects in this dataset. As for the *tuned-ml* scenario, we observe that the average deviation ranges between 8% and 23%, with three out of the five datasets (namely China, Maxwell and Miyazaki) having a deviation greater than 15%. In the worst case the deviation ranges between 16% and 50% with four out of the five datasets reaching over 30% deviation.

> **Finding 5:** The deviations in effort estimation among ML libraries, with respect to the actual effort, range on average between 11% and 24% (*out-of-the-box-ml* scenario), and between 8% and 23% (*tuned-ml* scenario). In the worst case it can

Table 5.4: RQ4: Rankings of prediction models based on the MAE results obtained by each of the ML libraries for each of the five datasets for the (a) *out-of-the-box-ml* scenario and the (b) *tuned-ml* scenario.

China			Desharnais			Kitchenham			Maxwell			Miyazaki		
SkLearn	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**
LR	LR	SVM	KNN	LR	LR	LR	LR	SVM	KNN	KNN	LR	KNN	KNN	SVM
KNN	KNN	LR	LR	KNN	SVM	KNN	KNN	LR	LR	SVM	SVM	SVM	SVM	LR
SVM	SVM	CART	SVM	SVM	CART	SVM	SVM	KNN	SVM	LR	KNN	LR	LR	CART
CART	CART	KNN	CART	CART	KNN	CART	CART	CART	CART	CART	CART	CART	CART	KNN

(a) *out-of-the-box-ml* scenario.

China			Desharnais			Kitchenham			Maxwell			Miyazaki		
SkLearn	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**	**SkLearn**	**Caret**	**Weka**
KNN	KNN	SVM	KNN	KNN	SVM	KNN	KNN	KNN	KNN	KNN	SVM	KNN	KNN	SVM
SVM	SVM	KNN	SVM	CART	KNN	SVM	SVM	SVM	SVM	SVM	KNN	SVM	SVM	KNN
CART	CART	CART	CART	SVM	CART	CART	CART	CART	CART	CART	CART	CART	CART	CART

(b) *tuned-ml* scenario.

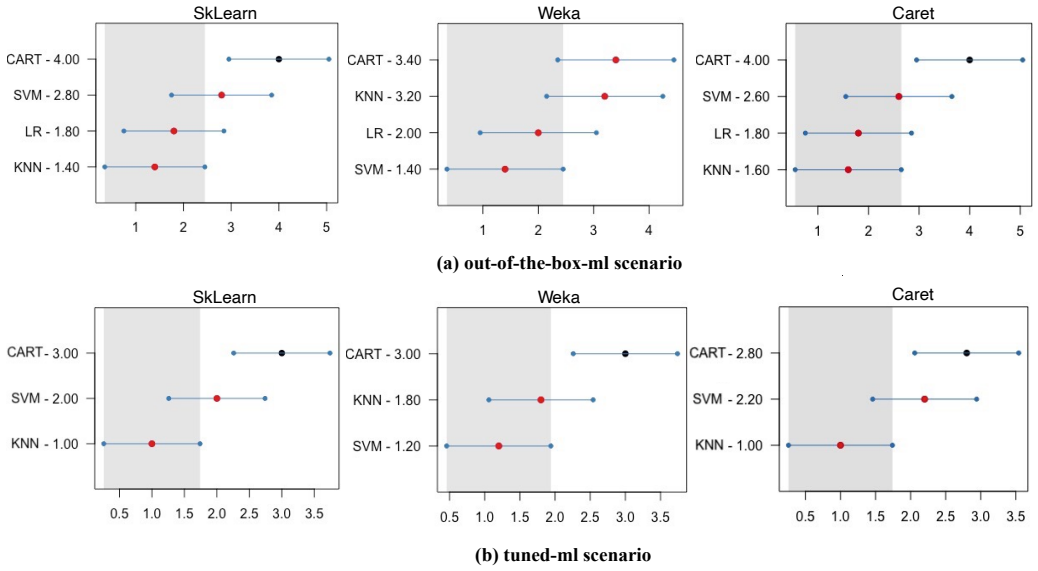


Figure 5.2: RQ4: Ranking of the ML techniques based on the results of the Nemenyi Test for the (a) *out-of-the-box-ml* scenario and the (b) *tuned-ml* scenario. The worst performing technique (with the highest MAE) is displayed at the top.

go up to half the time allocated for a project for both scenarios.

5.4.4 RQ4: Change in Ranking

To further understand the magnitude of the difference and to investigate whether it can have an impact on a study's conclusion, we analyse the ranking of the techniques, based on the MAE achieved, according to each library separately and assess the differences.

The results for *out-of-the-box-ml* scenario, presented in Tables 5.4a and 5.5a,

reveal that there is no case where all three techniques agree on a single ranking. The rankings obtained when using WEKA are different from those obtained by SCIKIT-LEARN and CARET for each of the datasets investigated. Whereas, SCIKIT-LEARN and CARET provide the same ranking for three out of the five (60%) datasets under study (i.e., China, Kitchenham, Miyazaki). As for the *tuned-ml* scenario, we can observe that all three tools provide the same ranking on only one dataset (i.e., Kitchenham). Whereas for the remaining four, CARET and SCIKIT-LEARN agree on three datasets (i.e., China, Maxwell, Miyazaki), whereas all three libraries disagree on the fourth (i.e., Desharnais). While SCIKIT-LEARN and CARET seem to generally agree at least on some of the ranks, WEKA tends to provide completely different rankings. These observations are confirmed by the statistical significance test analysis: The Friedman test shows that the difference between the results of the techniques is statistically significant (p-value < 0.05) for all ML libraries in both the *out-of-the-box-ml* and *tuned-ml* scenarios. The post-hoc Nemenyi Test (see Figure 5.2) reveals that, for the *out-of-the-box-ml* scenario, WEKA provides a different ranking than that obtained by SCIKIT-LEARN and CARET, with a statistical significant difference between the best and worst performing techniques (i.e., KNN and CART, respectively) in the case of SCIKIT-LEARN and CARET. While SCIKIT-LEARN and CARET rank KNN first and SVM third, WEKA ranks KNN in third place and SVM in the first in the *out-of-the-box-ml* scenario. As for the *tuned-ml* scenario, we can observe that WEKA also provides a different ranking than the other two ML libraries (i.e., SCIKIT-LEARN and CARET) as it ranks SVM and KNN first and second, respectively. Whereas SCIKIT-LEARN and CARET both agree on ranking KNN first and SVM second. All libraries show a statistically significant difference between the best and worst ranked techniques (i.e., KNN and CART for SCIKIT-LEARN and CARET and SVM and CART for WEKA). While, in some cases, SCIKIT-LEARN and CARET agree on the ranking, the results of RQ3 had revealed that the MAE values achieved by these two ML libraries are statistically significantly different. This shows that even when they output the same rankings of ML techniques, the results of the same techniques vary significantly depending

Table 5.5: RQ2-3: MAE results obtained by each of the ML libraries for each of the five datasets for the (a) *out-of-the-box-ml* scenario and the (b) *tuned-ml* scenario.

Dataset	CART			KNN			LR			SVM		
	SkLearn	Caret	Weka	SkLearn	Caret	Weka	SkLearn	Caret	Weka	SkLearn	Caret	Weka
China	3,606.42	2,963.80	3,129.69	2,798.93	2,699.45	3,532.56	2,647.40	2,647.56	3,067.63	3,108.56	2,749.33	2,471.66
Desharnais	2,906.33	2,510.66	3,078.18	2,199.65	2,310.67	3,171.84	2,277.26	2,277.23	2,051.05	2,754.36	2,314.77	2,177.32
Kitchenham	2,711.59	2,665.82	2,484.42	2,069.70	2,033.88	2,194.41	1,645.08	1,644.95	1,822.51	2,279.89	2,146.26	1,596.76
Maxwell	7,564.52	4,485.85	6,469.05	3,850.34	4,043.83	6,285.81	4,448.94	4,449.08	3,999.02	5,741.87	4,150.91	4,423.29
Miyazaki	12,058.73	13,710.94	11,480.30	8,644.25	8,718.85	9,162.33	11,700.77	11,700.95	12,930.09	10,346.04	9,632.12	7,158.33

(a) *out-of-the-box-ml* scenario

Dataset	CART			KNN			SVM		
	SkLearn	Caret	Weka	SkLearn	Caret	Weka	SkLearn	Caret	Weka
China	3,681.04	2,872.11	3,120.91	2,587.57	2,597.50	2,581.95	3,117.41	2,696.23	2,519.55
Desharnais	3,535.04	2,422.54	3,078.18	2,249.66	2,220.29	2,409.41	2,760.77	2,424.63	2,342.04
Kitchenham	2,478.44	2,663.92	2,725.83	1,985.66	1,999.46	1,960.95	2,282.58	2,032.15	2,045.81
Maxwell	7,232.53	4,717.38	6,580.89	4,285.81	3,864.54	4,953.90	5,743.15	4,699.31	4,005.16
Miyazaki	10,497.33	13,550.06	12,368.94	9,840.04	8,736.13	9,439.13	10,347.54	9,382.00	8,115.88

(b) *tuned-ml* scenario

on the library used, and therefore larger estimation errors can be committed when using one library over another.

> **Finding 6:** When comparing all three libraries together, there is no case where the rankings of ML techniques match. The statistical test results show a significant difference between the results of the techniques.

5.4.5 RQ5: API Analysis

We investigate various aspects of the API documentation and source code of the ML libraries in order to study the ways in which they are similar or different. Specifically we analyse (1) the algorithm reference for each machine learner; (2) the signature of the main method used to build each machine learner and the naming convention used; (3) the parameters' default values and the set of values that can be assigned for categorical parameters within each library and (4) the hidden parameters. A summary of the results is reported in Table 5.6, a detailed discussion follows.

Algorithm Reference. We observed that the documentation of these libraries does not always clearly state a reference algorithm. Specifically, no reference was provided in seven out of the 12 cases we looked into (with CARET never providing any reference for any of the algorithms). We also found that, among the cases where references are given, SCIKIT-LEARN usually provides a general list of references for a given algorithm. For example, it provides four references for CART (including

Wikipedia), without indicating the one actually being used for the implementation. Whereas among the cases where WEKA provides a reference, it is a single one.

> **Finding 7:** No reference was provided for the algorithms in more than half of the cases analysed.

Method Signature. We found that none of the libraries offer the same method signature to build each of the machine learners investigated herein (i.e., CART, KNN, LR and SVR). All three libraries provide a different *number of parameters* which can be directly manipulated by the user through the API as shown in Table 5.6. SCIKIT-LEARN provides the user with the highest number of parameters compared to CARET and WEKA across all techniques. For example, in SCIKIT-LEARN, a user can build CART by specifying 14 input parameters, while CARET and WEKA only allow the user to specify nine and six parameters, respectively. Similarly, a user can build KNN by specifying eight input parameters in SCIKIT-LEARN, while CARET and WEKA only provide the user with the ability to specify one and five parameters, respectively.

Among the parameters which are provided in an API but not in another, some do not directly have an effect on the accuracy of the ML techniques, but they play a role in the execution process (e.g., *n_jobs* in KNN controls the number of parallel jobs to run for neighbor search); whereas others control the machine learner hyper-parameters, therefore their use/setting can directly impact the accuracy of the models. For example, while CARET only allows the user to set the number of neighbours to be explored for KNN, WEKA permits setting the number of neighbours and the weight function, and SCIKIT-LEARN provides the user with the freedom to choose the number of nearest neighbours, the weight function of the neighbours, the algorithm used to compute the nearest neighbours, and the distance function. On average, across all four techniques, SCIKIT-LEARN provides the users with the ability to manipulate almost two times the number of parameters provided by CARET and three times those made available by WEKA. Overall, we can state that SCIKIT-

LEARN gives the user more control over the parameters and, therefore, the performance of these techniques.

We also observe that the number of *common parameters that perform the same functionality* across the three libraries is very low. The highest number of common parameters is only two (i.e., both CART and SVR have two common parameters, while KNN has only one and LR does not have any in common across the three libraries). If we consider each pair of libraries, we observe that SCIKIT-LEARN and WEKA have more parameters in common with respect to CARET, yet the numbers remain relatively low with the best case being six parameters in common between SCIKIT-LEARN and WEKA out of the 11 available ones for SVR. Specifically, in two cases out of 12 there is no common parameters between each pair of libraries. There is only one parameter in common in three cases, two common parameters in two cases, three common parameters in three cases. One case has four parameters in common and the remaining one has six common parameters as described above.

> **Finding 8:** The number of parameters that a user can access through the algorithm's method signature varies drastically between one ML library and another.

Even in the case where a method signature has a given parameter in common among these libraries, the parameter name, its default value, or possible categorical values, might differ across the libraries as further explained below.

Parameter Name. The libraries refer to a given parameter with different names, except for one case where all three ML libraries use the same name. This makes it non-trivial for a user to match concepts across different libraries. For example, the parameter used to specify the distance function for the KNN model is called `metric` in SCIKIT-LEARN, while in WEKA it is named `-A` and in CARET it is not even provided. Investigating this matter between each pair of libraries, we found that in only one case, out of a total of 12, two parameters have the same name, five cases had a single parameter with a common name, while all remaining cases (i.e., six cases) did not share any parameters using the same naming convention.

> **Finding 8:** The highest number of matching parameter name for an algorithm across the libraries does not exceed two parameters.

Parameter Value. We also inspected both the documentation and source code in order to extract the number of parameters which are set to the same default values in SCIKIT-LEARN, WEKA and CARET. This investigation revealed that no parameter has the same default value across all three libraries. We also compared the parameters' default values for each pair of libraries. We found that there were five cases where only one parameter had the same default value, while all the remaining cases (i.e., seven cases) did not have any parameters with the same default value. For example, SCIKIT-LEARN and WEKA set a different default value in KNN for the number of nearest neighbours to be explored by the algorithm. SCIKIT-LEARN gives its `n_neighbors` parameter a value of five, while WEKA sets its `K` parameter to one. Lastly, we observe that even if a given parameter is common among these libraries, they might provide the users with different value options to set as categorical ones. For example, when building KNN, WEKA allows the user to weigh neighbours by the inverse of their distance or by calculating $1-$ their distance. On the other hand, SCIKIT-LEARN allows the user to choose the value of this parameter among three different options: using uniform weights, using the inverse of their distance, or by creating a user-defined function which allows the user to assign specific weights.

> **Finding 9:** The ML libraries do not have any default parameter value in common. They also differ on the level of freedom they allow the users in choosing the value options of the parameters.

Hidden Parameters. A closer look at the implementation of these libraries revealed that some parameters that are made available to the user through API methods by one library are instead buried (i.e., *hidden*) in the source code of another library. By inspecting the source code, we found that each of WEKA and CARET does not directly expose to the user a number of parameters (i.e., they have hidden parameters)

that could instead be configured through the API of the other two libraries. Specifically, CARET has one hidden parameter in CART and four hidden ones in SVM, while WEKA has one in CART. SCIKIT-LEARN is the only library which does not have any hidden parameters that could have been matched with those made available by WEKA's and CARET's APIs. We also discovered that some of the hidden parameters in a given library are set to values which are different from the default values set in the API of another library, and such values cannot be changed unless one modifies the source code. For example, SCIKIT-LEARN and WEKA provide a parameter which defines the tolerance for the stopping criterion when building the SVR model, and both libraries set the default value to 0.001. However, this parameter is not provided by CARET's API, instead it can only be found by investigating the code, with its value being set to 0.01 in the called function's body.

Finding 10: Both CARET and WEKA do not expose all the parameters through the API method and documentation. The values of such *hidden parameters* often differ across libraries, and cannot be changed without modifying the source code.

Table 5.6: RQ5. Number of total parameters per ML library and machine learner, and number of parameters matching the same functionality, name, and default value across all and each pair of libraries.

	CART	*KNN*	*LR*	*SVM*
Total number of parameters				
Caret	9	1	1	3
SkLearn	14	8	4	11
Weka	6	5	0	10
Parameters matching functionality across				
All libraries	2	1	0	2
SkLearn & Weka	3	3	0	6
SkLearn & Caret	4	1	1	3
Caret & Weka	2	1	0	2
Parameters matching name across				
All libraries	0	0	0	1
SkLearn & Weka	0	0	0	1
SkLearn & Caret	1	0	1	2
Caret & Weka	0	1	0	1
Parameters matching default values across				
All libraries	0	0	0	0
SkLearn & Weka	0	1	0	1
SkLearn & Caret	1	0	1	1
Caret & Weka	0	0	0	0

5.5 Actionable Conclusions for Software Engineering Researchers

Essential Reporting: Our findings highlight that the choice of library is just as critical as the choice of a study's prediction techniques, benchmarks, validation procedures and evaluation measures [258], and as such researchers must include a description of the library used (including the version of the library) in their studies. In order to facilitate standard replication approaches, and to ensure the scientific validity of conclusions drawn, it is essential to reverse the declining reporting trend revealed in our study.

Stronger Conclusion Validity: As our findings reveal, the choice of tool and version, although theoretically not influential, can have an influence on a study's outcome. Given this potential impact on conclusion instability observed in the SEE literature, researchers should aim to experiment with multiple libraries when practical, otherwise report it as a threat. This should become one approach to tackle potential threats to conclusion validity.

Implications for the Wider Software Engineering Community: Software engineering research has been increasingly relying on artificial intelligence techniques in the past decade [279]. The impact on conclusion validity revealed by our study in the case of software effort estimation may well impact other applications of software engineering; although we have not studied this, there is nothing in our results to suggest that this might not be the case. It is therefore important for those working in other fields of study to conduct similar studies on the impact of library choice on their software engineering domains. Without this line of work, we can no longer rely on the conclusion validity of results from any Software Engineering domain that rests upon a single ML library. Furthermore, if a similar lack of reporting rigor is revealed, then these other SE domains will also have, not only threats to scientific conclusions, but also similar replicability problems.

Implications for ML Library Builders: Moreover, our *analysis of the API* documentation and code suggests that software engineers building open-source ML libraries should follow a more uniform approach by providing a reference to the

conceptual technique implemented by a given API; exposing the right number and type of parameters needed to build a given machine learner, otherwise explain any differences; testing the implementation of a given conceptual technique which uses other existing implementations of the same technique as an oracle [167, 166, 280].

5.6 Threats to Validity

Internal validity: To mitigate the threat of missing relevant information in our literature review (RQ1) as well as API manual analysis (RQ5), two authors examined all artefacts (i.e., papers, documentation and source code) independently, in order to ensure reliability and reduce researcher bias. The results were compared at the end of the process, and any inconsistencies were resolved by a joint analysis and discussion. Specifically, in order to minimise the probability of excluding a relevant article, two authors individually inspected a random selection of articles amounting for 15% of the total excluded papers, and had agreed on the exclusion of all papers inspected.

Moreover, we provide a detailed description about the methodology we followed and additional data in our online repository [262], so that our process can be reproducible, replicable and extendable.

The experimental setting used to answer RQs2–4 has been mainly dictated by the goal of soundly analysing the variance in performance solely due to the use of the different ML libraries investigated. As a result and to assure the validity of our work, we reverted to designing our own experiments by eliminating random inducing factors, as opposed to replicating previous work where the aim was not focused on eliminating these factors. While we acknowledge, that the use of other experimental settings could improve the overall prediction performance (e.g., using search-based approaches for hyper-parameter tuning [281, 237]), we explicitly avoid design choices that could introduce any stochasticity in the results. Nonetheless, we designed our empirical study in such a way that portrays comprehensive and widely used scenarios.

Construct and Conclusion Validity: We follow most recent best practice to

evaluate and compare prediction systems [218]. We use the MAE as a measure to evaluate and compare the predictions. The MAE is unbiased towards both over- and under-estimation and its use has been recommended [218, 277, 219] as opposed to other popular measures like MMRE and Pred(25) [282], which have been criticised for being biased towards underestimations and for behaving very differently when comparing prediction models [283, 284, 285, 286, 287, 288]. We also carefully calculated the performance measures and applied statistical tests by verifying all the required assumptions. We experimented with real-world datasets widely used to empirically evaluate SEE models, and to ensure a realistic scenario, we did not use any independent variable that is not known at prediction time and therefore cannot be used for prediction purposes [219].

External validity: Threats related to the generalizability of our findings may arise due to the open-source libraries, techniques and datasets we investigated. We have mitigated these threats by using those that are as representative as possible of the SEE literature, as well as by making our data and scripts publicly available [262] so that future studies could reproduce, replicate, and extend our work.

5.7 Conclusions and Future Work

We investigated and compared the use of three open-source ML libraries (CARET, SCIKIT-LEARN and WEKA) to build SEE prediction models with well-know machine learners and scenarios most commonly used in the literature.

Our results shed light on the fact that ML library users should consider the choice of ML library as part of their empirical design, similar to the choice of evaluation and validation criteria. We encourage users to justify their use of libraries which would allow for replicability and would motivate them to seek a deeper understanding of the ML libraries and algorithms. On the other hand, for this to be more achievable, the libraries' documentation should be improved to include more information about the algorithms implemented and their references. The developers of these libraries should provide a level of clarity in the documentation that would be comprehensible by all users regardless of their level of expertise.

The deficiencies we highlighted in the current API documentation, provide initial evidence of the need for further analysis of the existing APIs to derive a set of standard requirements for ML API documentation and API construction itself, as done, for example, in previous work for the documentation of Computer Vision Software [165]. Future studies can investigate the use of refactoring techniques to automatically recommend to developers suitable variable renaming in order to increase the consistency across different libraries. We envisage that the use of automated deep-parameter tuning [289, 290, 291] can aid to automatically improve the performance of prediction models built using those ML libraries that do not expose as many parameters in their APIs. Last but not least, future work can assess the extent to which discordant results arise when using proprietary ML libraries, as well as the impact of using different libraries for other ML-based software engineering tasks.

Data Availability

We make the data and the source code of our study publicly available to allow for replication and extension [262].

Chapter 6

On the Use of Evaluation Measures for Defect Prediction Studies

In the previous chapter, we assessed how the choice of the machine learning tool used to build effort prediction models can influence the accuracy of the final estimation, and therefore the results of a study. We also wanted to identify other factors that have been overlooked yet can influence the results of an empirical study assessing software prediction models. While the current literature in software effort estimation mostly follow best practices, software defect prediction studies have faced a few challenges in implementing robust measures. Specifically, the way in which the performance of prediction models is assessed is still a concern for software defect prediction, despite the various warnings have been previously raised. In this chapter, we further stress on the importance of the choice of appropriate measures in order to correctly assess strengths and weaknesses of a given defect prediction model.We also unveil the magnitude of the impact of assessing popular defect prediction models with several evaluation measures based, for the first time, on both statistical significance test and effect size analyses.

6.1 Introduction

Software bugs are costly. The most common cost of bugs is poor user experience, which causes software abandonment. For example, one of the most prominent reasons mobile users delete an application, often after a single use, is due to the app

crashing [292]. In the worst-case scenario, their cost can be life-threatening as in the case of the IT glitch of the UK National Health Service which put 10,000 patients at risk of being given the wrong medication in 2018 [76]. The earlier a bug is found and fixed, the less it costs.

Software defect prediction research aims to support engineers in identifying defective components, early in the development process. An ideal prediction model is one able to unveil as many defects as possible without raising false alarms (i.e., flagging clean components as defective).

However, as shown by Zhang et al. [293], achieving this in practise is challenging as optimising for one aspect often compromises the other (especially for problems such as the ones with large negative vs. positive ratios). On the other hand, defect prediction models with high detect capabilities and high false alarm, or vice-versa models with low detective capabilities but high precision, can still be considered effective depending on the business context [294, 295]. For example, a defect prediction model for safety critical software can be considered effective if it exhibits a high probability of detecting defects, even if it does so at the cost of generating a large number of false alarms. Similarly, a model that sacrifices certain detective capabilities in order to achieve a better precision can be desirable when there is, for example, a high cost in checking false alarms.

The use of appropriate evaluation measures guides practitioners and researchers to understand whether a given prediction model is fit for their purposes [23]. However, previous work has raised alarms about the way researchers have employed these measures to assess the effectiveness of the prediction models proposed in their work, especially in the presence of imbalanced data [23, 34, 35, 36, 37, 38]. The importance of adopting suitable measures has been often overlooked, thereby leading to discordant empirical results (i.e., conclusion instability) and hindering meta-analysis across different studies [296, 297, 298].

In this work we call on the community to reflect better on the selection of appropriate evaluation measures to support the scientific conclusions that are drawn on the effectiveness of defect prediction models.

To this end, we first analyse how researchers have selected and used these measures over the last decade by examining 111 defect prediction studies published between 2010 and 2020. We find that 59% of these studies do not properly motivate the use of their evaluation measures depending on the business context, and less than half acknowledge that their findings might change if the results are assessed by using a different evaluation measure. Furthermore, we find that, despite the warnings raised in previous studies [23, 34, 39], the use of some problematic measures has become more frequent with time. On the other hand, no growth in the adoption of more robust measures has been seen.

We also unveil and quantify the impact using different measures might have in practice by carrying out a comprehensive empirical study. Specifically, we investigate the use of six of the most widely used evaluation measures in literature to assess and compare the performance of seven popular defect prediction models in predicting defects for 15 different real-world software systems (for total of 24 datasets), under three different prediction scenarios.

We find that there is no case where all the measures agree on a same ranking of prediction models. Moreover, in 118 (83%) and in 122 (85%) out of the 144 cases analysed, the ranking produced by a given evaluation measure varies from the rankings produced by all the other evaluation measures, according to the Wilcoxon statistical significance test and the $\hat{A}_{12}$ the effect size measure, respectively. Besides, we find that assessing model performance based on a given measure would have changed the rank of a specific technique between 61% and 90% of the time, on average, depending on the measure used.

Overall, these results highlight the dramatic impact on the ability to draw meaningful conclusions across studies using different measures often not relevant or suitable to the business context.

We encourage researchers to select evaluation measures that fit the study's specific aim, model and data; as well as to include a more comprehensive and balanced measure to give an overall view of the performance of the proposed approach. This enables researchers and practitioners to assess and decide whether proposals made

in previous work can be applied for purposes different than the ones they were originally intended for. The rest of the chapter is organised as follows. We first provide the reader with some background on previous work discussing the matters arising from the use of different evaluation measures in defect prediction studies (Section 6.2), and with a comprehensive overview of the different metrics used in the literature (Section 6.3). The core contributions of our study are presented in Sections 6.4 to 6.6. In Section 6.4 we report our findings on the use of evaluation measures in 111 defect prediction studies published over the last decade. Whereas in Section 6.5 we describe the design of the empirical study we conducted and discuss its results in Section 6.6. Section 6.7 discusses possible threats to the validity of our study. Section 6.8 concludes this chapter and presents some recommendations for the selection of evaluation measures in future defect prediction studies.

6.2 Related Work

A few studies have highlighted possible differences resulting from the use of different evaluation measures, however no previous study has provided empirical evidence on the magnitude of such differences nor its statistical significance, and the effect it can potentially have on findings across various studies. In the following, we discuss these studies and highlight further differences with ours.

In 2008, Jiang et al. [39] provided a review of evaluation measures for defect prediction commonly used at that time, as well as an initial comparison of their use on the NASA data. They highlighted possible threats coming from the use of certain/different measures and suggested that evaluation measures should be carefully chosen and interpreted based on the specific needs of the project. Twelve years later, our analysis of the work published between 2010 and 2020 reveals that the threat has remained un-tackled by the community.

Moreover, our empirical study includes measures proposed more recently, as well as a more diverse and recent set of data and for the first time, the comparison is entirely based on statistical significant tests and the effect size measure.

Subsequently, Jingxiu and Shepperd [215] performed a meta-analysis of eight

papers on defect prediction in order to understand the differences resulting from the use of F-measure as opposed to MCC. They illustrated potential biases by using confusion matrices that portray different scenarios and found that the use of F-measure is problematic. However they did not quantify the differences resulting from the comparison, in fact as they state, their study "captures a change in direction of the effect, it does not, however, capture the magnitude of the effect"[215]). In our study, we specifically study the magnitude of the effect of using six different evaluation measures (including F-measure and MCC) based on both statistical and effect size analyses. This analysis is crucial to provide solid empirical evidence on whether the use of a measure over another significantly changes the way model performance is interpreted with respect to the business needs.

Other studies not directly targeting this issue yet highlighting that the problem exists are those by Arisholm et al. [34], Xuan et al. [299], and Hall et al. [23]. Arisholm et al. [34] investigated and compared the use of different classifiers and features to predict cross-release defects of an industrial legacy system. To this end, they used different evaluation measures (including Accuracy, Precision, Recall and ROC), and observed that what is considered the best model depends on the criteria that are used to evaluate and compare the models. Following the literature survey conducted by Hall et al. [23], Bowes et al. [300] proposed *DConfusion*, an approach used to re-compute the confusion matrix of studies provided that certain subsets of evaluation measures are present. While this is quite beneficial for studies to evaluate previously proposed approaches, we could not apply it in our work for two main reasons:

- In order to minimise any stochasticity in performance resulting from the use of different design choices, we opted to produce an empirical study where are evaluation measures are used in an identical empirical environment (i.e., same data, validation approach, data splits, etc.)

- During our investigation of the defect prediction literature between 2010 and 2020, we identified that more than 30% of the studies use only one evaluation measure, while a total of 71% of studies do not use more than three evaluation

measures, rendering it very difficult to use DConfusion in our case.

Xuan et al. [299] came to a similar conclusion by investigating the performance of several classifiers for within-project defect prediction in 10 open-source software systems, based on a large number of evaluation measures in order to find the best performing classifier. The comprehensive literature survey by Hall et al. [23] reviewed defect prediction studies published up to 2010. Despite the fact that the primary goal of their survey was not to investigate bias resulting from the use of different evaluation measures, they do observe that this is an issue and provide some guidelines to prevent it. As this study was published in 2012, one would have expected subsequent research to adopt/follow these guidelines, however, our analysis of the work published in the last decade shows that this has not been the case as we further articulate in Section 6.4.

6.3 A Hitchhiker's Guide to Defect Prediction Evaluation Measures

In this section we describe the most common binary classification evaluation measures, highlighting strengths and weaknesses of their use for defect prediction.

The formulae of these measures are reported in Table 6.1 and they originate from the confusion matrix (see Table 7.1). This matrix describes four types of instances: TP, defective modules correctly classified as defective; FP, non-defective modules wrongly classified as defective; FN, defective modules wrongly classified as non-defective; TN, non-defective modules correctly classified as non-defective.

The `Accuracy` measure has been one of the first measures used to assess defect prediction models performance, but it is nowadays widely recognised that this is a biased measure for defect prediction models and thus should not be used. The reason is that this measure is very sensitive to class imbalance and defect prediction data is very often imbalanced [39]. A simple trick to maximize accuracy when data is imbalanced is to always predict the instances as non-defective.

Subsequently, `Precision` and `Recall` and their harmonic mean, the `F-measure` (`F1`), have been adopted in numerous studies. These measures con-

Table 6.1: The definition of the measures.

Evaluation Measure	Definition
AUC	Area under the Receiver Operating Characteristic Curve
Recall (PD)	$\frac{TP}{TP+FN}$
Precision	$\frac{TP}{TP+FP}$
F1	$2 \times \frac{Precision \times PD}{Precision+PD}$
FPR (a.k.a PF)	$\frac{FP}{FP+TN}$
G-measure	$\frac{2 \times PD \times (1-PF)}{PD+(1-PF)}$
Balance	$1 - \frac{\sqrt{(0-FPR)^2+(1-PD)^2}}{\sqrt{2}}$
MCC	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
G-mean	$\sqrt{\text{PD} \times (1-\text{FPR})}$

sider the positive (i.e., defective) class as the only class of interest. Similarly, False Positive Rate (FPR), which is also known as Type I Error Rate, is also a single-focus measure and, as such, it explains only one aspect of a classifier.

While, this can be acceptable in some domains, like information retrieval, where the number of irrelevant documents not correctly retrieved is hard to quantify (i.e., it is essentially unbounded), it would not be acceptable in the medical domain, for example, where classifying a sick person as healthy is just as important as classifying a healthy person as sick. The latter can also be the case for defect prediction since the correct identification of negative classes (non-defective) becomes important when the cost of inspecting the components incorrectly classified as defective

Table 6.2: Confusion Matrix for Binary Classification.

	Predicted Value	
Actual Value	Defective	Non-Defective
Defective	True Positive (TP)	False Negative (FN)
Non-Defective	False Positive (FP)	True Negative (TN)

(i.e., false alarms) is high. These examples suggest that the choice of assessing a model with `Precision`, `Recall`, `F1` or (`FPR`) might vary according to the business needs.

The knowledge of the business domain can indeed guide in choosing the most appropriate way to evaluate whether a given prediction model is effective for the problem at hand. In other words, the relative importance assigned to precision and recall is an aspect of the problem. Weighted measures can be used to control such a delicate balance. An example is the `F`β measure, which can be used to control the balance of precision and recall by setting a β coefficient: $F\beta = ((1+\beta^2) * Precision * Recall)/(\beta^2 * Precision + Recall)$. However, determining meaningful weights is not trivial in practice [39].

Using only measures that give importance to only one class, might lead to biased evaluation when assessing prediction models in presence of highly imbalanced data [295]. In such a context, a good classifier is expected to produce high accuracy in detecting the defect class without significantly degrading the accuracy of detecting the non-defect class [38]. In other words, probability of detection (or Recall) and probability of false alarm (or FPR) are important performance metrics in the class imbalance context, and it is usually recommended to use balanced measures, such as `G-measure` and `G-mean2`, to properly assess the model's performance. These measures give equal importance to the positive and the negative class, i.e., they are formulated based taking into account both probability of detection and probability of false alarm, and thereby are able to show how much balance between two metrics is achieved overall.

Similarly, the `distance to heaven` (`d2h`) measure (a.k.a. `Balance = 1 - d2h`) computes the distance of `FPR` and `Recall` to the ideal (heaven) values of `FPR`=0 and `Recall`=1. Therefore, the smaller d2h (higher Balance), the better the performance of the classifier. Based on the definition of `d2h`, it is clear that a high `Recall` leads to a lower `d2h`; whereas a high `FPR` results in a higher `d2h`.

Ranking measures have also been used to assess prediction models. The most

popular one is the Area Under the ROC[1] Curve (AUC), which can be interpreted as the probability that a model ranks a random positive observation higher than a random negative observation. The AUC is a reliable heuristic to evaluate and compare the overall performance of classification models as it is scale-invariant (i.e., it measures how well predictions are ranked, rather than their absolute values) and threshold-invariant (i.e., it measures the quality of the model's predictions irrespective of the exact classification threshold chosen). However, it is not applicable in practice since a deployed classifier must have a particular classification threshold [301]. Thus, unless a classifier outperforms another for all possible threshold values, AUC cannot be used to compare and rank classifiers [99, 302].

A different approach based on statistics, is the phi-coefficient (ϕ) [303], also known as Matthews Correlation Coefficient (MCC) when applied to classifiers [304]. This approach considers the true class and the predicted class as two (binary) variables, and computes their correlation coefficient, in a similar way to the computation of the correlation coefficient between any two variables. The higher the correlation between true and predicted values, the better the prediction. MCC has the nice property of being perfectly symmetric, i.e., no class is more important than the other, and switching the positive and negative class yields to a same MCC value. Since MCC takes into account all four values in the confusion matrix, a coefficient close to 1 means that both the positive and negative classes are predicted accurately, even when the data is imbalanced.

The use of MCC allows to capture overall model performance, and thus it is often recommended for evaluating the performance of classification models across different tasks or in the presence of imbalanced data. Previous studies have argued that using MCC is more appropriate than using F1, (or AUC), alone for defect prediction studies [305, 215], yet our survey, (described in Section 6.4), shows that MCC has not been widely adopted yet (see Figure 6.2). A detailed comparison of MCC, F1 and AUC can be found elsewhere [37, 305, 215, 306].

The above measures are easy to compute, and this has generated the use of a

[1]The Receiver Operating Characteristic (ROC) is a graph showing model's performance in terms of True Positive Rate and False Positive Rate at all classification thresholds.

variety of different measures over the years, even if they were not always related to the specific model usage requirements as we further articulate in the next section.

6.4 Investigating the Use of Evaluation Measures in the Defect Prediction Literature

As discussed in Section 6.2, the use of different evaluation measures has raised a concern within the defect prediction community that dates back to over 10 years ago [39], but *what has changed since?* and *how have we handled this crucial aspect in our studies*? In the following we aim at answering these questions by reviewing prior studies published over the last decade.

6.4.1 Search Methodology

We used Scopus to search for research articles published between 2010 and 2020 (June 2nd) by using the query: ("Defect" || "Fault" || "Bug") & ("Prediction" || "Prone"). This search resulted in papers. Among these, we manually filter out irrelevant publications by following the three step process adopted in previous surveys [261, 307] as described below:

1. **Title:** First, we exclude all publications whose title clearly does not match our inclusion criteria;
2. **Abstract:** Second, we examine the abstract of every remaining publications. Publications whose abstract does not meet our inclusion criteria are excluded at this step;
3. **Body:** We then read, in full, all publications that had passed the previous two steps. Manuscripts are excluded if their content does not satisfy the inclusion criteria or contribute to this survey.

To ensure that the publications included in this survey are relevant to the context of binary defect prediction, at each of the above steps we apply the following inclusion criteria:

- The publication should investigate an experimental study of software defect prediction models, metrics or data.

- The publication predicts a dichotomous outcome (i.e., defect or not defect-prone).

Based on the above three-stage process and inclusion criteria, we iteratively reduce the amount of publications obtained from the Scopus search, until we end up with a set of 111 publications investigated herein. We provide the full list of papers in our online appendix [308].

We then manually examine the 111 papers to extract relevant information. In particular, in order to identify which evaluation measures were used in each study, and the rational for them, we analyze the section discussing the evaluation measure/procedure.

We consider a study to have explained its use of measures when it provides a clear reasoning for the choice of measures as opposed to others, or when otherwise, the reason is obvious from the research context.

Finally, we proceed by checking whether the study mentions any challenge arising from the choice of the evaluation measure(s), and hence whether any mitigation was put in place. We consider a study to have acknowledged and mitigated the importance of choosing proper evaluation measures if it discusses, for example, limitations of the used measure, the existence of measures other than the ones used, the fact that different measures might yield different results, the necessity to complement the used measures with additional ones, etc.

6.4.2 Results

We report the number of papers that use a specific measure in Table 6.3 as well as the number of studies that use one or multiple measures in Figure 6.1, and the frequency of measure use over time in Figure 6.2.

We can observe, from Table 6.3, that more than nine different evaluation measures have been used to assess defect prediction models over the past 10 years.

The most used measures are `AUC` and `F1` (used in 60% and 59% of papers, respectively), followed by `Recall` and `Precision` (used in 57% and 43% of papers, respectively). As explained in Section 6.3, measures such as AUC and F1 do not portray the full confusion-matrix and can lead to biased results when

Table 6.3: Number of studies using a given measure.

Evaluation Measure	No. of Studies (%)
AUC	60 (54%)
F1	59 (53%)
Recall (PD)	57 (51%)
Precision	43 (39%)
FPR (a.k.a. PF)	19 (17%)
G-measure	12 (11%)
Balance	10 (9%)
MCC	13 (12%)
Accuracy	12 (11%)
G-mean	9 (8%)
Others	10 (9%)

used for assessing classifier performance on unbalanced datasets, which is often the case for defect prediction data. A variety of symmetric and robust measures exist [99], however we found (see Table 6.3) that they have been adopted by much fewer studies (e.g., MCC 12%, G-measure 11%, G-mean 8%, Balance 9%).

Moreover, as shown in Figure 6.1, 30% of the papers analysed use only one measure to evaluate defect prediction models, among which AUC is the most common (53%), followed by F1 (33%).

Using only one measure might affect the validity of these studies especially if such measure is among those that have been shown to be problematic (see Section 6.3). This is the case of AUC and F1, which might be biased when imbalanced data is used, as further explained in Section 6.3.

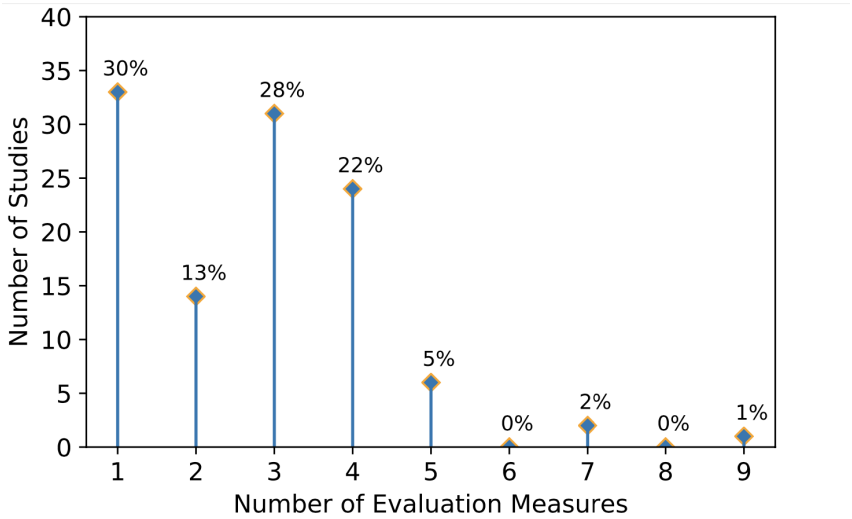


Figure 6.1: Number of measures used in prior studies.

The threat might still be present in past studies that use more than one measure. Indeed, among the 14 studies (13%) which use two measures, four of them (29%) use `AUC` and `F1` together; and among the 31 papers (28%) using three measures, 13 of them (42%) use `F1` and its constituent components (i.e., `Recall` and `Precision`). Furthermore, when analysing the studies that use four measures, 10 out of 24 (42%) use a set that may still be biased as it includes a subset of `Accuracy`, `F1`, `Precision`, `Recall`, and `AUC`. This is somehow worrying since these measures are known to be problematic [301, 39, 309, 310], as explained in Section 6.3. Their use should be complemented with the use of balanced measures such as G-measure or MCC.

Together, these results highlight that 56 out of the 111 papers examined (51%) might have drawn different conclusions had they considered a more comprehensive set of measures.

Further, when observing the use of the evaluation measures over the years, presented in Figure 6.2, results show that despite the warnings raised in previous studies about the use of `AUC`, `F1`, and its constituents [301, 39, 309, 310], their use has actually become more frequent with time. On the other hand, we do not observe as big of a growth in the adoption of more robust measures advocated in previous work [99]. For example, only one study has used MCC before 2016 and it has then been used at an average of less than three times per year since. This could have resulted from the fact that studies usually justify the usage of measures based on previous studies, and while measures like `AUC`, `F1` were popular in the past, the trend of following previous study practices creates an increasing trajectory.

Despite the warnings raised in previous work [23, 34, 35, 37], the results of our literature review reveal that the choice of evaluation measures is still a major concern in the community and that it has only been partially tackled so far. Over half of the studies (59%) do not justify the use of the measures, based on the aim of the study, neither the models investigated or the data at hand. 52% of studies also do not acknowledge that using different measures could lead to different results. Among those studies which acknowledge the threat (48%), 41 put in place some form of

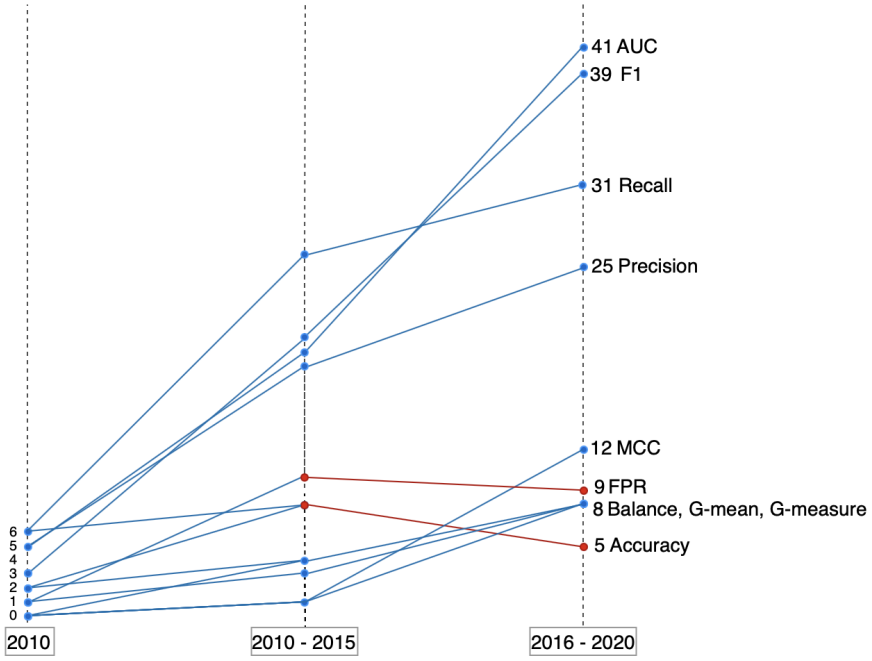


Figure 6.2: The frequency of measure use for 2010-2020. Blue lines signify an increase in measure usage while the red lines denote a decrease.

mitigation. However, only 15 out of the 41 studies recommend the use of unbiased measures, while the majority propose mitigation which might be perceived as unsatisfactory since they recommend, for example, that future work should investigate the use of other measures, or the use of measures widely seen in previous work, or the use of threshold independent measures, which has been criticised for not being applicable in practice since to deploy a classifier one must use a specific classification threshold [99, 301] as we explain in Section 6.3.

6.5 Empirical Study Design

In this section, we present a large-scale empirical study based on both, statistical tests and effect size to investigate the extent to which the conclusion of a study comparing the performance of defect prediction techniques may change based on the evaluation measure used to make the comparison.

This study investigates two research questions (RQs) and involves the use of seven classification techniques, 15 publicly available datasets, and six evaluation measures validated on three different defect prediction scenarios, as detailed in the

following subsections.

6.5.1 Research Questions

We pose two research questions investigating the use of the evaluation measures most used in prior studies (i.e., `AUC`, `Balance`, `F1`, `G-measure`, `MCC`, and `FPR` according to the literature [2] to measure the performance of seven different prediction techniques, described in Section 6.5.5.

Our first research question investigates, precisely, whether each of the six evaluation measures would rank the seven techniques in the same order. This is crucial because if these measures turn out to yield different rankings, then the conclusions made by previous studies using certain measures would change at a significant level. Therefore, we first ask:

RQ1. Ranking Disagreement: *How often would the ranking of techniques produced by a given measure differ from the ranking produced by another measure?*

To address RQ1, we first produce a ranking of the prediction techniques per measure according to the Wilcoxon Signed-Rank test and the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size [311]. Then, based on these rankings, we compute the *ranking disagreement* for each measure by counting the number of times it produces a same ranking with respect to using other measures.

The *ranking disagreement* tells us the extent to which a measure agrees with others. If the rank disagreement of a given measure is high, this would suggest that there is a non-trivial error when using different measures.

Our second research question analyses this error at a finer grain to gather further insights on the extent to which these rankings differ:

RQ2. Rank Disruption: *What is the percentage of cases in which a rank of a specific technique, based on a given measure, changes when assessed using the other measures?*

To assess the *rank disruption* of a ranking provided by a given evaluation measure, we assess the rank change, i.e., a prediction technique that would be believed

[2] We do not want to inflate our results, therefore we do not include `Accuracy`, `Recall` and `Precision` given they would generate opposing rankings due to the way they are defined.

to be better than others when using a given evaluation measure becomes worse when using a different measure. If the *rank disruption* proves to be high, some scientific conclusions drawn in previous studies could be reversed when accounting for the threat to validity posed by the choice of the evaluation measure.

We explain in details the way we compute the *rank disagreement* and *rank disruption* in Section 6.5.2.

6.5.2 Ranking Disagreement and Rank Disruption

To compute the ranking disagreement and the rank disruption, we run the prediction models on each dataset and evaluate their performance with each of the six evaluation measures under investigation. Then we apply the Wilcoxon test and the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size, and rank the prediction models, per measure, based on the results of these analyses. Once the rankings are obtained we compute:

- the *ranking disagreement* of a given measure by analysing the number of times its ranking varies from the rankings produced by each of the other evaluation measures per dataset. The *ranking disagreement* of a given measure varies from 0 (all measures agree on the same ranking) to 1 (all measures disagree, thus each ranking is unique).

- the *rank disruption* of a given measure by counting the number of times the rank of a specific technique changes when assessed according to each of the other measures.

In the following, we explain in detail how we computed the rankings based on the Wilcoxon test and $\hat{A}_{12}$ effect size.

Given an evaluation measure, we perform the Wilcoxon Signed-Rank test ($\alpha < 0.05$) on the results obtained by each pair of techniques by testing the null hypothesis: "The results achieved by $predictionModel_x$ in terms of a given evaluation measure m are worse than those achieved by $predictionModel_y$". We, then, summarise the results of this test by using the following win-tie-loss procedure [10]:

Table 6.4: An example of the procedure used to compute the ranking disagreement and the rank disruption.

	M_1	M_2	M_3
Rank 1	T_a	T_c	T_c
Rank 2	T_b, T_c	T_a	T_b
Rank 3	-	T_b	T_a

We count the number of times a prediction model scored a p–value$<$0.05 (win), p–value$>$0.95 (loss), and 0.05$\leq$ p–value $\geq$0.95 (tie). Finally, we rank the techniques based on the highest number of wins, where any ties are broken based on the number of losses. For example, based on the win-tie-loss procedure, Table 6.4 shows that according to measure M_1, technique T_a is the best performing one since it achieves the highest number of wins when compared to techniques T_b and T_c. Whereas, technique T_c is said to be the best performing one according to measures M_2 and M_3 given that this technique obtains the highest number of wins. However, all three measures (i.e., M_1, M_2 and M_3) disagree on the second and third rank. We also produce a ranking of the prediction models based on the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size [311] to validate whether the differences highlighted by Wilcoxon test are worthy of interest.

Similarly to the procedure described above, we compute the $\hat{A}_{12}$ effect size for each pair of techniques resulting in a statistically significant difference according to the Wilcoxon Signed-Rank test ($\alpha < 0.05$). The $\hat{A}_{12}$ statistic measures the probability that an algorithm A yields better values for a given performance measure M than that of another algorithm B. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. Given the first algorithm performing better than the second, $\hat{A}_{12}$ is considered small for $0.6 \leq \hat{A}_{12} < 0.7$, medium for $0.7 < \hat{A}_{12} < 0.8$, and large for $\hat{A}_{12} \geq 0.8$, although these thresholds are somewhat arbitrary [2]. Based on these thresholds, we count the number of small, medium and large effect sizes obtained by each prediction model, and rank the techniques based on the highest number of large effect size, where ties are broken based on the number of medium and small effect sizes in this order.

Following the above procedure, for each dataset we obtain a set of six rank-

ings (i.e., one per evaluation measure) based on the Wilcoxon test and a set of six rankings based on the $\hat{A}_{12}$. For each of these sets, we manually inspect the rankings produced by each of the measures to compute the ranking disagreement and rank disruption, as explained above.

6.5.3 Datasets

To answer RQ1, we carry out a large-scale empirical study using a total of 15 datasets available from two public repositories: NASA [102] and Realistic [312]. The former has been made publicly available in the early 2000 and contains software defect data about industrial NASA software systems, which have been widely used by the defect prediction community since. The latter was made publicly available in 2019 and consists of open-source projects characterised by a wider set of features than the ones extracted from the NASA data.

All data used herein was curated in previous work by following rigorous processes and made publicly available [102, 312]. In particular, we use six NASA datasets from the work of Petrić et al. [102], who applied rules to clean erroneous data from the original NASA repository [58], and data about nine open-source software systems, collected by Yatish et al. to ensure robust defect counts [312].These datasets have different nature and size: They vary in application domain, age, total number of modules (ranging from 94 to 8,847) and features (ranging from 38 to 65), and percentage of defective modules they contain as summarised in Table 7.2.

6.5.4 Validation Scenarios

Our experimental design tries to reflect as much as possible the scenarios where defect prediction has been applied so far. To this end we consider three common validation scenarios: Within-Project Defect Prediction, Cross-Version Defect prediction and Cross-Project Defect Prediction.

For the NASA datasets, given that each system consists of a single version, we investigate the Within-Project scenario applying the Hold-Out validation process where the data is randomly split into 80% training set and the other 20% which is used for testing. This procedure is repeated 30 times in order to reduce any possible

bias resulting from the validation splits [311].

For the Realistic datasets, given that each system consists of multiple releases we investigate both the Cross-Version and Cross-Project scenarios. In the Cross-Version scenario, for each of the software systems, we train on one release and test on a different one, i.e., we train on version v_x and test on version v_y, where $x < y$ as done in previous work (see e.g., [313]). In the Cross-Project scenario, for each of the software systems, we consider the version with the higher release number as the test set and train the model on the union of the versions of the other datasets with a lower release number. In both scenarios, the versions used as train and test sets are not subsequent releases nor are they the system's most recent ones. In addition, there is always a window of at least five months between these releases. This reduces the likelihood of the snoring effect or unrealistic labelling as noted in previous studies [314, 315, 316].

Table 6.5: Datasets used in our empirical study.

Repository	Dataset	Modules (% defective)
NASA	CM1	296 (12.84%)
	KC3	123 (13.00%)
	MW1	253 (10.67%)
	PC1	661 (7.87%)
	PC3	1043 (12.18%)
	PC5	94 (19.15%)
Realistic	Activemq_5.3.0	2367 (10.90%)
	Activemq_5.8.0	3420 (6.02%)
	Camel_2.10.0	7914 (2.91%)
	Camel_2.11.0	8846 (2.17%)
	Derby_10.3.1.4	2206 (30.33%)
	Derby_10.5.1.1	2705 (14.16%)
	Groovy_1_6_beta_1	821 (8.53%)
	Groovy_1_6_beta_2	884 (8.60%)
	Hbase_0.95.0	1669 (22.95%)
	Hbase_0.95.2	1834 (26.34%)
	Hive_0.10.0	1560 (11.28%)
	Hive_0.12.0	2662 (8.00 %)
	Jruby_1.5.0	1131 (7.25%)
	Jruby_1.7.0	1614 (5.39%)
	Lucene_3.0	1337 (11.59%)
	Lucene_3.1	2806 (3.81%)
	Wicket_1.3.0-beta2	1763 (7.37%)
	Wicket_1.5.3	2578 (4.07%)

6.5.5 Techniques

In order to study the magnitude of any discrepancy resulting from the use of different evaluation measures, we consider a variety of techniques, namely Dummy, Logistic Regression (LR), Naïve Bayes (NB), Random Forest (RF) and Support Vector Machine (SVM) implemented in `Scikit-learn 0.20.2` (`Python 3.6.8`), belonging to different classifier families. These are widely studied techniques in the defect prediction literature [23, 317]. Indeed, the chosen classifiers are also representative of those employed by the surveyed papers (see Section 6.4), as we found that LR is used in 67% of the surveyed papers (74/111), NB in 62% (69/111), RF in 47% (52/111) and SVM in 38% (42/111). We briefly describe them below:

Dummy is a simple baseline which generates predictions uniformly at random. We use the Scikit-learn's `dummy` class.

Naïve Bayes is a statistical classifier that uses the combined probabilities of the different attributes to predict the target variable, based on the principle of Maximum a Posteriori [318]. We use the Scikit-learn's `naive_bayes` class.

Logistic Regression is a statistical technique, introduced as an extension to linear regression, which models the probabilities for problems with a dichotomous dependent variable [319]. We use the Scikit-learn's `linear_model` class.

Random Forest is a decision tree-based classifier consisting of an ensemble of tree predictors [320]. It classifies a new instance by aggregating the predictions made by its decision trees. Each tree is constructed by randomly sampling N cases, with replacement, from the N sized training set. RF is known to perform well on large datasets and to be robust to noise. We use the RF classifier from the `ensemble` class in Scikit-learn. We also perform hyperparameter tuning, denoted as RF_T, where we explore different values for the following parameters:

- min_samples_leaf = [1, 20],
- min_samples_split = [2, 20],
- max_leaf_nodes = [default=None, 10, 50],
- n_estimators = [50, 75, 100, 125, 150]

Support Vector Machine is a widely known classification technique [270]. A lin-

ear model of this technique uses a hyperplane in order to separate data points into two categories. However, in many cases there might be several hyperplanes that can correctly separate the data. Hence, SVM seeks to find the hyperplane that has the largest margin, in order to achieve a maximum separation between the two categories. When the data is not linearly separable, SVM does the mapping from input space to feature space by using a kernel function instead of an inner product. In this work, we use Scikit-learn's `svm` class. We also perform hyperparameter tuning for this technique, denoted as SVM_T, where we try different values for C and γ with the Radial Basis Function kernel. Specifically, we explore the following grids:

- C = [0.01, 0.5, 1, 512, 8000, 32000],
- γ = [1/n_features, 0.000001, 0.0001, 0.001, 0.01, 0.1, 1, 2, 4, 8]

6.6 Empirical Study Results

In this section we report and discuss the results we obtained carrying out the empirical study described in Section 6.5.

6.6.1 RQ1. Ranking Disagreement

To address RQ1 we compare the performance of Dummy, LR, NB, RF, RF_T, SVM, SVM_T experimenting with the six most widely used evaluation measures in literature (shown in Table 6.3). The results of this comparison, based on both statistical significance test and effect size, are summarised in Tables 6.6 (Within-Project scenario) and 6.7 (Cross-Version and Cross-Project scenarios), and discussed in detail in the rest of the section. Each table reports whether a given measure disagrees with *"All"*, *"More than half*, *"Less than a half"*, or *"None"* of the other measures considered on a given dataset in our study[3].

When looking at the ranking of techniques obtained by each evaluation measure, based on the Wilcoxon test, our results show that across all scenarios (i.e., out of the 24 datasets) examined there is no case where the use of each measure leads to a same ranking of the techniques, i.e., the ranking disagreement is never 0. On

[3]Note that for the sake of space, we omit the columns *"None"* and *"Less than a half"* as our results show that there are no such cases.

Table 6.6: RQ1. Ranking disagreement results for the Within-Project scenario. For each dataset, we report whether a given evaluation measure disagrees with more than a half, or all the other measures, based on statistical significance and effect size analyses.

		>Half		*All*	
		Wilcoxon	**Eff. Size**	**Wilcoxon**	**Eff. Size**
CM1	F1			✓	✓
	G-meas.			✓	✓
	MCC			✓	✓
	Balance			✓	✓
	AUC			✓	✓
	FPR			✓	✓
KC3	F1		✓	✓	
	G-meas.	✓	✓		
	MCC			✓	✓
	Balance	✓			✓
	AUC			✓	✓
	FPR			✓	✓
MW1	F1		✓	✓	
	G-meas.		✓	✓	
	MCC			✓	✓
	Balance			✓	✓
	AUC			✓	✓
	FPR			✓	✓
PC1	F1			✓	✓
	G-meas.	✓			✓
	MCC			✓	✓
	Balance	✓			✓
	AUC			✓	✓
	FPR			✓	✓
PC3	F1			✓	✓
	G-meas.	✓			✓
	MCC			✓	✓
	Balance	✓			✓
	AUC			✓	✓
	FPR			✓	✓
PC5	F1			✓	✓
	G-meas.	✓			✓
	MCC			✓	✓
	Balance	✓			✓
	AUC			✓	✓
	FPR			✓	✓

the other hand, the number of times the ranking disagreement $= 1$ (i.e., a measure provides a unique ranking) is quite high as it adds up to a total of 119 out of the 144 cases analysed (83%). In the remaining cases, the ranking disagreement is always either greater or equal to 0.60.

Similar observations hold when analysing the results based on the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure. These results also show that

there is no case where all the measures produce the same ranking (i.e., ranking disagreement is never 0). When computing the number of times the ranking disagreement = 1, results show that this happens very frequently, specifically on a total of 123 out of the 144 cases studied (85%). The ranking disagreement on the remaining cases is always either equal or greater than 0.6. Below we discuss the specific results obtained in each of the scenarios.

When looking at the results obtained for the Within-Project scenario based on the Wilcoxon test, we observe that each of the measures produces a unique ranking on two out of the six NASA datasets (see Table 6.6). Whereas, four out of six measures produce a unique ranking on the other four datasets and only two (i.e., Balance and G-measure) agree with each other on the same ranking. An even stronger discordance than that shown by the Wilcoxon test is observed when analysing the results based on the $\hat{A}_{12}$ non-parametric effect size measure for this scenario. We found an agreement between two measures only (i.e., F1 and G-measure) on only two datasets out of the six under study. Whereas, each measure produces a unique ranking in all other cases (i.e., 34 out of 36) considered.

Based on the results for the Cross-Version scenario, shown in Table 6.7, we observe that, according to the Wilcoxon statistical significance test, each of the measures produces a unique ranking for four out of out nine datasets, while on the remaining ones, only two measures agree (G-measure and Balance on three datasets, F1 and Balance on one, and F1 and MCC on the other one). In the case of the analysis based on the Â12 effect size measure, each of the evaluation measures studied produces a unique ranking on five datasets. Similarly to the results observed by the Wilcoxon test, a maximum of two evaluation measures (the same ones denoted by the Wilcoxon test) agree on the remaining datasets. Specifically, G-measure and Balance agree on three of the remaining four datasets while F1 and Balance agree on another dataset and F1 and MCC on the remaining one.

The results based on the Wilcoxon test, in the Cross-Project scenario (shown in Table 6.7) are more discordant compared to those observed in the cross-version scenario. A unique ranking is produced by each measure on six out of the nine

dataset studied. On two out of the remaining three datasets, only two measures agree on a single ranking (i.e., Balance and G-measure). Whereas on the remaining dataset, Balance, F1 and G-measure agree on a ranking while the other measures provide unique ones. In addition, the $\hat{A}_{12}$ effect size, in line with the results based on the Wilcoxon Test, shows more discordant results compared to the cross-version scenario. While no measure agrees on any ranking for six of the datasets (i.e., each of the measures provides a unique ranking), at most two evaluation measures (i.e., Balance and G-measure) agree on a ranking for two of the remaining three datasets. As for the remaining dataset, results show that Balance, F1 and G-measure agree on the same ranking whereas the other evaluation measures produce unique rankings.

> **Answer to RQ1:** There is no case where all evaluation measures produce the same ranking. In fact, the evaluation measures produce a unique ranking in 83% and 85% of the cases according to the analysis based on the Wilcoxon test and $\hat{A}_{12}$ effect size, respectively. There are only very few cases where at most three (less than 1% according to both statistical and effect size analyses) or two evaluation measures (8% and 5% according to the Wilcoxon test and effect size, respectively) provide the same ranking, but even in those cases it is not always the same set of evaluation measures that agree on a ranking.

6.6.2 RQ2. Rank Disruption

Tables 6.8 and 6.9 report the rank disruption values obtained per measure per dataset for all scenarios when analysed based on the Wilcoxon Test and Vargha and Delaney's $\hat{A}_{12}$ effect size results, respectively.

Table 6.8: RQ2. Percentage of rank disruption per measure based on statistical significance analysis.

	Dataset	AUC	Balance	F1	FPR	G-meas.	MCC
Within-Project	**CM1**	69%	69%	74%	66%	66%	86%
	KC3	80%	71%	86%	71%	86%	91%
	MW1	63%	63%	69%	71%	71%	91%
	PC1	57%	49%	74%	49%	66%	71%
	PC3	77%	66%	91%	66%	83%	80%
	PC5	71%	66%	69%	66%	86%	77%
	Average	70%	64%	77%	65%	76%	83%
Cross-Version	**ActiveMQ**	69%	80%	71%	71%	80%	86%
	Camel	83%	69%	80%	71%	80%	86%
	Derby	86%	63%	80%	63%	89%	94%
	Groovy	94%	71%	77%	71%	91%	80%
	Hbase	71%	83%	74%	94%	91%	100%
	Hive	60%	66%	60%	60%	71%	100%
	JRuby	69%	80%	69%	74%	83%	89%
	Lucene	86%	63%	74%	63%	77%	100%
	Wicket	80%	80%	69%	80%	74%	86%
	Average	77%	73%	73%	72%	82%	91%
Cross-Project	**ActiveMQ**	86%	71%	77%	80%	89%	100%
	Camel	69%	66%	71%	71%	89%	80%
	Derby	51%	51%	83%	51%	89%	86%
	Groovy	77%	66%	77%	69%	89%	91%
	Hbase	71%	66%	69%	66%	97%	100%
	Hive	77%	71%	69%	71%	94%	91%
	JRuby	57%	83%	60%	66%	66%	86%
	Lucene	69%	63%	86%	63%	100%	94%
	Wicket	89%	74%	91%	80%	97%	89%
	Average	72%	68%	76%	69%	90%	91%

For the Within-Project scenario, the average rank disruption across the NASA datasets varies between 64% (Balance) and 83% (MCC) on average when basing the results on the Wilcoxon test and between 70% (Balance) and 86% (MCC) when the analysis is based on $\hat{A}_{12}$, depending on the evaluation measure used.

Similarly for the Cross-Version scenario the average rank disruption across the nine Realistic datasets ranges from 72% (FPR) to 91% (MCC) based on the Wilcoxon test and from 70% (FPR) to 92% (MCC) based on $\hat{A}_{12}$.

The rank disruption is still very high when looking at the Cross-Project scenario, with average results across the nine systems ranging from 68% (Balance) to 91% (MCC) based on the Wilcoxon test and from 67% (Balance) to 89% (G-measure and MCC) based on $\hat{A}_{12}$.

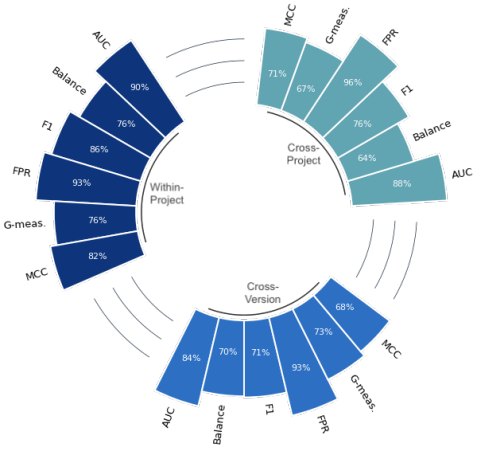
Table 6.9: RQ2. Percentage of rank disruption per measure based on effect size analysis.

	Dataset	AUC	Balance	F1	FPR	G-meas.	MCC
Within-Project	CM1	94%	83%	74%	74%	80%	80%
	KC3	63%	63%	86%	63%	89%	89%
	MW1	66%	66%	74%	69%	74%	91%
	PC1	77%	77%	80%	71%	77%	91%
	PC3	83%	74%	83%	77%	83%	97%
	PC5	60%	57%	57%	69%	100%	69%
	Average	74%	70%	76%	70%	84%	86%
Cross-Version	ActiveMQ	69%	86%	71%	71%	77%	83%
	Camel	83%	69%	80%	71%	80%	86%
	Derby	74%	63%	74%	63%	83%	94%
	Groovy	86%	69%	77%	69%	83%	86%
	Hbase	71%	83%	74%	94%	91%	100%
	Hive	57%	69%	60%	57%	74%	100%
	JRuby	66%	89%	66%	77%	83%	89%
	Lucene	69%	57%	80%	57%	77%	100%
	Wicket	77%	71%	71%	74%	77%	91%
	Average	72%	73%	73%	70%	81%	92%
Cross-Project	ActiveMQ	83%	77%	77%	86%	86%	94%
	Camel	71%	66%	60%	63%	91%	77%
	Derby	49%	49%	77%	49%	89%	89%
	Groovy	71%	57%	71%	71%	86%	83%
	Hbase	71%	66%	71%	71%	94%	100%
	Hive	80%	66%	80%	66%	91%	97%
	JRuby	54%	83%	57%	63%	69%	80%
	Lucene	69%	63%	86%	63%	100%	94%
	Wicket	89%	74%	91%	80%	97%	89%
	Average	71%	67%	75%	68%	89%	89%

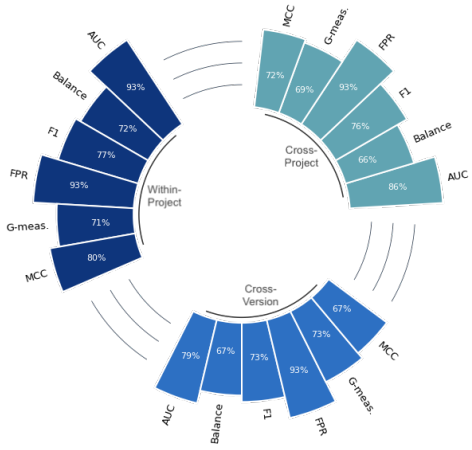
We can conclude that both analyses (i.e., Wilcoxon test and $\hat{A}_{12}$) provide similar rank disruption results for all the three scenarios.

We also perform a more fine grained analysis of our results to get a better understanding of the impact the use of these different evaluation measures can have on the conclusion validity of a study. We therefore examine the rank disruption of each evaluation measure when considering the top three ranked techniques, then considering the top two ranked and then the first ranked technique, for each scenario investigated. Figure 6.3 shows the average rank disruptions of the measures, across all datasets, for each scenario based on statistical test and effect measure analyses, respectively.

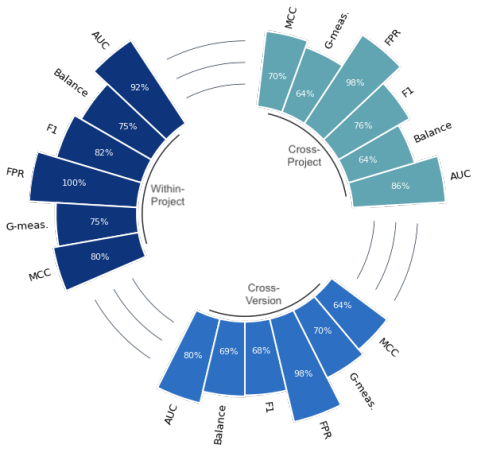
Top Three Ranked Techniques: We can observe from Figure 6.3 that the rank disruption remains very high when considering the top three ranked techniques for each scenario. Specifically, according to the Wilcoxon test (shown in Figure 6.3a), the rank disruption ranges from 76% (Balance and G-meas.) to 93% (FPR) when evaluating the Within-Project scenario, from 68% (MCC) to 93% (FPR) in the Cross-



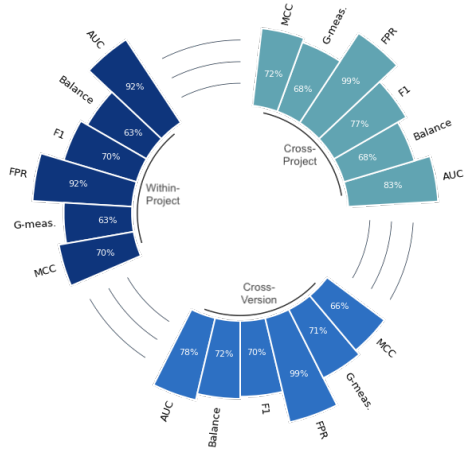
(a) Rank disruption of top three ranked techniques based on statistical significance.



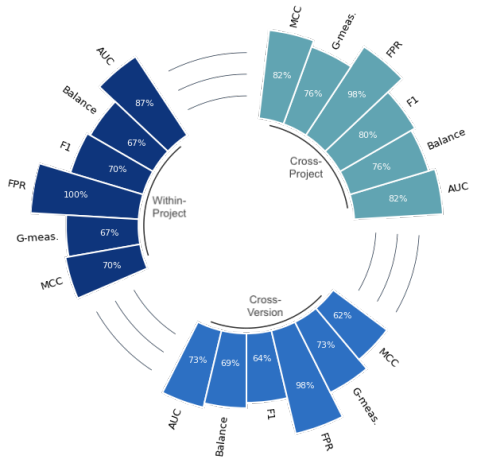
(d) Rank disruption of top three ranked techniques based on effect size measure.



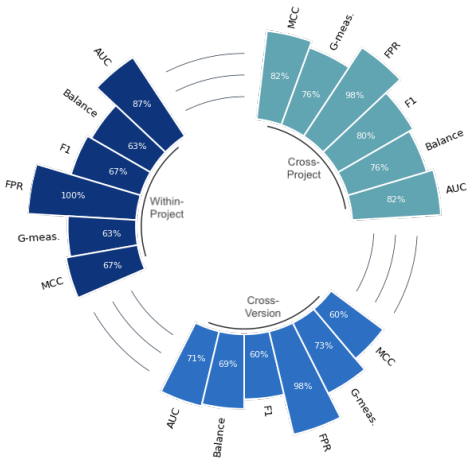
(b) Rank disruption of top two ranked techniques based on statistical significance.



(e) Rank disruption of top two ranked techniques based on effect size measure.



(c) Rank disruption of the top ranked technique based on statistical significance.



(f) Rank disruption of the top ranked technique based on effect size measure.

Figure 6.3: RQ2. Rank disruption (average across all datasets) of each evaluation measure for the top three techniques (a,d), top two (b,e) and first ranked technique (c,f) for each scenario investigated based on statistical significance analysis (a,b,c) and effect size measure (d,e,f).

Version scenario and from 64% (Balance) to 96% (FPR) in the Cross-Project scenario. Similar observations hold when analysing the rank disruption based on $\hat{A}_{12}$, with its values ranging between 71% (G-meas.) and 93% (AUC and FPR) for the Within-Project scenario, between 67% (Balance and MCC) and 93% (FPR) when evaluating predictions across versions (i.e., Cross-Version scenario) and between 66% (Balance) and 93% (FPR) in the Cross-Project scenario.

Top Two Ranked Techniques: This high level of rank disruption is also consistent when considering the top two ranked techniques only. Figure 6.3b shows that, based on the Wilcoxon text, the rank disruption ranges between 75% (Balance and G-meas.) to 100% (FPR) when evaluating the Within-Project scenario, between 64% (MCC) and 98% (FPR) in the Cross-Version scenario, and between 64% (Balance and G-meas.) and 98% (FPR) when evaluating predictions across projects (i.e., Cross-Project scenario). The analysis based on $\hat{A}_{12}$ also agrees with that of the statistical significance test. Figure 6.3e reports that the average rank disruption, across all datasets, ranges between 63% (Balance and G-meas.) and 92% (AUC and FPR) in the Within-Project scenario. When considering the Cross-Version and Cross-Project scenarios, the average rank disruption is a bit higher, with its values ranging between 66% (MCC) and 99% (FPR) and between 68% (Balance and G-meas.) and 99% (FPR), respectively.

Top Ranked Technique: High values of rank disruption are also be observed when only the top ranked technique is considered. The average rank disruption based on the the Wilcoxon test (see Figure 6.3c) varies between 67% (Balance and G-meas.) and 100% (FPR) in the Within-Project scenario, between 62% (MCC) and 98% (FPR) in the Cross-Version scenario, and between 76% (Balance and G-meas.) and 98% (FPR) in the Cross-Project scenario. Similar to previous analyses, the results based on the effect size measure agree with those based on the Wilcoxon test. As shown in Figure 6.3f, the rank disruption based on the $\hat{A}_{12}$ ranges between 63% (Balance and G-meas.) and 100% (FPR) in the Within-Project scenario, between 60% (F1 and MCC) and 98% (FPR) when evaluations are carried out on predictions of different versions within a same software project (i.e., Cross-Version scenario),

and between 76% (Balance and G-meas.) and 98% (FPR) in the Cross-Project scenario.

> **Answer to RQ2:** The rank disruption for each of the measures investigated is high on average, ranging from 64% to 92% depending on the measure and validation scenario. Our results also show that high disruption is also present when investigating the top ranked techniques only, with an average rank disruption ranging between 60% and 100% depending on the validation scenario, the measure and the number of top ranked techniques considered.

6.7 Threats to Validity

We discuss below how we mitigate possible threats to the internal, construct and external validity of this study.

Internal Validity: To mitigate the threat of missing relevant work or information in our literature review, we have specified the query we used for our search, defined clear inclusion criteria and followed a rigours procedure recommended and used in previous work to filter out irrelevant articles [261, 321]. Although we cannot and do not claim that the set of 111 studies we investigates is exhaustive, it is reasonable to believe that it is representative of the current state-of-the-art. As a matter of fact, we note that two prominent defect prediction literature surveys [23, 99] published in IEEE TSE in 2012 and 2014, reviewed, respectively, 36 and 42 papers, while in this study we analyse 111 articles. The gathering and filtering procedure was performed by both authors, to ensure reliability and reduce researcher bias.

Construct Validity: We carefully calculated the performance measures used in the study, and applied the statistical tests, verifying all the required assumptions.

External Validity: As happens with most empirical studies, the subjects used in our study might not be representative of the whole population. However, we have designed our study aiming at using measures, techniques, datasets, and validation scenarios, which are as representative as possible of the defect prediction literature. First of all, we focused on binary evaluation measure as these are the most common

in literature [23] and we strove to consider the measures most used in previous work (see Section 6.4). To increase the relevance to the defect prediction literature we also investigated class level defect binary prediction models in three validation scenarios (i.e., within-project, cross-release and cross-project) widely investigated in the literature [322, 297] as opposed, for example, to more recent (and therefore less studied) ones such as effort-aware models [323, 324, 325, 326] or binary prediction at line level [327], method level [328] or commit/change level [329, 330, 331, 332]. Similarly, we considered traditional classification techniques widely used in previous studies [23] as described in Section 6.5.5. Moreover, we used a publicly available implementation of these techniques provided by the `Scikit-learn` library, to reduce possible biases and errors arising from the use of ad-hoc implementations. We also used publicly available datasets previously investigated in the literature, which are of different nature and size, and which have been carefully curated in previous work as explained in Section 6.5.3.

6.8 Conclusions

In this chapter, we investigate the effect of using different evaluation measures for comparing the performance of software defect prediction models.

Our review of previous work published over the last decade, has revealed that the majority of the studies do not provide rationale for the measures used with regard to the characteristic of the datasets and/or aim of their study. Moreover, they often use measures which only partially reflect the performance of defect prediction models, and that the measures used are often the most susceptible ones to data imbalance. Further, our empirical study reveals that different evaluation measures provide unique rankings in 82% and 85% of the cases studied according to the Wilcoxon test and effect size measure, respectively. Moreover, the rank disruption for each of the measures investigated is high (ranging from 61% to 90% on average depending on the measure and validation scenario). These results suggest that in the majority of the cases, a prediction technique that would be believed to be better than others when using a given evaluation measure becomes worse when using a

different one. Our results also show that the percentage of disruption when only considering the top ranked techniques is just as significant as when all ranks are studied, rendering the results even more striking.

We hope that the empirical evidence provided herein on the significant differences that arise from the use of evaluation measures will encourage the community to act upon this matter, and carefully select the measures based on factors which are specific to the problem at hand [23]. These include (1) the class distribution of the training data; (2) the way in which the model will be used; (3) the way in which the model has been built. Moreover, we recommend to include in the set of evaluation measures, at least one able to capture the full picture of the confusion matrix (i.e., the correctly and incorrectly classified instances), such as MCC, so that it is possible to assess whether proposals made in previous work can be applied for purposes different than the ones they were originally intended for. Besides, we recommend to report, whenever possible, the raw confusion matrix from which the results were extracted as this can enable other researchers to compute any measure of interest and facilitate them to draw meaningful observations across different studies.

Table 6.7: RQ1. Ranking disagreement results for the Cross-Version and Cross-Project scenarios. For each dataset, we report whether a given evaluation measure disagrees with *more than a half*, or *all* the other measures, based on statistical significance and effect size analyses.

		Cross-Version				**Cross-Project**			
		>Half		***All***		***>Half***		***All***	
		Wilcoxon	**Eff. Size**	**Wilcoxon**	**Eff. Size**	**Wilcoxon**	**Eff. Size**	**Wilcoxon**	**Eff. Size**
ActiveMQ	F1			✓	✓			✓	✓
	G-meas.			✓	✓			✓	✓
	MCC			✓	✓			✓	✓
	Balance			✓	✓			✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Camel	F1			✓	✓			✓	✓
	G-meas.			✓	✓			✓	✓
	MCC			✓	✓			✓	✓
	Balance			✓	✓			✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Derby	F1			✓	✓	✓	✓		
	G-meas.	✓	✓			✓	✓		
	MCC			✓	✓			✓	✓
	Balance	✓	✓			✓	✓		
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Groovy	F1			✓	✓			✓	✓
	G-meas.	✓	✓					✓	✓
	MCC			✓	✓			✓	✓
	Balance	✓	✓					✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
HBase	F1			✓	✓			✓	✓
	G-meas.			✓	✓			✓	✓
	MCC			✓	✓			✓	✓
	Balance			✓	✓			✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Jruby	F1	✓	✓					✓	✓
	G-meas.			✓	✓			✓	✓
	MCC	✓	✓					✓	✓
	Balance			✓	✓			✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Lucene	F1			✓	✓			✓	✓
	G-meas.	✓	✓			✓	✓		
	MCC			✓	✓			✓	✓
	Balance	✓	✓			✓	✓		
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Hive	F1	✓	✓			✓	✓		
	G-meas.			✓	✓			✓	✓
	MCC			✓	✓			✓	✓
	Balance	✓	✓			✓	✓		
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓
Wicket	F1			✓	✓			✓	✓
	G-meas.			✓	✓			✓	✓
	MCC			✓	✓			✓	✓
	Balance			✓	✓			✓	✓
	AUC			✓	✓			✓	✓
	FPR			✓	✓			✓	✓

Chapter 7

MEG: Multi-objective Ensemble Generation for Software Defect Prediction

In the previous chapter, we assessed the impact of using different evaluation measures for defect prediction models. The user's knowledge of the business domain is an important factor to choose an appropriate way to evaluate whether a given prediction model is effective for the problem at hand. In this chapter, we introduce a novel approach that allows the user to automatically build defect prediction ensembles guided by the user's own choice of evaluation measure based on their business objective.

7.1 Introduction

A variety of automated approaches, ranging from traditional classification models to more sophisticated learning approaches, have been explored for the early detection of software defects. Among these, recent studies have found the use of ensemble prediction models (i.e., aggregation of multiple base classifiers) to achieve more accurate results than those that would have been obtained by relying on a single classifier. However, designing an ensemble requires a non-trivial amount of effort and expertise with respect to the choice of the set of base classifiers, their hyper-parameter tuning, and the choice of the strategy used to aggregate the predictions.

An inappropriate choice of any of these aspects can lead to over- or under-fitting, thereby heavily worsening the performance of the ensemble. Examining all possible combinations is not computationally affordable, as the search space is too large, and there is a strong interaction among these aspects, which cannot be optimized separately. Such a large search space makes Search-Based Software Engineering a suitable solution for the problem of automatically generating effective ensembles for DP.

In this chapter, we propose a novel use of multi-objective evolutionary algorithms to automatically generate defect prediction ensembles. We dub our proposed approach Multi-objective **E**nsemble **G**eneration (MEG).

MEG is novel with respect to the existing proposals in the more general area of evolutionary generation of ensembles, which are all based on *Pareto-ensemble generation* as opposed to the concept of *Whole-ensemble generation* we introduce herein. Moreover, our study is the first to investigate the effectiveness of evolutionary ensemble for defect prediction.

In order to assess the effectiveness of MEG, we conduct a large-scale empirical study by benchmarking it against traditional base classifiers (as a sanity check), against the state-of-the-art multi-objective ensemble approach proposed by Petrić et al. [202] which, to the best of our knowledge, is the only one to use a diversity measure in the ensemble Defect Prediction literature, and against DIVACE proposed by Chandra et al. [207], which is considered a seminal work for ensemble generation in the Evolutionary Computation literature. These are the work most closely related to ours [202, 207], which motivated us to compare MEG to them. To assess the effectiveness of MEG we carried out a large-scale empirical study involving a total of 24 real-world software versions and 16 cross-version defect prediction scenarios, assessed according to the latest best practice for the evaluation of defect prediction and search-based approaches [333, 258, 334].

Our results show that MEG is able to generate ensembles with similar or more accurate predictions than those achieved by all the other approaches considered in 73% of the cases (with large effect sizes in 80% of them). Not only does MEG yield

good results, but it also relieves the engineer from the error-prone, burdensome, and time-consuming task of manually designing and experimenting with different ensemble configurations in order to find an optimal one for the problem at hand.

The main contributions of this chapter are: (1) The proposal of MEG, a novel multi-objective approach for the automated generation of ensembles based on the concept of *whole-ensemble generation*. (2) A large-scale empirical evaluation of the effectiveness of MEG for defect prediction which involves eight open-source Java software systems for a total of 24 software versions and 16 cross-version defect prediction scenarios. (3) The comparison of MEG to baseline defect prediction classifiers, the more recent approach to build ensembles for defect prediction [202], and a state-of-the-art multi-objective Pareto-ensemble generation (which has never applied to defect prediction before) [207]. We make the source code of MEG publicly available to facilitate its uptake for both researchers and practitioners [335]. We also share a replication package to allow for reproduction and extension [335].

7.2 Background

Given the multi-disciplinary nature of this work, this section provides some background on Ensemble Learning, and Multi-Objective Evolutionary Optimisation.

7.2.1 Ensemble Learning

Base models can be used to design more complex models by combining a number of them according to a given ensemble learning approach. The aim is to achieve a more robust model able to reduce both (1) the bias error, which arises from erroneous assumptions made by a single base learner and which usually causes underfitting (i.e., missing relevant relations between features and target outputs); (2) the variance, which arises from the base learner sensitivity to small fluctuations in the training set and causes overfitting (i.e., the algorithm models the noise present in the training data).

Ensemble approaches can be classified into two types, homogeneous ensembles and heterogeneous ones. Homogeneous ensembles consist of members having a single-type base learning algorithm, whereas heterogeneous ensembles consist of

members having different base learning algorithms. The choice of base learners and their combination to build an ensemble is extremely important for building a successful model. To this end, several algorithms have been proposed in the literature [204]. These include both, simple aggregation strategies, such as majority voting, weighted majority voting, average voting, and more advanced ones like bagging, boosting, and stacking.

Majority Voting is a simple aggregation strategy that considers the prediction of each base classifier, for a new instance, as a single vote. It then assigns the class which obtains the largest number of votes to the new instance.

Weighted Majority Voting is an extension to the Majority Voting strategy, except that it gives more weight to the best base classifier by counting its prediction twice. It then follows the same strategy as Majority Voting as it assigns, to the new instance, the class which obtains the largest number of votes.

Average Voting assigns to the new instance an averaged value of the predictions of all base classifiers. In binary classification problems, such as defect prediction, where the output is either "defective" or "non-defective", this strategy computes the average of the prediction probabilities yielded by each base classifier. If the final averaged probability is lower than 0.5, the instance is classified as non-defective, otherwise it is assigned to the defective class.

Stacking is an ensemble machine learning algorithm which uses a meta-classifier to learn how to best combine the predictions from two or more base machine learning algorithms. Stacking can harness the power of a range of well-performing models on a classification or regression task and can make predictions that achieve better performance than any single model in the ensemble.

Bagging and *Boosting* are considered homogeneous learners, while Stacking considers heterogeneous base learners. Besides, Bagging mainly focuses on producing an ensemble model with less variance than its components whereas Boosting and Stacking will mainly try to produce strong models less biased than their components.

7.2.2 Multi-Objective Evolutionary Optimisation

Evolutionary Algorithms (EAs) are evolution-based optimisation algorithms used to find approximation solutions in a feasible amount of time to otherwise hard search problems [205, 336]. EAs work by iteratively evolving a population of chromosomes (solutions), each of which containing a set of genes. These solutions are encoded in a pre-defined structure, also known as the *representation*, e.g., an array of bits, integers, or floating points. The quality of a solution is assessed by a *fitness function*, which measures the extent to which a solution is fit to solve the problem. At each generation, the solutions of the population (parents) undergo crossover and random mutations in order to generate new candidate solutions (offspring). These operations carry genetic information from the fittest parents to the offspring and introduce diversity into the population, respectively. At the end of the generation, the fittest solutions survive and become parents in the subsequent generation. When a given stopping condition is reached, the fittest solution is returned. However, in most real-world scenarios (such as in Software Engineering), there are many conflicting objectives with equal weights that should be considered when analysing the quality of solutions [205, 337, 338]. They are said to be *conflicting* because they often cannot be optimised simultaneously in full, i.e., by improving one objective, the others are likely to deteriorate. Therefore, Multi-Objective Evolutionary Algorithms (MOEAs) try to find a balance between the many objectives and eventually output a set of non-dominated solutions using the concept of Pareto dominance [338, 205] as explained below. Let Z be a set of minimisation objectives and x and y two different solutions. Solution x is said to dominate solution y ($x \succ y$) if: $\forall z \in Z : z(x) \leq z(y) and \exists z \in Z : z(x) < z(y)$. If these conditions do not hold, then x and y are said to be non-dominated, i.e., they represent equally feasible solutions with acceptable trade-offs for the problem at hand.

7.3 MEG: Multi-objective Ensemble Generation

Figure 7.1 depicts an overview of MEG. The main purpose of MEG is to automatically generate ensemble classifiers by choosing a set of base classifiers, tuning them,

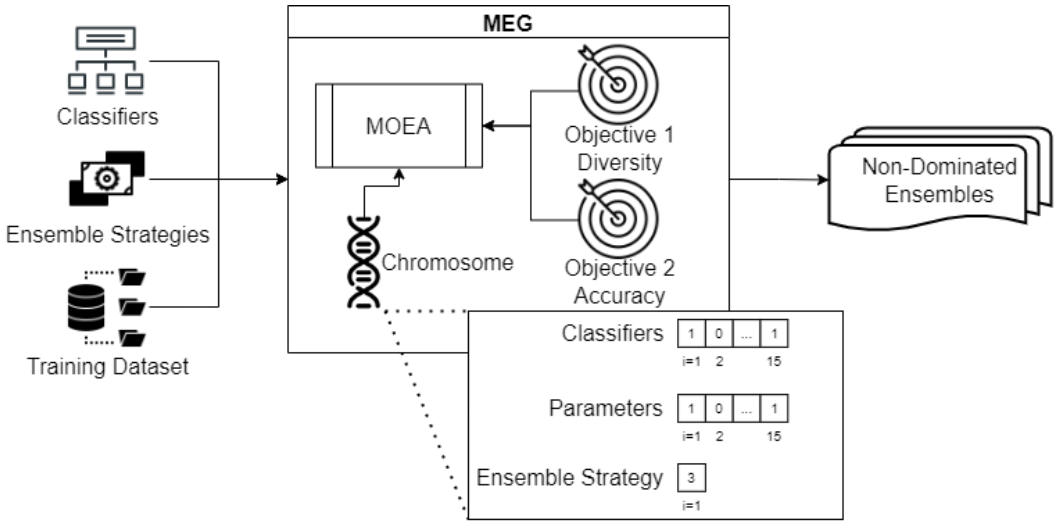


Figure 7.1: An overview of our proposed Multi-objective Ensemble Generation (MEG) approach.

and then selecting an aggregation strategy to produce the final ensemble. MEG is based on MOEAs, thus it iteratively evolves a population of ensembles across multiple generations and outputs the ensembles with the best trade-off between diversity of base classifiers and overall accuracy of predictions (Section 7.3.2 describes the objectives).

MEG differs from related work [207, 211, 212, 213, 214] in many ways. While algorithms like DIVACE [207] work as a Pareto-ensemble technique (evolve base classifiers and aggregate the non-dominated ones), MEG takes a more direct and intuitive approach where each solution in the population is a whole ensemble. Other work use a more similar representation as the one adopted by MEG [211, 212, 213, 214]. However, such approaches focus on selecting a set of pre-defined base classifiers, as opposed to designing, configuring, and building ensembles and the constituent parts.

MEG differs from related work by selecting base classifiers, optimising their hyper-parameters, and finally picking an ensemble strategy that best suits the context. The result is not a single ensemble, but rather a set of evolved ensembles from which the engineer can choose the one that best fits their needs. This allows for a robust evolution of ensembles with more flexibility in how they are designed and built. We define our proposed technique, MEG, as *whole ensemble generation*,

which is not only novel for the defect prediction literature, but also for the general ensemble literature.

7.3.1 Representation

The representation of MEG consists of three arrays: i) Classifiers – binary array; ii) Parameters – double/floating points array; and iii) Ensemble/Aggregation Strategy – integer array.

The classifier array contains 15 bits, where each index corresponds to a specific classifier. If the bit in index i is 1, it signifies that the i-th classifier is active and will be included in the ensemble, otherwise, it will not be included. MEG explores three different types of classifiers. These are namely, Naive Bayes (NB) at indexes 1..3, three k-Nearest Neighbors algorithms (k-NN) at indexes 4..6, four Support Vector Machines (SVM) at indexes 7..10, and five Decision Trees (DT) at indexes 11..15. We included those models in the representation because the most related work [202] to ours used the same. This allows a fair comparison in our empirical study, however future work can extend MEG to incorporate other classifiers.

The parameter array consists of the hyper-parameters of each of the 15 classifiers.

For k-NN, SVM, and DT, the parameters are represented by double values indicating the number of neighbours, cache size, and pruning confidence, respectively. For NB, we use a categorical value where 0 indicates the normal density distribution and 1 represents a kernel density estimator. MEG can be extended to handle additional hyper-parameters for each classifier, however in this work we experimented with the ones used in the work of Petrić et al. [202].

Finally, the strategy array consists of a single integer value (later converted into a categorical one) representing the strategy to be used by the ensemble to aggregate the predictions of all constituent classifiers. Given that we aim at investigating ensembles composed by different types of base classifiers, we consider the following heterogeneous aggregation strategies: i) majority voting; ii) weighted majority voting; iii) stacking; and iv) average voting.[1]

[1]We do not consider Bagging and Boosting as they are both homogeneous ensemble strategies

The proposed representation allows MEG to simultaneously select classifiers, optimise their parameters, and select the aggregation strategy. For instance, differently from algorithms such as DIVACE [207], the strategy array allows MEG to optimise for a specific ensemble strategy, as opposed to forcing the engineer to choose one. Furthermore, allowing the selection of classifiers and their tuning online also reduces the engineering effort, as the engineer is not required to pre-train the classifiers before the optimisation process, such as in the work of Fletcher et al. [211].

7.3.2 Fitness Functions

MEG uses two fitness functions to guide the search for ensembles: diversity and accuracy. The accuracy of an ensemble depicts how well it can predict the labels of the instances under consideration, which in the context of this work are "defective" (true) or "non-defective" (false). Naturally, the more accurate the ensemble, the better. On the other hand, the diversity measures assess how different the predictions of the classifiers in the ensemble are. Diversity is an important factor when designing ensembles, since a diverse ensemble is more likely to predict "corner cases" instances. However, in a classification problem, the more the classifiers disagree in their predictions, the less accurate the ensemble tends to be. For instance, for a given instance with the true label "defective", if two classifiers each predict "defective" and "non-defective" respectively, then we obtain diverse results, but with 50% accuracy. Hence, these two measures are conflicting, but MEG still aims at optimising both for a good trade-off.

There are many accuracy and diversity measures in the literature [204]. In this work, we use Mathews Correlation Coefficient (MCC) [99] as the accuracy objective, and Disagreement [339] as the diversity measure. MCC represents the correlation coefficient between the actual and predicted classifications. Equation 7.1 shows the formula for the assessment of MCC.

$$\uparrow Accuracy = MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{7.1}$$

which take into account a single type of base classifier.

where True Positives (TP) are defective modules correctly classified as defective; False Positives (FP) are non-defective modules falsely classified as defective; False Negatives (FN) are defective modules falsely classified as non-defective; True Negatives (TN) are non-defective modules correctly classified as non-defective (see Table 7.1).

MCC outputs a value between -1 and $+1$, where a value of $+1$ indicates a perfect prediction, a value of 0 signifies that the prediction is no better than random guessing, and -1 represents a completely miss-classified output. With that in mind, the greater the MCC, the better the solution. We opted to use MCC to assess and compare the accuracy of models as this measure has been strongly recommend in alternative to other previously popular measures, such as F-measure, which have been shown to be biased [99, 215, 258] when the data is imbalanced (as it is frequently the case in DP). MCC is a more balanced measure which, unlike the other measures, takes into account all the values of the confusion matrix [99, 258].

The second objective, diversity, is computed based on the disagreement measure [339], which measures (as the name implies) the prediction disagreement between groups of classifiers. Previous work has proposed various disagreement measures in the literature. To better understand whether there is a significant difference between the performance of these measures, we conducted a preliminary study comparing four different diversity measures. These are namely Disagreement, Double Fault, Q Statistic, and Correlation Coefficient. The results of our preliminary investigation showed that these measures provide the same or similar diversity values, so using any of the measures would yield to very similar results. Therefore, we use the Disagreement measure in order to maintain a fair comparison with the traditional ensemble defect prediction model proposed by [202] to which we compare MEG.

Table 7.1: Confusion Matrix for Binary Classification.

Actual Value	Predicted Value	
	Defective (1)	Non-Defective (0)
Defective (1)	TP	FN
Non-Defective (0)	FP	TN

Equation 7.2 depicts the formula for computing Disagreement:

$$\uparrow Diversity = DIS_{i,j} = \frac{N^{10} + N^{01}}{N^{11} + N^{00} + N^{10} + N^{01}} \tag{7.2}$$

where N^{10} represents the number of instances correctly classified by the i^{th} classifier and incorrectly by the j^{th} classifier of the ensemble. N^{01}, N^{00} and N^{11} can be interpreted similarly. In summary, disagreement measures how many of the base classifiers' predictions contradict each other. Disagreement values can be any value within the range $[0,1]$, where the higher the value, the more different the two classifiers. To calculate the diversity of the ensemble, we find the average pairwise disagreement between all pairs of its constituent members. An ideal ensemble would be the one that yields a high MCC and a high disagreement between its constituent classifiers. However, as mentioned at the beginning of this section, this is hard to achieve in practice, thus the need of using a MOEA to strike an optimal trade-off between these two competing goals.

7.3.3 Genetic Operators

Since there are three arrays in the representation (Section 7.3.1), the crossover and mutation happen in three parts, each of which with the respective operators adapted for the type of genes.

MEG uses a Single Point Crossover operator with 95% probability for all three representation arrays. This crossover operator takes two parents and combines their genes to generate two children. First, it chooses an index from the parents' genes array at random. It cuts the parents on that index into two parts, left and right. It then combines the left part of parent one with the right part of parent two to generate the first child, and the right genes of parent one with the left genes of parent two to create the second child.

After crossover, the children undergo mutation with a lower probability. Since the classifiers array (bit array) and parameters array (double array) have 15 indexes, the mutation probability is set to 0.07. Thus, it is expected that each child will have one of its bit/double gene mutated. For the ensemble strategy (int array) with

one index, the probability is set to 0.25. Hence, it is expected that the ensemble aggregation strategy is mutated once in every four children. MEG uses a Bit Flip Mutation operator for the classifier array, and a Simple Random Mutation operator for the parameter and strategy arrays. The former simply flips the bit by changing it to 1 if the gene is 0, or to 0 if the gene is 1. The latter generates a random number for a mutated gene.

We set the population size to 100 and the stopping condition to 10,000 fitness evaluations. This means that the evolutionary process runs for 100 generations. In the end, all the non-dominated solutions are returned. In our experiments, the same algorithm, operators, and configuration are used with DIVACE (Sections 7.4 and 7.5).

7.3.4 Implementation Aspects

MEG was implemented using jMetal 5.10,[2] a widely-used framework for realising MOEAs in Java. jMetal provides multiple MOEAs, and mutation and crossover operators, which suits well the purpose of MEG. We implement MEG by using the NSGA-II [340] algorithm. We chose NSGA-II due to its popularity, easily adaptable nature with jMetal, performance, and because it has shown to be a robust algorithm in the SBSE literature [341]. For the ML implementation, we use the Weka 3.9 framework.[3] Our source-code is publicly available online, along with a replication package, containing the datasets, the raw results and the scripts we realised to analyse them [335].

7.4 Experimental Design

In order to validate the effectiveness of MEG, we benchmark it against base classifiers, the state-of-the-art ensemble stacking for defect prediction as proposed by Petrić et al. [202], and the state-of-the-art Pareto-ensemble generation approach DIVACE [207], in the context of Cross-Version Defect Prediction (CVDP). Specifically, we aim at answering three research questions (RQs) as detailed below.

[2] https://github.com/jMetal/jMetal
[3] https://www.cs.waikato.ac.nz/ml/weka/

7.4.1 Research Questions

ML base classifiers have been widely proposed and considered as benchmarks in previous defect prediction studies. Therefore, our first research question investigates and compares the performance of MEG with that of traditional ML base classifiers. We consider this a "sanity check" given that any newly proposed model that cannot generally outperform base classifiers cannot be considered a scientific advancement in the state-of-the-art. To this end we ask:

RQ1 – MEG Vs. Base Classifiers: How does MEG compare to traditional base classifiers?

If MEG passes this sanity check, then we investigate whether our proposed approach can also outperform the aggregation of multiple ML base classifiers. We therefore compare the performance of MEG with that of existing ensemble for defect prediction. In order to verify this, we pose our second research question:

RQ2 – MEG Vs. Traditional Ensemble: How does MEG compare to ensemble stacking?

In particular, as nobody has previously proposed the use of MOEAs to generate ensemble models for defect prediction, we answer the above question by verifying whether MEG actually performs better than stacking, which has proven to be better than other ensemble approaches in previous defect prediction work [202].

Since our approach, MEG, is also novel with respect to existing multi-objective evolutionary approaches designed to generate ensemble for general-purpose classification problems, we also benchmark it with the state-of-the-art Pareto-ensemble generation technique DIVACE, thereby motivating our last research question:

RQ3 – MEG Vs. Pareto-ensemble: How does MEG compare to Pareto-ensemble generation, more specifically DIVACE, the state-of-the-art ensemble generation technique based on MOEAs?

In the following subsections we describe the techniques and datasets used as a benchmark to answer RQs 1–3, and the validation and evaluation criteria we employed to assess the performance of the prediction models we compare.

7.4.2 Benchmark Techniques

7.4.2.1 Base Classifiers

As a baseline benchmark, we compare the ensemble produced by MEG to the single based learners, as the purpose to build an ensemble is to achieve prediction performance which are at at least comparable or superior to individual classifiers [202, 342, 343]. We use the same four base classifiers used by Petrić et al. [202], namely Naïve Bayes (NB)[318], k-Nearest Neighbour (k-NN), Support Vector Machine (SVM) [270], and Decision Tree (DT) [344]. To insinuate a fair comparison with the ensembles, these are the same classifiers used by both MEG and DIVACE.

Based on the study by Petrić et al. [202], we train the base classifiers with the following parameters and select the configuration with the best training/validation MCC, as explained in Section 7.4.4:

i) NB – NB with conventional distribution estimation and NB-K with kernel density estimator;
ii) DT – pruning confidence values of 0.25, 0.2, 0.15, 0.1, and 0.05;
iii) k-NN – number of neighbours (k) of 3, 5, and 7;
iv) SVM – C of 1, 10, 25, and 50

7.4.2.2 Stacking Ensemble

The Stacking ensemble proposed by Petrić et al. [202] is composed by at most 15 base classifiers, each of which has a pre-defined configuration. The best stacking composition is the best combination of 2 to 15 base classifiers with the best training/validation MCC value. Moreover, since the other algorithms and MEG also use MCC, this was the natural choice for a fair comparison. To find the best Stacking combination, we followed the procedure proposed by Petrić et al. [202]. We first train the 15 base classifiers and order them in descending order according to their MCC. We then build 14 combinations from sizes 2 to 15, incrementally adding the next best base classifier, and setting the best classifier as the meta-classifier. We then select the best combination for a given program obtained on the train/validation set

and use it during the testing phase, as detailed in Section 7.4.

7.4.2.3 Pareto-ensemble Generation: DIVACE

DIVACE is a Multi-objective Genetic Algorithm that uses the Pareto-ensemble approach to search for optimal ensemble classifiers, i.e., it joins the resulting non-dominated predictors into one single ensemble. DIVACE had never been assessed for defect predition in prior studies. It was originally proposed for a regression task (more specifically for building neural networks), but it has served as a reference since then for other ensemble generation techniques. It makes use of specialised continuous operators to generate weights and drive solutions towards specific and relevant parts of the search space. For more details on the approach we refer the reader to the original paper by Chandra et al. [207].

DIVACE, as originally proposed, uses the Negative Correlation Learning (NCL) measure for diversity, and MCC for the accuracy objective. DIVACE evolves the same four base classifiers as MEG by using NSGA-II, to allow for a fair comparison. Since DIVACE does not automatically select the aggregation strategy during the evolutionary process, one has to manually experiment with different aggregation strategies to identify the most suitable for the problem at hand. We therefore tried different strategies (i.e., Majority Voting, Stacking, and Average Rule) and found that Majority Voting generally generates the best results for the investigated datasets.

At the end of the evolution process, DIVACE aggregates all non-dominated solutions based on NCL and MCC into a single Majority Voting ensemble, thus producing a single optimal solution.

7.4.3 Datasets

In our empirical study we used the corpus made publicly available by Yatish et al. [345]. This data has been collected using a realistic approach based on two main criteria: (i) the use of an Issue Tracking System with the availability of high numbers of closed or fixed issue reports; and (ii) issue reports collected for each studied system are traceable back to the code. Following this procedure has resulted

in obtaining less erroneous defect counts and hence representing a more realistic scenario of defective module collection.

We experiment with eight software systems considering three releases for each system, as listed in Table 7.2, which has been shown to be preferable to other types of validation such as 10-fold cross validation or bootstrapping [346, 313].

7.4.4 Validation Criteria

We validate the effectiveness of the proposed approach using the CVDP scenario. As previously described in Chapter 2, for each of the software systems, we train on one release and test on a later version, i.e., we train on version v_x and test on version v_y, where $x < y$, as done in previous work [313].

Training and Validation: For all algorithms included in this study, to prevent overfitting during the training phase, we apply an internal bootstrapping procedure with replacement using 80% of the data for training and 20% for validation (as suggested by Tantithamthavorn et al. [347]). We use this bootstrapping procedure to cater for the randomness of the data as confounding factors for the results. During the MOEA evolution, each candidate solution's fitness is the result of the predictions over the unseen 20% validation. We perform the same procedure to train the Stacking (state-of-the-art ensemble), DIVACE, and all traditional classifiers.

For each of the approaches investigated herein, the best performing solutions, are selected based on MCC on the validation data, and then used to train the final model on the whole training data. Such a model is then evaluated on an unseen test set. Given that MEG generates a set of ensembles (rather than a single solution as done by the base classifiers and DIVACE), we choose the solution in the Pareto-front with the highest MCC value on the validation data as MEG's final solution. After a preliminary but comprehensive assessment, such a heuristic also showed to be the best one among other investigated ones such as selecting the most diverse solution, the one in between, or a random solution. We provide a summary of this preliminary assessment in our online artefact.

Testing: For each of the systems and prediction techniques considered herein, we build two prediction models. One by using the first available version in Yatish's

corpus [345] as training data, given that recent work has proven that it is effective to use early defect data for training purposes [316, 348], and one by using the penultimate available version, as done in most of previous CVDP work. Both models are then tested on the latest version available in the corpus, which is completely unseen/untouched during the training process. By following this procedure, we ensure that the training and choice of best solution in the previous phases reflects a real-world scenario where the engineer does not possess information about the software components for which they are trying to predict defects. Moreover, the versions we used as train and test sets are not immediately subsequent releases nor are they the system's most recent ones. In addition, there is always a window of at least five months between these releases. This reduces the likelihood of the snoring effect (i.e., when defects are discovered several releases after their introduction, which can cause a class to be labeled as defect-free while it is not, and is, therefore referred to as "snoring") or unrealistic labelling as described in previous studies [314, 315, 316].

Finally, to mitigate for the variability induced by the use of stochastic algorithms, we run the above procedure 30 times [334, 349]. In order to maintain a fair comparison among all algorithms, we ensured that the bootstrapping procedure samples the same data to train each compared approach within a same run by using a same seed for all, yet different runs use different seeds.

7.4.5 Evaluation Criteria

We use MCC to evaluate the prediction performance of the models given that we do not target a specific business context [258, 295], and, as explained in Section 7.3.2, MCC is a comprehensive measure, which provides a full picture of the confusion matrix by assessing all its aspects equally. It is also not sensitive to highly imbalanced data and is widely used in the defect prediction and machine learning literature [258, 99, 215].

In order to show whether there is any statistical significance between the results obtained by the models, we perform the Mann-Whitney U [350] setting the confidence limit, α, at 0.05 and applying the Bonferroni correction (α/K, where K is

Table 7.2: Total number of modules and percentage of faulty components for each of the datasets used in our empirical study. We used the two lowest version numbers (one at time) for training the prediction models and the highest one for testing them.

Dataset	No. of modules (faulty %)	Dataset	No. of modules (faulty %)
activemq-5.0.0	1884 (15.55%)	hive-0.9.0	1560 (11.28%)
activemq-5.3.0	2367 (10.90%)	hive-0.10.0	1560 (11.28%)
activemq-5.8.0	3420 (6.02%)	hive-0.12.0	2662 (8.00 %)
derby-10.2.1.6	1963 (33.67%)	jruby-1.1.0	731 (11.9%)
derby-10.3.1.4	2206 (30.33%)	jruby-1.5.0	1131 (7.25%)
derby-10.5.1.1	2705 (14.16%)	jruby-1.7.0	1614 (5.39%)
groovy-1.5.7	757 (3.43%)	lucene-2.3	805 (24.35%)
groovy-1.6-B1	821 (8.53%)	lucene-3.0	1337 (11.59%)
groovy-1.6-B2	884 (8.60%)	lucene-3.1	2806 (3.81%)
hbase-0.94.0	1059 (20.59%)	wicket-1.3.0-B1	1763 (7.37%)
hbase-0.95.0	1669 (22.95%)	wicket-1.3.0-B2	1763 (7.37%)
hbase-0.95.2	1834 (26.34%)	wicket-1.5.3	2578 (4.07%)

the number of hypotheses) when multiple hypotheses are tested. Unlike parametric tests, the Mann-Whitney U raises the bar for significance, by making no assumptions about underlying data distributions. Moreover, we used effect size to assess whether the statistical significance has practical significance effect size [334]. To this end we use the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure, as it is recommended to use a standardised measure rather than a pooled one like the Cohen's d when not all samples are normally distributed [334], as in our case. The $\hat{A}_{12}$ statistic measures the probability that an algorithm A yields greater values for a given performance measure M than another algorithm B, based on the following equation:

$$\hat{A}_{12} = (R_1/m - (m+1)/2)/n \tag{7.3}$$

where R_1 is the rank sum of the first data group we are comparing, and m and n are the number of observations in the first and second data sample, respectively. Values between $(0.44, 0.56)$ represent negligible differences, values between $[0.56, 0.64)$ and $(0.36, 0.44]$ represent small differences, values between $[0.64, 0.71)$ and $(0.29, 0:44]$ represent medium differences, and values between $[0.0, 0.29]$ and $[0.71, 1.0]$ represent large differences.

7.4.6 Threats to Validity

Threats to External Validity: As it is the case for most software engineering empirical studies, the datasets (i.e., programs) and prediction approaches used in this work might not be fully representative of the population, which is a threat to the generalisation of our conclusions. The datasets we use are all based on open-source Java projects [345], which limits the generalisability of our results to proprietary software or projects written in other languages. In order to mitigate this threat, we used a variety of programs of different sizes and imbalanced nature, and which have been used in previous work [345]. Another external threat is linked to the choice of techniques which MEG was benchmarked against. We benchmarked MEG against the most relevant related work in the literature: traditional base ML classifiers, widely used in defect prediction work [23], the only multi-objective ensemble proposed thus far for defect prediction [202] and the state-of-the-art evolutionary ensemble DIVACE. Moreover, we implemented both DIVACE and the Stacker approach with Weka which is the tool used by Petric et al. [202] to closely reproduce the state-of-the art multi-objective ensemble DP [202]. This allowed us to share as many aspects as possible between all approaches and reduce any confounding factors that could arise from different implementations or configurations [351, 352]. While we believe that it would be interesting to carry out a large-scale empirical study comparing other ensemble approaches which have been proposed in the more general ML literature but have not been used for defect prediction (including for example auto-sklearn and AutoFolio [75]), this is out of the scope of our work and it deserves an investigation on its own right due to both the different aim, scope and technical challenges involved. For example, in order to compare various ensembles provided in ML tools other than Weka, such as auto-sklearn or AutoFolio which are currently available only in Python, it might require one to re-implement all approaches in a same tool to ensure a fair empirical comparison as using different ML tools might lead to different results [351, 352].

Threats to Internal Validity: The most prominent threat to internal validity relates to the correctness of our own implementations of DIVACE and the Stack-

ing build procedure. We followed all the details provided in the reference papers [207, 202], but it is still possible that some differences were introduced. In order to mitigate this threat, all authors made sure to rule out any ambiguity by verifying the code and comparing it to the reference papers. In occasions where we could not agree on the way to resolve an ambiguity, we opted for design decisions which allow us to compare all the approaches on a level playing field.

Threats to Construct Validity: DIVACE and MEG are stochastic by nature. Consequently, the results of these techniques may differ from one run to another, causing unwanted variations in our experimental analysis. In order to mitigate this variability, and to provide a robust analysis, both DIVACE and MEG were executed 30 times and the median MCC values were reported (as opposed to means given that the latter is known to be more susceptible to outliers [333]), as suggested by best practices for the assessment of randomised optimisation algorithms and prediction systems [334, 349, 258, 333]. Moreover, we perform bootstrapping during training by using a same random seed for all techniques used across all our experiments. This ensures that any variation or difference in results is due to the nature of the techniques themselves and not due to a different random data sampling. Finally, we used a robust and unbiased measure, such as MCC, to evaluate the prediction capabilities of all the approaches investigated herein, and performed statistical tests, including both hypothesis testing and effect size, by carefully checking all the required assumption, such that our conclusions could be backed up by scientifically sound evidence [258].

7.5 Results

In this section, we present and discuss the results of all research questions addressed in our work.

7.5.1 Answer to RQ1 – MEG vs. Base Classifiers

As a sanity check, we compare the performance of MEG to that of traditional classifiers (i.e., DT, NB, k-NN and SVM) known to perform well for the task of defect prediction.

Table 7.3: MCC values obtained on the test set by the base learners (NB, DT, k-NN, SVM), MEG, DIVACE, over 30 runs.

Version (training data)	MEG	DIVACE	Stacking	NB	DT	k-NN	SVM
activemq-5.0.0	0.29	0.20	0.24	0.29	0.15	0.25	0.30
derby-10.2.1.6	0.40	0.25	0.22	0.37	0.24	0.16	-0.05
groovy-1.5.7	0.23	0.31	0.22	0.21	0.31	0.33	0.24
hbase-0.94.0	0.30	0.27	0.25	0.07	0.27	0.25	0.31
hive-0.9.0	0.20	0.19	0.10	0.21	0.19	0.07	0.22
jruby-1.1	0.23	0.28	0.16	0.22	0.30	0.29	0.20
lucene-2.3.0	0.18	0.13	0.12	0.18	0.13	0.11	0.12
wicket-1.3.0-B1	0.08	0.20	0.13	0.18	0.20	0.16	0.15
activemq-5.3.0	0.31	0.27	0.27	0.28	0.24	0.23	0.29
derby-10.3.1.4	0.39	0.31	0.31	0.37	0.30	0.28	0.38
groovy-1.6-B1	0.20	0.26	0.27	0.21	0.26	0.47	0.26
hbase-0.95.0	0.30	0.18	0.19	0.33	0.26	0.20	0.31
hive-0.10.0	0.28	0.27	0.27	0.18	0.26	0.19	0.23
jruby-1.5.0	0.19	0.28	0.10	0.17	0.28	0.27	0.17
lucene-3.0.0	0.06	0.05	0.09	0.19	0.04	0.14	0.00
wicket-1.3.0-B2	0.15	0.00	0.00	0.19	0.10	0.11	0.12

Table 7.3 shows that, out of the four classifiers investigated, NB achieves the best performance overall. When compared to MEG, results show that MEG performs similarly or better than NB in 69% of the cases with 73% of those cases being statistically significant and having a large effect size.

Our results also show that MEG is able to generate ensembles that perform similarly or better than traditional classifiers in 45 (70%) out of the 64 cases investigated. Moreover, MEG strictly outperforms traditional classifiers in 39 out of the 64 (61%) cases considered with the difference in 90% of those cases being statistically significant.

> **Answer to RQ1:** MEG generates statistically better or at least equivalent ensembles to traditional classifiers in 70% of the cases.

7.5.2 Answer to RQ2 – MEG vs. Traditional Ensemble

When compared to the state-of-the-art Stacking ensemble proposed by Petric et al. [353], results show that MEG is able to generate similarly or enhance the predictions in 14 out of the 16 cases under study (88%) with the difference in 86% of those cases (12 out of 14) being statistically significant with a large effect size.

To understand the performance of the ensemble, we analyse the non-dominated

Table 7.4: Mann-Withney U pair-wise test results / Vargha-Delaney $\hat{A}_{12}$ effect sizes obtained comparing MEG with DIVACE, Stacking, and base classifiers (NB, DT, k-NN, SVM). $\hat{A}_{12}$: Large – L; Medium – M; Small – S; Negligible – N. Cells highlighted in blue (p-value < 0.05 and effect size > 0.5) indicate that MEG is statistically significantly better than the algorithms in the corresponding columns. Cells highlighted in orange (p-value < 0.05 and effect sizes < 0.5) indicate that MEG is significantly statistically worse. The last three rows show the number of times MEG yields better, equivalent, and worse results, respectively.

Version (training data)	DIVACE	Stacking	NB	DT	k-NN	SVM
activemq-5.0.0	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 0.0 (L)
derby-10.2.1.6	<0.01 / 0.8 (L)	<0.01 / 0.83 (L)	<0.01 / 0.8 (L)	<0.01 / 0.83 (L)	<0.01 / 0.83 (L)	<0.01 / 1.0 (L)
groovy-1.5.7	<0.01 / 0.14 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.03 (L)
hbase-0.94.0	<0.01 / 0.93 (L)	<0.01 / 0.93 (L)	<0.01 / 1.0 (L)	<0.01 / 0.93 (L)	<0.01 / 0.93 (L)	0.058 / 0.37 (S)
hive-0.9.0	0.197 / 0.6 (S)	<0.01 / 1.0 (L)	<0.01 / 0.0 (L)	1.0 / 0.5 (N)	<0.01 / 1.0 (L)	<0.01 / 0.0 (L)
jruby-1.1	<0.01 / 0.24 (L)	0.638 / 0.53 (N)	0.638 / 0.53 (N)	<0.01 / 0.03 (L)	<0.01 / 0.03 (L)	0.638 / 0.53 (N)
lucene-2.3.0	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 0.83 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)
wicket-1.3.0-B1	<0.01 / 0.25 (L)	0.64 / 0.47 (N)	0.345 / 0.43 (N)	<0.01 / 0.0 (L)	0.64 / 0.47 (N)	0.64 / 0.47 (N)
activemq-5.3.0	<0.01 / 0.83 (L)	<0.01 / 1.0 (L)	0.151 / 0.6 (S)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	0.151 / 0.6 (S)
derby-10.3.1.4	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 0.77 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 0.77 (L)
groovy-1.6-B1	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)
hbase-0.95.0	<0.01 / 0.96 (L)	<0.01 / 1.0 (L)	<0.01 / 0.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	0.057 / 0.37 (S)
hive-0.10.0	<0.01 / 0.92 (L)	<0.01 / 0.9 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)	<0.01 / 1.0 (L)
jruby-1.5.0	<0.01 / 0.12 (L)	<0.01 / 1.0 (L)	<0.01 / 0.9 (L)	<0.01 / 0.0 (L)	<0.01 / 0.0 (L)	<0.01 / 0.9 (L)
lucene-3.0.0	0.587 / 0.54 (N)	<0.01 / 0.07 (L)	<0.01 / 0.0 (L)	<0.01 / 0.8 (L)	<0.01 / 0.0 (L)	<0.01 / 1.0 (L)
wicket-1.3.0-B2	<0.01 / 0.99 (L)	<0.01 / 1.0 (L)	<0.01 / 0.1 (L)	<0.01 / 0.93 (L)	<0.01 / 0.93 (L)	<0.01 / 0.93 (L)
MEG is better	9	12	8	10	10	7
MEG is equivalent	2	2	3	1	1	5
MEG is worse	5	2	5	5	5	4

solutions generated by MEG. We discovered that Stacking, as an aggregation strategy, was selected in only 3% of the cases; the least common type of generated ensembles. On the other hand, we found that the most selected ensemble strategy by MEG was Weighted Majority Voting, which simply counts the best classifier's vote twice and the other classifiers once. This strategy was selected in 78% of the non-dominated ensembles.

> **Answer to RQ2:** MEG is able to generate ensembles that perform similarly or enhance the state-of-the-art Stacking model in 88% of the cases. The difference in results obtained by MEG is statistically significantly better in 75% of the cases with a large effect size.

7.5.3 Answer to RQ3 – MEG vs. Pareto-ensemble

We also compare the ensembles generated by MEG to those generated by DIVACE. Results show that MEG outperforms DIVACE in 11 out of 16 cases studied (69%).

This finding is also supported by the statistical tests showing that, in 82% of these cases, the difference is statistically significantly better and the effect size is large.

We further investigated the solutions generated by both these approaches and we found some interesting behavioural differences when it comes to the nature of the solutions. While MEG produced solutions consisting of a more heterogeneous set of classifiers with a mean and median number of classifiers equal to 2.41 and 2, respectively, DIVACE's behaviour was more homogeneous. In most cases, DIVACE generated non-dominated solutions that were retained across generations resulting in ensembles comprised of 100 classifiers, out of which ≈98 were entirely of a homogeneous nature. MEG, on the other hand, generated a smaller set of heterogeneous ensembles with the largest number of classifiers being equal to 11. This shows that an ensemble consisting of a small heterogeneous set of classifiers yields better results than one comprising of a large set of homogeneous classifiers.

> **Answer to RQ3**: MEG generates ensembles that are similar or statistically better to those produced by DIVACE in 69% of the cases, with the difference in 82% of them being statistically significant.

7.5.4 Final Remarks

MEG yielded statistically significantly better results than the other algorithms in most of the cases. To be precise, if we consider all comparisons between MEG and the other techniques, MEG is able to generate ensembles producing similar or more accurate predictions than those produced by the other approaches in 73% of the cases (with favourable large effect sizes in 80% of them). In the minority of the cases (27%) where MEG does not statistically significantly outperform all other approaches we observed that the number of faulty modules in the training data is lower than 10%, or the training data consists of less than 1,000 modules. For these cases, we also observed that MEG's ensembles achieve better MCC values than those produced by traditional ML and Pareto-ensemble in the training phase, thus suggesting that its ensembles may overfit when the training data is small or contain few of defective instance.

The positive results obtained by using MEG in the majority of the cases investigated herein, suggest its use could be convenient for several reasons.

First, MEG removes from the engineer hands the burdensome, error-prone, and time-consuming task of building, or even selecting, an ensemble/classifier. Since it is all automated, the engineer can simply use MEG to generate robust ensembles.

Second, MEG can generate ensembles that can yield good predictions for future versions of a software. This is shown by the results achieved by the generated ensembles when trained on the first available software version, and tested on the latest.

Third, MEG can be extended and adapted to the engineers' needs. In this work, we focused on MCC as a measure of performance accuracy given that it is a comprehensive and balanced measure [258]. However, there might be cases where it may be more important to minimize one type of classification error. For example, an engineer might want to reduce the number of false negatives to the very minimum when predicting defects for mission and safety critical software systems (e.g., software controlling autopilot, medical devices) in which even a single failure can have serious adverse effects [258]. In this case, MEG can be extended to be guided by a fitness function devised for a specific goal.

Regarding the qualitative results, we analysed possible correlations between the results of the generated ensembles and possible imbalance in the data with respect to the percentages of defective and non-defective classes. However, we did not find any clear pattern. It seems that the results of the ensembles generated by MEG are more influenced by the individual predictions of each base classifier. For instance, when predicting defects for *Derby*, SVM showed very good results during training (training and validation MCC values higher than 0.7), but very poor performance during testing (negative MCC value). When analysing the ensembles generated by MEG for this program, we found out that when SVM was added to the ensemble, the resulting MCC values would drop from around 0.40 to 0.02. The drop is noticeable and undeniably significant, but luckily it is an exception, not the rule.

In our experiments, each of MEG's independent runs took from 6–48 hours, which is greater than the time needed to perform the automated hyper-parameter tuning of the state-of-the-art algorithms (2–24 hours). While the time taken by the state-of-the-art is generally lower as it includes only training and possibly tuning, MEG's running time is higher because it also encompasses the automatic selection of classifiers, voting strategies, and tuning of parameters. This procedure is analogous to the experimentation, tuning, and selection of many different algorithms that are usually done manually by the engineers. If we compare the time and expertise that would be required by the engineer to perform these tasks, using MEG saves time and effort overall. This is a common trade-off for automated search based approaches, as observed in previous work [354].

7.6 Conclusions and Future Work

In this chapter, we proposed MEG, a Multi-objective Ensemble Generation algorithm for defect prediction.

MEG is a novel technique, which, unlike previous MOEA-based ensemble generation, evolves a population of ensembles (i.e., *whole-ensemble generations*) rather than a population of base classifiers (i.e, *Pareto-ensemble generation*). Furthermore, MEG relies on both accuracy and diversity of ensembles to guide the search towards robust ensembles, which has not been previously evaluated in the context of defect prediction.

In our large-scale empirical study involving a total of 24 software versions from 8 different open-source projects, and both baselines and state-of-the-art benchmarks, we show that MEG is able to generate ensembles that achieve comparable or statistically significantly more accurate predictions for 73% of the comparisons, with large effect sizes in 80% of these cases.

These results show that our novel *whole-ensemble generation* approach, MEG, not only advances the state-of-the-art of ensemble in defect prediction (where multi-objective evolutionary ensemble had never been investigated) but it is also more effective than *Pareto-ensemble generation* algorithm. In addition, the benefits of

MEG do not only lie on the higher accuracy of the generated ensembles, but also on the benefits of having an approach that automatically searches for an optimal design: This spares the engineer from the tedious, time-consuming, and error-prone activity of manually designing and experimenting.

The proposal of a novel *whole-ensemble generation*, such as MEG, opens up a variety of avenues for future work, including but not limited to the following:

- Investigating the characteristics of the optimal ensembles built by MEG further, as well as the characteristics of the bugs that are most/least predictable.
- Investigating the effectiveness of *alternative solution representations* considering additional learners and aggregation strategies.
- Investigating the *effect of using other measures as fitness function* to guide the ensemble evolution, in combination or in alternative to the measures explored herein. These include the use of other diversity measures, and performance measures such as the classification stability [355] and the effort required for source code inspection [356, 357].
- Investigating *MOEAs other than NSGA-II* (e.g., MOEA/D, IBEA, MOCell, SPEA2) in order to assess to what extent the effectiveness of MEG varies depending on the underlying multi-objective algorithm [358].
- Investigating *Deep-Learning (DL) in combination with* M*EG*, to assess if this would further increase the ensemble prediction performance. DL approaches have recently been proposed to extract from the source code features which are subsequently used as an input (together or in alternative to traditional code metrics) to train a base binary classifier [359] for defect prediction. Our proposed approach, MEG, uses traditional code metrics yet it combines multiple different binary classifiers automatically searching for the best configuration. These approaches are different in nature: DL augments the training data, while MEG searches for an optimal configuration of multiple binary classifiers. It would be interesting in future work to explore such a combined use.

Chapter 8

Conclusion

The work presented in this thesis contributes to the advancement of the research in predictive models for software engineering, specifically in the areas of software development effort estimation and software defect prediction. We have proposed two novel methods, LFM and MEG, to aid human-machine collaboration towards achieving accurate effort and defect prediction, respectively, and to facilitate the adoption of automated models in practice. Moreover, the work in this thesis aimed at increasing the robustness of empirical studies by investigating crucial factors that can undermine scientific conclusion stability, namely empirically evaluating the magnitude of the threat posed by using different evaluation measures to the scientific conclusions that have been drawn on defect prediction models as well as revealing a new threat posed by the use of different ML libraries for the replicability of software estimation studies. We envisage that the contribution made in this thesis will help ameliorate the gap between humans and machines. In each of the chapters, we outlined specific future work, whereas herein we discuss the most pressing challenges that lie ahead for applying prediction models in practice.

More work is needed to further incorporate human factors as input features into automated prediction models. After all, software development is a knowledge-intensive and social activity [360].

For example, factors that relate to the development team such as the number of developers involved in producing the system and their level of expertise are good proxies for understanding and predicting human expert errors. However, during the

study on LFM, we have learnt that data about human-factors is seldom available for effort estimation and gathering the data we used in the study was the most demanding task. Further collaborations with industry is needed in order to encompass a variety of human factors. However these types of features are not easy to access for researchers and practitioners. In fact, small to medium companies cannot even afford to collect them in most cases while, big company can collect them but cannot share them due to high confidentiality.

Future work on prediction models should also aim to be more user-centered, i.e., the end user should play a more active role in achieving automated predictive models to be exploited in practice. The threat posed by the use of different ML libraries for software estimation studies calls for a number of mitigations for different stakeholders, e.g. researchers need to report the library and version used in their study to allow for replicability, and ML library developers need to improve the documentation as well as the transparency of the source code. Whereas, the empirical evidence we provided on the magnitude threat posed by using different evaluation measures encourages the community to reflect on the selection of appropriate evaluation measures that fit the study's specific aim, model and data, and also to take into account the given software application domain and business goal. MEG, for example, allows the user to specify their own objective function based on their business needs. We show that it is possible to fully automate the construction of accurate defect prediction ensemble, and to bridge the gap between the prediction modeler and the decision maker by using multi-objective evolutionary computation. Such an approach opens up the possibility of training prediction models optimal with respect to multiple business goals. In this work we have exposed the tip of the iceberg, future work can take into account the inclusions of effort-aware measures to guide the search for optimal models, as well as, Explainable AI measures to let the users make a more informed decision about the models to adopt in practice. In fact, we have been assisting in a recent application of Explainable AI (xAI) in Software Engineering activities [361], including software effort estimation [362] and defect prediction [363, 364].

Appendix A

Mathematical Formulation of Linear Programming

In this appendix we explain the mathematical formulation of the Linear Programming model we used in RQ2 to predict the MisestimationMagnitude.

Linear Programming (LP) [365] aims to achieve the best outcome from a mathematical model with a linear objective function subject to linear equality and inequality constraints. The feasible region is given by the intersection of the constraints and the Simplex (linear programming algorithm) is able to find a point in the polyhedron where the function has the smallest value (minimisation) in polynomial time.

Here, we generalize the model proposed for the effort estimation by Sarro and Petrozziello [29].

In the original implementation, the model is subject to an inequality constraint imposing that the value estimated for each of the observations in the training set has to fall in R_0^+

Here, we remove the inequality constraints allowing the model to use both positive and negative feature values as well as to optimize for both positive and negative values, as follows:

$$\begin{aligned} \text{minimise} \quad & \sum_{i=1}^{n} |\sum_{j=1}^{m} a_{ij} x_j - ActualValue_i| \\ & x_j \text{free}, \qquad\qquad j = 1, ..., m \end{aligned} \tag{A.1}$$

where a_{ij} represents the coefficient of the j^{th} feature for the i^{th} project, x_j is the value of the j^{th} feature, and $ActualValue_i$ is the actual effort of the i^{th} project.

Due to the non-linearity of the absolute value function, the above model has been linearised as follows:

$$\begin{aligned}
\text{minimise} \quad & \sum_{i=1}^{n} t_i \\
\text{subject to} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}x_j - ActualValue_i - t_i \leq 0 \\
& \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}x_j - ActualValue_i + t_i \geq 0 \\
& x_j \text{ free}, \qquad j = 1,...,m \\
& t_i \text{ free}, \qquad i = 1,...,n
\end{aligned} \tag{A.2}$$

Let $X_i, \forall i$ be the part of Eq. (A.1) wrapped in the absolute value. $\forall i$, the slack variable t_i and the following two constraints have been added to the model:

$$X_i \leq t_i$$

$$-X_i \leq t_i$$

Therefore we can have one of the following cases:

$X_i > 0$: The second constraint, $-X_i \leq t_i$, is always fulfilled as $-X_i$ is negative and t_i is implicitly ≥ 0. Since t_i is minimised by the objective function and $0 \leq X_i \leq t_i$, the first constraint, $X_i \leq t_i$, is satisfied and t_i is $abs(X)$.

$X_i < 0$: The first constraint, $X_i \leq t_i$, is always fulfilled as X_i is negative and t_i is implicitly ≥ 0. Since t_i is minimised by the objective function and $0 \leq -X_i \leq t_i$, the second constraint, $-X_i \leq t_i$, is satisfied and t_i is $abs(X)$.

$X_i = 0$: Both constraints are always fulfilled since t_i is implicitly ≥ 0. Since t_i is minimised by the objective function, $0 = X_i = t_i$. So t_i is $abs(X)$.

Bibliography

[1] Tim Menzies and Thomas Zimmermann. Software analytics: so what? *IEEE Software*, 30(4):31–37, 2013.

[2] Federica Sarro, Alessio Petrozziello, and Mark Harman. Multi-objective software effort estimation. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 619–630. IEEE, 2016.

[3] Ekrem Kocaguneli, Tim Menzies, and Jacky W Keung. On the value of ensemble effort estimation. *IEEE TSE*, 38(6):1403–1416, 2011.

[4] Kjetil Moløkken and Magne Jørgensen. A review of surveys on software effort estimation. 2003.

[5] Jianfeng Wen, Shixian Li, Zhiyong Lin, Yong Hu, and Changqin Huang. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, 54(1):41–59, 2012.

[6] Anna Corazza, Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, Federica Sarro, and Emilia Mendes. Using tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering*, 18(3):506–546, 2013.

[7] Filomena Ferrucci, Carmine Gravino, Rocco Oliveto, and Federica Sarro. Genetic programming for effort estimation: an analysis of the impact of different fitness functions. In *2nd International Symposium on Search Based Software Engineering*, pages 89–98. IEEE, 2010.

[8] Federica Sarro, Filomena Ferrucci, and Carmine Gravino. Single and multi objective genetic programming for software development effort estimation. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1221–1226. ACM, 2012.

[9] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[10] Federica Sarro, Mark Harman, Yue Jia, and Yuanyuan Zhang. Customer rating reactions can be predicted purely using app features. In *Procs. of RE*, pages 76–87. IEEE, 2018.

[11] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *IEEE transactions on software engineering*, 37(3):356–370, 2010.

[12] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[13] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[14] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341, 2016.

[15] Qinbao Song, Yuchen Guo, and Martin Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE TSE*, 45(12):1253–1269, 2018.

[16] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.

[17] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[18] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 215–226. IEEE, 2017.

[19] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. Improving code review by predicting reviewers and acceptance of patches. *Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006)*, pages 1–18, 2009.

[20] Lionel C Briand and Isabella Wieczorek. Resource estimation in software engineering. *Encyclopedia of software engineering*, 2002.

[21] Adam Trendowicz and Ross Jeffery. Software project effort estimation. *Foundations and Best Practice Guidelines for Success, Constructive Cost Model, COCOMO*, pages 277–293, 2014.

[22] Steve McConnell. *Software estimation: demystifying the black art*. Microsoft press, 2006.

[23] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software

engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[24] Armin Najafi, Peter C Rigby, and Weiyi Shang. Bisecting commits and modeling commit risk during testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 279–289, 2019.

[25] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. Predicting pull request completion time: a case study on large scale cloud services. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 874–882, 2019.

[26] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 91–100. IEEE Press, 2019.

[27] Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction. *EMSE*, 17(1):1–17, 2012.

[28] Jacky Keung, Ekrem Kocaguneli, and Tim Menzies. Finding conclusion stability for selecting the best effort predictor in software effort estimation. *Automated Software Engineering*, 20:543–567, 2013.

[29] Federica Sarro and Alessio Petrozziello. Linear programming as a baseline for software effort estimation. *ACM transactions on software engineering and methodology (TOSEM)*, 27(3):1–28, 2018.

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

[31] Kjetil Molkken and Magne Jörgensen. A review of surveys on software effort estimation. In *Proc. of ISESE'03*, pages 223–230, 2003.

[32] Tanja M Gruschke and Magne Jørgensen. The role of outcome feedback in improving the uncertainty assessment of software development effort estimates. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):1–35, 2008.

[33] Muhammad Usman, Emilia Mendes, and Jürgen Börstler. Effort estimation in agile software development: a survey on the state of the practice. In *Proceedings of the 19th international conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[34] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83:2–17, 1 2010.

[35] Mauno Vihinen. How to evaluate performance of prediction methods? measures and their interpretation in variation effect analysis. In *BMC genomics*, volume 13, 2012.

[36] Mohamed Bekkar, Hassiba Kheliouane Djemaa, and Taklit Akrouf Alitouche. Evaluation measures for models assessment over imbalanced data sets. *J Inf Eng Appl*, 3(10), 2013.

[37] Davide Chicco and Giuseppe Jurman. The advantages of the mcc over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):6, 2020.

[38] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[39] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *EMSE*, 13(5):561–595, 2008.

[40] Fred J Heemstra. Software cost estimation. *Information and software technology*, 34(10):627–639, 1992.

[41] Lawrence H Putnam and Ware Myers. *Measures for excellence: reliable software on time, within budget*. Prentice Hall Professional Technical Reference, 1991.

[42] Albert L Lederer and Jayesh Prasad. Information systems software cost estimating: a current assessment. *Journal of information technology*, 8(1):22–33, 1993.

[43] Ning Nan and Donald E Harter. Impact of budget and schedule pressure on software development cycle time and effort. *IEEE Transactions on Software Engineering*, 35(5):624–637, 2009.

[44] T Capers Jones. *Estimating software costs*. McGraw-Hill, Inc., 1998.

[45] Graham C. Low and D. Ross Jeffery. Function points in the estimation and evaluation of the software process. *IEEE transactions on Software Engineering*, 16(1):64–71, 1990.

[46] Chris F Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987.

[47] FJ Heemstra and RJ Kusters. Function point analysis: Evaluation of a software cost estimation model. *European Journal of Information Systems*, 1(4):229–237, 1991.

[48] Allan J Albrecht. Measuring application development productivity. In *Proc. Joint Share, Guide, and IBM Application Development Symposium, 1979*, 1979.

[49] Cigdem Gencel and Onur Demirors. Functional size measurement revisited. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3):1–36, 2008.

[50] Iso/iec 14143-1. `https://www.iso.org/standard/38931.html`. Accessed: 2020-11-08.

[51] A. Abran, J. Desharnais, A. Lesterhuis, B. Londeix, R. Meli, P. Morris, S. Oligny, M. O'Neil, T. Rollo, G. Rule, L. Santillo, C. Symons, and H. Toivonen. The COSMIC Functional Size Measurement Method – Measurement Manual, version 4.0.1 In http://www.cosmicon.com/portal/public/MMv4.0.1.pdf., 2015.

[52] Harold van Heeringen and Edwin Van Gorp. Measure the functional size of a mobile app: Using the cosmic functional size measurement method. In *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, pages 11–16. IEEE, 2014.

[53] F. Ferrucci, C. Gravino, P. Salza, and F. Sarro. Investigating functional and code size measures for mobile applications. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 365–368, 2015.

[54] Filomena Ferrucci, Carmine Gravino, Pasquale Salza, and Federica Sarro. Investigating functional and code size measures for mobile applications: A replicated study. In *International Conference on Product-Focused Software Process Improvement*, pages 271–287. Springer, 2015.

[55] L. De Marco, F. Ferrucci, C. Gravino, F. Sarro, S.M. Abrahão, and J. Gómez. Functional versus design measures for model-driven web applications: a case study in the context of web effort estimation. In *Proceedings of the 3rd International Workshop on Emerging Trends in Software Metric (WETSoM)*, pages 21–27, 2012.

[56] Beatriz Marín, Oscar Pastor, and Alain Abran. Towards an accurate functional size measurement procedure for conceptual models in an MDA environment. *Data Knowl. Eng.*, 69(5):472–490, 2010.

[57] Silvia Abrahão, Lucia De Marco, Filomena Ferrucci, Jaime Gómez, Carmine Gravino, and Federica Sarro. Definition and evaluation of a COSMIC mea-

surement procedure for sizing web applications in a model-driven development environment. *Information and Software Technology*, 104:144–161, 2018.

[58] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases., 2005.

[59] ISBSG. The international software benchmarking standards group, 2019.

[60] Emilia Mendes, Ian Watson, Chris Triggs, Nile Mosley, and Steve Counsell. A comparative study of cost estimation models for web hypermedia applications. *Empirical Software Engineering*, 8(2):163–196, 2003.

[61] Martin Auer, Adam Trendowicz, Bernhard Graser, Ernst Haunschmid, and Stefan Biffl. Optimal project feature weights in analogy-based cost estimation: Improvement and limitations. *IEEE Transactions on Software Engineering*, 32(2):83–92, 2006.

[62] Ekrem Kocaguneli, Tim Menzies, Ayse Bener, and Jacky W Keung. Exploiting the essential assumptions of analogy-based effort estimation. *IEEE Transactions on Software Engineering*, 38(2):425–438, 2011.

[63] Yan-Fu Li, Min Xie, and Thong Ngee Goh. A study of project selection and feature weighting for analogy based software cost estimation. *Journal of Systems and Software*, 82(2):241–252, 2009.

[64] Lionel C Briand, Khaled El Emam, Dagmar Surmann, Isabella Wieczorek, and Katrina D Maxwell. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 313–323. IEEE, 1999.

[65] Lionel C Briand, Tristen Langley, and Isabella Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. In

Proceedings of the 22nd international conference on Software engineering, pages 377–386, 2000.

[66] Ayse Bakir, Ekrem Kocaguneli, Ayse Tosun, Ayse Bener, and Burak Turhan. Xiruxe: an intelligent fault tracking tool. In *2009 International Conference on Artificial Intelligence and Pattern Recognition, AIPR 2009*, pages 293–300, 2009.

[67] Burak Turhan, Onur Kutlubay, and Ayse Bener. Evaluation of feature extraction methods on software cost estimation. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 497–497. IEEE, 2007.

[68] Jacky Keung and Barbara Kitchenham. Experiments with analogy-x for software cost estimation. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 229–238. IEEE, 2008.

[69] Jacky W Keung. Theoretical maximum prediction accuracy for analogy-based software cost estimation. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 495–502. IEEE, 2008.

[70] Ekrem Kocaguneli, Gregory Gay, Tim Menzies, Ye Yang, and Jacky W Keung. When to use data from other projects for effort estimation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 321–324, 2010.

[71] Jingzhou Li and Guenther Ruhe. Analysis of attribute weighting heuristics for analogy-based software effort estimation method aqua+. *Empirical Software Engineering*, 13(1):63–96, 2008.

[72] Martin Shepperd and Chris Schofield. Estimating software project effort using analogies. *IEEE Transactions on software engineering*, 23(11):736–743, 1997.

[73] Ekrem Kocaguneli and Tim Menzies. Software effort models should be assessed via leave-one-out validation. *Journal of Systems and Software*, 86(7):1879–1890, 2013.

[74] Boyce Sigweni, Martin Shepperd, and Tommaso Turchi. Realistic assessment of software effort estimation models. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6, 2016.

[75] Jasper Jolly. Passenger anger as tens of thousands hit by ba systems failure, 2019.

[76] Henry Bodkin, 2019.

[77] Capers Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill Education Group, 3rd edition, 2008.

[78] Barry Boehm and Victor R Basili. Top 10 list [software development]. *Computer*, 34(1):135–137, 2001.

[79] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[80] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[81] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[82] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[83] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *JSS*, 83(1):2–17, 2010.

[84] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[85] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[86] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[87] Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of systems and software*, 56(1):63–75, 2001.

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

[89] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[90] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461, 2006.

[91] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[92] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings*

of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pages 309–311, 2008.

[93] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering*, 67:15–24, 2018.

[94] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

[95] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88. IEEE, 2009.

[96] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18, 2007.

[97] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[98] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321, 2011.

[99] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[100] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the nasa metrics data program data sets for automated software

defect prediction. In *15th annual conference on evaluation & assessment in software engineering (EASE 2011)*, pages 96–103. IET, 2011.

[101] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[102] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. The jinx on the nasa software defect data sets. In *Procs. of EASE*, pages 1–5, 2016.

[103] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, 2012.

[104] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417–428. IEEE, 2004.

[105] Ahmed Iqbal, Shabib Aftab, Umair Ali, Zahid Nawaz, Laraib Sana, Munir Ahmad, and Arif Husen. Performance analysis of machine learning techniques on software defect prediction using nasa datasets. *Int. J. Adv. Comput. Sci. Appl*, 10(5), 2019.

[106] Ömer Faruk Arar and Kürşat Ayan. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, 33:263–277, 2015.

[107] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[108] Kwabena Ebo Bennin, Koji Toda, Yasutaka Kamei, Jacky Keung, Akito Monden, and Naoyasu Ubayashi. Empirical evaluation of cross-release

effort-aware defect prediction models. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 214–221. IEEE, 2016.

[109] Swapnil Shukla, T Radhakrishnan, K Muthukumaran, and Lalita Bhanu Murthy Neti. Multi-objective cross-version defect prediction. *Soft Computing*, 22(6):1959–1980, 2018.

[110] Jie Zhang, Jiajing Wu, Chuan Chen, Zibin Zheng, and Michael R Lyu. Cds: A cross–version software defect prediction model with data selection. *IEEE Access*, 8:110059–110072, 2020.

[111] Zhou Xu, Shuai Li, Yutian Tang, Xiapu Luo, Tao Zhang, Jin Liu, and Jun Xu. Cross version defect prediction with representative data via sparse subset selection. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 132–13211. IEEE, 2018.

[112] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[113] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.

[114] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the" imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[115] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. How far we have progressed in the jour-

ney? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(1):1–51, 2018.

[116] Lionel C. Briand and Isabella Wieczorek. Software resource estimation. *Encyclopedia of Software Engineering*, pages 1160–1196, 2002.

[117] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE TSE*, 23(11):736–743, 2000.

[118] Jianfeng Wen, Shixian Li, Zhiyong Lin, Yong Hu, and Changqin Huang. Systematic literature review of machine learning based software development effort estimation models. *Inf. Softw. Technol.*, 54(1):41–59, 2012.

[119] Ali Idri, Mohamed Hosni, and Alain Abran. Systematic literature review of ensemble effort estimation. *Journal of Systems and Software*, 118(C):151–175, 2016.

[120] Filomena Ferrucci, Mark Harman, and Federica Sarro. *Search-Based Software Project Management*, pages 373–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[121] Cigdem Gencel. How to use cosmic functional size in effort estimation models? In *Software Process and Product Measurement*, pages 196–207. Springer, 2008.

[122] Marcos de Freitas Junior, Marcelo Fantinato, and Violeta Sun. Improvements to the function point analysis method: A systematic literature review. *IEEE Transactions on Engineering Management*, 62(4):495–506, 2015.

[123] Filomena Ferrucci, Carmine Gravino, and Federica Sarro. Conversion from ifpug fpa to cosmic: within-vs without-company equations. In *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 293–300. IEEE, 2014.

[124] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro. Web effort estimation: Function point analysis vs. COSMIC. *Information and Software Technology*, 72:90–109, 2016.

[125] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. Assessing the effectiveness of approximate functional sizing approaches for effort estimation. *Information and Software Technology*, 123:106308, 2020.

[126] Emilia Mendes, Marcos Kalinowski, Daves Martins, Filomena Ferrucci, and Federica Sarro. Cross- vs. within-company cost estimation studies revisited: an extended systematic review. In *18th International Conference on Evaluation and Assessment in Software Engineering, EASE*, pages 12:1–12:10, 2014.

[127] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2013.

[128] Ekrem Kocaguneli, Tim Menzies, and Emilia Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, 20(3):813–843, 2015.

[129] Leandro Minku, Federica Sarro, Emilia Mendes, and Filomena Ferrucci. How to make best use of cross-company data for web effort estimation? In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[130] Stephen G. MacDonell and Martin J. Shepperd. Combining techniques to optimize effort predictions in software project management. *Journal of Systems and Software*, 66(2):91–98, 2003.

[131] Magne Jorgensen. Realism in assessment of effort estimation uncertainty: It matters how you ask. *IEEE Transactions on Software Engineering*, 30(4):209–217, 2004.

[132] Federica Sarro, Alessio Petrozziello, and Mark Harman. Multi-objective software effort estimation. In *Proc. of the 38th International Conference on Software Engineering ICSE*, pages 619–630, 2016.

[133] M. Jørgensen and D.I.K. Sjöberg. An effort prediction interval approach based on the empirical distribution of previous estimation accuracy. *Information and Software Technology*, 45(3):123–136, 2003.

[134] Magne Jørgensen and Dag I. K. Sjoeberg. An effort prediction interval approach based on the empirical distribution of previous estimation accuracy. *Information and software Technology*, 45(3):123–136, 2003.

[135] Magne Jørgensen. Looking back on previous estimation error as a method to improve the uncertainty assessment of benefits and costs of software development projects. In *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 19–24. IEEE, 2018.

[136] Karl Halvor Teigen and Magne JØrgensen. When 90% confidence intervals are 50% certain: On the credibility of credible intervals. *Applied Cognitive Psychology: The Official Journal of the Society for Applied Research in Memory and Cognition*, 19(4):455–475, 2005.

[137] Magne Jørgensen and Kjetil Moløkken. Combination of software development effort prediction intervals: Why, when and how? In *Proc. of SEKE'02*, pages 425–428, 2002.

[138] Magne Jørgensen, Karl Halvor Teigen, and Kjetil MoløKken. Better sure than safe? over-confidence in judgement based software development effort prediction intervals. *Journal of Systems and Software*, 70(1-2):79–93, 2004.

[139] Magne Jørgensen. The ignorance of confidence levels in minimum-maximum software development effort intervals. *Lecture Notes on Software Engineering*, 2(4):327, 2014.

[140] Albert L Lederer and Jayesh Prasad. Causes of inaccurate software development cost estimates. *Journal of systems and software*, 31(2):125–134, 1995.

[141] Terry Connolly and Doug Dean. Decomposed versus holistic estimates of effort required for software writing tasks. *Management Science*, 43(7):1029–1045, 1997.

[142] Andrew R Gray, Stephen G MacDonell, and Martin J Shepperd. Factors systematically associated with errors in subjective estimates of software development effort: the stability of expert judgment. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, pages 216–227. IEEE, 1999.

[143] Magne Jørgensen. Regression models of software development effort estimation accuracy and bias. *Empirical Software Engineering*, 9(4):297–314, 2004.

[144] Magne Jorgensen and Stein Grimstad. Software development estimation biases: The role of interdependence. *IEEE Transactions on Software Engineering*, 38(3):677–693, 2012.

[145] Magne Jørgensen. Forecasting of software development work effort: Evidence on expert judgement and formal models. *International Journal of Forecasting*, 23(3):449–462, 2007.

[146] Harry Levi Hollingworth. The central tendency of judgment. *The Journal of Philosophy, Psychology and Scientific Methods*, 7(17):461–469, 1910.

[147] Daniel Kahneman and Amos Tversky. Intuitive prediction: Biases and corrective procedures. Technical report, Decisions and Designs Inc Mclean Va, 1977.

[148] Michael M Roy and Nicholas JS Christenfeld. Bias in memory predicts bias in estimation of future task duration. *Memory & Cognition*, 35(3):557–564, 2007.

[149] Michael M Roy, Scott T Mitten, and Nicholas JS Christenfeld. Correcting memory improves accuracy of predicted task duration. *Journal of Experimental Psychology: Applied*, 14(3):266, 2008.

[150] Magne Jorgensen and Kjetil Molokken-Ostvold. Reasons for software effort estimation error: impact of respondent role, information collection approach, and data analysis method. *IEEE Transactions on Software engineering*, 30(12):993–1007, 2004.

[151] Yigit Kultur, Burak Turhan, and Ayse Basar Bener. Enna: software effort estimation using ensemble of neural networks with associative memory. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 330–338, 2008.

[152] Asad Ali and Carmine Gravino. A systematic literature review of software effort prediction using machine learning methods. *Journal of Software: Evolution and Process*, 31(10):e2211, 2019.

[153] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 207–210, New York, NY, USA, 2011. Association for Computing Machinery.

[154] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluch? Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey. *Artif. Intell. Rev.*, 52(1):77–124, June 2019.

[155] Sridevi Bonthu and K Hima Bindu. Review of leading data analytics tools. *International Journal of Engineering & Technology*, 7(3.31):10–15, 2017.

[156] Shraddha Dwivedi, Paridhi Kasliwal, and Suryakant Soni. Comprehensive study of data analytics tools (rapidminer, weka, r tool, knime). In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–8. IEEE, 2016.

[157] I. Myrtveit, E. Stensrud, and M. Shepperd. Reliability and validity in comparative studies of software prediction models. *IEEE Transactions on Software Engineering*, 31(5):380–391, 2005.

[158] Federica Sarro. Predictive analytics for software testing: keynote paper. In Juan Pablo Galeotti and Alessandra Gorla, editors, *Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE*, page 1. ACM, 2018.

[159] F. Sarro, R. Moussa, A. Petrozziello, and M. Harman. Learning from mistakes: Machine learning enhanced human expert effort estimates. *IEEE Transactions on Software Engineering*.

[160] Md Johirul Islam, Hoan Anh Nguyen, Rangeet Pan, and Hridesh Rajan. What do developers ask about ml libraries? a large-scale study using stack overflow. *arXiv preprint arXiv:1906.11940*, 2019.

[161] Yalda Hashemi, Maleknaz Nayebi, and Giuliano Antoniol. Documentation of machine learning software. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 666–667. IEEE, 2020.

[162] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23, 2014.

[163] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[164] Tushar Sharma Federica Sarro Ying Zou Stefanos Georgiou, Maria Kechagia. Green AI: Do deep learning frameworks have different costs? In

44th IEEE/ACM International Conference on Software Engineering (ICSE). IEEE/ACM, 2022.

[165] Alex Cummaudo, Rajesh Vasa, John Grundy, and Mohamed Abdelrazek. Requirements of api documentation: A case study into computer vision services. *IEEE Transactions on Software Engineering*, 2020.

[166] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[167] Cynthia C. S. Liem and Annibale Panichella. Oracle issues in machine learning and where to find them. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 483–488, New York, NY, USA, 2020. Association for Computing Machinery.

[168] Tenzin Doleck, David John Lemay, Ram B Basnet, and Paul Bazelais. Predictive analytics in education: a comparison of deep learning frameworks. *Education and Information Technologies*, 25(3):1951–1963, 2020.

[169] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: an analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783, 2020.

[170] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Empirical Softw. Engg.*, 22(5):2543–2584, October 2017.

[171] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. A benchmark study on sentiment analysis for software engineering research. In *Proceedings of*

the 15th International Conference on Mining Software Repositories, MSR '18, page 364–375, New York, NY, USA, 2018. Association for Computing Machinery.

[172] Nicole Novielli, Fabio Calefato, Filippo Lanubile, and Alexander Serebrenik. Assessment of off-the-shelf se-specific sentiment analysis tools: An extended replication study. *Empir. Softw. Eng.*, 26(4):77, 2021.

[173] Jarernsri Mitrpanont, Wudhichart Sawangphol, Thanita Vithantirawat, Sinattaya Paengkaew, Prameyuda Suwannasing, Atthapan Daramas, and Yi-Cheng Chen. A study on using python vs weka on dialysis data analysis. In *2017 2nd International Conference on Information Technology (INCIT)*, pages 1–6. IEEE, 2017.

[174] Satish CR Nandipati, Chew XinYing, and Khaw Khai Wah. Hepatitis c virus (hcv) prediction by machine learning techniques. *Applications of Modelling and Simulation*, 4:89–100, 2020.

[175] Stephen R Piccolo, Terry J Lee, Erica Suh, and Kimball Hill. Shinylearner: A containerized benchmarking tool for machine-learning classification of tabular data. *GigaScience*, 9(4):giaa026, 2020.

[176] Stephen R Piccolo, Avery Mecham, Nathan P Golightly, Jérémie L Johnson, and Dustin B Miller. Benchmarking 50 classification algorithms on 50 gene-expression datasets. *bioRxiv*, 2021.

[177] Cynthia Liem and Annibale Panichella. Run, forest, run? on randomization and reproducibility in predictive software engineering. *arXiv preprint arXiv:2012.08387*, 2020.

[178] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[179] Venkata Udaya B Challagulla, Farokh B Bastani, I-Ling Yen, and Raymond A Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

[180] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology*, 52(11):1242–1257, 2010.

[181] Seyedrebvar Hosseini, Burak Turhan, and Mika Mäntylä. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology*, 95:296–312, 2018.

[182] Rebecca Moussa and Danielle Azar. A pso-ga approach targeting fault-prone software modules. *Journal of Systems and Software*, 132:41–49, 2017.

[183] Kehan Gao, Taghi M Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[184] Marian Jureczko. Significance of different software metrics in defect prediction. *Software Engineering: An International Journal*, 1(1):86–95, 2011.

[185] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2018.

[186] Shamsul Huda, Kevin Liu, Mohamed Abdelrazek, Amani Ibrahim, Sultan Alyahya, Hmood Al-Dossari, and Shafiq Ahmad. An ensemble oversampling model for class imbalance problem in software defect prediction. *IEEE access*, 6:24184–24195, 2018.

[187] Sousuke Amasaki. Cross-version defect prediction using cross-project defect prediction approaches: Does it work? In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 32–41, 2018.

[188] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[189] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *LASER*, pages 1–59, 2010.

[190] Federica Sarro. Search-based predictive modelling for software engineering: How far have we gone? In Shiva Nejati and Gregory Gay, editors, *Proceedings of the 11th International Symposium on Search-Based Software Engineering (SSBSE)*, volume 11664 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2019.

[191] Ruchika Malhotra, Megha Khanna, and Rajeev R. Raje. On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions. *Swarm and Evolutionary Computation*, 32:85–109, 2017.

[192] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015.

[193] Mark Harman, Syed Islam, Yue Jia, Leandro L. Minku, Federica Sarro, and Komsan Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In *Search-Based Software Engineering*, pages 240–246, Cham, 2014. Springer International Publishing.

[194] Federica Sarro, Sergio Di Martino, Filomena Ferrucci, and Carmine Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1215–1220, New York, NY, USA, 2012. ACM.

[195] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2019.

[196] Faseeha Matloob, Taher M Ghazal, Nasser Taleb, Shabib Aftab, Munir Ahmad, Muhammad Adnan Khan, Sagheer Abbas, and Tariq Rahim Soomro. Software defect prediction using ensemble learning: A systematic literature review. *IEEE Access*, 2021.

[197] Xiaoxing Yang, Xin Li, Wushao Wen, and Jianmin Su. An investigation of ensemble approaches to cross-version defect prediction. pages 437–442, July 2019.

[198] Issam H. Laradji, Mohammad Alshayeb, and Lahouari Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.

[199] Santosh Rathore and Sandeep Kumar. An empirical study of ensemble techniques for software fault prediction. *Applied Intelligence*, 51:1–30, June 2021.

[200] Arsalan Ahmed Ansari, Amaan Iqbal, and Bibhudatta Sahoo. Heterogeneous defect prediction using ensemble learning technique. In Subhransu Sekhar Dash, C. Lakshmi, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, pages 283–293, Singapore, 2020. Springer Singapore.

[201] Hamad Alsawalqah, Neveen Hijazi, Mohammed Eshtay, Hossam Faris, Ahmed Al Radaideh, Ibrahim Aljarah, and Yazan Alshamaileh. Software defect prediction using heterogeneous ensemble classification based on segmented patterns. *Applied Sciences*, 10(5), 2020.

[202] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. Building an ensemble for software defect prediction based on diversity selection. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, September 2016.

[203] Wasif Afzal and Richard Torkar. Review: On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Syst. Appl.*, 38(9):11984–11997, sep 2011.

[204] Ye Ren, Le Zhang, and P.N. Suganthan. Ensemble classification and regression-recent developments, applications and future directions [review article]. *IEEE Computational Intelligence Magazine*, 11(1):41–53, 2016.

[205] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[206] Hussein A Abbass. Pareto neuro-evolution: Constructing ensemble of neural networks using multi-objective optimization. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 3, pages 2074–2080. IEEE, 2003.

[207] Arjun Chandra and Xin Yao. Ensemble learning using multi-objective evolutionary algorithms. *Journal of Mathematical Modelling and Algorithms*, 5(4):417–445, March 2006.

[208] Christian Gagné, Michèle Sebag, Marc Schoenauer, and Marco Tomassini. Ensemble learning for free with evolutionary algorithms? In *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*. ACM Press, 2007.

[209] Leandro L. Minku and Xin Yao. An analysis of multi-objective evolutionary algorithms for training ensemble models based on different performance measures in software effort estimation. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*. ACM, October 2013.

[210] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. Autostacker: A Compositional Evolutionary Learning System. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO'18, pages 402—-409, 2018.

[211] Sam Fletcher, Brijesh Verma, and Mengjie Zhang. A non-specialized ensemble classifier using multi-objective optimization. *Neurocomputing*, 409:93–102, October 2020.

[212] Kate Han, Tien Pham, Trung Hieu Vu, Truong Dang, John McCall, and Tien Thanh Nguyen. VEGAS: A variable length-based genetic algorithm for ensemble selection in deep ensemble learning. In *Intelligent Information and Database Systems*, pages 168–180. Springer International Publishing, 2021.

[213] Shenkai Gu and Yaochu Jin. Generating diverse and accurate classifier ensembles using multi-objective optimization. In *2014 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making (MCDM)*. IEEE, December 2014.

[214] E.M. Dos Santos, R. Sabourin, and P. Maupin. Single and multi-objective genetic algorithms for the selection of ensemble of classifiers. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE, 2006.

[215] Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: why using the matthews correlation coefficient matters. In *Procs. of EASE*, pages 120–129. 2020.

[216] Xiao Xuan, David Lo, Xin Xia, and Yuan Tian. Evaluating defect prediction approaches using a massive set of metrics: An empirical study, 2015.

[217] Magne Jørgensen and Torleif Halkjelsvik. Sequence effects in the estimation of software development effort. *Journal of Systems and Software*, 159:110448, 2020.

[218] Martin J. Shepperd and Steven G. MacDonell. Evaluating prediction systems in software project estimation. *Information and Software Technology*, 54(8):820–827, 2012.

[219] Federica Sarro and Alessio Petrozziello. Linear programming as a baseline for software effort estimation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(3):12:1–12:28, 2018.

[220] Mohamed Aly. Survey on multiclass classification methods, 2005.

[221] Fiona Walkerden and Ross Jeffery. An empirical study of analogy-based software effort estimation. *Empirical software engineering*, 4(2):135–158, 1999.

[222] M. Choetkiertikul, H. K. Dam, T. Tran, T. T. M. Pham, A. Ghose, and T. Menzies. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2018.

[223] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *STVR*, 24(3):219–250, 2014.

[224] William B. Langdon, José Javier Dolado, Federica Sarro, and Mark Harman. Exact mean absolute error of baseline predictor, MARP0. *Information and Software Technology*, 73:16–18, 2016.

[225] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.

[226] Chakkrit Tantithamthavorn and Ahmed E Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 286–295, 2018.

[227] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, 1983.

[228] Çigdem Gencel and Onur Demirörs. Functional size measurement revisited. *ACM Trans. Softw. Eng. Methodol.*, 17(3), 2008.

[229] Marta Fernández-Diego and Fernando González L. Guevara. Potential and limitations of the isbsg dataset in enhancing software engineering research: A mapping review. *Information and Software Technology*, 56, 2014.

[230] Barbara Kitchenham, Shari Lawrence Pfleeger, Beth McColl, and Suzanne Eagan. An empirical study of maintenance and development estimation accuracy. *Journal of Systems and Software*, 64(1):57–77, 2002.

[231] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[232] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[233] Pat Langley, Wayne Iba, Kevin Thompson, et al. An analysis of bayesian classifiers. In *Aaai*, volume 90, pages 223–228, 1992.

[234] Abraham Charnes and William W Cooper. Programming with linear fractional functionals. *Naval Research logistics quarterly*, 9(3-4):181–186, 1962.

[235] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[236] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2018.

[237] Anna Corazza, Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, Federica Sarro, and Emilia Mendes. Using tabu search to configure support vector regression for effort estimation. *EMSE*, 18(3):506–546, 2013.

[238] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 321–332. ACM, 2016.

[239] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[240] Brian D Ripley. *Pattern recognition and neural networks*. Cambridge university press, 2007.

[241] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., 2005.

[242] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[243] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[244] David J. Hand and Robert J. Till. A simple generalisation of the area under the roc curve for multiple class classification problems. *Machine Learning*, 45(2):171–186, 2001.

[245] Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association*, 32(200):675–701, 1937.

[246] PB Nemenyi. Distribution-free multiple comparisons (doctoral dissertation, princeton university, 1963). *Dissertation Abstracts International*, 25(2):1233, 1963.

[247] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.

[248] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition, 1988.

[249] Robert Rosenthal, H Cooper, and L Hedges. Parametric measures of effect size. *The handbook of research synthesis*, 621:231–244, 1994.

[250] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[251] James A Rosenthal. Qualitative descriptors of strength of association and effect size. *Journal of social service Research*, 21(4):37–59, 1996.

[252] B. Kitchenham, L. Pickard, and S.L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

[253] E. Mendes, S. Counsell, N. Mosley, C. Triggs, and I. Watson. A comparative study of cost estimation models for web hypermedia applications. *EMSE*, 8(23):163–196, 2003.

[254] Peter A. Whigham, Caitlin A. Owen, and Stephen G. Macdonell. A baseline model for software effort estimation. *ACM TOSEM*, 24(3):20:1–20:11, 2015.

[255] Lionel C. Briand and Jürgen Wüst. Modeling development effort in object-oriented systems using design properties. *IEEE TSE*, 27(11):963–986, 2001.

[256] Tron Foss, Erik Stensrud, Barbara Kitchenham, and Ingunn Myrtveit. A simulation study of the model evaluation criterion mmre. *IEEE Transactions on Software Engineering*, 29(11):985–995, 2003.

[257] Barbara Kitchenham and Emilia Mendes. Why comparative effort prediction studies may be invalid. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE, New York, NY, USA, 2009. ACM.

[258] Rebecca Moussa and Federica Sarro. On the use of evaluation measures for defect prediction studies. In *2022 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery (ACM), 2022.

[259] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[260] Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17:1–17, 2012.

[261] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE TSE*, 43(9):817–847, 2016.

[262] Mark Harman Rebecca Moussa and Federica Sarro. The Role of Open Source Machine Learning Libraries in Effort Estimation Studies, 2022. On-line appendix, `https://figshare.com/s/d8bc05d38f0a32be2d53`.

[263] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[264] Max Kuhn. A short introduction to the caret package. *R Found Stat Comput*, 1, 2015.

[265] Remco R. Bouckaert, Eibe Frank, Mark A. Hall, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. Weka—experiences with a java open-source project. *J. Mach. Learn. Res.*, 11:2533–2541, December 2010.

[266] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

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

[268] Ian H Witten. Data mining. 2016.

[269] George AF Seber and Alan J Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

[270] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[271] The seacraft repository of empirical software engineering data, 2017.

[272] Federica Sarro, Sergio Di Martino, Filomena Ferrucci, and Carmine Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1215–1220, 2012.

[273] F. Ferrucci, M. Harman, and F. Sarro. Search-based software project management. In *Software Project Management in a Changing World*, pages 373–399. Springer, 2014.

[274] Ekrem Kocaguneli, Tim Menzies, and Jacky W. Keung. On the value of ensemble effort estimation. *IEEE TSE*, 38(6):1403–1416, 2012.

[275] Federica Sarro, Rebecca Moussa, Alessio Petrozziello, and Mark Harman. Learning from mistakes: Machine learning enhanced human expert effort estimates. *IEEE Transactions on Software Engineering*, 2020.

[276] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 147–157, 2013.

[277] William B. Langdon, Javier Dolado, Federica Sarro, and Mark Harman. Exact mean absolute error of baseline predictor, MARP0. *Information and Software Technology*, 73:16–18, 2016.

[278] RF Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.

[279] Mark Harman. The role of artificial intelligence in software engineering. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 1–6. IEEE, 2012.

[280] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[281] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes. How effective is tabu search to configure support vector regression for effort estimation? In *Proc. of PROMISE'10*, pages 4:1–4:10, 2010.

[282] D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software engineering metrics and models*. Benjamin/Cummings Publishing Company, Inc., 1986.

[283] Tron Foss, Erik Stensrud, Barbara Kitchenham, and Ingunn Myrtveit. A simulation study of the model evaluation criterion MMRE. *IEEE TSE*, 29(11):985–995, 2003.

[284] B. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd. What accuracy statistics really measure. *IEEE Proc. Software*, 148(3):81–85, 2001.

[285] Marcel Korte and Dan Port. Confidence in software cost estimation results based on mmre and pred. In *Proc. of PROMISE'08*, pages 63–70, 2008.

[286] Dan Port and Marcel Korte. Comparative studies of the model evaluation criterions mmre and pred in software cost estimation research. In *Proc. of ESEM'08*, pages 51–60, 2008.

[287] Martin Shepperd, Michelle Cartwright, and Gada Kadoda. On building prediction systems for software engineers. *EMSE*, 5(3):175–182, 2000.

[288] Erik Stensrud, Tron Foss, Barbara Kitchenham, and Ingunn Myrtveit. A further empirical investigation of the relationship between MRE and project size. *EMSE*, 8(2):139–161, 2003.

[289] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382, 2015.

[290] Bobby R Bruce, Jonathan M Aitken, and Justyna Petke. Deep parameter optimisation for face detection using the viola-jones algorithm in opencv. In *International Symposium on Search Based Software Engineering*, pages 238–243. Springer, 2016.

[291] Mahmoud A Bokhari, Bobby R Bruce, Brad Alexander, and Markus Wagner. Deep parameter optimisation on android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1501–1508, 2017.

[292] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[293] Hongyu Zhang and Xiuzhen Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, 2007.

[294] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[295] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision: A response to "comments on 'data mining static

code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering*, 33(9):637–640, 2007.

[296] David Bowes, Tracy Hall, and David Gray. Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix. In *Procs. of PROMISE*, pages 109–118, 2012.

[297] B. Turhan, T. Zimmermann, F. Shull, L. Layman, A. Marcus, A. Butcher, D. Cok, and T. Menzies. Local versus global lessons for defect prediction and effort estimation. *IEEE TSE*, 39(06):822–834, 2013.

[298] Tim Menzies and Martin J. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17:1–17, 2011.

[299] Xiao Xuan, David Lo, Xin Xia, and Yuan Tian. Evaluating defect prediction approaches using a massive set of metrics: An empirical study. In *Procs. of ACM SAC*, page 1644–1647, 2015.

[300] David Bowes, Tracy Hall, and David Gray. Dconfusion: a technique to allow cross study performance evaluation of fault prediction studies. *Automated Software Engineering*, 21:287–313, 2014.

[301] David J Hand. Measuring classifier performance: a coherent alternative to the area under the roc curve. *Machine learning*, 77(1):103–123, 2009.

[302] Sandro Morasca and Luigi Lavazza. On the assessment of software defect prediction models via roc curves. *Empirical Software Engineering*, 25(5):3977–4019, 2020.

[303] Harald Cramir. Mathematical methods of statistics. *Princeton U. Press*, 1946.

[304] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[305] Q. Song, Y. Guo, and M. Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269, 2019.

[306] George Forman and Martin Scholz. Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. *ACM Sigkdd Explorations Newsletter*, 12(1):49–57, 2010.

[307] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance optimization for mobile applications. *IEEE TSE*, 2021.

[308] Rebecca Moussa and Federica Sarro. On the Use of Evaluation Measures for Defect Prediction Models, 2022. On-line appendix, `https://github.com/SOLAR-group/dpevalmeasures`.

[309] Peter Flach and Meelis Kull. Precision-recall-gain curves: Pr analysis done right. In *Advances in neural information processing systems*, pages 838–846, 2015.

[310] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[311] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[312] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn. Mining software defects: Should we consider affected releases? volume 2019-May, pages 654–665, 2019.

[313] Mark Harman, Syed Islam, Yue Jia, Leandro L Minku, Federica Sarro, and Komsan Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In *International Symposium on Search Based Software Engineering*, pages 240–246. Springer, 2014.

[314] Aalok Ahluwalia, Massimiliano Di Penta, and Davide Falessi. On the need of removing last releases of data when using or validating defect prediction models. *arXiv preprint arXiv:2003.14376*, 2020.

[315] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 695–705, 2019.

[316] Abdul Ali Bangash, Hareem Sahar, Abram Hindle, and Karim Ali. On the time-based conclusion stability of cross-project defect prediction models. *Empirical Software Engineering*, 25(6):5047–5083, 2020.

[317] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, 44(9):811–833, 2017.

[318] Ian H Witten, Eibe Frank, and Mark A Hall. Practical machine learning tools and techniques. *Morgan Kaufmann*, page 578, 2005.

[319] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. *Logistic regression*. Springer, 2002.

[320] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[321] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance optimization for mobile applications. *IEEE TSE*, 2021.

[322] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *Procs. of SANER*, pages 33–45, 2016.

[323] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, and N. Ubayashi. Empirical evaluation of cross-release effort-aware defect prediction models. In *Procs. of QRS*, pages 214–221, 2016.

[324] Mariam El Mezouar, Feng Zhang, and Ying Zou. Local versus global models for effort-aware defect prediction. In *Procs. of CASCON*, page 178–187, 2016.

[325] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE TSE*, 2020.

[326] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E. Hassan, and Xindong Zhang. *Effort-Aware Just-in-Time Defect Identification in Practice: A Case Study at Alibaba*, page 1308–1319. ACM, 2020.

[327] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Hideaki Tantithamthavorn, Chakkrit Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. In *IEEE IEEE TSE*, 2020.

[328] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software*, 161, 3 2020.

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

[330] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 4 2019.

[331] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE TSE*, pages 1–1, 2020.

[332] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect

prediction using cross-project models. In *Procs. of MSR*, page 172–181, 2014.

[333] Martin Shepperd and Steve MacDonell. Evaluating prediction systems in software project estimation. *Information and Software Technology*, 54(8):820–827, 2012.

[334] Andrea Arcuri and Lionel Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, May 2014.

[335] Rebecca Moussa, Giovani Guizzo, and Federica Sarro. Meg - on-line, 2022. GitHub Repository, `https://github.com/SOLAR-group/MEG`.

[336] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[337] Eckart Zitzler and Simon Künzli. Indicator-Based Selection in Multiobjective Search. In *International Conference on Parallel Problem Solving from Nature*, pages 832–842, 2004.

[338] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, June 2000.

[339] Ludmila I Kuncheva and Christopher J Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine learning*, 51(2):181–207, 2003.

[340] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE TEVC*, 6(2):182–197, 2002.

[341] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Lecture*

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7007 LNCS:1–59, 2011.

[342] Gavin Brown, Jeremy Wyatt, Rachel Harris, and Xin Yao. Diversity creation methods: a survey and categorisation. *Information Fusion*, 6(1):5–20, March 2005.

[343] Yijun Bian and Huanhuan Chen. When does diversity help generalization in classification ensembles? *IEEE Transactions on Cybernetics*, pages 1–17, 2021.

[344] Steven L Salzberg. C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993, 1994.

[345] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. Mining software defects: should we consider affected releases? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 654–665. IEEE, 2019.

[346] Davide Falessi, Jacky Huang, Likhita Narayana, Jennifer Fong Thai, and Burak Turhan. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering*, pages 1–26, 2020.

[347] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[348] NC Shrikanth, Suvodeep Majumder, and Tim Menzies. Early life cycle software defect prediction. why? how? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 448–459. IEEE, 2021.

[349] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio

Filieri, et al. Empirical standards for software engineering research. *arXiv preprint arXiv:2010.03525*, 2020.

[350] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[351] Rebecca Moussa and Federica Sarro. Do not take it for granted: Comparing open-source libraries for software development effort estimation, 2022.

[352] Cynthia C. S. Liem and Annibale Panichella. Run, forest, run? on randomization and reproducibility in predictive software engineering, 2020.

[353] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. Building an ensemble for software defect prediction based on diversity selection. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[354] Giovani Guizzo, Federica Sarro, Jens Krinke, and Silvia Regina Vergilio. Sentinel: A hyper-heuristic for the generation of mutant reduction strategies. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[355] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[356] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 107–116, 2010.

[357] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170, 2017.

[358] Vali Tawosi, Federica Sarro, Alessio Petrozziello, and Mark Harman. Multi-objective software effort estimation: A replication study. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[359] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[360] Claes Wohlin, Darja Šmite, and Nils Brede Moe. A general theory of software engineering: Balancing human, social and organizational capitals. *Journal of Systems and Software*, 109:229–242, 2015.

[361] Chakkrit Tantithamthavorn, Jürgen Cito, Hadi Hemmati, and Satish Chandra. Explainable ai for se: Challenges and future directions. *IEEE Software*, 40(3):29–33, 2023.

[362] Michael Fu and Chakkrit Tantithamthavorn. Gpt2sp: A transformer-based agile story point estimation approach. *IEEE Transactions on Software Engineering*, 49(2):611–625, 2022.

[363] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 432–443. IEEE, 2021.

[364] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 48(5):1480–1496, 2020.

[365] John C Nash. The (dantzig) simplex method for linear programming. *Computing in Science and Engineering*, 2(1):29–31, 2000.