Methods and Tools for Focusing and Prioritizing the Testing Effort

A dissertation submitted to
University of Salerno

for the degree of
Doctor of Philosophy

presented by
Dario Di Nucci

March 2018

Software Engineering Lab, University of Salerno
http://www.sesa.unisa.it

This research was done under the supervision of Prof. Andrea De Lucia with the financial support of the University of Salerno, from December 1st of 2014 to November 30th of 2017.

The final version of the thesis has been revised based on the reviews made by Prof. Harald Gall and Prof. Federica Sarro.

First release, December 2017.
Revised release, February 2018.
Defense date, 15th March 2018.

ABSTRACT

Software testing is widely recognized as an essential part of any software development process, representing however an extremely expensive activity. The overall cost of testing has been estimated at being at least half of the entire development cost, if not more. Despite its importance, however, recent studies showed that developers rarely test their application and most programming sessions end without any test execution. Indeed, new methods and tools able to better allocating the developers effort are needed in order to increment the system reliability and to reduce the testing costs.

The resources available should be allocated effectively upon the portions of the source code that are more likely to contain bugs. In this thesis we focus on three activities able to prioritize the testing effort, specifically bug prediction, test case prioritization, and detection of code smell able to fix energy issues. Indeed, despite the effort devoted by the research community in the last decades through the conduction of empirical studies and the devising of new approaches led to interesting results, in the context of our research we highlighted some aspects that might be improved and proposed empirical investigations and novel approaches.

In the context of bug prediction, we devised two novel metrics, namely the DEVELOPER'S STRUCTURAL AND SEMANTIC SCATTERING. These metrics exploit the presence of scattering changes that make developers more error-prone. The results of our the empirical study show the superiority of our model with respect to baselines based on product metrics and process metrics. Afterwards, we devised a "hybrid" model providing an average improvement in terms of prediction accuracy. Besides analyzing on predictors, we proposed a novel adaptive prediction classifier, which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class. The models based on this classifier are able to outperform models based on stand-alone classifiers, as well as those based on the VALIDATION AND VOTING ensemble technique in the context of within-project bug prediction. Lately, we performed a differentiated replication study in the contexts of cross-project and within-project bug prediction. We analyzed the

behavior of seven ensemble methods. The results show that the problem is still far from being solved and that the use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers, independently from the strategy adopted to build model. Finally, we confirmed, in the context of ensemble-based models, the findings of previous studies that demonstrated that cross-project bug prediction models perform worse than within-project ones, being however more robust to performance variability.

With respect to the test case prioritization problem, we proposed a genetic algorithm based on the hypervolume indicator. We provided an extensive evaluation of Hypervolume-based and state-of-the-art approaches when dealing with up to five testing criteria. Our results suggest that the test ordering produced by HGA is more cost-effective than those produce by state-of-the-art algorithms. Moreover, our algorithm is much more faster and its efficiency does not decrease as the size of the software program and of the test suite increase.

To cope with energy efficiency issues of mobile applications and thus reducing the effort needed to test this non-functional aspect, we devised two novel software tools. PETrA is able to extract the energy profile of mobile applications, while aDoctor is a code smell detector able to identify 15 Android-specific code smells defined by Reimann *et al.*. We analyzed the impact of these smells, by a large empirical study with the aim of determining to what extent code smells affecting source code methods of mobile applications influence energy efficiency and whether refactoring operations applied to remove them directly improve the energy efficiency of refactored methods. The results of our study highlight that methods affected by code smells consume up to 385% more energy than methods not affected by any smell. A fine-grained analysis reveals the existence of four specific energy-smells. Finally, we also shed light on the usefulness of refactoring as a way for improving energy efficiency by code smell removal. Specifically, we found that it is possible to improve the energy efficiency of source code methods by up to 90% through refactoring code smells.

Finally, we provide a set of open issues that should be addressed by the research community in the future.

SOMMARIO

Il testing del software è largamente riconosciuto come una parte essenziale del processo di sviluppo software, rappresentando comunque un'attività estremamente costosa. Il costo totale del testing è stato stimato costituire almeno metà del costo totale di sviluppo. Nonostante la sua importanza, comunque, studi recenti hanno mostrato che gli sviluppatori raramente testano le loro applicazioni e la maggioranza delle sessioni di programmazione finiscono senza che nessun test sia stato eseguito. Quindi, nuovi metodi e strumenti capaci di meglio allocare gli sviluppatori sono necessari per aumentare la robustezza dei sistemi e ridurre i costi del testing.

Le risorse disponibili dovrebbero essere efficacemente allocate tra le parti del codice sorgente che hanno più probabilità di contenere difetti. In questa tesi ci focalizziamo su tre attività per focalizzare e prioritizzare il costo del testing, in particolare predizione dei difetti, prioritizzazione dei casi di test e rilevazione dei code smell relativi a problemi energetici. Quindi, nonostante lo sforzo profuso dalla comunità scientifica nelle ultime decadi attraverso la conduzione di studi empirici e la proposta di nuovi approcci che hanno portato risultati interessanti, nel contesto della nostra ricerca abbiamo sottolineato alcuni aspetti che potrebbero essere migliorati e proposto studi empirici e nuovi approcci.

Nel contesto della predizione dei difetti, abbiamo proposto due nuove misure, DEVELOPER'S STRUCTURAL AND SEMANTIC SCATTERING. Queste metriche sfruttano la presenza di cambiamenti dispersi che rendono gli sviluppatori più inclini ad introdurre difetti. I risultati del nostro studio empirico mostrano la superiorità del nostro modello rispetto a quelli basati su metriche di processo e di prodotto. In seguito, abbiamo sviluppato un modello "ibrido" che fornisce un miglioramento medio in termini di accuratezza. Oltre ad analizzare i predittori, abbiamo sviluppato un nuovo classificatore adattivo, che dinamicamente raccomanda il classificatore capace di predire in maniera migliore la difettosità di una classe, basandosi sulle caratteristiche strutturali della stessa. I modelli basati su questo classificatori riesco ad essere più efficaci rispetto a quelli basati su classificatori semplici, così come quelli basati sulla tecnica di ensemble detta VALIDATION AND VOTING nel contesto della predizione dei difetti intra-progetto. In seguito

abbiamo proposto uno studio replica nel contesto della predizione di difetti intra- e inter-progetto. Abbiamo analizzato il comportamento di sette metodi ensemble. I risultati mostrano che il problema è ancora lontano dall'essere risolto e che l'uso delle tecniche di ensemble non fornisce benefici evidenti rispetto ai classificatori semplici, indipendentemente dalla strategia utilizzata per costruire il modello. Infine, abbiamo confermato, nel contesto dei modelli basati su tecniche di ensemble, i risultati di studi precedenti che hanno dimostrato che i modelli per la predizione dei difetti inter-progetto funzionano peggio di quelli intra-progetto, essendo comunque più robusti alla variabilità delle performance.

Rispetto al problema di prioritizzazione dei casi di test, abbiamo proposto un algoritmo genetico basato sull'indicatore dell'ipervolume. Abbiamo fornito una validazione estesa degli approcci basati sull'ipervolume e dello stato dell'arte utilizzando fino a cinque criteri di testing. I nostri risultati suggeriscono che l'ordinamento fornito da HGA è più efficace rispetto a quelli prodotti dagli algoritmi dello stato dell'arte. Inoltre, il nostro algoritmo è molto più veloce e la sua efficacia non diminuisce quando la dimensione del programma software o della test suite cresce.

Per gestire i problemi relativi all'efficienza energetica delle applicazioni mobile e quindi ridurre il costo del testing di questo aspetto non funzionale, abbiamo sviluppato due nuovi strumenti software. PETrA è capace di estrarre il profilo energetico delle applicazioni mobile, mentre aDoctor è un rilevatore di code smell capace di identificare 15 dei code smells specifici per applicazioni Android definiti da Reimann *et al.*. Abbiamo analizzato l'impatto di questi smell, attraverso un grande studio empirico con l'obiettivo di determinare in che modo i code smell relativi ai metodi del codice sorgente delle applicazioni mobile influenzano il consumo energetico e se le operazioni di refactoring applicate per rimuoverli migliorano l'efficienza energetica dei metodi rifattorizzati. I risultati del nostro studio sottolineano che i metodi affetti da code smell consumano fino a 385% più energia rispetto ai metodi non affetti da smell. Un'analisi a grana fine rivela l'esistenza di quattro energy smell. Infine, abbiamo sottolineato l'utilità del refactoring come un mezzo per migliorare l'efficienza energetica attraverso la rimozione dei code smell. In dettaglio, abbiamo trovato che sia possibile migliorare l'efficienza energetica dei metodi del codice sorgente fino al 90% attraverso il refactoring dei code smell.

Infine, forniamo un insieme di problemi aperti che dovrebbero essere affrontati dalla comunità scientifica nel futuro.

ACKNOWLEDGEMENTS

I would like to thank all the people that supported me during these years of research. My sincere apologies to anyone I inadvertently omitted.

First and foremost I want to thank my tutor, **Andrea De Lucia**. Andrea has been an example since the first days I began my Ph.D. He taught me how to dig into the problems, trying to find the best solution without stopping after the first insights. His dedication to the work and his passion for Software Engineering have been sources of inspiration during these research years. Without his precious support, it would not have been possible to conduct this research.

I have to thank **Andy Zaidman** from the SERG Group, for the fruitful experience I had at the Technische Universiteit Delft. I think that Andy is the perfect example of how a research should be and how he should act. His integrity, his passion to work, and his code of conduct have been an example that I will try to follow for the upcoming years.

A special thanks to **Coen De Roover** that let me work with his research group. I will be always grateful to Coen for the great opportunity he gave to me.

During this path I collaborated with many brilliant researchers, I would like to thank **Fabio Palomba**, **Rocco Oliveto**, **Annibale Panichella**, **Andy Zaidman**, **Gabriele Bavota**, **Filomena Ferrucci**, **Denys Poshyvanyk**, **Alexander Serebrenik**, **Damian Andrew Tamburri**, **Pasquale Salza**, **Giuseppe De Rosa, Antonio Prota**, **Sandro Siravo**, and **Michele Tufano**. I was lucky to work with them and they all deserve my immense gratitude.

I thank my fellow labmates, now friends, of the SESA Lab, **Fabio Palomba**, **Pasquale Salza**, and **Gemma Catolino**, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun experiences we had in the last three years.

During these years, I met a lot of people and each of them influenced my goals, my choices, and my life. I thank all of them for contributing to the puzzle of my life that will lead to who I am. Last but not least, I would like to thank my family and my parents for supporting me in any possible way.

RINGRAZIAMENTI

Vorrei ringraziare tutte le persone che mi hanno supportato durante questi anni di ricerca. Mi scuso con tutti quelli che ho inavvertitamente omesso.

Prima di tutto vorrei ringraziare il mio tutor, **Andrea De Lucia**. Andrea è stato un esempio dai primi giorni del mio dottorato. Mi ha insegnato come scavare nei problemi, cercando di trovare le migliori soluzioni senza fermarmi dopo le prime intuizioni. La sua dedica al lavoro e la sua passione per l'Ingegneria del Software sono state fonti di ispirazione durante questi anni di ricerca. Senza il suo prezioso supporto, non sarebbe stato possibile condurre questa ricerca.

Devo ringraziare **Andy Zaidman** del SERG Group, per la fruttuosa esperienza che ho avuto alla Technische Universiteit Delft. Penso che Andy sia un esempio perfetto di come un ricercatore dovrebbe essere e di come dovrebbe comportarsi. La sua integrità, la sua passione per il lavoro e il suo codice di condotta sono stati un esempio che cercherò di seguire nei prossimi anni.

Un ringraziamento speciale a **Coen De Roover** che mi ha permesso di lavorare nel suo gruppo di ricerca. Sarò sempre grato a Coen per la grande opportunità che mi ha dato.

Durante questo percorso ho collaborato con molti ricercatori brillanti, vorrei ringraziare **Fabio Palomba**, **Rocco Oliveto**, **Annibale Panichella**, **Andy Zaidman**, **Gabriele Bavota**, **Filomena Ferrucci**, **Denys Poshyvanyk**, **Alexander Serebrenik**, **Damian Andrew Tamburri**, **Pasquale Salza**, **Giuseppe De Rosa, Antonio Prota**, **Sandro Siravo** e **Michele Tufano**. Sono stato fortunato a lavorare con loro e tutti loro meritano la mia immensa gratitudine.

Ringrazio i miei colleghi di laboratorio, ora amici, del SESA Lab, **Fabio Palomba**, **Pasquale Salza** e **Gemma Catolino**, per le discussioni stimolanti, le notti insonni in cui abbiamo lavorato insieme prima delle scadenze e per tutte le esperienze divertenti che abbiamo avuto negli ultimi tre anni.

Durante questi anni, ho incontrato molte persone e ognuna di esse ha influenzato i miei obiettivi, le mie scelte e la mia vita. Ringrazio tutte loro per aver contribuito al puzzle della mia vita. Per ultimi ma non meno importanti, vorrei ringraziare la mia famiglia e i miei genitori per avermi supportato in tutti i modi possibili.

LIST OF PUBLICATIONS

The complete list of publications is reported below. The ⋆ symbol highlights the publications discussed in this dissertation.

INTERNATIONAL JOURNAL PAPERS

J1 - **D. Di Nucci**, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia. *A Developer Centered Bug Prediction Model*. IEEE Transactions on Software Engineering (TSE), 2018, Volume 44 Issue 1, 21 pages, 5-24. ⋆

J2 - **D. Di Nucci**, F. Palomba, R. Oliveto, A. De Lucia. *Dynamic Selection of Classifiers in Bug Prediction: an Adaptive Method.*. IEEE Transactions on Emerging Topics in Computational Intelligence (TETCI), 2017, Volume 1 Issue 3, 11 pages, 202-212. ⋆

J3 - **D. Di Nucci**, F. Palomba, A. Panichella, A. Zaidman, A. De Lucia. *On the Impact of Code Smells on the Energy Consumption of Mobile Applications*. Submitted to Elsevier Information and Software Technology (IST).⋆

J4 - **D. Di Nucci**, A. Panichella, A. Zaidman, A. De Lucia. *A Test Case Prioritization Genetic Algorithm guided by the Hypervolume Indicator*. Submitted to IEEE Transactions on Software Engineering (TSE).⋆

INTERNATIONAL CONFERENCE PAPERS

C1 - **D. Di Nucci**, A. Panichella, A. Zaidman, A. De Lucia. *Hypervolume-based Search for Test Case Prioritization*. Proceedings of the Symposium on Search-Based Software Engineering (SSBSE 2015), - Bergamo, Italy, 2015, 15 pages, 157-172. ⋆

C2 - F. Palomba, **D. Di Nucci**, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. De Lucia. *Landfill: an Open Dataset of Code Smells with Public Evaluation*. Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR 2015) - Florence, Italy, 2015, 4 pages, 482-485.

C3 - **D. Di Nucci**, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, A. De Lucia. *On the Role of Developer's Scattered Changes in Bug Prediction*. In Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME 2015) - Bremen, Germany, 2015, 10 pages, 241-250.⋆

C4 - F. Palomba, **D. Di Nucci**, A. Panichella, R. Oliveto, A. De Lucia. *On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study*. In Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST 2016) - Austin, TX, United States, 2016, 10 pages, 5-14.

C5 - S. Scalabrino, G. Grano, **D. Di Nucci**, R. Oliveto, A. De Lucia. *Search-based Testing of Procedural Programs: Iterative Single-Target or Multi-Target Approach?* In Proceedings of the Symposium on Search-Based Software Engineering (SSBSE 2016) - Raleigh, NC, United States, 2016, 15 pages, 64-79.

C6 - **D. Di Nucci**, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia. *Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable?* In Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017) - Klagenfurt, Austria, 2017, 12 pages, 103-114.⋆

C7 - F. Palomba, **D. Di Nucci**, A. Panichella, A. Zaidman, A. De Lucia. *Lightweight Detection of Android-specific Code Smells: the aDoctor Project*. In Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017) - Tool Track, Klagenfurt, Austria, 2017, 5 pages, 487-491.⋆

C8 - **D. Di Nucci**, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia. *PETrA: a Software-Based Tool for Estimating the Energy Profile of Android Applications*. In Proceedings of the 39th International Conference on Software Engineering (ICSE 2017) – Demonstrations Track, Buenos Aires, Argentina, 2017, 4 pages, 3-6.⋆

C9 - **D. Di Nucci**, A. De Lucia. *The Role of Meta-Learners in the Adaptive Selection of Classifiers*. In Proceedings of the 2nd Workshop on Machine Learning for Software Quality Evaluation (MaLTeSQuE) – Campobasso, Italy, 2018, 6 pages, to appear.

C10 - **D. Di Nucci**, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia. *Detecting Code Smells using Machine Learning Techniques: Are We There Yet?*. In Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018) – Campobasso, Italy, 2018, 10 pages, to appear.

CONTENTS

1

PROBLEM STATEMENT

1.1 CONTEXT AND MOTIVATION

Software testing is widely recognized as an essential part of any software development process, representing however an extremely expensive activity. The overall cost of testing has been estimated at being at least half of the entire development cost, if not more [1]. Despite its importance, however, recent studies [2, 3] showed that developers rarely test their application and most programming sessions end without any test execution. Indeed, new methods and tools able to better allocating the developers effort are needed in order to increase the system reliability and to reduce the testing costs.

The resources available should be allocated effectively upon the portions of the source code that are more likely to contain bugs. One of the most powerful techniques aimed at dealing with the testing-resource allocation is the creation of *bug prediction models* [4] which allow to predict the software components that are more likely to contain bugs and need to be tested more extensively. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "not buggy" code components. The first set of techniques exploits *product metrics* (*i.e.*, metrics capturing intrinsic characteristics of the code components, like their size and complexity) [5, 6, 7, 8, 9, 10, 11], while the second one focuses on *process metrics* (*i.e.*, metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [12, 13, 14, 15, 16, 17, 18, 19, 20]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [13, 17, 21] there is a general consensus on the fact that no technique is the best in all contexts [22, 23]. For this reason, the research community is still spending effort in investigating under which circumstances

and during which coding activities developers tend to introduce bugs (see *e.g.*, [24, 25, 26, 27, 28, 29, 30]).

Bug prediction models rely on machine learning classifiers (*e.g.,* Logistic Regression [31]). The model can be trained using a sufficiently large amount of data available from the project under analysis, *i.e., within-project* strategy, or using data coming from other (similar) software projects, *i.e., cross-project* strategy. A factor that strongly influences the accuracy of bug prediction models is represented by the classifier used to predict buggy components. Specifically, Ghotra *et al.* [32] found that the accuracy of a bug prediction model can increase or decrease up to 30% depending on the type of classification applied [32]. Also, Panichella *et al.* [33] demonstrated that the predictions of different classifiers are highly complementary despite the similar prediction accuracy. Based on such findings, an emerging trend is the definition of prediction models which are able to combine multiple classifiers (a.k.a., *ensemble* techniques [34]) and their application to bug prediction [33, 35, 36, 37, 38, 39, 40, 41]. Such models can be trained using a sufficient amount of labeled data coming from (i) the previous history of the same project where the model is applied to, *i.e.,* using a *within-project* strategy, or (ii) other similar projects, *i.e.,* using a *cross-project* strategy. Previous studies showed how within-project bug prediction models have higher capabilities than cross-project ones since they rely on data that better represents the characteristics of the source code elements of the project where the model have to be applied [42]. As a drawback, a within-project training strategy cannot often be adopted in practice since new projects might not have enough data to setup a bug prediction model [43]. As a consequence, the research community has started investigating ways to make cross-project bug prediction models more effective with the aim of allowing a wider adoption of bug prediction models [44, 45]. For instance, Menzies *et al.* [40] recently proposed the concept of *local* bug prediction introducing a technique that (i) firstly clusters homogeneous data coming from different projects with the aim of reducing its heterogeneity and (ii) then builds for each cluster a different model using the Naive Bayes classifier [46]. The empirical analyses showed how such technique can significantly improve the performances of cross-project bug prediction models as well as the performance of effort estimation models [47, 48, 49], thus paving the way for a re-visitation of cross-project techniques.

Another effective way for reducing the testing effort is to optimize regression testing [50, 51], which remains a particular expensive post-maintenance

activity [52]. One of these approaches is *test case prioritization* (TCP) [53, 54], whose goal is to execute the available test cases in a specific order that increases the likelihood of revealing regression faults earlier [55]. Since fault detection capability is unknown before test execution, most of the proposed techniques for TCP use coverage criteria [50] as surrogates with the idea that test cases with higher code coverage will have a higher probability to reveal faults. Once a coverage criterion is chosen, search algorithms can be applied to find the ordering maximizing the selected criterion.

Finally, testing is not only related to the functional properties of the software system, but also on its non-functional properties. Among these, energy efficiency is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. Although the problem is mainly concerned with hardware efficiency, researchers have recently shown how even software may be at the root of energy leaks [56]. The problem is even more evident in the context of mobile applications (*a.k.a.*, "apps"), where billions of customers rely on smartphones every day for social and emergency connectivity [57].

As well as functional testing, also testing the energy efficiency of mobile apps is particularly expensive. For this reason, it would be reasonable to focus the testing on those software components that are more likely to consume energy. In the context of mobile apps development, a set of new peculiar bad programming practices of Android developers has been defined by Reimann *et al.* [58]. These Android-specific smells may threat several non-functional attributes of mobile apps, such as security, data integrity, and source code quality [58]. As highlighted by Hetch *et al.* [59], these type of smells can also lead to performance issues.

1.2 RESEARCH STATEMENT

Even though the effort devoted by the research community to focus and prioritizing the testing effort through the conduction of empirical studies and the definition of new approaches led to interesting results, in the context of our research we highlighted some aspects that might be improved, summarized as follow:

- **Chapter 3 - A Developer Centered Bug Prediction Model**. Although previous studies showed the potential of human-related factors in bug

prediction, this information is not captured in state-of-the-art models based on process metrics extracted from version history. Indeed, these models exploit predictors based on (i) the number of developers working on a code component [15, 16]; (ii) the analysis of change-proneness [17, 18, 21]; (iii) the entropy of changes [14], and (iv) the micro interaction metrics that leverage the developer interaction information [19, 20]. Thus, despite the previously discussed finding by Posnett *et al.* [30], none of the proposed bug prediction models considers how focused the developers performing changes are and how scattered these changes are.

- **Chapter 4 - Dynamic Selection of Classifiers in Bug prediction**. As highlighted by Bowes *et al.* [60], traditional ensemble approaches miss the predictions of a large part of bugs that are correctly identified by a single classifier and, therefore, "ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of bugs" [60]. Moreover, previous empirical studies on the use of ensemble techniques for bug prediction [37, 35, 61] have some critical limitations that could have impacted on the performances of classifiers thus possibly threating the conclusions provided so far. These issue were related to (i) data quality, (ii) data preprocessing, (iii) data analysis, and (iv) limited size. Furthermore, these studies exposed an unclear relationship between local learning and ensemble classifiers and did not compare the performance achieved by cross-project models with respect to within-project ones.

- **Chapter 5 - Hypervolume Genetic Algorithm for Test Case Prioritization**. We observed that the AUC metric used in the related literature for TCP represents a simplified version of the well-known *hypervolume* [62], which is a metric used in many-objective optimization. We argue that the *hypervolume* can be used to condense not only a single cumulative code coverage criteria (as done by previous AUC metrics used in TCP literature) but also multiple testing criteria, such as the test case execution cost or further coverage criteria (*e.g.*, branch and past-fault coverage), in only one scalar value and it can be used as fitness function in a search-based algorithm.

- **Chapter 6 - On the Impact of Code Smells on the Energy Consumption of Mobile Applications**. Although several important research steps have

been made and despite the ever-increasing number of empirical studies aimed at understanding the reasons behind the presence of energy leaks in the source code, little knowledge is available in literature on the potential impact on energy consumption of the Android-specific code smells defined by Reimann *et al.* [58]. These smells are detectable through static analysis, hence they could be used to efficiently detect energy leaks. Unfortunately, while the impact of these smells on energy consumption has been theoretically supposed by Reimann *et al.* [58], there exists only a few empirical evidence on it. For this reason, we aim at conducting a large empirical study to analyze the impact of the Android-specific smells on the energy consumption of mobile apps.

The work presented in this thesis has the goal to overcome the limitations mentioned above, by performing large-scale empirical investigations and defining new approaches. Specifically, the high-level research questions considered in this thesis are the following:

- **Chapter 3**: *To what extent developer's scattering metrics are able to improve a bug prediction model based on state-of-the-art metrics?*
- **Chapter 4**: *To what extent a technique able to select a classifier based on the characteristics of the class is able to out-perform state-of-the-art classifiers?*
- **Chapter 5**: *What are the cost-effectiveness, the efficiency, and scalability of a genetic algorithm based on the hypervolume, compared to state-of-the-art test case prioritization techniques?*
- **Chapter 6**: *To what extent Android-specific code smells can be used to focus energy testing of mobile apps?*

These research question are further detailed in the related chapters. The final goal is to provide developers new approaches and tools, able to (i) focus their effort on bug-prone components, (ii) produce test cases permutations able to find faults earlier, and (iii) quickly detect energy flaws in the source code of mobile apps.

1.3 RESEARCH CONTRIBUTION

This thesis provides several contributions aimed at answering the research questions formulated in the previous Section. Basically, the contribution of our

work concern four aspects. Table 1.1 reports, for each aspect, the list of the references to papers published or submitted. Specifically:

Table 1.1: Summary of the Thesis Contribution

Aspect	Ref.
A Developer Centered Bug Prediction Model	[63, 64]
Dynamic Selection of Classifiers in Bug Prediction	[65]
Hypervolume Genetic Algorithm fo Test Case Prioritization	[66, 67]
On the Impact of Code Smells on the Energy Consumption of Mobile Applications	[68, 69, 70, 71]

1. **A Developer Centered Bug Prediction Model**. To answer the research questions in Chapter 3, we defined two metrics, namely the DEVELOPER'S STRUCTURAL AND SEMANTIC SCATTERING. The first assesses how "structurally far" in the software project the code components modified by a developer in a given time period are. The second measure (*i.e.*, the *semantic scattering*) is instead meant to capture how much spread in terms of implemented responsibilities the code components modified by a developer in a given time period are. The conjecture behind the proposed metrics is that high levels of STRUCTURAL AND SEMANTIC SCATTERING make the developer more error-prone. To verify this conjecture, we built two predictors exploiting the proposed metrics, and we used them in a bug prediction model. Firstly, we conducted a case study on 26 software systems and compared our model with respect to four models based on state-of-the-art metrics. Secondly, we devised and discussed the results of an hybrid bug prediction model, based on the best combination of predictors exploited by the five prediction models experimented. Publications [63] and [64] discuss the contribution.

2. **Dynamic Selection of Classifiers in Bug Prediction**. To answer the research questions in Chapter 4, we proposed a novel adaptive prediction model, coined as ASCI (**A**daptive **S**election of **C**lass**I**fiers in bug prediction), which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class. To build and evaluate our approach, we carry out a preliminary investigation aimed at understanding whether a set of five different classifiers is complementary in the context of within-project bug prediction. We provided a *differentiated* replication study [72] in which not only we

corroborate previous empirical research on the performances of ensemble classifiers for cross-project bug prediction, but also extend previous knowledge by assessing the extent to which local bug prediction can benefit of the usage of ensemble techniques. The study was conducted on a PROMISE dataset composed of 21 software systems, where we applied a number of corrections suggested by Shepperd *et al.* [73] to make it cleaned and suitable for our purpose. We took into account several different ensemble techniques, belonging to six categories, measuring their performances using the two metrics recommended by previous work, *i.e.*, AUC-ROC and Matthew's Correlation Coefficient [74, 75]. Publications [65] and [76] present this contribution.

3. **Hypervolume Genetic Algorithm for Test Case Prioritization**. To answer the research questions in Chapter 5, we proposed HGA (HYPERVOLUME-BASED GENETIC ALGORITHM) and provided an extensive evaluation of Hypervolume-based and state-of-the-art approaches for TCP when dealing with up to five testing criteria. In particular, we carried out two different case studies to assess the cost-effectiveness, the efficiency, and the scalability of the various approaches. In the first study, we compared HGA with respect to two state-of-the-art techniques: a cost cognizant additional greedy algorithm [53, 77]; and `NSGA-II`, a multi-objective search-based algorithm [78, 79, 80, 81]. To assess the scalability of HGA, we conducted a second case study by considering up to five testing criteria (*i.e.*, objectives). This further evaluation is needed since previous studies [82, 83] showed the benefits of optimizing multiple criteria in regression testing with respect to considering each criterion individually. Moreover, a well-known problem in many-objective optimization is that traditional multi-objective evolutionary algorithms do not scale when handling more than three criteria. This is because the number of non-dominated solutions increases exponentially with the number of objectives [84]. Therefore, we compared the performance of HGA with those achieved by two many-objective algorithms. Publications [66] and [67] discuss this contribution.

4. **On the Impact of Code Smells on the Energy Consumption of Mobile Applications**. To answer the research questions in Chapter 6, we proposed two novel tools and a large empirical study. ADOCTOR, a novel code smell detector that identifies 15 Android-specific code smells. The tool exploits

the Abstract Syntax Tree of the source code and navigates it by applying detection rules based on the exact definitions of the smells provided by Reimann *et al.* [58]. We also conducted an empirical study to evaluate the overall ability of our tool in recommending portions of code affected by a design flaw. In particular, we experimented aDoctor against the source code of 18 Android apps and compared the set of candidate code smells given by the tool with a manually-built oracle. PETrA is a novel tool for extracting the energy profile of mobile applications specific for Android OS. We evaluated PETrA on 54 mobile applications belonging to a publicly available dataset [85]. We compared the energy measurements provided by PETrA against the actual energy consumption computed using the Monsoon hardware toolkit [86] for the same apps and using the same hardware/software setting. We provided a deeper investigation to determine (i) to what extent code smells affecting source code methods of mobile applications influence energy efficiency, and (ii) whether refactoring operations applied to remove them directly improve the energy efficiency of refactored methods. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann *et al.* [58] in the context of 60 Android apps belonging to the dataset provided by Choudhary *et al.* [87]. To the best of our knowledge, this is up to date the largest study aimed at practically investigating the actual impact of such code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency. Publications [68], [69], [70], and [71] present this contribution.

Besides the contributions described above, two further common contributions were made:

- **Large-scale Empirical Studies**. All the studies conducted in our research have been conducted on a large set of software systems, in order to ensure the generalizability of the findings.

- **Publicly Available Tools and Replication Packages**. The construction of several tools, scripts, and datasets was needed to effectively perform the analyses. We made all of them publicly available by providing tools (publications [70, 68]) and online replication packages.

1.4 STRUCTURE OF THE THESIS

Before describing in depth the single contributions of this thesis, Chapter 2 overviews the related literature on (i) bug prediction, (ii) test case prioritization, and (iii) energy efficiency and code smells. The core of the thesis is organized in four main blocks, each reported in Table 1.1. Specifically:

- *A Developer Centered Bug Prediction Model.* Chapter 3 describes two new metrics for representing developers scattered changes. It also reports the study aimed at evaluating the performances of the bug prediction models based on these metrics and those of an *hybrid* model, based on the best combination of predictors exploited in the study.

- *Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method.* Chapter 4 presents ASCI: a novel adaptive prediction model, which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class. It also reports the study aimed at evaluating the performances of the bug prediction models based on this classifier in contexts of within-project and cross-project bug prediction.

- *A Test Case Prioritization Genetic Algorithm guided by the Hypervolume Indicator.* Chapter 5 presents HGA: a genetic algorithm that use as fitness function the hypervolume indicator. Two studies aimed at evaluating the cost-effectiveness, efficiency, and scalability of HGA are also reported.

- *On the Impact of Code Smells on the Energy Consumption of Mobile Applications.* Chapter 6 provides a large empirical study to determine (i) to what extent code smells detected by aDoctor influence energy efficiency, and (ii) whether the refactoring operations applied to remove them are able to improve the energy efficiency of refactored methods. To conduct our analyses, we built upon two tools that we previously developed and evaluated, *i.e.*, PETRA and ADOCTOR. The former is a novel *Android-specific* code smell detector, while the latter is a software-based tool that estimates the energy profile of mobile applications [69].

Finally, Chapter 7 concludes the thesis and discusses the future directions and challenges for reducing the effort required by testing activities.

2

BACKGROUND AND RELATED WORK

Software testing is widely recognized as an essential part of any software development process, representing however an extremely expensive activity. The overall cost of testing has been estimated at being at least half of the entire development cost, if not more [1]. Writing good test cases able to test the code components more prone to be affected by defects represents probably the most expensive activity in the entire testing process. Hence, several techniques able to reduce the testing effort received more and more attention by researchers and practitioners in order to increment the system reliability and to reduce testing costs. This chapter provides a comprehensive literature review about (i) bug prediction, (ii) test case prioritization, and (iii) bad smell able to discover non functional issues such as energy leakages.

2.1 BUG PREDICTION

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. This section discusses the related literature about metrics, training strategies and classifiers for bug prediction.

2.1.1 *Metrics*

Many bug prediction techniques have been proposed in the literature in the last decade. Such techniques mainly differ for the specific predictors they use, and can roughly be classified in those exploiting *product metrics* (*e.g.*, lines of code, code complexity, etc), those relying on *process metrics* (*e.g.*, change- and fault-proneness of code components), and those exploiting a mix of the two. Table 2.1 summarizes the related literature, by grouping the proposed techniques on the basis of the metrics they exploit as predictors.

Table 2.1: Prediction models proposed in literature

Type of Information Exploited	Prediction Model	Predictors
Product metrics	Basili *et al.* [5]	CK metrics
	El Emam *et al.* [88]	CK metrics
	Subramanyam *et al.* [89]	CK metrics
	Nikora *et al.* [90]	CFG metrics
	Gyimothy *et al.* [6]	CK metrics, LOC
	Ohlsson *et al.* [7]	CFG metrics, complexity metrics, LOC
	Zhou *et al.* [91]	CK metrics, OO metrics, complexity metrics, LOC
	Nagappan *et al.* [22]	CK metrics, CFG metrics, complexity metrics
	Hall *et al.* [10]	Code smell
	Bowes *et al.* [11]	Mutants
Process metrics	Khoshgoftaar *et al.* [12]	debug churn
	Nagappan *et al.* [92]	relative code churn
	Hassan and Holt [93]	entropy of changes
	Hassan and Holt [94]	entropy of changes
	Kim *et al.* [95]	previous fault location
	Hassan [14]	entropy of changes
	Ostrand *et al.* [16]	number of developers
	Nagappan *et al.* [96]	consecutive changes
	Bird *et al.* [28]	social network analysis on developers' activities
	Ostrand *et al.* [15]	number of developers
	Posnett *et al.* [30]	module activity focus, developer attention focus
	Lee *et al.* [19, 20]	micro interactions
Product and process metrics	Graves *et al.* [13]	various code and change metrics
	Nagappan and Ball [8]	LOC, past defects
	Bell *et al.* [97]	LOC, age of files, number of changes, program type
	Zimmerman *et al.* [9]	complexity metrics, CFG metrics, past defects
	Moser *et al.* [21]	various code and change metrics
	Moser *et al.* [17]	various code and change metrics
	Bell *et al.* [18]	various code and change metrics
	D'Ambros *et al.* [23]	various code and change metrics

The Chidamber and Kemerer (CK) metrics [98] have been widely used in the context of bug prediction. Basili *et al.* [5] investigated the usefulness of the CK suite for predicting the probability of detecting faulty classes. They showed that five of the experimented metrics are actually useful in characterizing the bug-proneness of classes. The same set of metrics has been successfully exploited in the context of bug prediction by El Emam *et al.* [88] and Subramanyam *et al.* [89]. Both works reported the ability of the CK metrics in predicting buggy code components, regardless of the size of the system under analysis.

Still in terms of product metrics, Nikora *et al.* [90] showed that measuring the evolution of structural attributes (*e.g.*, number of executable statements, number

of nodes in the control flow graph, etc.) it is possible to predict the number of bugs introduced during the system development. Later, Gyimothy *et al.* [6] performed a new investigation on the relationship between CK metrics and bug proneness. Their results showed that the *Coupling Between Object* metric is the best in predicting the bug-proneness of classes, while other CK metrics are untrustworthy.

Ohlsson *et al.* [7] focused the attention on the use of design metrics to identify bug-prone modules. They performed a study on an Ericsson industrial system showing that at least four different design metrics can be used with equivalent results. The metrics performance are not statistically worse than those achieved using a model based on the project size. Zhou *et al.* [91] confirmed their results showing that size-based models seem to perform as well as those based on CK metrics except than the *Weighted Method per Class* on some releases of the Eclipse system. Thus, although Bell *et al.* [97] showed that more complex metric-based models have more predictive power with respect to size-based models, the latter seem to be generally useful for bug prediction.

Nagappan and Ball [8] exploited two static analysis tools to early predict the pre-release bug density. The results of their study, conducted on the Windows Server system, show that it is possible to perform a coarse grained classification between high and low quality components with a high level of accuracy. Nagappan *et al.* [22] analyzed several complexity measures on five Microsoft software systems, showing that there is no evidence that a single set of measures can act universally as bug predictor. They also showed how to methodically build regression models based on similar projects in order to achieve better results. Complexity metrics in the context of bug prediction are also the focus of the work by Zimmerman *et al.* [9]. Their study reports a positive correlation between code complexity and bugs.

Hall *et al.* investigated the relationship between faults and code smells. Their results suggest that some smells have a positive correlation with an increased number of faults. However, where smells did significantly affect faults, the size of that effect was small (always under 10 percent). They conclude that arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness.

Bowes *et al* [11] introduced mutation-aware fault prediction, which leverages additional guidance from metrics constructed in terms of mutants and the test cases that cover and detect them. Their empirical study on 3 large real-world

systems (both open and closed source) showed that mutation-aware predictors can improve fault prediction performance.

Differently from the previous discussed techniques, other approaches try to predict bugs by exploiting process metrics. Khoshgoftaar *et al.* [12] analyzed the contribution of debug churns (defined as the number of lines of code added or changed to fix bugs) to a model based on product metrics in the identification of bug-prone modules. Their study, conducted on two subsequent releases of a large legacy system, shows that modules exceeding a defined threshold of debug churns are often bug-prone. The reported results show a misclassification rate of just 21%.

Nagappan *et al.* [92] proposed a technique for early bug prediction based on the use of relative code churn measures. These metrics relate the number of churns to other factors such as LOC or file count. An experiment performed on the Windows Server system showed that relative churns are better than absolute value.

Hassan and Holt [93] conjectured that a chaotic development process has bad effects on source code quality and introduced the concept of entropy of changes. Later they also presented the top-10 list [94], a methodology to highlight to managers the top ten subsystems more likely to present bugs. The set of heuristics behind their approach includes a number of process metrics, such as considering the *most recently modified*, the *most frequently modified*, the *most recently fixed* and the *most frequently fixed* subsystems.

Bell *et al.* [18] pointed out that although code churns are very effective bug predictors, they cannot improve a simpler model based on the code components' change-proneness. Kim *et al.* [95] presumed that faults do not occur in isolation but in burst of related faults. They proposed the bug cache algorithm that predicts future faults considering the location of previous faults. Similarly, Nagappan *et al.* [96] defined change burst as a set of *consecutive changes over a period of time* and proposed new metrics based on change burst. The evaluation of the prediction capabilities of the models was performed on Windows Vista, achieving high accuracy.

Graves *et al.* [13] experimented both product and process metrics for bug prediction. They observed that history-based metrics are more powerful than product metrics (*i.e.*, change-proneness is a better indicator than LOC). Their best results were achieved using a combination of module's age and number

of changes, while combining product metrics had no positive effect on the bug prediction. They also saw no benefits provided by the inclusion of a metric based on the number of developers modifying a code component.

Moser *et al.* [21] performed a comparative study between product-based and process-based predictors. Their study, performed on Eclipse, highlights the superiority of process metrics in predicting buggy code components. Later, they performed a deeper study [17] on the bug prediction accuracy of process metrics, reporting that the *past number of bug-fixes performed on a file* (*i.e.*, bug-proneness), the *maximum changeset size occurred in a given period*, and the *number of changes involving a file in a given period* (*i.e.*, change-proneness) are the process metrics ensuring the best performances in bug prediction.

D'Ambros *et al.* [23] performed an extensive comparison of bug prediction approaches relying on process and product metrics, showing that no technique based on a single metric works better in all contexts.

Hassan [14] analyzed the complexity of the development process. In particular he defined the entropy of changes as the scattering of code changes across time. He proposed three bug prediction models, namely Basic Code Change Model (BCCM), Extended Code Change Model (ECCM), and File Code Change Model (FCCM). These models mainly differ for the choice of the temporal interval where the bug proneness of components is studied. The reported study indicates that the proposed techniques have a stronger prediction capability than a model purely based on the amount of changes applied to code components or on the number of prior faults.

Ostrand *et al.* [15, 16] proposed the use of the *number of developers who modified a code component in a give time period* as a bug predictor. Their results show that combining developers' information poorly, but positively, impact the detection accuracy of a prediction model. Our work does not use a simple count information of developers who worked on a file, but also takes in consideration the change activities they carry out.

Bird *et al.* [28] investigated the relationship between different ownership measures and pre- and post-releases failures. Specifically, they analyzed the developers' contribution network by means of social network analysis metrics, finding that developers having low levels of ownership tend to increase the likelihood of introducing defects. Our scattering metrics are not based on code

ownership, but on the "distance" between the code components modified by a developer in a given time period.

The "focus" metrics presented by Posnett *et al.* [30] is based on the idea that a developer performing most of her/his activities on a single module (a module could be a method, a class, etc.) has a higher focus on the activities she/he is performing and is less likely to introduce bugs. Following this conjecture, they defined two symmetric metrics, namely the Module Activity Focus metric (shortly, MAF), and the Developer Attention Focus metric (shortly, DAF) [30]. The former is a metric which captures to what extent a module receives focused attention by developers. The latter measures how focused are the activities of a specific developer.

Lee *et al.* firstly considered the effects of developer behavior on bug prediction. They proposed the micro interaction metrics, that leverage developer interaction information. The developer interactions, such as file editing and browsing events in task sessions, are captured and stored as information by Mylyn. Their experimental evaluation demonstrates that these metrics can significantly improve the overall bug prediction accuracy when combined with existing software measures.

2.1.2 *Training Strategies*

Prediction approaches can be defined by training a classification model on past data of the same software project (*within-project* strategy) [36, 37, 38, 60, 99, 100, 101, 102, 103, 104] or belonging to different projects (*cross-project* strategy) [33, 35, 39, 40, 41, 61, 105, 43].

Within-project strategy can be applied only on mature projects, where a sufficiently large amount of project history (*i.e.*, past faults) is available. Thus, such a strategy cannot be used on new projects. In this scenario, the cross-project strategy can be used. The main problem of the cross-project strategy is represented by the heterogeneity of data [44, 106]. To overcome this issue, Menzies *et al.* [40] introduced the concept of local bug prediction. This approach firstly cluster software components into homogeneous groups to reduce the differences among such classes, and then train the model on each cluster. With the same purpose, Nam *et al.* [107] introduced the concept of heterogeneous

defect prediction, that is building defect prediction models using different metrics in different project.

2.1.3 *The Choice of the Classifier*

2.1.3.1 *Classifiers in Machine Learning*

A general prediction model can be viewed as a function, which takes as input a set of predictors and returns a scalar value that measures the likelihood that a specific software entity is defect-prone. The estimated defect proneness is used to rank the source code entities of a system in order to identify the most defect-prone entities to be investigated in quality assurance activities. The difference between the various machine learning techniques depends on the specific function that is used to map the predictors to the estimated defect proneness. In this section we briefly analyze five machine learning techniques, namely Logistic Regression [108], Naive Bayes [109], Radial Basis Function Network [110], Multi-layer Perceptron [111], Decision Tree [112], and Support Vector Machine [113].

Logistic Regression (LOG) [108] is one of the most used machine learning techniques for predicting the defect proneness of software entities. This classifier is a generalization of the linear regression to binary classification, *i.e.,* either a file is defect-prone or it is not in our case. Its general equation is the following:

$$P(c_i) = \frac{e^{\alpha+B_1x_1+B_2x_2+\cdots+B_nx_n}}{1+e^{\alpha+B_1x_1+B_2x_2+\cdots+B_nx_n}} \tag{2.1}$$

where $P(c_i)$ is the estimated defect proneness of the entity c_i, the scalars $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ are linear coefficient and $x_1, x_2, \ldots, x_n$ are the predictors *i.e.,* the software metrics. Since the equation cannot be solved analytically, the maximum likelihood procedure is used to estimate the coefficients that minimize the prediction error.

A Bayesian network is a directed graph composed of V vertices and E edges, where each vertex represents a predictor (*i.e.,* software metric) and each edge denotes the causal relationship of one predictor to its successor in the network. Since the search space tends to grow exponentially when the number of nodes increases, heuristic algorithms are generally used to find the network configura-

tion that best fits the training data. Naive Bayes [109] is a simpler technique that assume that the value of a particular feature is independent of the value of any other feature, given the class variable.

Radial Basis Function Network (RBF) [110] is a type of neural network which uses radial basis function as activation functions. It has three different layers: (i) the input layer which corresponds to the predictors, (*i.e.,* software metrics; (ii) the output layer which maps the outcomes to predict, *i.e.,* the defect proneness of entities; (iii) the hidden layer used to connect the input layer with the output layer. The radial basis function used to activate the hidden layer is a Gaussian function. The prediction model is defined as follows:

$$P(c_i) = \sum_{k=1}^{n} \alpha_k \epsilon^{\frac{-\|x-\gamma_k\|}{\beta}} \tag{2.2}$$

where $P(c_i)$ is the predicted defect proneness of the entity c_i, α_1, α_2, ..., α_n is a set of linear weights and γ_k are the centers of the radial basis function. Finally $\epsilon^{\frac{-\|x-\gamma_k\|}{\beta}}$ is the radial basis function.

Multi-layer Perceptron (MLP) [111] is another type of neural network that is trained using a back-propagation algorithm. In general multi-layer perceptron consists of multiple layers of nodes in a directed graph: an input layer, one or more hidden layers, and one output layer. The output from a layer is used as input to nodes in the subsequent layer. The formal definition of the model has the following mathematical formulation:

$$P(c_i) = \sum_{k=1}^{n} w_k \frac{1}{1 + \epsilon^{\alpha + B_1 x_1 + B_2 x_2 + \cdots + B_n x_n}} \tag{2.3}$$

where $P(c_i)$ is the predicted defect proneness of the entity c_i, α_1, α_2, ..., α_n is a set of linear weights and w_k are the weights of each layer.

A Decision Tree (DTree) [112] consists of a tree structure composed of leaf and decision nodes: leaf nodes contain the predicting outcomes, *i.e.,* entity defect-proneness, while each decision nodes contain the a specific decision rule. When a given instance has to be classified, the tree is traversed from the root node to bottom until a leaf node is reached. Each leaf node is a linear regression model. Hence, the classification of a given instance is performed by following

all paths for which all decision nodes are true and summing the predicting values that are traversed along the corresponding path.

A Support Vector Machine (SVM) [113] is a discriminative classifier formally defined by a separating hyperplane. Given a set of training instances, the algorithm builds an optimal hyperplane which categorizes new examples. In a two dimensional space the hyperplane is a line dividing a plane in two parts where in each class lay in either side. The learning of the hyperplane in linear SVM is performed by a kernel function. SVM is effective in high dimensional spaces, also if the number of dimensions is higher than the number of samples. Moreover, different kernel functions can be specified for the decision function.

2.1.3.2 *Classifiers for Bug Prediction*

Several machine learning classifiers have been used in literature [4], *e.g.,* Logistic Regression (LOG), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), Bayesian Network (BN), Decision Trees (DTree)).

However, results of previous studies demonstrated no clear winner among these classifiers [99, 100, 101, 102, 103]. In particular, depending on the dataset employed, researchers have found different classifiers achieving higher performances with respect to the others, *i.e.,* (i) RBF and its modified version, namely RBF trained with enhanced Dynamic Decay Adjustment algorithm (RBF-eDDA) [99, 100], (ii) Dynamic Evolving Neuro-Fuzzy Inference System (DENFIS), Support Vector Regression (SVR), and Regression Tree (RT) [101], (iii) ADTrees [102], (iv) Naive Bayes [103], and (v) Multilayer Perceptron [103].

Moreover, Lessman *et al.* [114] conducted an empirical study with 22 classification models to predict the bug proneness of 10 publicly available software development data sets from the NASA repository, reporting no statistical differences among the top-17 models. As demonstrated by Shepperd *et al.* [115], the NASA dataset that is used in the work by Lessman *et al.* [114] was noisy and biased. A subsequent investigation on the cleaned NASA dataset performed by Ghotra *et al.* [32] found that the impact of classifiers on the performance of a bug prediction model is instead relevant. Indeed, the performance of a bug prediction model can increase or decrease up to 30% depending on the type of classifier applied [32].

2.1.3.3 *Ensemble of Classifiers in Machine Learning*

Ensemble of classifiers have been used with the aim of combining different classifiers to achieve better classification performances. In the following, we report some of the most used ensemble techniques along with their usage in the context of defect prediction.

A Boosting technique iteratively use the models built in previous iterations to manipulate the training set [34]. In this way the focus of the next iteration of the model is on those instances that are more difficult to predict. A well-known Boosting technique is Adaptive Boosting (AdaBoost) [116]. During the training phase, AdaBoost repetitively trains a *weak* classifier, *i.e.,* a learner which is only slightly more accurate than a random one, on subsequent training data. Specifically, at each iteration a weight is assigned to each instance of the training set. Higher weights are assigned to misclassified instances of the training set that thus have more chances to be correctly predicted by the new models. At the end of the training phase, a weight is assigned to each model in a way that models having higher overall accuracy have higher weights. It is important to note that the overall accuracy takes in consideration the weights of the instances, thus the models able to correctly classify the hardest instances are rewarded. During the test phase, the prediction of a new instance is performed by voting of all models. The results are thus combined using the weights of the models, in case of binary classification a threshold of 0.5 is applied.

Bootstrap Aggregating (Bagging) [117] creates a composite classifier combining the output of various models in a single prediction. During the training phase, m datasets with the same size as the original one are generated by performing sampling with replacement (Bootstrap) from the training set. Hence m models are trained using a *weak* classifier on the m datasets. During the test phase, for each instance the composite classifier use a majority voting rule to combine the output of the m models into a single prediction.

Validation and Voting [118] (also called Voting) is a weighting method. It combines the confidence scores obtained by the underlying classifiers. Indeed, each classifier returns for each instance a confidence score between 0 and 1. An operator combines the scores and an instance is predicted as buggy if the combined score is larger than 0.5, while it is predicted as clean otherwise.

CODEP [33] is a Stacking technique [119] that uses a meta-classifier (*e.g.,* Logistic Regression) to infer the bugginess of classes [34]. In particular, during

the training phase CODEP builds the base classifiers on the training set. Then, a new dataset is created by collecting the confidence scores assigned by the classifiers on each instance. Finally, a meta-classifier is built on such a new dataset with the aim of combining the outputs of the base classifiers.

Random Forest [120] depends on an ensemble of pruned decision trees. Each decision tree is built by using111 Bootstrap from the training set as showed for Bagging. Another similarity with respect to Bagging is the combination of the predictions that are performed by the single trees. Indeed this operation is made by using majority voting. Despite this, a major difference between C45Bagging and Random Forest is that even if both apply feature selection. The latter selects, for each tree, a random subset of the features. This process is called *feature bagging* [121].

2.1.3.4 *Ensemble of Classifiers for Bug Prediction*

Based on the fact that different classifiers are able to correctly classify different sets of buggy classes [33], methodologies aimed at combining them have been investigated.

Misirli *et al.* [36] proposed the Validation and Voting approach, which consists of a combination of the results achieved by different classifiers using an aggregating function with the aim of improving the prediction performances. In particular, if the majority of models (obtained using different classifiers on the same training set) predict the bugginess of a class, then it is predicted as bug-prone; otherwise, it is predicted as bug-free. Wang *et al.* [37] benchmarked the performances of 7 ensemble techniques in the context of within-project bug prediction, showing that often Validation and Voting achieves better results. Also, when employed in cross-project bug prediction, this technique seems to provide better results according to the findings by Liu *et al.* [35], who experimented 17 different classifiers.

Besides Validation and Voting, other ensemble techniques have also been proposed. Kim *et al.* [38] and He *et al.* [39] devised approaches similar to the Bagging ensemble technique [34] which combines the outputs of different models trained on a sample of instances taken with a replacement from the training set. Specifically, Kim *et al.* [38] combined multiple training data obtained applying a random sampling, while He *et al.* [39] proposed an approach

to automatically select training data from other projects in the cross-project context.

More recently, some approaches inspired to the STACKING ensemble technique [34] have been proposed [33, 41]. They use a meta-learner to induce which classifiers are reliable and which are not and consider the predictions of different classifiers as input for a new classifier. Specifically, Panichella *et al.* [33] devised CODEP, an approach that firstly applies a set of classifiers independently, and then uses the output of the first step as predictors of a new prediction model based on Logistic Regression. Zhang *et al.* [61] conducted a similar study as the one performed by Panichella *et al.* [33] comparing different ensemble approaches. They found that there exist several ensemble techniques that improve the performances achieved by CODEP, and Validation and Voting is often one of them. Petric *et al.* [41] used 4 families of classifiers in order to build a Stacking ensemble technique [34] based on the diversity among classifiers in the cross-project context. Their empirical study showed that their approach can perform better than other ensemble techniques and that the diversity among classifiers is an essential factor.

Finally, Herbold *et al.* [45] performed a wide replication study, comparing 24 approaches devised for Cross-Project Defect Prediction. In their work they suggest some approaches that should be applied to have better performance such as data normalization, proposed by Camargo Cruz and Ochimizu [122]. Moreover, their results indicate that Cross-Project Defect Predition is not yet ready for being applied in practice.

2.2 TEST CASE PRIORITIZATION

The Test Case Prioritization (TCP) problem consists of generating a test case ordering $\tau' \in PT$ that maximizes fault detection rate f [55]:

Definition 2.1. Given: a test suite T, the set of all permutations PT of test cases in T, and a function f: $PT \rightarrow \mathbb{R}$.
Problem: find $\tau' \in PT$ such that $(\forall \tau'')(\tau'' \in PT)(\tau'' \neq \tau')[f(\tau') \geq f(\tau'')]$

However, the fault detection capability of a test case is not known to the tester before its execution. Therefore, researchers have proposed to use surrogate metrics, which are in some way correlated with the fault detection rate [50],

to determine test case execution ordering. They can be divided in two main categories [123]: *white-box* metrics and *black-box* metrics.

Code coverage is the most widely used metric among *white-box* ones, *e.g.*, branch coverage [53], statement coverage [54], block coverage [124], and function or method coverage [125]. Other prioritization criteria were also used instead of structural coverage, such as interactions [126, 127], requirement coverage [128], statement and branch diversity [129, 130], and additional spanning statement and branches [131].

Other than *white-box* metrics also *black-box* metrics have been proposed. For example, Bryce *et al.* proposed the *t-wise* approach that considers the maximum interactions between t model inputs [132, 133] [134]. Other approaches considered the input diversity calculated using NCD [135], the Jaccard distance [136, 137], and the Levenshtein distance [138, 139] between inputs. Finally, Henard *et al.* considered also the number of killed model mutants [137, 140].

Henard *et al.* [123] compared various *white-box* and *black-box* criteria for TCP, showing that there is a "little difference between black-box and white-box performance".

In all the aforementioned works, once a prioritization criterion is chosen, a greedy algorithm is used to order the test cases according to the chosen criterion. Two main greedy strategies can be applied [141] [142]: the *total* strategy selects test cases according to the number of code elements they cover, whereas the *additional* strategy iteratively selects the test case that yields the maximal coverage of code elements not covered yet by previously selected test cases. Recently, Hao *et al.* [141] and Zhang *et al.* [142] proposed a hybrid approach that combines *total* and *additional* coverage criteria showing that their combination can be more effective than the individual components. Greedy algorithms have also been used to combine multiple testing criteria such as code coverage and cost. For example, Elbaum et al. [143] and Malishevsky *et al.* [144] considered code coverage and execution cost, where the additional greedy algorithm was customized to condense the two objectives in only one function (coverage per unit cost) to maximize. Three-objective greedy algorithms have been also used to combine statement coverage, historical fault coverage and execution cost [50, 83].

2.2.1 *Search-Based Test Case Prioritization*

Other than greedy algorithms, meta-heuristics have been investigated as alternative search algorithms to test case prioritization. Li *et al.* [51] compared additional greedy algorithm, hill climbing and genetic algorithms for code coverage based TCP. To enable the application of meta-heuristics they developed proper fitness functions: APBC (Average Percentage Block Coverage), APDC (Average Percentage Decision Coverage) or APSC (Average Percentage Statement Coverage). For a generic coverage criterion (e.g., branch coverage), the corresponding fitness function is defined as follows:

Definition 2.2. Let $E = \{e_1, \ldots, e_m\}$ be a set of target elements to cover; let $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ be a given test case ordering; and let TE_i be the position of the first test in τ that covers the element $e_i \in E$; the Average Percentage of *Element* Coverage, i.e., the AUC metric, is defined as:

$$APEC = 1 - \frac{\sum_{i=1}^{m} TE_i}{n \times m} + \frac{1}{2 \times n} \tag{2.4}$$

In the definition above, the target elements in E can be branches (Equation 2.2 would correspond to APDC), statements (APSC), basic blocks (APBC), etc. Equation 2.2 condenses the cumulative coverage scores (e.g., branch coverage) achieved when considering the test cases in the given order τ using the Area Under Curve (AUC) metric. This area is delimited by the cumulative points whose y-coordinates are the cumulative coverage scores (e.g., statement coverage) achieved when varying the number of executed test cases (x-coordinates) according to a specified ordering [51].

Equation 2.4 relies on the assumption that all test cases have equal cost. However, such an assumption is unrealistic in practice and, as consequence, test orderings optimizing Equation 2.4 may become sub-optimal when measuring the test execution cost. Hence, a "cost-cognizant" variant of Equation 2.4 has been also used in literature [81]:

Definition 2.3. Let $E = \{e_1, \ldots, e_m\}$ be a set of target elements to cover; let $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ be a given test case ordering; let $C = \{c_1, \ldots, c_m\}$ be the cost of tests in τ; and let TE_i be the position of the first test in τ that covers the

element $e_i \in E$; the "Cost-cognizant" Average Percentage of *Element* Coverage is defined as:

$$APEC_c = \frac{\sum_{i=1}^{m} \left(\sum_{j=TE_i}^{n} c_j - \frac{1}{2} c_{TF_i} \right)}{\sum_{i=1}^{n} c_i \times m} \tag{2.5}$$

When assuming that all tests have the same cost (i.e., $\forall c_i \in C, c_i = 1$), Equation 2.5 becomes equivalent to Equation 2.4 [145]. This "cost-cognizant" variant measures the AUC delimited by the cumulative points whose y-coordinates are the cumulative coverage scores (e.g., statement coverage) while their x-coordinates are the cumulative test execution costs for a specified test ordering τ.

Since these metrics allow to condense multiple cumulative points in only one scalar value, single-objective genetic algorithms can be applied to find an ordering maximizing the AUC. According to the empirical results in [51], in most cases the difference between the effectiveness of permutation-based genetic algorithms and additional greedy approaches is not significant.

2.2.2 *Multi-objective Test Case Prioritization*

Later works highlighted that given the multi-objective nature of the TCP problem, permutation-based genetic algorithms should consider more than one testing criterion. For example, Li *et al.* [78] proposed a two-objective permutation-based genetic algorithm to optimize APSC and execution cost required to reach the maximum statement coverage (cumulative cost). They use a multi-objective genetic algorithm, namely NSGA-II, to find a set of Pareto optimal test case orderings representing optimal compromises between the two corresponding AUC-based criteria.

Based on the concept of *Pareto optimality*, in this formulation of the problem, a test cases permutation τ_A is better than another permutation τ_B, (and vice versa), if and only if τ_A outperforms τ_B in at least one objective and it is not worse in all other objectives. Formally:

Definition 2.4. Given two permutations of test cases τ_A and τ_B and a set of n functions (objectives) $f : PT \rightarrow \mathbb{R}$, τ_A dominates τ_B ($\tau_A \prec \tau_B$) if an only if:

$$\begin{gathered} f_i(\tau_A) \geq f_i(\tau_B), \forall i \in 1, 2, \cdots, n \\ \textit{and} \\ \exists i \in 1, 2, \cdots, n : f_i(\tau_A) > f_i(\tau_B) \end{gathered} \tag{2.6}$$

Definition 2.5. Given the concept of Pareto dominance and a set of feasible solutions Ω, a solution τ^* is Pareto optimal if a solution able to dominate it does not exist, namely:

$$\nexists \tau_A \in \Omega : \tau_A \prec \tau^* \tag{2.7}$$

Definition 2.6. A Pareto Front is a set composed of Pareto optimal solutions.

$$P^* = \{\tau^* \in \Omega\} \tag{2.8}$$

It is worth considering that multi-objective approaches for test case prioritization return a Pareto front of permutations, that is a set of Pareto optimal test orderings.

Islam *et al.* [79], and Marchetto *et al.* [80] used NSGA-II to find Pareto optimal test case orderings representing trade-offs between three different AUC-based criteria: (i) cumulative code coverage, (ii) cumulative requirement coverage, and (iii) cumulative execution cost. Similarly, Epitropakis *et al.* [81] compared greedy algorithms, MOEAs (NSGA-II e TAEA), and hybrid algorithms. As already done by Islam *et al.* [79] and Marchetto *et al.* [80], they considered different AUC-based fault surrogates: statement coverage (APSC), Δ-coverage (APDC), and past fault coverage (APPFD). They showed that 3-objective MOEAs and hybrid algorithms are able to produce more effective solutions with respect to those produced by additional greedy algorithms based on a single AUC metric.

We notice that these approaches [78, 79, 80, 81] to test case prioritization have important drawbacks. First of all, these measures are computed considering the Area Under Curve obtained plotting the value of the metric with respect to the test cases position in a Cartesian plan [80], and then computing a numerical approximation of the Area Under Curve, using the Trapezoidal rule

[146]. These values are projections of a manifold of cumulative points (*e.g.*, a projection of a volume into two areas). Therefore, despite the AUC metrics being strictly dependent on each other, the different AUC metrics are calculated independently.

Moreover, the tester has to inspect the Pareto front in order to find the most suitable solution with respect to the criteria, no guidelines are provided for selecting the ordering (Pareto optimal solution) to use. Indeed, in the obtained Pareto front each solution represents a permutation of test and selecting a different solution (permutation) requires to re-evaluate all the test cases in the new permutation.

Another important limitation of these classical multi-objective approaches is that they lose their effectiveness as the problem dimensionality increases, as demonstrated by previous work in numerical optimization [147]. Therefore, other non-classical many-objective solvers must be investigated in order to deal with multiple (many) testing criteria. Finally, in [51, 78, 79, 81] there is a lack of strong empirical evidence of the effectiveness of `MOEAs` with respect to simple heuristics, such as greedy algorithms, in terms of cost-effectiveness.

2.3 ENERGY EFFICIENCY AND CODE SMELLS

This section discusses the related literature about code smells and energy consumption.

2.3.1 *Energy Efficiency and Mobile Applications*

The attention toward energy efficiency issues have driven the research community in spending a lot of effort on the construction of new methods to extract the energy profiles of devices, as well as providing guidelines to help developers in writing green code. This section describes on the one hand the tools proposed in recent years to measure energy consumption, and on the other hand the empirical studies conducted in the context of software maintenance and evolution.

2.3.1.1 *Measuring the Energy Profile of Hardware Devices*

A first category of strategies to measure the energy consumption of devices is hardware-based since specific hardware toolkits are required to perform measurements. While such methodologies are quite popular in other research communities, such as high performance analysis [148] or large scale integration systems [149], they have been only partially explored in the context of software engineering.

Flinn and Satyanarayanan [150] proposed a tool named PowerScope. It is based on the adoption of a digital multi-meter connected to a computer, which is used to monitor the energy variations – recorded by the multi-meter – of processes that are running on a laptop. Hindle *et al.* devised GreenMiner [151], a hardware mining testbed based on an Arduino board with an INA219 chip [152]. Besides the extraction of the energy consumption of mobile devices, GreenMiner also provides a web application[1] for (i) automating the testing of applications running on a device, and (ii) analyzing the results. Finally, other researchers exploited the Monsoon power monitor [86] to measure energy consumption of APIs of Android apps [85].

The costly hardware requirements needed for hardware-based energy profiling encouraged researchers to find alternative ways to approximate the energy consumption. A proxy measure can be computed by constructing models, which are based on the definition of specific functions to estimate the energy consumed by a device during its usage. Bourdon *et al.* [153] defined PowerAPI, an approach that leverages on analytical models characterizing the consumption of various hardware components (*e.g.*, CPU). Noureddine *et al.* [154] introduced Jalen, a Java agent which uses statistical sampling for the energy estimations. The model proposed by Pathak *et al.* [155] [156] is based on system calls, and it was implemented in eprof, an energy counterpart of `gprof`, the `gnu` profiler tool, for profiling application energy drain. V-edge [157] considers the battery voltage dynamics for generating a power model. It neither needs external power meters nor relies on the battery current sensing capability. Along the same line, Balasubramanian *et al.* [158] defined an energy consumption model, named TailEnder, to estimate to what extent modules such as 3G and GSM contribute to the battery drain by mobile apps. Ding *et al.* [159] proposed SEMO, a monitoring tool powered by an energy model based on the usage of

1 http://softwareprocess.es/static/GreenMining.html

the battery and its temperature. Zhang *et al.* [160] proposed a model-based solution with PowerBooter and PowerTutor. PowerBooter is a technique for automated power model construction that relies on battery voltage sensors and knowledge of battery discharge behavior. It does not require external power meters. PowerTutor uses the model provided by PowerBooter for generating online power estimation. Lastly, it is worth mentioning Microsoft's Joulemeter tool[2], which uses energy models specific for each hardware configuration.

Finally, software-based approaches exclusively use the system functionalities to estimate the power consumption, without constructing any specific model or requiring any additional hardware. An example of this kind of functionality is ACPI (`Advanced Configuration and Power Interface`), an industry specification for efficiently handling the power consumption in desktop and mobile computers. For this reason, these approaches can be considered hardware-assisted. In this category, Do *et al.* [161] developed ptop, an approach proposed that takes into account CPU frequency, hard disk and memory consumption as sources of information to estimate the joules consumed by a process. eLens [162] provides a more fine-grained estimation of energy consumption at method, path or line-of-source level. It relies on a combination of program analysis and energy modeling and it produces visual feedback to help developers in better understanding the application behavior.

Differently from the techniques/tools discussed above, PETrA does not require any additional hardware equipment and therefore any strong experience in the setup of the test bed. It uses reliable tools coming from the Android Toolkit and does not exploit energy models that need to be calibrated. Finally, unlike the tools measuring energy consumption at process level, PETrA works at method-level granularity. In addition to that, most of the approaches proposed in the literature (including ptop and eLens) are not publicly available.[3]

2.3.1.2 *Empirical Studies in Green Software Engineering*

On top of estimation tools, researchers have studied ways to predict the energy consumption using empirical data [163, 164] or dynamic analysis [165, 166], to study how changes across software versions affect energy consumption [56], or to estimate the energy consumed by single lines of code [167]. At the same time, several empirical investigations have been carried out. Procacciant *et al.* [168]

2 http://tinyurl.com/jkvo9qa
3 PETrA is available at `http://tinyurl.com/je2nxkd`.

provided evidence on the beneficial effect of using *green* practices such as query optimization in MySQL Server and the usage of the `sleep` instruction in the Apache web server. Sahin *et al.* [169] studied the influence of code obfuscation on energy consumption, finding that the magnitudes of such impacts are unlikely to impact end users. Sahin *et al.* [170] also studied the role of design patterns, highlighting that some patterns (*e.g.*, the *Decorator* pattern) negatively impact the energy efficiency of an application. Similar results have been found by Noureddine *et al.* [171].

Hasan *et al.* [172] investigated the impact of the Collections type used by developers, demonstrating how the application of the wrong type of data structure can increase the energy consumption by up to 300%. Along the same lines, other researchers focused their attention on the behavior of sorting algorithms [173], lock-free data structures [174], GUI optimizations [175], API usage of Android apps [85], providing findings on how to efficiently use different programming structures and algorithms.

Sahin *et al.* [176] studied the effect of the refactorings defined by Fowler [177] on energy consumption. Specifically, they evaluated the behavior of 6 types of refactoring, finding that some of them, such as *Extract Local Variable* and *Introduce Parameter Object*, can both increase or decrease the amount of energy used by an application [176]. On the other hand, Park *et al.* [178] experimented with 63 different refactorings and propose a set of 33 energy-efficient refactorings.

Li *et al.* [179] conducted a small-scale empirical investigation - involving four code snippets - into some programming practices believed to be energy-saving, such as the strategies for invoking methods, accessing fields, and setting the length of arrays. The results of this study confirmed the expectations, showing that such practices help in reducing the energy consumption of mobile apps. While some of the practices considered by Li *et al.* [179] are similar to the code smells we considered (*e.g.*, the way developers access fields), it is worth noting the they analyzed the behavior of a small set of programming practices, missing several other program characteristics possibly affecting the energy efficiency of mobile apps. More importantly, their study did not quantify how much energy can be saved by removing the analyzed poor programming practices. Furthermore, they focused their attention only on a few code snippets, rather than considering a large variety of mobile apps.

Finally, the study proposed by Carette *et al.* [180] aimed at investigating the impact of three Android-specific code smells on the energy consumption of

mobile apps. Specifically, they considered the behavior of *Internal Getter/Setter*, *Inefficient Data Structure*, and *Member Ignoring Method* smells on a set of five mobile applications, and measured the energy consumption before and after their removal. The authors found that the removal of single code smells increase the energy efficiency up to 3.86%, while the correction of all the code smells can reduce the energy consumption of mobile apps by up to 4.83%.

2.3.2 *Code Smells and Refactoring*

The traditional code smells defined by Fowler [177] have been widely studied in the past. Several studies demonstrated their negative effects on program comprehension [181], change- and fault-proneness [182], and maintainability [183, 184]. At the same time, several approaches and tools, relying on different sources of information, have been proposed to automatically detect [185, 186, 187], and fix them through the application of refactoring operations [188, 189]. Traditional code smells have also been studied in the context of mobile apps by Mannan *et al.* [190], who demonstrate that, despite the different nature of mobile applications, the variety and density of code smells is similar. A more detailed overview about code smells and refactoring can be found in Palomba *et al.* [191], Bavota *et al.* [192] and Martin *et al.* [193].

As for the Android-specific code smells defined by Reimann *et al.* [58], there is little knowledge about them. Indeed, while the authors of the catalogue assumed the existence of a relationship between the presence of code smells and non-functional attributes of source code, they never empirically assessed it [58]. The unique investigation on the impact of three code smells on the performance of Android applications has been carried out by Hecht *et al.* [59], who found some positive correlations between the studied smells and the decreasing performance in term of delayed frames and CPU usage.

Finally, Morales *et al.* [194] proposed EARMO, a refactoring tool that, besides code quality, takes into account the energy consumption when refactoring code smells detected in mobile apps. It is important to note that the authors mostly considered the code smells proposed by Fowler [177], while our work aims at understanding the impact of a large variety of Android-specific code smells on energy efficiency as well as the role of refactoring on the performance improvement of mobile apps.

3

A DEVELOPER CENTERED BUG PREDICTION MODEL

3.1 INTRODUCTION

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (*i.e.*, metrics capturing intrinsic characteristics of the code components, like their size and complexity) [5, 6, 7, 8, 9], while the second one focuses on *process metrics* (*i.e.*, metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [12, 13, 14, 15, 16, 17, 18]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [13, 21, 17] there is a general consensus on the fact that no technique is the best in all contexts [22, 23]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see *e.g.*, [24, 25, 26, 27, 28, 29, 30]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson *et al.* [25] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [26] partly contradicted the study by Eyolfson *et al.* by showing that the experience of a developer has no clear link with the bug introduction. Bird *et al.* [28] found that high levels of ownership are associated with fewer bugs. Finally, Posnett *et al.* [30] showed that focused developers (*i.e.*, developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

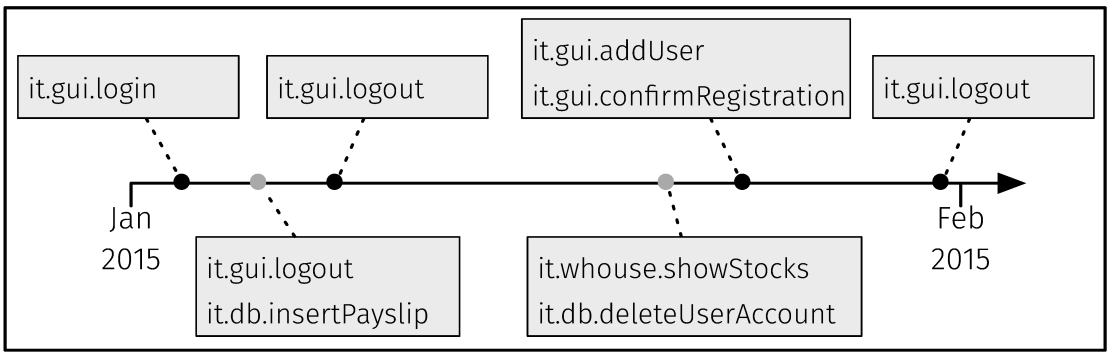
Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i) the number of developers working on a code component [15, 16]; (ii) the analysis of change-proneness [17, 18, 21]; and (iii) the entropy of changes [14]. Thus, despite the previously discussed finding by Posnett *et al.* [30], none of the proposed bug prediction models considers how focused the developers performing changes are and how scattered these changes are. In this chapter we study the role played by *scattered changes* in bug prediction. We defined two metrics, namely the developer's *structural* and *semantic scattering*. The first assesses how "structurally far" in the software project the code components modified by a developer in a given time period are.

The "structural distance" between two code components is measured as the number of subsystems one needs to cross in order to reach one component from the other.

The second measure (*i.e.*, the *semantic scattering*) is instead meant to capture how much semantically spread the code components modified by a developer in a given time period are. The conjecture behind the proposed metrics is that high levels of *structural* and *semantic scattering* make the developer more error-prone. To verify this conjecture, we built two predictors exploiting the proposed metrics and we used them in a bug prediction model. The results achieved on 26 software systems showed the superiority of our model with respect to (i) a prediction model based on structural code metrics [195], (ii) the Basic Code Change Model (BCCM) built using the entropy of changes [14], (iii) a model using the number of developers working on a code component as predictor [15, 16], and (iv) a model based on the focus metrics proposed by Posnett *et al.* [30]. It is worth noting that to support our experiments, we provide a comprehensive replication package [196] including all the raw data and working data sets of our studies.

The achieved results confirm the superiority of our model, achieving a F-Measure 10.3% higher, on average, than the change entropy model [14], 53.7% higher, on average, with respect to what achieved by exploiting the number of developers working on a code component as predictor [15], 13.3% higher, on average, than the F-Measure obtained by using the developers' focus metric by Posnett *et al.* [30] as predictor, and 29.3% higher, on average, with respect to

Figure 3.1: Example of two developers having different levels of "scattering"



the prediction model built on top of product metrics [5]. The two scattering metrics showed their complementarity with the metrics used by the alternative prediction models. Thus, we devised a "hybrid" model providing an average boost in prediction accuracy (*i.e.*, F-Measure) of +5% with respect to the best performing model (*i.e.*, the one proposed in this chapter).

3.2 COMPUTING DEVELOPER'S SCATTERING CHANGES

We conjecture that the developer's effort in performing maintenance and evolution tasks is proportional to the number of involved components and their spread across different subsystems. In other words, we believe that a developer working on different components scatters her attention due to continuous changes of context. This might lead to an increase of the developer's *"scattering"* with a consequent higher chance of introducing bugs.

To get a better idea of our conjecture, consider the situation depicted in Figure 3.1, where two developers, d_1 (black point) and d_2 (grey point) are working on the same system, during the same time period, but on different code components. The tasks performed by d_1 are very focused on a specific part of the system (she mainly works on the system's GUI) and on a very targeted topic (she is mainly in charge of working on GUIs related to the users' registration and login features). On the contrary, d_2 performs tasks scattered across different parts of the system (from GUIs to database management) and on different topics (users' accounts, payslips, warehouse stocks).

Our conjecture is that during the time period shown in Figure 3.1, the contribution of d_2 might have been more "scattered" than the contribution of d_1, thus having a higher likelihood of introducing bugs in the system.

To verify our conjecture we define two metrics, named the *structural* and the *semantic scattering* metrics, aimed at assessing the scattering of a developer d in a given time period p. Note that both metrics are meant to work in object oriented systems at the class level granularity. In other words, we measure how scattered are the changes performed by developer d during the time period p across the different classes of the system. However, our metrics can be easily adapted to work at other granularity levels.

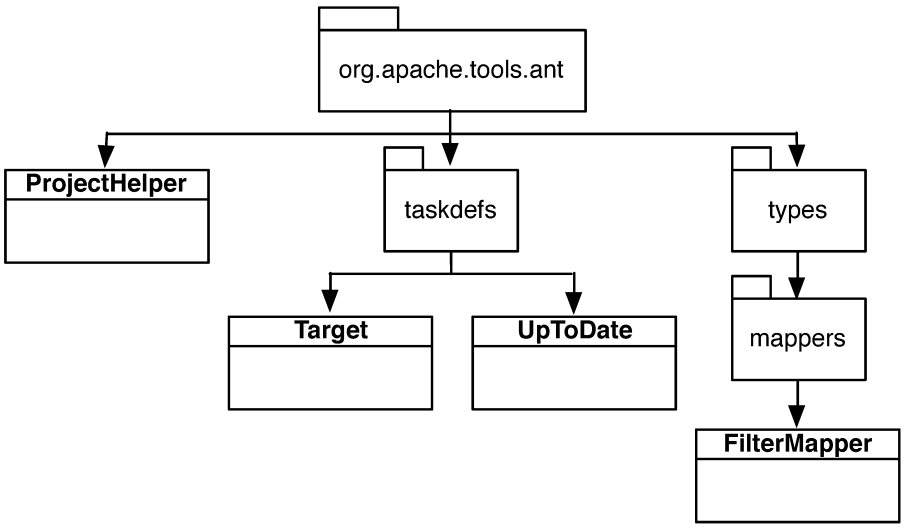
3.2.1 *Structural Scattering*

Let $CH_{d,p}$ be the set of classes changed by a developer d during a time period p. We define the *structural scattering* measure as:

$$\text{StrScat}_{d,p} = |CH_{d,p}| \times \underset{\forall c_i, c_j \in CH_{d,p}}{average} [dist(c_i, c_j)] \tag{3.1}$$

where *dist* is the number of packages to traverse in order to go from class c_i to class c_j; *dist* is computed by applying the shortest path algorithm on the graph representing the system's package structure. For example, the *dist* between two classes `it.user.gui.`c_1 and `it.user.business.db.`c_2 is three, since in order to reach c_1 from c_2 we need to traverse `it.user.business.db`, `it.user.business`, and `it.user.gui`. We (i) use the *average* operator for normalizing the distances between the code components modified by the developer during the time period p and (ii) assign a higher scattering to developers working on a higher number of code components in the given time period (see $|CH_{d,p}|$). Note the the choice to use the average to normalize the distances is driven by the fact that other central operators, such as the median, are not affected by the outliers. Indeed, suppose that a developer performs a change (*i.e.*, commit) C, modifying four files F1, F2, F3, and F4. The first three files are in the same package, while the fourth one (F4) is in a different subsystem. When computing the structural scattering for C, the median would not reflect the scattering of the change performed on F4, since half of the six pairs of files involved in the change (and in particular, F1-F2, F1-F3, F2-F3) have zero as structural distance (*i.e.*, they are in the same package). Thus, the median would not capture the fact that C was,

Figure 3.2: Example of structural scattering



at least in part, a scattered change. This is instead captured by the mean that is influenced by outliers.

To better understand how the *structural scattering* measure is computed and how it is possible to use it in order to estimate the developer's scattering in a time period, Figure 3.2 provides a running example based on a real scenario we found in *Apache Ant*[1], a tool to automate the building of software projects.

The tree shown in Figure 3.2 depicts the activity of a single developer in the time period between 2012-03-01 and 2012-04-30.

In particular, the leafs of the tree represent the classes modified by the developer in the considered time period, while the internal nodes (as well as the root node) illustrate the package structure of the system. In this example, the developer worked on the classes `Target` and `UpToDate`, both contained in the package `org.apache.tools.ant.taskdefs` grouping together classes managing the definition of new commands that the *Ant*'s user can create for customizing her own building process. In addition, the developer also modified `FilterMapper`, a class containing utility methods (*e.g.*, map a java String into an array), and the class `ProjectHelper` responsible for parsing the build file and creating java instances representing the build workflow. To compute the *structural scattering* we compute the distance between every pair of classes modified by the developer. If two classes are in the same package, as in the case of the classes `Target`

1 http://ant.apache.org/

Table 3.1: Example of structural scattering computation

Changed components		Distance
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.Target	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.UpToDate	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.types.mappers.FilterMapper	2
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.taskdefs.UpToDate	0
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.types.mappers.FilterMapper	3
org.apache.tools.ant.taskdefs.UpToDate	org.apache.tools.ant.types.mappers.FilterMapper	3
Structural Developer scattering		**6.67**

and `UpToDate`, then the distance between them will be zero. Instead, if they are in different packages, like in the case of `ProjectHelper` and `Target`, their distance is the minimum number of packages one needs to traverse to reach one class from the other. For example, the distance is one between `ProjectHelper` and `Target` (we need to traverse the package `taskdefs`), and three between `UpToDate` and `FilterMapper` (we need to traverse the packages `taskdefs`, `types` and `mappers`).

After computing the distance between every pair of classes, we can compute the *structural scattering*. Table 3.1 shows the structural distances between every pair of classes involved in our example as well as the value for the *structural scattering*. Note that, if the developer had modified only the `Target` and `UpToDate` classes in the considered time period, then her *structural scattering* would have been zero (the lowest possible), since her changes were focused on just one package. By also considering the change performed to `ProjectHelper`, the *structural scattering* raises to 2.01. This is due to the number of classes involved in change set (3) and the average of the distance among them (0.67).

Finally the *structural scattering* reaches the value of 6.67 when also considering the change to the `FilterMapper` class. In this case the change set is composed of 4 classes and the average of the distances among them is 1.67. Note that the *structural scattering* is a direct scattering measure: the higher the measure, the higher the *estimated* developer's scattering.

3.2.2 *Semantic Scattering*

Considering the package structure might not be an effective way of assessing the similarity of the classes (*i.e.*, to what extent the modified classes implement similar responsibilities). Because of the software "aging" or wrong design decisions, classes grouped in the same package may have completely different responsibilities [197]. In such cases, the *structural scattering* measure might

provide a wrong assessment of the level of developer's scattering, by considering classes implementing different responsibilities as similar only because grouped inside the same package. For this reason, we propose the *semantic scattering* measure, based on the textual similarity of the changed software components. Textual similarity between documents is computed using the *Vector Space Model* (VSM) [198]. In our application of VSM we (i) used *tf-idf* weighting scheme [198], (ii) normalized the text by splitting the identifiers (we also have maintained the original identifiers), (iii) applied a stop word removal, and (iv) stemmed the words to their root (using the well known Porter stemmer). The semantic scattering measure is computed as:

$$\text{SemScat}_{d,p} = |CH_{d,p}| \times \frac{1}{\underset{\forall c_i, c_j \in CH_{d,p}}{average} \; [sim(c_i, c_j)]} \tag{3.2}$$

where the *sim* function returns the textual similarity between the classes c_i and c_j as a value between zero (no textual similarity) and one (the textual content of the two classes is identical). Note that, as for the *structural scattering*, we adopt the *average* operator and assign a higher scattering to developers working on a higher number of code components in the given time period.

Figure 3.3 shows an example of computation for the *semantic scattering* measure. Also in this case the figure depicts a real scenario we identified in *Apache Ant* of a single developer in the time period between 2004-04-01 and 2004-06-30. The developer worked on the classes `Path`, `Resource` and `ZipScanner`, all contained in the package `org.apache.tools.ant.types`.

`Path` and `Resource` are two data types and have some code in common, while `ZipScanner` is an archives scanner. While the *structural scattering* is zero for the example depicted in Figure 3.3 (all classes are from the same package), the *semantic scattering* is quite high (24.32) due to the low textual similarity between the pairs of classes contained in the package (see Table 3.2). To compute the *semantic scattering* we firstly calculate the textual similarity between every pair of classes modified by the developer, as reported in Table 3.2. Then we calculate the average of the textual similarities (≈ 0.12) and we apply the inverse operator (≈ 8.11). Finally the *semantic scattering* is calculated multiplying the obtained value by the number of elements in the change set, that is 3, achieving the final result of ≈ 24.32.

Figure 3.3: Example of semantic scattering measure

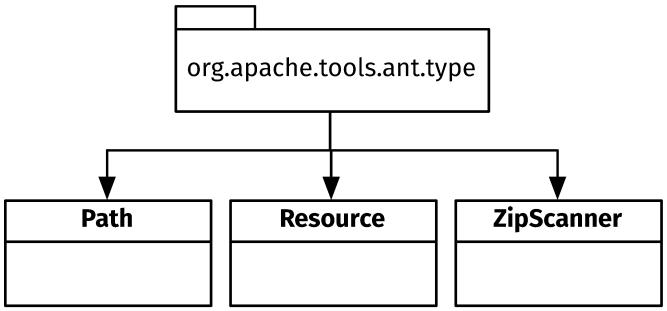


Table 3.2: Example of semantic scattering computation

Changed components		Text. sim.
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.Resource	0.22
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.ZipScanner	0.05
org.apache.tools.ant.type.Resource	org.apache.tools.ant.type.ZipScanner	0.10
Semantic Developer scattering		**24.32**

3.2.3 *Applications of Scattering Metrics*

The scattering metrics defined above could be adopted in different areas concerned with monitoring maintenance and evolution activities. As an example, a project manager could use the scattering metrics to estimate the workload of a developer, as well as to re-allocate resources. In the context of this chapter, we propose the use of the defined metrics for class-level bug prediction (**i.e.,** to predict which classes are more likely to be buggy). The basic conjecture is that *developers having a high scattering are more likely to introduce bugs during code change activities*.

To exploit the defined scattering metrics in the context of bug prediction, we built a new prediction model called *Developer Changes Based Model* (DCBM) that analyzes the components modified by developers in a given time period. The model exploits a machine learning algorithm built on top of two predictors. The first, called *structural scattering predictor*, is defined starting from the structural scattering measure, while the second one, called *semantic scattering predictor*, is based on the semantic scattering measure.

The predictors are defined as follow:

$$\text{StrScatPred}_{c,p} = \sum_{d \in developers_{c,p}} \text{StrScat}_{d,p} \tag{3.3}$$

$$\text{SemScatPred}_{c,p} = \sum_{d \in developers_{c,p}} \text{SemScat}_{d,p} \tag{3.4}$$

where the $developers_{c,p}$ is the set of developers that worked on the component c during the time period p.

3.3 EVALUATING SCATTERING METRICS IN THE CONTEXT OF BUG PREDICTION

The *goal* of the study is to evaluate the usefulness of the developer's scattering metrics in the prediction of bug-prone components, with the *purpose* of improving the allocation of resources in the verification & validation activities focusing on components having a higher bug-proneness. The *quality focus* is on the detection accuracy and completeness of the proposed technique as compared to competitive approaches. The *perspective* is of researchers, who want to evaluate the effectiveness of using information about developer scattered changes in identifying bug-prone components.

The *context* of the study consists of 26 Apache software projects having different size and scope. Table 3.3 reports the characteristics of the analyzed systems, and in particular (i) the software history we investigated, (ii) the mined number of commits, (iii) the size of the active developers base (those who performed at least one commit in the analyzed time period), (iv) the system's size in terms of KLOC and number of classes, and (v) the percentage of buggy files identified (as detailed later) during the entire change history. All data used in our study are publicly available [196].

3.3.1 *Research Questions and Baseline Selection*

In the context of the study, we formulated the following research questions:

- **RQ1.1:** *What are the performances of a bug prediction model based on developer's scattering metrics and how it compares to baseline techniques proposed in literature?*

Table 3.3: Characteristics of the software systems used in the study

Project	Period	#Commits	#Dev.	#Classes	KLOC	% buggy classes
AMQ	Dec 2005 - Sep 2015	8,577	64	2,528	949	54
Ant	Jan 2000 - Jul 2014	13,054	55	1,215	266	72
Aries	Sep 2009 - Sep 2,ì015	2,349	24	1,866	343	40
Camel	Mar 2007 - Sep 2015	17,767	128	12,617	1,552	30
CXF	Apr 2008 - Sep 2015	10,217	55	6,466	1,232	26
Drill	Sep 2012 - Sep 2015	1,720	62	1,951	535	63
Falcon	Nov 2011 - Sep 2015	1,193	26	581	201	25
Felix	May 2007 - May 2015	11,015	41	5,055	1,070	18
JMeter	Sep 1998 - Apr 2014	10,440	34	1,054	192	37
JS2	Feb 2008 - May 2015	1,353	7	1,679	566	34
Log4j	Nov 2000 - Feb 2014	3,274	21	309	59	58
Lucene	Mar 2010 - May 2015	13,169	48	5,506	2,108	12
Oak	Mar 2012 - Sep 2015	8,678	19	2,316	481	43
OpenEJB	Oct 2011 - Jan 2013	9,574	35	4,671	823	36
OpenJPA	Jun 2007 - Sep 2015	3,984	25	4,554	822	38
Pig	Oct 2010 - Sep 2015	1,982	21	81,230	48,360	16
Pivot	Jan 2010 - Sep 2015	1,488	8	11,339	7,809	22
Poi	Jan 2002 - Aug 2014	5,742	35	2,854	542	62
Ranger	Aug 2014 - Sep 2015	622	18	826	443	37
Shindig	Feb 2010 - Jul 2015	2,000	27	1,019	311	14
Sling	Jun 2009 - May 2015	9,848	29	3,951	1,007	29
Sqoop	Jun 2011 - Sep 2015	699	22	667	134	14
Sshd	Dec 2008 - Sep 2015	629	8	658	96	33
Synapse	Aug 2005 - Sep 2015	2,432	24	1,062	527	13
Whirr	Jun 2010 - Apr 2015	569	17	275	50	21
Xerces-J	Nov 1999 - Feb 2014	5,471	34	833	260	6

- **RQ1.2:** *What is the complementarity between the proposed bug prediction model and the baseline techniques?*

- **RQ1.3:** *What are the performances of a "hybrid" model built by combining developer's scattering metrics with baseline predictors?*

In the first research question we quantify the performances of a prediction model based on developer's scattering metrics (DCBM). Then, we compare its performances with respect to four baseline prediction models, one based on product metrics and the other three based on process metrics.

The first model exploits as predictor variables the CK metrics [5], and in particular size metrics (*i.e.*, the Lines of Code—LOC—and the Number of Methods—NOM), cohesion metrics (*i.e.*, the Lack of Cohesion of Method—LCOM), coupling metrics (*i.e.*, the Coupling Between Objects—CBO—and the

Response for a Class—RFC), and complexity metrics (*i.e.*, the Weighted Methods per Class—WMC). We refer to this model as CM.

We also compared our approach with three prediction models based on process metrics. The first is the one based on the work by Ostrand *et al.* [16], and exploiting the number of developers that work on a code component in a specific time period as predictor variable (from now on, we refer to this model as DM).

The second is the Basic Code Change Model (BCCM) proposed by Hassan and using code change entropy information [14]. This choice is justified by the superiority of this model with respect to other techniques exploiting change-proneness information [17, 18, 21]. While such a superiority has been already demonstrated by Hassan [14], we also compared these techniques before choosing BCCM as one of the baselines for evaluating our approach. We found that the BCCM works better with respect to a model that simply counts the number of changes. This is because it filters the changes that differ from the code change process (i.e., fault repairing and general maintenance modifications) considering only the *Feature Introduction modifications* (FI), namely the changes related to adding or enhancing features. However, we observed a high overlap between the BCCM and the model that use the number of changes as predictor (almost 84%) on the dataset used for the comparison, probably due to the fact that the nature of the information exploited by the two models is similar. The interested reader can find the comparison between these two models in our online appendix [196].

Finally, the third baseline is a prediction model based on the *Module Activity Focus* metric proposed by Posnett *et al.* [30]. It relies on the concept of predator-prey food web existing in ecology (from now on, we refer to this model as MAF). The metric is based on the measurement of the degree to which a code component receives focused attention by developers. It can be considered as a form of ownership metric of the developers on the component. It is worth noting that we do not consider the other *Developer Attention Focus* metric proposed by Posnett *et al.*, since (i) the two metrics are symmetric, and (ii) in order to provide a probability that a component is buggy, we need to qualify to what extent the activities on a file are focused, rather than measuring how are developers' activities focused. Even if Posnett *et al.* have not proposed a prediction model based on their metric, the results of this comparison will provide insights

on the usefulness of developer's scattering metrics for detecting bug-prone components.

Note that our choice of the baselines is motived by the will of: (i) considering both models based on product and process metrics, and (ii) covering a good number of different process metrics (since our model exploits process metrics), including approaches exploiting information similar to the ones used by our scattering metrics.

In the second research question we aim at evaluating the complementarity of the different models, while in the third one we build and evaluate a "hybrid" prediction model exploiting as predictor variables the scattering metrics we propose as well as the metrics used by the four experimented competitive techniques (*i.e.*, DM, BCCM, MAF, and CM). Note that we do not limit our analysis to the experimentation of a model including all predictor variables, but we exercise all 2,036 possible combinations of predictor variables to understand which is the one achieving the best performances.

3.3.2 *Experimental Process and Oracle Definition*

To evaluate the performances of the experimented bug prediction models we need to define the machine learning classifier to use. For each prediction technique, we experimented several classifiers belonging to different families, namely ADTree [199], Decision Table Majority [200], Logistic Regression [201], Multilayer Perceptron [202] and Naive Bayes [203]. We empirically compared the results achieved by the five different models on the software systems used in our study (more details on the adopted procedure later in this section). For all the prediction models the best results in terms of F-Measure were obtained using the Majority Decision Table (the comparison among the classifiers can be found in our online appendix [196]). Thus, we exploit it in the implementation of the five models. This classifier can be viewed as an extension of one-valued decision trees [200]. It is a rectangular table where the columns are labeled with predictors and rows are sets of decision rules.

Each decision rule of a decision table is composed of (i) a pool of conditions, linked through and/or logical operators which are used to reflect the structure of the if-then rules; and (ii) an outcome which mirrors the classification of a software entity respecting the corresponding rule as bug-prone or non bug-prone. Majority Decision Table uses an attribute reduction algorithm to find

a good subset of predictors with the goal of eliminating equivalent rules and reducing the likelihood of over-fitting the data.

To assess the performance of the five models, we split the change-history of the object systems into three-month time periods and we adopt a three-month sliding window to *train* and *test* the bug prediction models. Starting from the first time period TP_1 (*i.e.*, the one starting at the first commit), we train each model on it, and test its ability in predicting buggy classes on TP_2 (*i.e.*, the subsequent three-month time period). Then, we move three months forward the sliding window, training the classifiers on TP_2 and testing their accuracy on TP_3. This process is repeated until the end of the analyzed change history (see Table 3.3) is reached. Note that our choice of considering three-month periods is based on: (i) choices made in previous work, like the one by Hassan *et al.* [14]; and (ii) the results of an empirical assessment we performed on such a parameter showing that the best results for all experimented techniques are achieved by using three-month periods. In particular, we experimented with time windows of one, two, three, and six months. The complete results are available in our replication package [196].

Finally, to evaluate the performances of the five experimented models we need an oracle reporting the presence of bugs in the source code.

Although the PROMISE repository collects a large dataset of bugs in open source systems [204], it provides oracles at release-level. Since the proposed metrics work at time period-level, we had to build our own oracle. Firstly, we identified bug fixing commits happened during the change history of each object system by mining regular expressions containing issue IDs in the change log of the versioning system (*e.g.*, *"fixed issue #ID"* or *"issue ID"*). After that, for each identified issue ID, we downloaded the corresponding issue report from the issue tracking system and extracted the following information: *product name*; *issue's type* (*i.e.*, whether an issue is a bug, enhancement request, etc); *issue's status* (*i.e.*, whether an issue was closed or not); *issue's resolution* (*i.e.*, whether an issue was resolved by fixing it, or it was a duplicate bug report, or a "works for me" case); *issue's opening date*; *issue's closing date*, if available.

Then, we checked each issue's report to be correctly downloaded (*e.g.*, the issue's ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (*e.g.*, enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. Basically, we

restricted our attention to (i) issues that were related to bugs as we used them as a measure of fault-proneness, and (ii) issues that were neither duplicate reports nor false alarms.

Once collected the set of bugs fixed in the change history of each system, we used the SZZ algorithm [205] to identify when each fixed bug was introduced. The SZZ algorithm relies on the annotation/blame feature of versioning systems. In essence, given a bug-fix identified by the bug ID, k, the approach works as follows:

1. For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision $\textit{rel-fix}_{i,k}$, we extract the file revision just *before* the bug fixing ($\textit{rel-fix}_{i,k} - 1$).

2. starting from the revision $\textit{rel-fix}_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [206]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions $\textit{rel-bug}_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

By adopting the process described above we are able to approximate the periods of time where each class of the subject systems was affected by one or more bugs (*i.e.*, was a buggy class). In particular, given a bug-fix BF_k performed on a class c_i, we consider c_i buggy from the date in which the bug fixed in BF_k was introduced (as indicated by the SZZ algorithm) to the date in which BF_k (*i.e.*, the patch) was committed in the repository.

3.3.3 *Metrics and Data Analysis*

Once defined the oracle and obtained the predicted buggy classes for every three-month period, we answer **RQ1.1** by using three widely-adopted metrics, namely accuracy, precision and recall [198]:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.5}$$

$$precision = \frac{TP}{TP + FP} \tag{3.6}$$

$$recall = \frac{TP}{TP + FN} \tag{3.7}$$

where TP is the number of classes containing bugs that are correctly classified as bug-prone; TN denotes the number of bug-free classes classified as non bug-prone classes; FP and FN measure the number of classes for which a prediction model fails to identify bug-prone classes by declaring bug-free classes as bug-prone (FP) or identifying actually buggy classes as non buggy ones (FN). As an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall} \tag{3.8}$$

Finally, we also report the Area Under the Curve (AUC) obtained by the prediction model. The AUC quantifies the overall ability of a prediction model to discriminate between buggy and non-buggy classes. The closer the AUC to 1, the higher the ability of the classifier to discriminate classes affected and not by a bug. On the other hand, the closer the AUC to 0.5, the lower the accuracy of the classifier. To compare the performances obtained by DCBM with the competitive techniques, we performed the bug prediction using the four baseline models BCCM, DM, MAF, and CM on the same systems and the same periods on which we ran DCBM.

To answer **RQ1.2**, we analyzed the orthogonality of the different metrics used by the five experimented bug prediction models using Principal Component Analysis (PCA). PCA is a statistical technique able to identify various orthogonal dimensions (principal components) from a set of data. It can be used to evaluate the contribution of each variable to the identified components. Through the analysis of the principal components and the contributions (scores) of each predictor to such components, it is possible to understand whether different predictors contribute to the same principal components. Two models are complementary if the predictors they exploit contribute to capture different principal components. Hence, the analysis of the principal components provides insights on the complementarity between models.

Such an analysis is necessary to assess whether the exploited predictors assign the same bug-proneness to the same set of classes.

However, PCA does not tell the whole story. Indeed, using PCA it is not possible to identify to what extent a prediction model complements another and *vice versa*. This is the reason why we complemented the PCA by analyzing the overlap of the five prediction models. Specifically, given two prediction models m_i and m_j, we computed:

$$corr_{m_i \cap m_j} = \frac{|corr_{m_i} \cap corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \tag{3.9}$$

$$corr_{m_i \setminus m_j} = \frac{|corr_{m_i} \setminus corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \tag{3.10}$$

where $corr_{m_i}$ represents the set of bug-prone classes correctly classified by the prediction model m_i, $corr_{m_i \cap m_j}$ metrics the overlap between the sets of true positives correctly identified by both models m_i and m_j, $corr_{m_i \setminus m_j}$ metrics the percentage of bug-prone classes correctly classified by m_i only and missed by m_j. Clearly, the overlap metrics are computed by considering each combination of the five experimented detection techniques (e.g., we compute $corr_{BCCM \cap DM}$, $corr_{BCCM \cap DCBM}$, $corr_{BCCM \cap CM}$, $corr_{DM \cap DCBM}$, etc.). In addition, given the five experimented prediction models m_i, m_j, m_k, m_p, m_z, we computed:

$$corr_{m_i \setminus (m_j \cup m_k \cup m_p \cup m_z)} = \frac{|corr_{m_i} \setminus (corr_{m_j} \cup corr_{m_k} \cup corr_{m_p} \cup corr_{m_z})|}{|corr_{m_i} \cup corr_{m_j} \cup corr_{m_k} \cup corr_{m_p} \cup corr_{m_z}|}\% \tag{3.11}$$

that represents the percentage of bug-prone classes correctly identified only by the prediction model m_i. In the chapter, we discuss the results obtained when analyzing the complementarity between our model and the baseline ones. The other results concerning the complementarity between the baseline approaches are available in our online appendix [196].

Finally, to answer **RQ1.3** we build and assess the performances of a "hybrid" bug prediction model exploiting different combinations of the predictors used by the five experimented models (*i.e.,* DCBM, BCCM, DM, MAF, and CM). Firstly, we assess the boost in performances (if any) provided by our scattering metrics when plugged-in the four competitive models, similarly to what has been done by Bird *et al.* [28], who explained the relationship between ownership

metrics and bugs building regression models in which the metrics are added incrementally in order to evaluate their impact on increasing/decreasing the likelihood of developers to introduce bugs.

Then, we create a "comprehensive baseline model" featuring all predictors exploited by the four competitive models and again, we assess the possible boost in performances provided by our two scattering metrics when added to such a comprehensive model. Clearly, simply combining together the predictors used by the five models could lead to sub-optimal results, due for example to model overfitting.

Thus, we also investigate the subset of predictors actually leading to the best prediction accuracy. To this aim, we use the wrapper approach proposed by Kohavi and John [207]. Given a training set built using all the features available, the approach systematically exercises all the possible subsets of features against a test set, thus assessing their accuracy. Also in this case we used the Majority Decision Table [200] as machine learner.

In our study, we considered as training set the penultimate three-month period of each subject system, and as test set the last three-month period of each system. Note that this analysis has not been run on the whole change history of the software systems due to its high computational cost. Indeed, experimenting all possible combinations of the eleven predictors means the run of 2,036 different prediction models across each of the 26 systems (52,936 overall runs). This required approximately eight weeks on four Linux laptops having two dual-core 3.10 GHz CPU and 4 Gb of RAM.

Once obtained all the accuracy metrics for each combination, we analyzed these data in two steps. Firstly, we plot the distribution of the average F-measure obtained by the 2,036 different combinations over the 26 software systems. Then we discuss the performances of the top five configurations comparing the results with the ones achieved by (i) each of the five experimented models, (ii) the models built plugging-in the scattering metrics as additional features in the four baseline models, and (iii) the comprehensive prediction models that include all the metrics exploited by the four baseline models plus our scattering metrics.

3.4 ANALYSIS OF THE RESULTS

In this section we discuss the results achieved aiming at answering the formulated research questions.

Table 3.4: AUC-ROC, Accuracy, Precision, Recall, and F-Measure of the five bug prediction models

System	DCBM					DM					BCCM				
	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure
AMQ	83%	53%	42%	53%	47%	58%	24%	18%	19%	19%	61%	52%	33%	49%	39%
Ant	88%	69%	66%	72%	69%	69%	26%	28%	37%	31%	67%	63%	67%	68%	68%
Aries	86%	56%	51%	54%	52%	65%	23%	23%	25%	24%	58%	50%	34%	45%	39%
Camel	81%	55%	51%	55%	53%	51%	27%	18%	28%	22%	50%	39%	39%	46%	42%
CFX	79%	94%	88%	94%	91%	54%	25%	19%	25%	21%	71%	79%	86%	84%	85%
Drill	63%	53%	45%	48%	46%	58%	23%	22%	39%	28%	52%	39%	14%	25%	18%
Falcon	75%	98%	96%	98%	97%	50%	25%	20%	21%	21%	75%	89%	86%	90%	88%
Felix	88%	70%	69%	67%	68%	50%	25%	17%	30%	22%	59%	61%	60%	65%	63%
JMeter	91%	77%	72%	68%	70%	50%	29%	24%	53%	33%	69%	65%	65%	63%	64%
JS2	62%	87%	83%	86%	84%	50%	26%	22%	17%	19%	58%	81%	70%	74%	72%
Log4j	89%	71%	62%	66%	64%	50%	19%	13%	26%	17%	52%	43%	36%	78%	49%
Lucene	77%	84%	79%	83%	81%	54%	27%	22%	30%	26%	63%	72%	61%	86%	71%
Oak	67%	97%	95%	97%	96%	52%	27%	15%	29%	19%	66%	95%	92%	80%	86%
OpenEJB	82%	98%	97%	98%	98%	50%	22%	25%	20%	22%	78%	95%	81%	91%	85%
OpenJPA	83%	79%	71%	77%	74%	51%	20%	20%	38%	26%	78%	72%	61%	68%	64%
Pig	79%	89%	79%	89%	84%	50%	22%	21%	37%	27%	73%	71%	64%	75%	69%
Pivot	78%	86%	75%	86%	80%	53%	26%	19%	24%	21%	68%	69%	71%	79%	75%
Poi	87%	68%	88%	59%	71%	50%	25%	34%	16%	22%	66%	60%	74%	49%	59%
Ranger	77%	95%	90%	95%	93%	50%	28%	18%	19%	19%	76%	92%	83%	91%	87%
Shindig	73%	66%	50%	65%	56%	50%	24%	23%	23%	23%	58%	58%	43%	61%	50%
Sling	62%	85%	76%	84%	80%	57%	21%	17%	18%	18%	61%	80%	62%	68%	65%
Sqoop	78%	98%	96%	98%	97%	55%	26%	19%	32%	23%	77%	97%	90%	89%	90%
Sshd	86%	70%	59%	70%	64%	55%	24%	19%	36%	25%	69%	52%	49%	54%	52%
Synapse	67%	62%	50%	62%	56%	53%	23%	17%	24%	20%	64%	49%	48%	56%	52%
Whirr	76%	98%	95%	98%	97%	52%	26%	20%	24%	21%	74%	96%	84%	88%	86%
Xerces-J	83%	94%	94%	88%	91%	52%	49%	28%	35%	31%	71%	74%	59%	80%	68%
System	CM					MAF									
	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure					
AMQ	55%	43%	37%	41%	39%	56%	56%	38%	45%	41%					
Ant	58%	38%	28%	33%	30%	60%	59%	60%	62%	61%					
Aries	56%	38%	28%	33%	30%	51%	45%	30%	43%	35%					
Camel	41%	42%	44%	41%	42%	50%	38%	35%	38%	36%					
CFX	53%	52%	46%	55%	50%	76%	75%	82%	73%	77%					
Drill	50%	34%	26%	32%	29%	52%	32%	22%	29%	25%					
Falcon	51%	52%	45%	54%	49%	71%	81%	70%	81%	75%					
Felix	53%	55%	53%	51%	52%	67%	56%	62%	65%	63%					
JMeter	50%	43%	44%	43%	43%	68%	58%	61%	59%	60%					
JS2	50%	43%	44%	43%	43%	62%	80%	72%	78%	75%					
Log4j	50%	35%	38%	31%	34%	52%	51%	44%	58%	52%					
Lucene	50%	35%	38%	31%	34%	65%	66%	66%	76%	70%					
Oak	52%	46%	54%	55%	54%	64%	88%	89%	78%	83%					
OpenEJB	61%	62%	66%	57%	61%	80%	78%	77%	79%	78%					
OpenJPA	51%	55%	59%	45%	51%	67%	70%	57%	67%	62%					
Pig	57%	62%	58%	52%	55%	69%	68%	62%	68%	65%					
Pivot	50%	41%	65%	40%	43%	66%	64%	65%	69%	67%					
Poi	58%	65%	61%	65%	63%	61%	55%	58%	56%	57%					
Ranger	60%	65%	61%	65%	63%	76%	81%	77%	82%	79%					
Shindig	52%	48%	36%	46%	40%	54%	55%	39%	59%	47%					
Sling	55%	38%	35%	41%	38%	61%	76%	59%	63%	61%					
Sqoop	53%	59%	59%	64%	61%	78%	92%	86%	84%	87%					
Sshd	50%	28%	26%	31%	28%	67%	48%	46%	52%	49%					
Synapse	56%	43%	47%	52%	49%	61%	47%	47%	53%	50%					
Whirr	54%	61%	55%	63%	59%	69%	82%	82%	82%	82%					
Xerces-J	58%	61%	55%	63%	59%	65%	71%	68%	75%	72%					

3.4.1 *RQ1.1: On the Performances of DCBM and Its Comparison with the Baseline Techniques*

Table 3.4 reports the results—in terms of AUC-ROC, accuracy, precision, recall, and F-measure—achieved by the five experimented bug prediction models, *i.e.*, our model, exploiting the developer's scattering metrics (DCBM), the BCCM proposed by Hassan [14], a prediction model that uses as predictor the number of developers that work on a code component (DM) [15], [16], the prediction model based on the degree to which a module receives focused attention by developers (MAF) [30], and a prediction model exploiting product metrics capturing size, cohesion, coupling, and complexity of code components (CM) [5].

The achieved results indicate that the proposed prediction model (*i.e.*, DCBM) ensures better prediction accuracy as compared to the competitive techniques. Indeed, the area under the ROC curve of DCBM ranges between 62% and 91%, outperforming the competitive models.

In particular, the Developer Model achieves an AUC between 50% and 69%, the Basic Code Change Model between 50% and 78%, the MAF model between 50% and 78%, and the CM model between 41% and 61%. Also in terms of accuracy, precision and recall (and, consequently, of F-measure) DCBM achieves better results. In particular, across all the different object systems, DCBM achieves a higher F-measure with respect to DM (mean=+53.7%), BCCM (mean=+10.3%), MAF (mean=+13.3%), and CM (mean=+29.3%). The higher values achieved for precision and recall indicates that DCBM provides less false positives (*i.e.*, non-buggy classes indicated as buggy ones) while also being able to identify more classes actually affected by a bug as compared to the competitive models. Moreover, when considering the AUC, we observed that DCBM reaches higher values with respect the competitive bug prediction approaches. This result highlights how the proposed model performs better in discriminating between buggy and non-buggy classes.

Interesting is the case of `Xerces-J` where DCBM is able to identify buggy classes with 94% of accuracy (see Table 3.4), as compared to the 74% achieved by BCCM, 49% of DM, 71% of MAF, and 59% of CM. We looked into this project to understand the reasons behind such a strong result. We found that the `Xerces-J`'s buggy classes are often modified by few developers that, on average, perform a small number of changes on them. As an example, the class

`XSSimpleTypeDecl` of the package `org.apache.xerces.impl.dv.xs` has been modified only twice between May 2008 and July 2008 (one of the three-month periods considered in our study) by two developers. However, the sum of their structural and semantic scattering in that period was very high (*161* and *1,932*, respectively). It is worth noting that if a low number of developers work on a file, they have higher chances to be considered as the owner of that file. This means that, in the case of the MAF model, the probability that the class is bug-prone decreases. At the same time, models based on the change entropy (BCCM) or on the number of developers modifying a class (DM) experience difficulties in identifying this class as buggy due to the low number of changes it underwent and to the low number of involved developers, respectively. Conversely, our model does not suffer of such a limitation thanks to the exploited developers' scattering information.

Finally, the CM model relying on product metrics fails in the prediction since the class has code metrics comparable with the average metrics of the system (*e.g.*, the CBO of the class 12, while the average CBO of the system is 14).

Looking at the other prediction models, we can observe that the model based only on the number of developers working on a code component never achieves an accuracy higher than 49%. This result confirms what previously demonstrated by Ostrand *et al.* [16], [15] on the limited impact of individual developer data on bug prediction.

Regarding the other models, we observe that the information about the ownership of a class as well as the code metrics and the entropy of changes have a stronger predictive power compared to number of developers. However, they still exhibit a lower prediction accuracy with respect to what allowed by the developer scattering information.

In particular, we observed that the MAF model has good performances when it is adopted on well-modularized systems, *i.e.*, systems grouping in the same package classes implementing related responsibilities. Indeed, MAF achieved the highest accuracy on the `Apache CFX`, `Apache OpenEJB`, and `Apache Sqoop` systems, where the average modularization quality (MQ) [208] is of 0.84, 0.79, and 0.88, respectively. The reason behind this result is that a high modularization quality often correspond to a good distribution of developers activities. For instance, the average number of developers per package working on `Apache CFX` is 5. As a consequence, the focus of developers on specific code entities is high. The same happens on `Apache OpenEJB` and `Apache Sqoop`, where the

average number of developers per package is 3 and 7, respectively. However, even if the developers mainly focus their attention on few packages, in some cases they also apply changes to classes contained in other packages, increasing their chances of introducing bugs. This is the reason why our prediction model still continue to work better in such cases. A good example is the one of the class `HBaseImportJob`, contained in the package `org.apache.sqoop.mapreduce` of the project `Apache Sqoop`. Only two developers worked on this class over the time period between July 2013 and September 2013, however the same developers have been involved in the maintenance of the class `HiveImport` of the package `com.cloudera.sqoop.hive`. Even if the two classes shared the goal to import data from other projects into `Sqoop`, they implement significantly different mechanisms for importing data. This results in a higher proneness of introducing bugs. The sum of the structural and semantic scattering in that period for the two developers reached *86* and *92*, respectively, causing the correct prediction of the buggy file for our model, and an error in the prediction of the MAF model.

The BCCM [14] often achieves a good prediction accuracy. This is due to the higher change-proneness of components being affected by bugs. As an example, in the `JS2` project, the class `PortalAdministrationImpl` of the package `org.apache.jetspeed.administration` has been modified 19 times between January and March 2010. Such a high change frequency led to the introduction of a bug. However, not always such a conjecture is valid. Let us consider the *Apache Aries* project, in which BCCM obtained a low accuracy (recall=45%, precision=34%). Here we found several classes with high change-proneness that were not subject to any bug. For instance, the class `AriesApplicationResolver` of the package `org.apache.aries.application.managament` has been changed 27 times between November 2011 and January 2012.

It was the class with the higher change-proneness in that time period, but this never led to the introduction of a bug. It is worth noting that all the changes to the class were applied by only one developer.

The model based on structural code metrics (CM) obtains fluctuating performance, with quite low F-measure achieved on some of the systems, like the `Sshd` project (28%). Looking more in depth into such results, we observed that the structural metrics achieve good performances in systems where the developers tend to repeatedly perform evolution activities to the same subset of classes. Such a subset of classes generally centralizes the system behavior, is composed

of complex classes, and exhibits a high fault-proneness. As an example, in the AMQ project the class `activecluster.impl.StateServiceImpl` controls the state of the services provided by the system and it experienced five changes during the time period between September 2009 and November 2009. In this period, developers heavily worked on this class increasing its size from 40 to 265 lines of code. This sudden growth of the class size resulted in the introduction of a bug, correctly predicted by the CM model.

Table 3.5: Wilcoxon's t-test p-values of the hypothesis F-Measure achieved by DCBM $>$ than the compared model. Statistically significant results are reported in bold face. Cliff Delta d values are also shown.

Compared models	p-value	Cliff Delta	Magnitude
DCBM - CM	$<$ **0.01**	0.81	large
DCBM - BCCM	0.07	0.29	small
DCBM - DM	$<$ **0.01**	0.96	large
DCBM - MAF	$<$ **0.01**	0.44	medium

We also statistically compare the F-measure achieved by the five experimented prediction models. To this aim, we exploited the Mann-Whitney test [209] (results are intended as statistically significant at $\alpha = 0.05$). We also estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [210] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [210]. Table 3.5 reports the results of this analysis. The proposed DCBM model obtains a significant higher F-measure with respect to the other baselines (p-value$<$0.05), with the only exception of the model proposed by Hassan [14], for which the p-value is partially significant (p-value=0.07). At the same time, the magnitude of the the differences is large in the comparison with the model proposed by Ostrand *et al.* [15] and the one based on product metrics [195], medium in the comparison with the model based on the Posnett *et al.* metric [30], and small when our model is compared with the model based on the entropy of changes [14].

Summary for RQ1.1. Our approach showed quite high accuracy in identifying buggy classes. Among the 26 object systems its accuracy ranges between 53% and 98%, while the F-measure between 47% and 98%. Moreover, DCBM performs better than the baseline approaches, demonstrating its superiority in correctly predicting buggy classes.

Table 3.6: Results achieved applying the Principal Component Analysis

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
Proportion of Variance	0.39	0.16	0.11	0.10	0.06	0.05	0.03	0.03	0.03	0.02	0.02
Cumulative Variance	0.39	0.55	0.66	0.76	0.82	0.87	0.90	0.92	0.95	0.97	1.00
Structural scattering predictor	**0.69**	-	-	0.08	0.04	-	-	-	-	-	-
Semantic scattering predictor	-	**0.51**	0.33	0.16	0.03	-	-	-	-	-	-
Change entropy	0.07	0.34	**0.45**	0.25	0.11	0.22	-	0.01	-	-	-
Number of Developers	-	-	0.05	0.02	0.29	-	0.04	0.05	0.01	-	0.07
MAF	0.04	0.11	-	**0.38**	**0.45**	-	0.21	0.04	0.06	-	0.1
LOC	0.04	-	0.01	-	0.03	0.07	0.18	0.21	0.11	0.09	**0.33**
CBO	0.1	0.04	0.05	0.07	-	**0.56**	0.2	**0.33**	0.21	**0.44**	0.12
LCOM	0.01	-	0.04	-	0.01	-	**0.24**	0.1	0.06	0.09	0.05
NOM	0.03	-	0.01	0.01	-	0.11	-	0.12	**0.43**	0.22	0.1
RFC	0.01	-	0.04	0.01	0.03	-	0.13	0.06	0.12	0.1	0.09
WMC	0.01	-	0.02	0.02	0.01	0.04	-	0.08	-	0.06	0.14

3.4.2 *RQ1.2: On the Complementarity between DCBM and Baseline Techniques*

Table 3.6 reports the results of the Principal Component Analysis (PCA), aimed at investigating the complementarity between the predictors exploited by the different models. The different columns (PC1 to PC11) represent the components identified by the PCA as those describing the phenomenon of interest (in our case, bug-proneness). The first row (*i.e.*, the proportion of variance) indicates on a scale between zero and one how much each component contributes to the phenomenon description (the higher the proportion of variance, the higher the component's contribution). The identified components are sorted on the basis of their "importance" in describing the phenomenon (*e.g.*, the PC1 in Table 3.6 is the most important, capturing 39% of the phenomenon as compared to the 2% brought by PC11). Finally, the values reported at row i and column j indicate how much the predictor i contributes in capturing the PC j (*e.g.*, structural scattering captures 69% of PC1). The *structural scattering* predictor is mostly orthogonal with respect to the other ten, since it is the one capturing most of PC1, the most important component. As for the other predictors, the semantic scattering and the change entropy information seem to be quite related by capturing the same components (*i.e.*, PC2 and PC3), while the MAF predictor is the one better capturing PC4 and PC5. The number of developers is only able to partially capture PC5, while the product metrics are the most important to capture the remaining components (PC6 to PC11). From these results, we can firstly conclude that the information captured by our predictors is strongly orthogonal with respect to the competitive ones. Secondly, we also observe a high complementarity between the MAF predictor and the others, while the

predictor based on the number of developers working on a code component only partially capture the phenomenon, demonstrating again its limited impact in the context of bug prediction. Finally, the code metrics capture portions of the phenomenon that none of the other (process) metrics is able to capture. Such results highlight the possibility to achieve even better bug prediction models by combining predictors capturing orthogonal information (we investigate this possibility in $\mathbf{RQ}_3$).

As a next step toward understanding the complementarity of the five prediction models, Tables 3.7, 3.8, 3.9, and 3.10 report the overlap metrics computed between DCBM-DM, DCBM-BCCM, DCBM-CM, and DCBM-MAF, respectively.

In addition, Table 3.11 shows the percentage of buggy classes correctly identified only by each of the single bug prediction models (*e.g.*, identified by DCBM and not by DM, BCCM, CM and MAF). While in this chapter we only discuss in details the overlap between our model and the alternative ones, the interested readers can find the analysis of the overlap among the other models in our online appendix [196].

Regarding the overlap between our predictor (DCBM) and the one built using the number of developers (DM), it is interesting to observe that there is high complementarity between the two models, with an overall 73% of buggy classes correctly identified only by our model, 13% only by DM, and 14% of instances correctly classified by both models. This result is consistent on all the object systems (see Table 3.7).

An example of buggy class identified only by our model is represented by `LuceneIndexer` contained in the package `org.apache.camel.component.lucene` of the `Apache Lucene` project. This class, between February 2012 and April 2012, has been modified by one developer that in the same time period worked on five other classes (the sum of structural and semantic scattering reached 138 and 192, respectively). This is the reason why our model correctly identified this class as buggy, while DM was not able to detect it due to the single developer who worked on the class. On the other side, DM was able to detect few instances of buggy classes not identified by DCBM. This generally happens when developers working on a code component apply less scattered changes over the other parts of the system, as in the case of the `Apache Sling` project, where the class `AbstractSlingRepository` of the package `org.apache.sling.jrc.base` was modified by four developers between March 2011 and May 2011. Such developers did not apply changes to other classes, thus having a low structural

Table 3.7: Overlap analysis between DCBM and DM

System	DCBM ∩ DM%	DCBM \ DM%	DM \ DCBM%
AMQ	14	81	5
Ant	9	74	17
Aries	12	65	23
Camel	16	67	17
CXF	12	66	22
Drill	27	72	1
Falcon	12	84	4
Felix	14	65	21
JMeter	8	89	3
JS2	22	75	3
Log4j	13	75	12
Lucene	18	75	7
Oak	19	81	0
OpenEJB	17	80	3
OpenJPA	22	71	7
Pig	16	74	10
Pivot	18	80	2
Poi	11	72	17
Ranger	11	76	13
Shindig	20	61	18
Sling	16	62	21
Sqoop	19	71	10
Sshd	22	64	14
Synapse	12	79	9
Whirr	19	66	15
Xerces	32	55	13
Overall	**14**	**73**	**13**

and semantic scattering. DM was instead able to correctly classify the class as buggy.

A similar trend is shown in Table 3.8, when analyzing the overlap between our model and BCCM. In this case, our model correctly classified 42% of buggy classes that are not identified by BCCM that is, however, able to capture 29% of buggy classes missed by our approach (the remaining 29% of buggy classes are correctly identified by both models). Such complementarity is mainly due to the fact that the change-proneness of a class does not always correctly suggest buggy classes, even if it is a good indicator. Often it is important to discriminate in which situations such changes are done. For example, the class `PropertyIndexLookup` of the package `oak.plugins.index.property` in the `Apache Oak` project, during the time period between April 2013 and June 2013, has been changed 4 times by 4 developers that worked, in the same period, on

Table 3.8: Overlap Analysis between DCBM and BCCM

System	DCBM ∩ BCCM %	DCBM \ BCCM %	BCCM \ DCBM %
AMQ	23	32	45
Ant	39	37	24
Aries	24	39	37
Camel	19	43	38
CXF	20	44	36
Drill	27	47	26
Falcon	34	40	26
Felix	29	38	34
JMeter	28	45	27
JS2	21	40	39
Log4j	16	67	17
Lucene	16	45	39
Oak	29	37	34
OpenEJB	36	35	28
OpenJPA	19	36	45
Pig	31	39	30
Pivot	34	46	20
Poi	37	33	30
Ranger	40	44	16
Shindig	31	33	36
Sling	16	31	53
Sqoop	32	49	19
Sshd	18	36	46
Synapse	20	31	49
Whirr	40	48	12
Xerces	22	43	35
Overall	**29**	**42**	**29**

other 6 classes. This caused a high scattering (both structural and semantic) for all the developers, and our model correctly marked the class as buggy.

Instead, BCCM did not classify the component as buggy since the number of changes applied on it is not high enough to allow the model to predict a bug. However, the model proposed by Hassan [14] is able to capture several buggy files that our model does not identify. For example, in the `Apache Pig` project the class `SenderHome` contained in the package `com.panacya.platform.service.bus.sender` experienced 27 changes between December 2011 and February 2012. Such changes were made by two developers that touched a limited number of related classes of the same package. Indeed, the sum of structural and semantic scattering was quite low (13 and 9, respectively) thus not allowing our model to classify the class as buggy. Instead, in this case the number of changes represent a good predictor.

Table 3.9: Overlap Analysis between DCBM and CM

System	DCBM ∩ CM %	DCBM \ CM %	CM \ DCBM %
AMQ	10	65	25
Ant	8	68	24
Aries	12	58	30
Camel	22	53	25
CXF	7	84	9
Drill	5	73	22
Falcon	18	79	3
Felix	15	68	17
JMeter	15	78	7
JS2	6	88	6
Log4j	11	87	2
Lucene	11	77	12
Oak	14	83	3
OpenEJB	6	88	6
OpenJPA	18	67	15
Pig	16	75	9
Pivot	13	78	9
Poi	14	75	11
Ranger	21	75	5
Shindig	7	82	11
Sling	7	82	11
Sqoop	9	86	5
Sshd	15	72	13
Synapse	18	63	19
Whirr	8	85	7
Xerces2-j	39	59	2
Overall	**13**	**78**	**9**

Regarding the overlap between our model and the code metrics-based model (Table 3.9), also in this case the set of code components correctly predicted by both the models represents only a small percentage (13% on average). This means that the two models are able to predict the bug-proneness of different code components. Moreover, the DCBM model captures 78% of buggy classes missed by the code metrics model that is able to correctly predict 9% of code components missed by our model. For example, the DCBM model is able to correctly classify the `pivot.serialization.JSONSerializer` class of the `Apache Pivot` project, having low (good) values of size, complexity, and coupling, but modified by four developers in the quarter going from January 2013 to March 2013.

As for the overlap between MAF and our model, DCBM was able to capture 45% of buggy classes not identified by MAF. On the other hand, MAF correctly captured 29% of buggy classes missed by DCBM, while 26% of the

Table 3.10: Overlap Analysis between DCBM and MAF

System	DCBM ∩ MAF %	DCBM \ MAF %	MAF \ DCBM %
AMQ	24	47	29
Ant	23	46	31
Aries	32	47	21
Camel	19	51	29
CXF	20	50	31
Drill	23	43	34
Falcon	19	42	39
Felix	24	56	20
JMeter	25	53	22
JS2	23	40	37
Log4j	26	45	30
Lucene	31	41	28
Oak	26	46	28
OpenEJB	28	49	24
OpenJPA	22	46	32
Pig	25	44	31
Pivot	26	55	19
Poi	27	44	29
Ranger	27	41	32
Shindig	27	46	27
Sling	21	37	42
Sqoop	33	43	24
Sshd	19	40	41
Synapse	21	56	23
Whirr	27	53	20
Xerces2-j	30	42	28
Overall	**26**	**45**	**29**

buggy classes were correctly classified by both models. An example of class correctly classified by DCBM and missed by MAF can be found in the package `org.apache.drill.common.config` of the `Apache Drill` project, where the class `DrillConfig` was changed by three developers during the time period between November 2014 and January 2015. Such developers mainly worked on this and other classes of the same package (they can be considered as owners of the `DrillConfig` class), but they also applied changes to components structurally distant from it. For this reason, the sum of structural and semantic scattering increased and our model was able to correctly classify `DrillConfig` as buggy. On the other hand, an example of class correctly classified by MAF and missed by DCBM is `LogManager` of the package `org.apache.log4j` from the `Log4j` project. Here the two developers working on the component between March 2006 and May 2006 applied several changes to this class, as well as

Table 3.11: Overlap Analysis considering each Model independently

System	DCBM \ (BCCM ∪ DM ∪ CM ∪ MAF) %	BCCM \ (DCBM ∪ DM ∪ CM ∪ MAF) %	DM \ (DCBM ∪ BCCM ∪ CM ∪ MAF) %	MAF \ (CM ∪ DCBM ∪ BCCM ∪ DM) %	CM \ (DCBM ∪ BCCM ∪ CM ∪ DM) %
AMQ	44	24	9	17	6
Ant	40	25	8	20	7
Aries	41	22	10	19	8
Camel	39	21	6	22	12
CXF	45	25	9	14	7
Drill	44	25	8	18	5
Falcon	46	27	8	18	2
Felix	43	21	5	19	12
JMeter	42	23	7	17	11
JS2	45	26	10	15	4
Log4j	43	20	8	19	10
Lucene	44	23	8	20	5
Oak	39	26	9	19	7
OpenEJB	43	24	8	16	9
OpenJPA	41	26	9	18	6
Pig	44	25	9	20	2
Pivot	45	25	8	19	3
Poi	39	23	9	17	12
Ranger	48	19	10	14	9
Shindig	46	24	6	17	7
Sling	41	25	9	16	9
Sqoop	41	26	7	19	7
Sshd	44	22	10	19	5
Synapse	41	22	7	20	10
Whirr	40	23	8	18	11
Xerces	47	23	9	12	9
Overall	**43**	**24**	**8**	**18**	**7**

related classes belonging to different packages. Such related updates decreased the semantic scattering accumulated by developers.

Thus, DCBM did not classify the instance as buggy, while MAF correctly detect less focused attention on the class and marked the class as buggy.

Finally, looking at Table 3.11, we can see that our approach identifies 43% of buggy classes missed by the other four techniques, as compared to 24% of BCCM, 8% of DM, 18% of MAF, and 7% of CM. This confirms that (i) our model captures something missed by the competitive models, and (ii) by combining our model with BCCM/DM/MAF/CM ($\mathbf{RQ_3}$) we could further improve the detection accuracy of our technique. An example of a buggy class detected only by DCBM can be found in the *Apache Ant* system. The class `Exit` belonging to the package `org.apache.tools.ant.taskdefs` has been modified just once by a single developer in the time period going from January 2004 to April 2004. However, the sum of the structural and semantic scattering in that period was very high for the involved developer (*461.61* and *5,603.19*, respectively), who modified a total of 38 classes spread over 6 subsystems. In the considered time period the DM does not identify `Exit` as buggy given the single developer who

worked on it, and the BCCM fails too due to the single change `Exit` underwent between January and April 2004. Similarly, the CM model is not able to identify this class as buggy due to its low complexity and small size.

Conversely, an example of buggy class not detected by DCBM is represented by the class `AbstractEntityManager` belonging to the package `org.apache.ivory.resource` of the *Apache Falcon* project.

Here we found 49 changes occurring on the class on the time period going from October 2012 to January 2013 applied by two developers. The sum of the structural and semantic scattering metrics in this time period was very low for both the involved developers (*14.77* is the sum for the first developer, *18.19* for the second one). Indeed, the developers in that period only apply changes to another subsystem. This is the reason why our prediction model is not able to mark this class as buggy. On the other hand, BCCM and MAF prediction models successfully identify the buggyness of the class exploiting the information about the number of changes and ownership, respectively. DM fails due to the low number of developers involved in the change process of the class. Finally, CM is not able to correctly classify this class as buggy because of the low complexity of the class.

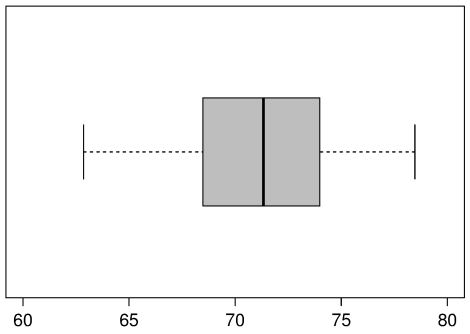
Summary for RQ1.2. The analysis of the complementarity between our approach and the four competitive techniques showed that the proposed scattering metrics are highly complementary with respect to the metrics exploited by the baseline approaches, paving the way to "hybrid" models combining multiple predictors.

3.4.3 *RQ1.3: A "Hybrid" Prediction Model*

Table 3.12 shows the results obtained while investigating the creation of a "hybrid" bug prediction model, exploiting a combination of predictors used by the five experimented models.

The top part of Table 3.12 (*i.e.*, Performances of each experimented model) reports the average performances—in terms of AUC-ROC, accuracy, precision, recall, and F-measure—achieved by each of the five experimented bug prediction models. As already discussed in the context of RQ_1, our DCBM model substantially outperforms the competitive ones. Such values only serve as a

Figure 3.4: Boxplot of the average F-Measure achieved by the 2,036 combinations of experimented predictors.



reference to better interpret the results of the different hybrid models we discuss in the following.

The second part of Table 3.12 (*i.e.*, Boost provided by our scattering metrics to each baseline model), reports the performances of the four competitive bug prediction models when augmented with our predictors.

The boost provided by our metrics is evident in all the baseline models. Such a boost goes from a minimum of +8% in terms of F-Measure (for the model based on change entropy) up to +49% for the model exploiting the number of developers as predictor. However, it is worth noting that the combined models do not seem to improve the performances of our DBCM model.

The third part of Table 3.12 (*i.e.*, Boost provided by our scattering metrics to a comprehensive baseline model) seems to tell a different story. In this case, we combined all predictors belonging to the four baseline models into a single, comprehensive, bug prediction model, and assessed its performances. Then, we added our scattering metrics to such a comprehensive baseline model and assessed again its performances. As it can be seen from Table 3.12, the performances of the two models (*i.e.*, the one with and the one without our scattering metrics) are almost the same (F-measure=71% for both of them). This suggests the absence of any type of impact (positive or negative) of our metrics on the model's performances, which is something unexpected considered the previously performed analyses.

Such a result might be due to the high number of predictor variables exploited by the model (eleven in this case), possibly causing model overfitting on the training sets with consequent bad performances on the test set. Again, the

Table 3.12: RQ_3: Performances of "hybrid" prediction models

	Avg. AUC-ROC	Avg. Accuracy	Avg. Precision	Avg. Recall	Avg. F-measure
Performances of each experimented model					
DM	51	24	19	25	21
BCCM	63	70	61	69	64
CM	52	46	44	45	44
MAF	62	65	59	64	61
DCBM	76	77	72	77	74
Boost provided by our scattering metrics to each baseline model					
DM + Struct-scattering + Seman-scattering	78	71	73	68	70
BCCM + Struct-scattering + Seman-scattering	77	70	76	69	72
CM + Struct-scattering + Seman-scattering	76	70	73	70	71
MAF + Struct-scattering + Seman-scattering	77	70	73	70	71
Boost provided by our scattering metrics to a comprehensive baseline model					
# Developers, Entropy, LOC, CBO, LCOM, NOM, RFC, WMC, MAF	78	69	73	68	71
# Developers, Entropy, LOC, CBO, LCOM, NOM, RFC, WMC, MAF, Struct-scattering, Seman-scattering	76	71	72	71	71
Top-5 predictors combinations obtained from the wrapper selection algorithm					
CBO, Change Entropy, Struct-scattering, Seman-scattering, MAF	90	85	77	81	79
LOC, LCOM, Change Entropy, Seman-scattering, # Developers, MAF	78	72	77	77	77
LOC, NOM, WMC, Change Entropy, Struct-scattering	78	70	77	75	76
LOC, LCOM, NOM, Seman-scattering	77	70	75	75	75
LOC, CBO, LCOM, NOM, RFC, Struct-scattering, Seman-scattering	77	71	76	73	75

combination of predictors does no seem to improve the performances of our DBCM model. Thus, as explained in Section 3.3.3, to verify the possibility to build an effective hybrid model we investigated in an exhaustive way the combination of predictors that leads to the best prediction accuracy by using the wrapper approach proposed by Kohavi and John [207].

Figure 3.4 plots the average F-measure obtained by each of the 2,036 combinations of predictors experimented. The first thing that leaps to the eyes is the very high variability of performances obtained by the different combinations of predictors, ranging between a minimum of 62% and a maximum of 79% (mean=70%, median=71%). The bottom part of Table 3.12 (*i.e.*, Top-5 predictors combinations obtained from the wrapper selection algorithm) reports the performances of the top five predictors combinations. The best configuration, achieving an average F-Measure of 79% exploits as predictors the CBO coupling metric [5], the change entropy by Hassan [14], the structural and semantic scattering defined in this chapter, and the module activity focus by Posnett et al. [30]. Such a configuration also exhibits a very high AUC (90%) and represents a substantial improvement in prediction accuracy over the best model used in isolation (*i.e.*, DCBM with an average F-Measure of 74% and an AUC=76%) as well as over the comprehensive model exploiting all the baselines' predictors in combinations (+8% in terms of F-Measure). Such a result supports our conjecture that blindly combining predictors (as we did in the comprehensive model) could result in sub-optimal performances likely due to model overfitting.

Interestingly, the best combination of baselines' predictors (*i.e.*, all predictors from the four competitive models) obtained as result of the wrapper approach is composed of BCCM (i.e., entropy of changes), MAF, and the RFC and WMC metrics from the CM model, and achieves 70% in terms of F-Measure (9% less with respect to the best combination of predictors which also exploits our scattering metrics).

We also statistically compare the prediction accuracy obtained across the 26 subject systems by the best-performing "hybrid" configuration and the best performing model. Also in this case, we exploited the Mann-Whitney test [209] for this statistical test, as well as the Cliff's Delta [210] to estimate the magnitude of the measured differences. We observed a statistically significant difference (p-value=0.03) with a medium effect size ($d = 0.36$).

Looking at the predictors more frequently exploited in the five most accurate prediction models, we found that:

1. *Semantic-scattering, LOC*. Our semantic predictor and the LOC are present in 4 out of the 5 most accurate prediction models. This confirms the well-known bug prediction power of the size metrics (LOC) and suggests the importance for developers to work on semantically related code components in the context of a given maintenance/evolution activity.

2. *Change entropy, LCOM, Structural-scattering*. These predictors are present in 3 out of the 5 most accurate prediction models. This confirms that (i) the change entropy is a good predictor for buggy code components [14]), (ii) classes exhibiting low cohesion can be challenging to maintain for developers [5], and (iii) scattered changes performed across different subsystems can increase the chances of introducing bugs.

In general, the results of all our three research questions seem to confirm the observations made D'Ambros *et al.* [23]: no technique based on a single metric works better in all contexts. This is why the combination of multiple predictors can provide better results. We are confident that plugging other orthogonal predictors in the "hybrid" prediction model could further increase the prediction accuracy.

Summary for RQ1.3. By combining the eleven predictors exploited by the five prediction models subject of our study it is possible to obtain a boost of prediction accuracy up to +5% with respect to the best performing model (*i.e.*, DCBM) and +9% with respect to the best combination of baseline predictors. Also, the top five "hybrid" prediction models include at least one of the predictors proposed in this work (*i.e.*, the structural and semantic scattering of changes) and the best model includes both.

3.5 THREATS TO VALIDITY

metricsThis section describes the threats that can affect the validity our study. Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important type of threat for our study and it is related to:

- *Missing or wrong links between bug tracking systems and versioning systems [211]*: although not much can be done for missing links, as explained in

the design we verified that links between commit notes and issues were correct;

- *Imprecision due to tangled code changes [212].* We cannot exclude that some commits we identified as bug-fixes grouped together tangled code changes, of which just a subset represented the committed patch.
- *Imprecision in issue classification made by issue-tracking systems [27]*: while we cannot exclude misclassification of issues (*e.g.*, an enhancement classified as a bug), at least all the systems considered in our study used Bugzilla as issue tracking system, explicitly pointing to bugs in the issue type field;
- *Undocumented bugs present in the system*: while we relied on the issue tracker to identify the bugs fixed during the change history of the object systems, it is possible that undocumented bugs were present in some classes, leading to wrong classifications of buggy classes as "clean" ones.
- *Approximations due to identifying fix-inducing changes using the SZZ algorithm [205]*: at least we used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of fix-inducing changes.

Threats to *internal validity* concern external factors we did not consider that could affect the variables being investigated. We computed the developer's scattering metrics by analyzing the developers' activity on a single software system. However, it is well known that, especially in open source communities and ecosystems, developers contribute to multiple projects in parallel [213]. This might negatively influence the "developer's scattering" assessment made by our metrics. Still, the results of our approach can only improve by considering more sophisticated ways of computing our metrics.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. The metrics used in order to evaluate our defect prediction approach (*i.e.*, accuracy, precision, recall, F-Measure, and AUC), are widely used in the evaluation of the performances of defect prediction techniques [23]. Moreover, we used appropriate statistical procedures, (*i.e.*, PCA [214]), and the computation of overlap metrics to study the orthogonality between our model and the competitive ones.

Since we had the necessity to exploit change-history information to compute the scattering metrics we proposed, the evaluation design adopted in our

study is different from the k-fold cross validation [215] generally exploited while evaluating bug prediction techniques. In particular, we split the change-history of the object systems into three-month time periods and we adopted a three-month sliding window to train and test the experimented bug prediction models. This type of validation is typically adopted when using process metrics as predictors [14], although it might be penalizing when using product metrics, which are typically assessed using a ten-fold cross validation. Furthermore, although we selected a model exploiting a set of product metrics previously shown to be effective in the context of bug prediction [5], the poor performances of the CM model might be due to the fact that the model relies on too many predictors, resulting in a model overfitting. This conjecture is supported by the results achieved in the context of $\mathbf{RQ}_3$, where we found that the top five "hybrid" prediction models include only a subset of code metrics.

Threats to *external validity* concern the generalization of results. We analyzed 26 Apache systems from different application domains and with different characteristics (number of developers, size, number of classes, etc).

However, systems from different ecosystems should be analyzed to corroborate our findings.

3.6 CONCLUSION

A lot of effort has been devoted in the last decade to analyze the influence of the development process on the likelihood of introducing bugs. Several empirical studies have been carried out to assess under which circumstances and during which coding activities developers tend to introduce bugs. In addition, bug prediction techniques built on top of process metrics have been proposed. However, changes in source code are made by developers that often work under stressing conditions due to the need of delivering their work as soon as possible.

The role of developer-related factors in the bug prediction field is still a partially explored area. This chapter makes a further step ahead, by studying the role played by the *developer's scattering* in bug prediction. Specifically, we defined two metrics that consider the amount of code components a developer modifies in a given time period and how these components are spread structurally (*structural scattering*) and in terms of the responsibilities they implement (*semantic scattering*). The defined metrics have been evaluated as bug predictors

in an empirical study performed on 26 open source systems. In particular, we built a prediction model exploiting our metrics and compared its prediction accuracy with four baseline techniques exploiting process metrics as predictors. The achieved results showed the superiority of our model and its high level of complementarity with respect to the considered competitive techniques. We also built and experimented a "hybrid" prediction model on top of the eleven predictors exploited by the five competitive techniques. The achieved results show that (i) the "hybrid" is able to achieve a higher accuracy with respect to each of the five models taken in isolation, and (ii) the predictors proposed in this chapter play a major role in the best performing "hybrid" prediction models.

Our future research agenda includes a deeper investigation of the factors causing scattering to developers, and negatively impacting their ability of dealing with code change tasks. We plan to reach such an objective by performing a large survey with industrial and open source developers. We also plan to apply our technique at different levels of granularity, to verify if we can point out buggy code components at a finer granularity level (*e.g.,* methods).

4

DYNAMIC SELECTION OF CLASSIFIERS IN BUG PREDICTION: AN ADAPTIVE METHOD

4.1 INTRODUCTION

Continuous changes, close deadlines, and the need to ensure the correct behaviour of the functionalities being issued are common challenges faced by developers during their daily activities [23]. However, limited time and manpower represent serious threats to the effective testing of a software system. Thus, the resources available should be allocated effectively upon the portions of the source code that are more likely to contain bugs. One of the most powerful techniques aimed at dealing with the testing-resource allocation is the creation of *bug prediction models* [4] which allow to predict the software components that are more likely to contain bugs and need to be tested more extensively.

Roughly speaking, a bug prediction model is a supervised method where a set of independent variables (the predictors) are used to predict the value of a dependent variable (the bug-proneness of a class) using a machine learning classifier (*e.g.,* Logistic Regression [31]). The model can be trained using a sufficiently large amount of data available from the project under analysis, *i.e., within-project* strategy, or using data coming from other (similar) software projects, *i.e., cross-project* strategy.

A factor that strongly influences the accuracy of bug prediction models is represented by the classifier used to predict buggy components. Specifically, Ghotra *et al.* [32] found that the accuracy of a bug prediction model can increase or decrease up to 30% depending on the type of classification applied [32]. Also, Panichella *et al.* [33] demonstrated that the predictions of different classifiers are highly complementary despite the similar prediction accuracy.

Based on such findings, an emerging trend is the definition of prediction models which are able to combine multiple classifiers (a.k.a., *ensemble* techniques [34]) and their application to bug prediction [33, 35, 36, 216, 37, 38, 39, 40, 41]. For instance, Tosun *et al.* [216] experimented the *Validation and Voting* (VV) strategy, an approach where the prediction of the bug-proneness of a class is

assigned by considering the output of the majority of the classifiers. Panichella *et al.* [33] devised CODEP, an approach that uses the outputs of 6 classifiers as predictors of a new prediction model, which is trained using Logistic Regression (LOG). However, as highlighted by Bowes *et al.* [60], traditional ensemble approaches miss the predictions of a large part of bugs that are correctly identified by a single classifier and, therefore, *"ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of bugs"* [60].

To understand whether ensemble techniques actually perform better than stand-alone classifiers, Liu *et al.* [35] performed an empirical study in which they showed that the usage of the VALIDATION AND VOTING technique allows bug prediction models to work better. Furthermore, Zhang *et al.* [61] benchmarked eight ensemble techniques, belonging to five categories, confirming the findings on the superiority of the VALIDATION AND VOTING technique. While the findings by Liu *et al.* [35] and Zhang *et al.* [61] represent an important source of information for researchers and practitioners interested in the application of ensemble techniques in practice, in the context of our research we figured out six critical limitations of previous investigations on the performances of ensemble classifiers that possibly threat the conclusions provided so far:

- **Data quality.** Both Liu *et al.* [35] and Zhang *et al.* [61] exploited datasets coming from the PROMISE repository [204]. Unfortunately, such datasets might contain noisy and/or erroneous entries that have possibly biased the results achieved in previous work [73].

- **Data preprocessing.** As widely demonstrated in literature [217, 74, 218, 219], data preprocessing techniques such as (i) data normalization, (ii) feature selection, and (iii) training data balancing should always be applied before running bug prediction models to limit conclusion instability. However, Liu *et al.* [35] and Zhang *et al.* [61] did not perform a complete data preprocessing before building the experimented bug prediction models: as a consequence, their findings might not provide a correct overview of the performances of ensemble techniques.

- **Data Analysis.** To evaluate the performances of bug prediction models, previous work heavily relied on metrics such as the precision, recall, and F-Measure [198]. Nevertheless, these are threshold-dependent metrics

whose usage is strongly discouraged since they often do not allow a clear interpretation of the actual effectiveness of bug prediction models [74].

- **Limited size of previous analysis.** The study by Liu *et al.* [35] has been conducted on seven systems and considering the behavior of two ensemble techniques, *i.e.,* BAGGING and VALIDATION AND VOTING, while *et al.* [61] took into account a dataset composed of ten systems analyzing seven ensemble approaches classified in four categories, *i.e.,* RANDOM FOREST, BAGGING, BOOSTING, and VALIDATION AND VOTING. As a consequence, a wider analysis that includes the recent ensemble techniques proposed in literature may be beneficial.

- **Unclear relationship between local learning and ensemble classifiers.** While the potential usefulness of *local* learning on the performances of bug prediction models has been shown [40], to the best of our knowledge there have not been attempts to investigate the extent to which such local learning strategy can benefit of the usage of ensemble techniques.

- **Missing comparison with within-project prediction models.** One of the key promises of cross-project bug prediction models is to be competitive with respect to within-project ones. As recommended in previous work [220, 221], whenever a certain technique is experimented in a cross-project setting, it should be applied in a within-project setting as well in order to fairly benchmarking its real capabilities. The studies by Liu *et al.* [35] and Zhang *et al.* [61] did not tested how ensemble approaches applied in cross-project bug prediction work when compared with their adoption in within-project models.

This chapter proposes two contributions. Firstly, based on the results of previous approaches, we conjecture that a successful way to combine classifiers can be obtained by *choosing the most suitable classifier based on the characteristics of classes, rather than combining the output of different classifiers*. To verify this conjecture, we propose a novel adaptive prediction model, coined as **ASCI** (**A**daptive **S**election of **C**lass**I**fiers in bug prediction), which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class (*i.e.,* product metrics). Specifically, given a set of classifiers our approach firstly trains these classifiers using the

structural characteristics of the classes in the training set, then a decision tree is built where the internal nodes are represented by the structural characteristics of the classes contained in the training set, and the leafs are represented by the classifiers able to correctly classify the bug-proneness of instances having such structural characteristics. In other words, we use a decision tree learning approach to train a classifier able to predict which classifier should be used based on the structural characteristics of the classes.

Secondly, to cope with the issues of previous empirical studies, we provide a *differentiated* replication study [72] in which not only we **corroborate** previous empirical research on the performances of ensemble classifiers for cross-project bug prediction, but also **extend** previous knowledge by assessing the extent to which local bug prediction can benefit of the usage of ensemble techniques. The study was conducted on a PROMISE dataset composed of 21 software systems, where we applied a number of corrections suggested by Shepperd *et al.* [73] in order to make it cleaned and suitable for our purpose. We took into account several different ensemble techniques, belonging to six categories, measuring their performances using the two metrics recommended by previous work, *i.e.*, AUC-ROC and Matthew's Correlation Coefficient [74, 75].

The results suggest that the problem of identifying buggy classes using external sources of information is still far from being solved. The use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers, but in general the VALIDATION AND VOTING [216] and ASCI [65] techniques should be preferred among the others. These observations also hold when applying ensemble methodologies to local bug prediction; moreover, we found that global and local models are mostly statistically equivalent. Finally, ensemble-based cross-project models perform worse than within-project ones in terms of prediction accuracy, yet being more robust (their performances do not vary as much as the ones of within-projects models). Furthermore, we observed that also in a within-project scenario the use of ensemble classifiers does not guarantee better prediction performances with respect to models adopting stand-alone machine learners.

Our findings provide some key implications for both researchers and practitioners. For researchers, our results confirm that the problem of cross-project bug prediction still needs noticeable attention in order to devise methodologies to properly transfer external information into a new context, especially because existing local learning methods and ensemble techniques do not represent yet

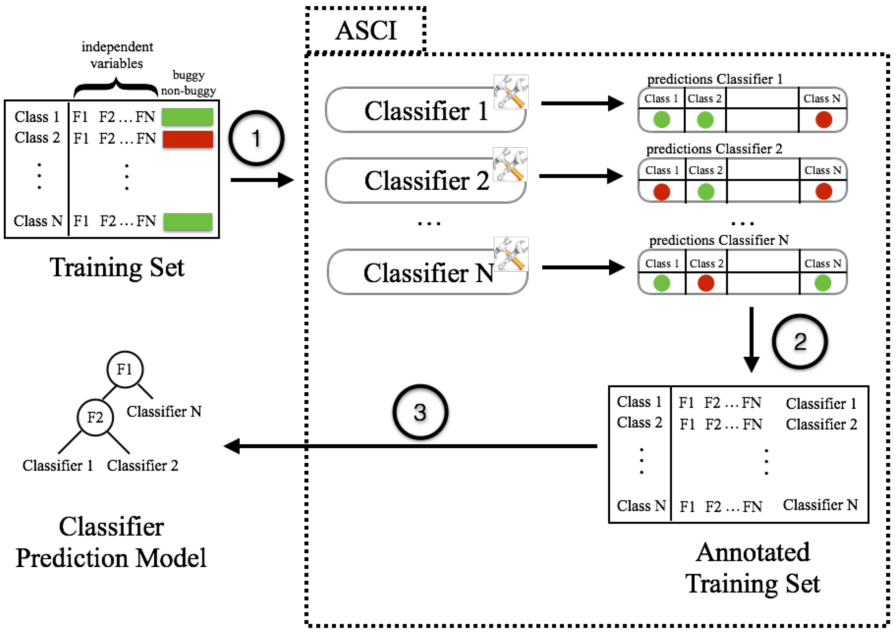


Figure 4.1: The workflow of ASCI.

a suitable solution; moreover, within-project models are not bulletproof and their robustness cannot be improved by means of ensemble techniques: thus, more research is needed to overcome such limitation. For practitioners, our findings suggest that the use of more sophisticated techniques to mix different classifiers does not provide immediate advantages: thus, they need to carefully evaluate the suitability of ensemble methods before using them. Similarly, collecting bug-related data from the project under development still represents the most effective way to apply bug prediction in practice, even because the use of cross-project models might actually lead to higher inspection costs.

4.2 ASCI: AN ADAPTIVE METHOD FOR BUG PREDICTION

In this section we present ASCI (**A**daptive **S**election of **C**lassifiers in bug pred**I**ction), our solution to dynamically select classifiers in bug prediction. The implementation of ASCI is available in our online appendix [222].

Figure 4.1 depicts the three main steps that our approach employs to recommend which classifier should be used to evaluate the bugginess of a given class. In particular:

1. Let $C = \{c_1, ..., c_n\}$ be a set of n different classifiers, and let $T = \{e_1, ..., e_m\}$ be the set of classes composing the training set. Each $c_i \in C$ is experimented against the set T, in order to find its configuration. As an example,

for the *Logistic Regression* [31], this step will configure the parameters to use in the logistic function. At the same time, each $c_i \in C$ outputs the predictions regarding the bug-proneness of each $e_j \in T$. Note that the proposed technique is independent from the underlying pool of classifiers, however it would be desirable that the base classifiers exhibit some level of complementarity. The time efficiency of this step is influenced by (i) the number and type of classifiers to configure and (ii) the size of the training set: however, on systems similar to those we analyzed in the evaluation of the model (see Section 4.4) it requires few seconds.

2. At the end of the first step, each $e_j \in T$ is labeled with the information about the best classifier $c_i \in C$ which correctly identified its bugginess. In this stage, there are two possible scenarios to consider. If a unique machine learning classifier c_i is able to predict the bug-proneness of e_j, then e_j will be associated with c_i. On the other hand, if more classifiers or none of them correctly identified the bugginess of e_j, we assign to e_j the classifier c_i having the highest F-Measure on the whole training set. The output of this step is represented by an annotated training set T'. Note that, while there would be other alternatives to build the annotated training set (*e.g.,* the usage of multi-label classifiers [223]), we empirically evaluated them observing that our solution results in higher performances. A detailed comparison between our approach, the multi-label solution, and a baseline where the choice is made randomly is available in our online appendix [222].

3. Finally, based on T', we build a *classifier prediction model* using a decision tree DT as classifier. Specifically, given the structural characteristics of the classes in the annotated training set (independent variables), the goal of this final step is to build a model able to predict the classifier $c_i \in C$ to use (dependent variable). In other words, the role of the DT is to predict a nominal variable indicating the name of the classifier $c_i \in C$ most suitable for classifying a class that is characterized by the structural properties reported as independent variables. Note that we decide to use a decision tree learning approach since a decision tree captures the idea that if different decisions were to be taken, then the structural nature of a situation (and, therefore, of the model) may have changed drastically [199]. This is in line with what we want to obtain, namely a model where a

change in the structural properties of a class implies a different evaluation of suitability of a classifier. In order to build *DT*, we propose to use Random Forest [224], a classifier widely used in literature built from a combination of tree predictors.

Once the adaptive model has been built, the bugginess of a new class is predicted by using the classifier that the *DT* has selected to be the most suitable one and not all the base classifiers as required by other ensemble techniques, such as VV, Boosting, and Stacking.

4.3 ON THE COMPLEMENTARITY OF MACHINE LEARNING CLASSIFIERS

This section describes the design and the results of the empirical study we conducted in order answer our $\mathbf{RQ}_0$ with the purpose of verifying whether the investigated classifiers are complementary and thus good candidates for being combined by ASCI.

- **RQ2.0:** *Are different classifiers complementary to each other when used in the context of within-project bug prediction?*

4.3.1 *Empirical Study Design*

The *goal* of the empirical study is to assess the complementarity of different classifiers when used to predict bugs at class level granularity, with the *purpose* of investigating whether they classify different sets of software components as bug-prone. The *quality focus* is on the improvement of the effectiveness of bug prediction approaches in the context of *within-project* bug prediction, while the *perspective* is of a researcher who is interested to understand to what extent different classifiers complement each other when used to predict buggy classes. Indeed, if classifiers are complementary it could be worth to combine them using an ensemble technique.

The *context* of the study consists of 30 software systems from the `Apache Software Foundation` ecosystem[1]. Table 4.1 reports the specific release taken into account as well as the characteristics of the projects considered in the study in terms of size, expressed as number of classes and KLOC, and number

1 http://www.apache.org

Table 4.1: Characteristics of the software systems used in the study

#	Project	Release	Classes	KLOC	Buggy Classes	(%)
1	Ant	1.7	745	208	166	22%
2	ArcPlatform	1	234	31	27	12%
3	Camel	1.6	965	113	188	19%
4	E-Learning	1	64	3	5	8%
5	InterCafe	1	27	11	4	15%
6	Ivy	2.0	352	87	40	11%
7	jEdit	4.3	492	202	11	2%
8	KalkulatorDiety	1	27	4	6	22%
9	Log4J	1.2	205	38	180	92%
10	Lucene	2.4	340	102	203	60%
11	Nieruchomosci	1	27	4	10	37%
12	pBeans	2	51	15	10	20%
13	pdfTranslator	1	33	6	15	45%
14	Poi	3.0	442	129	281	64%
15	Prop	6.0	660	97	66	10%
16	Redaktor	1.0	176	59	27	15%
17	Serapion	1	45	10	9	20%
18	Skarbonka	1	45	15	9	20%
19	SklepAGD	1	20	9	12	60%
20	Synapse	1.2	256	53	86	34%
21	SystemDataManagement	1	65	15	9	14%
22	SzybkaFucha	1	25	1	14	56%
23	TermoProjekt	1	42	8	13	31%
24	Tomcat	6	858	300	77	9%
25	Velocity	1.6	229	57	78	34%
26	WorkFlow	1	39	4	20	51%
27	WspomaganiePI	1	18	5	12	67%
28	Xalan	2.7	909	428	898	99%
29	Xerces	1.4	588	4	437	74%
30	Zuzel	1	39	14	13	45%

and percentage of buggy classes. All the systems are publicly available in the PROMISE repository [204], which provides for each project (i) the independent variables, *i.e.*, LOC and CK metrics [225], and (ii) the dependent variable, *i.e.*, a boolean value indicating whether a class is buggy or not.

In order to answer $\mathbf{RQ}_0$, we run five different machine learning classifiers [31], namely Binary Logistic Regression (LOG), Naive Bayes (NB), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), and Decision Trees (DTree). The selection of the machine learning classifiers is not random. On the one hand, they have been used in many previous work on bug prediction [33, 35, 60, 100, 103, 61], while on the other hand they are based on different learning peculiarities (*i.e.*, regression functions, neural networks, and decision trees). This choice increases the generalizability of our results.

As evaluation procedure, we adopt the 10-fold cross validation strategy [215]. This strategy randomly partitions the original set of data, *i.e.*, data of each system, into 10 equal sized subset. Of the 10 subsets, one is retained as *test*

set, while the remaining 9 are used as *training* set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the *test* set exactly once [215]. We use this test strategy since it allows all observations to be used for both training and test purpose, but also because it has been widely-used in the context of bug prediction (e.g., see [15, 226, 227, 27]).

As evaluation methodology, we firstly evaluate the performances of the experimented classifiers using widely-adopted metrics, such as accuracy, precision and recall [228]. In addition, we also computed (i) the F-measure, *i.e.,* the harmonic mean of precision and recall, and (ii) the Area Under the Curve (AUC), which quantifies the overall ability of a prediction model to discriminate between buggy and non-buggy classes. Note that the analysis of the accuracy achieved by different classifiers is necessary to corroborate previous findings which report how different classifiers exhibit similar accuracy, even if they are complementary to each other [33], [60].

Due to space limitations, we report and discuss the boxplots of the distributions of the accuracy, the F-Measure, and the AUC achieved by the single classifiers independently on the 30 considered systems. A complete report of the results is available in our online appendix [222]. Furthermore, we verified whether the differences are statistically significant by exploiting the Mann-Whitney U test [209]. The results are intended as statistically significant at $\alpha = 0.05$. We also estimated the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [210] for ordinal data. We followed the guidelines in [210] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

After the analysis of the performance of the different classifiers, we analyze their complementarity by computing the overlap metrics. Specifically, given two sets of predictions obtained by using classifiers c_i and c_j, we compute:

$$corr_{c_i \cap c_j} = \frac{|corr_{c_i} \cap corr_{c_j}|}{|corr_{c_i} \cup corr_{c_j}|}\% \tag{4.1}$$

$$corr_{c_i \setminus c_j} = \frac{|corr_{c_i} \setminus corr_{c_j}|}{|corr_{c_i} \cup corr_{c_j}|}\% \tag{4.2}$$

where $corr_{c_i}$ represents the set of bug-prone classes correctly classified by the classifier c_i, $corr_{c_i \cap c_j}$ measures the overlap between the set of buggy classes correctly identified by both classifiers c_i and c_j, and $corr_{c_i \setminus c_j}$ measures bug-prone classes correctly classified by c_i only and missed by c_j. Also in this case, we aggregated the results of the 30 considered systems by summing up the single $corr_{c_i \cap c_j}$, $corr_{c_i \setminus c_j}$, and $corr_{c_j \setminus c_i}$ obtained after the evaluation of the complementarity between two classifiers on a system. The fine-grained results are available in our online appendix [222].

Figure 4.2: Boxplots of the accuracy, F-Measure, and AUC-ROC achieved by the single classifiers.

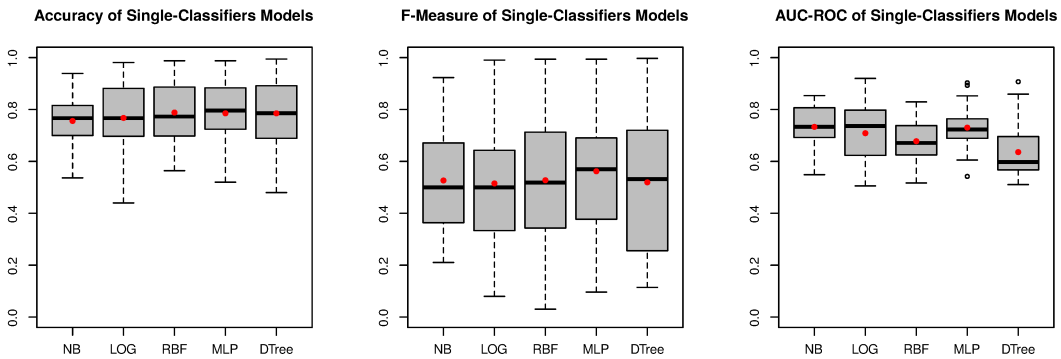


Table 4.2: p-values and Cliff's Delta obtained between each pair of single classifiers. $p-values$ that are statistically significant are reported in bold face.

	NB			LOG			RBF			MLP			DTree		
	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$	$p-value$	d	$magn.$
NB	-	-	-	0.28	0.03	neg.	0.16	0.04	neg.	0.44	-0.04	neg.	0.34	0.02	neg.
LOG	0.73	-0.03	neg.	-	-	-	0.18	0.01	neg.	0.96	-0.07	neg.	0.64	-0.04	neg.
RBF	0.73	-0.03	neg.	1.00	-0.00	neg.	-	-	-	0.96	-0.07	neg.	0.64	-0.04	neg.
MLP	0.57	0.04	neg.	**0.04**	**0.07**	**neg.**	**0.04**	**0.07**	**neg.**	-	-	-	0.49	0.05	neg.
DTree	0.67	-0.02	neg.	0.36	0.04	neg.	0.36	0.04	neg.	0.52	-0.05	neg.	-	-	-

4.3.2 *Analysis of the Results*

The results achieved running the stand-alone classifiers over all the considered software systems are reported in Figure 4.2.

Looking at the boxplots, we can confirm previous findings in the field [33, 60], demonstrating once again that there is no a clear winner among different classifiers in bug prediction. Indeed the differences in the terms of accuracy, F-Measure, and AUC achieved by the classifiers are quite small, as highlighed by

the median values presented in Figure 4.2. Despite this, the average F-Measure (0.56) achieved by MLP is slightly higher with respect to the other classifiers (*i.e.*, NB=+3%, LOG=+4%, RBF=+2%, DTree=+4%). As shown in Table 4.2 such superiority is statistically significant when considering the differences between the performances of MLP and the ones achieved by LOG and RBF, even though with negligible effect size.

Particularly interesting is the discussion of the results achieved on the `Apache Xalan` project. Here all the classifiers achieve good precision and recall. This is due to the fact that in this project there is 99% of buggy classes. The classifiers can be mainly trained using data related to buggy components and, thus, they do not have enough information to distinguish those components not affected by bugs. However, the presence of an extremely low number of non-buggy classes (11) does not influence too much the performances of the classifiers, which correctly predicted most of the buggy classes.

Another interesting result concerns LOG. As reported in recent papers [23, 33], this classifier is the more suitable in the context of cross-project bug prediction. According to our findings, this is not true when the training set is built using a within-project strategy. A possible explanation of the result is that LOG generally requires much more data in the training set to achieve stable and meaningful results [229]. In the cross-project strategy, the training set is built using a bunch of data coming from external projects, while in the within-project solution the construction of the training set is limited to previous versions of the system under consideration. As a consequence, LOG is not more suitable than other classifiers in this context.

Table 4.3: Overlap analysis among the classifiers considered in the preliminary study in all the considered classes.

	A=NB			A=Log			A=RBF			A=MLP			A=DTree		
	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A
B=NB	-	-	-	83	12	5	91	5	4	92	4	4	90	5	5
B=Log	83	5	12	-	-	-	83	5	12	81	6	13	80	6	14
B=RBF	91	4	5	83	12	5	-	-	-	89	5	6	89	5	6
B=MLP	92	4	4	81	13	6	89	6	5	-	-	-	90	5	5
B=DTree	90	5	5	80	14	6	89	6	5	90	5	5	-	-	-

Concerning the analysis of the complementarity, Table 4.3 summarizes the results achieved. In particular, for each pair of classifiers, the table reports the percentage of classes that are correctly classified as bug-prone by (i) both the compared classifiers (i.e., column A∩B), (ii) only the first classifier (i.e., column A-B), and (iii) only the second classifier (i.e., column B-A). From this table, we can observe that the overlap between the set of correctly classified instances by

a classifiers pair is at least 80%, *i.e.*, 80% of the instances are correctly classified by both the classifiers in the comparison. This means that the bug-proneness of most of the classes may be correctly predicted by an arbitrary classifier, while less than the remaining 20% of predictions are correctly classified by only one of the classifiers. However, the majority of the non-overlapping predictions are related to buggy classes (83%): such missing predictions can result in a reduction up to 35% of the performances of a bug prediction model.

More importantly, we noticed that these performances may be increased by analyzing how different classifiers behave on classes having different structural characteristics. As an example of the impact of the characteristics of classes on the effectiveness of bug prediction classifiers, consider the predictions provided by NB and MLP in the `Apache Velocity` project. We found the former more effective in predicting the bugginess of classes having more than 500 lines of code: indeed, although the F-Measure achieved by NB on this system is quite low (35%), the correct predictions refer to large classes containing bugs. An example is the `io.VelocityWriter`, which has 551 LOCs, and a low values cohesion and coupling metrics (*e.g.*, Coupling Between Methods (CBM) = 5). On the other hand, MLP is the classifier obtaining the highest F-Measure (62%), however its correct predictions refer to classes having a high level of coupling: indeed, most of the classes correctly predicted as buggy are the ones having the value of the Coupling Between Methods (CBM) metric higher than 13 and a limited number of lines of code (<400). For instance, the class `velocity.Template` contains 320 LOCs, but it has a CBM = 19. Thus, an adequate selection of the classifiers based on the characteristics of classes may increase the effectiveness of bug prediction.

Summary for RQ2.0. Despite some differences, the five experimented machine learning classifiers achieve comparable results in terms of accuracy. However, their complementarity could lead to combined models able to achieve better results.

4.4 EVALUATING ENSEMBLES OF CLASSIFIERS IN BUG PREDICTION

The *goal* of the empirical study is twofold: in the first place, we aim at enriching the investigation of the capabilities of ASCI, an ensemble methodology we proposed in a previous work [65], by comparing its performances with those

achieved by a larger set of existing ensemble techniques in the context of within-project bug prediction; on the other hand, we aim at evaluating how ASCI works when adopted for cross-project bug prediction and what is the effect of local learning on its performances. The *purpose* of the study is the better allocation of resources dedicated to testing activities. The *perspective* is of researchers interested in understanding how much the selection of an ensemble technique has effect on bug prediction capabilities, as well as of practitioners who want to evaluate the usability of ensemble-based bug prediction models.

Specifically, the research questions formulated in the study are the following:

- **RQ2.1:** *How does* ASCI *work in the context of within-project bug prediction when compared to existing ensemble techniques?*
- **RQ2.2:** *How does* ASCI *work in the context of cross-project bug prediction when compared to existing ensemble techniques?*
- **RQ2.3:** *To what extent can local learning improve the performances of* ASCI*?*

The first research question (**RQ2.1**) aimed at replicating our previous empirical study [65] by considering a wider set of ensemble techniques as alternatives to ASCI. In the second research question (**RQ2.2**), we assessed the performances of our approach in the context of cross-project bug prediction, while **RQ2.3** was focused on the combination between local learning and the use of ensemble methodologies.

4.4.1 *Context Selection and Data Preprocessing*

The *context* of the study was composed of the 21 software systems shown in Table 4.4. Specifically, we considered projects having different scope (*e.g.*, build or workflow management systems) and different size (*e.g.*, from 3 to 300 KLOC). Table 4.4 reports the specific releases taken into account as well as the detailed characteristics of the projects considered in terms of (i) size, expressed as number of classes and KLOC, and (ii) number and percentage of buggy classes. To properly select the dataset we considered two main factors. Firstly, we considered only publicly available datasets to guarantee a full replication of our experiments. Secondly, we selected software systems from various application domains and having different characteristics to reduce the threats to external validity of our study [32, 219]. Thus, we picked up a sample of 21

Table 4.4: Characteristics of the software systems used in the study

#	Project	Release	Classes	KLOC	Buggy Classes	(%)
1	Ant	1.7	745	208	166	22%
2	ArcPlatform	1	234	31	27	12%
3	Camel	1.6	965	113	188	19%
4	E-Learning	1	64	3	5	8%
5	InterCafe	1	27	11	4	15%
6	Ivy	2.0	352	87	40	11%
7	jEdit	4.3	492	202	11	2%
8	KalkulatorDiety	1	27	4	6	22%
9	Nieruchomosci	1	27	4	10	37%
10	pBeans	2	51	15	10	20%
11	pdfTranslator	1	33	6	15	45%
12	Prop	6.0	660	97	66	10%
13	Redaktor	1.0	176	59	27	15%
14	Serapion	1	45	10	9	20%
15	Skarbonka	1	45	15	9	20%
16	Synapse	1.2	256	53	86	34%
17	SystemDataManagement	1	65	15	9	14%
18	TermoProjekt	1	42	8	13	31%
19	Tomcat	6	858	300	77	9%
20	Velocity	1.6	229	57	78	34%
21	Zuzel	1	39	14	13	45%

systems from the initial set of 30 systems mined by Jureczko *et al.* [230] and available in the PROMISE dataset [204], after applying the guidelines proposed by Tantithamthavorn *et al.* [75] to ensure data robustness: specifically, we did not consider systems having more than 50% of buggy classes.

It is important to highlight that the considered dataset **already contained both independent and dependent variables used to build the bug prediction models**. More specifically, for each class of the considered systems the independent variables were represented by LOC and Chidamber and Kemerer metrics [231], while the dependent variable was represented by a boolean value indicating the bugginess of each class.

Once we selected the dataset, we applied the following data preprocessing activities:

1. **Data Cleaning**. As shown by Shepperd *et al.* [73], the PROMISE repository might contain noise and/or erroneous entries that possibly bias the results of bug prediction models. To deal with this issue, they proposed a data cleaning procedure composed of 13 corrections aimed at increasing the data quality. We applied these 13 steps to remove instances with conflicting

values or presenting missing values, etc. From the initial dataset composed of 5,422 instances, we removed $\simeq$ 1% of instances. Thus the final dataset was composed of 5,361 instances.

2. **Data Normalization**. A second element that could badly affect the performance of the prediction models is related to the different levels of design-complexity metrics [122, 232]. To overcome this issue, we applied the normalization filter implemented in WEKA [233] which linearly normalizes the data in the [0,1] interval. It is important to note that the choice of this normalization technique came from the results provided by Nam *et al.* [232] and Herbold *et al.* [234], who showed that such a technique represents the best one for this task.

3. **Feature Selection**. Highly correlated independent variables can negatively affect the capabilities of bug prediction models [235]. To avoid this issue, we applied the Correlation-based Feature Selection (CFS) approach [236]. This method uses correlation measures and a heuristic search strategy to identify a subset of actually relevant features for a model. It is worth noting that we applied CFS for each training set obtained from the *Leave-One-Out* cross validation process, described later in Section 4.4.3. Thus, we firstly combined the training instances belonging to different software systems and then we applied the feature selection algorithm, as recommended by Hall *et al.* [237].

4. **Data Balancing**. Bennin *et al.* [238] demonstrated that the problem of data unbalancing, *i.e.*, datasets having a number of buggy classes much lower than non-buggy ones, can bias the performance of bug prediction models. For this reason, we applied the *Synthetic Minority Over-sampling TEchnique*, *i.e.*, SMOTE [239] to ensure a similar proportion of buggy and non-buggy classes in the training sets.

It is important to note that the order of the preprocessing steps have been guided by the framework proposed by Song *et al.* [219], who suggested an ideal sequence of operations to perform before training a bug prediction model. The final preprocessed datasets are available in our online appendix [240].

4.4.2 *Baseline Selection*

We firstly compared the results obtained by ASCI with those achieved by the model relying on the Naive Bayes classifier, which was found to be the best stand-alone machine learner over our dataset. More specifically, we ran seven stand-alone classifiers, *i.e.*, Multi-Layer Perceptron, Naive Bayes, Logistic Regression, Radial Basis Function, C4.5, Decision Table, and Support Vector Machine on the same set of systems considered in the study and using the same validation methodology. As a result, we found that the use of Naive Bayes led to the best results in terms of MCC. For this reason, we considered such a classifier as our baseline. A complete overview of the results achieved by the stand-alone classifiers is available in our online appendix [240].

In the second place, we benchmarked ASCI with a set of ensemble techniques that were previously experimented for bug prediction. Specifically:

Boosting. The Boosting technique iteratively uses a set of models built in previous iterations to manipulate the training set [34]. At the next iteration, the model focuses on those instances more difficult to predict. Adaptive Boosting (AdaBoost) [116] is a well-known Boosting technique. During the training phase, AdaBoost repetitively trains a *weak* classifier on subsequent training data. At each iteration a weight is assigned to each instance of the training set, with the purpose of assigning higher weights to misclassified instances that should have more chances to be correctly predicted by the new models. At the end of the training phase, a weight is assigned to each model in order to reward models having higher overall accuracy. During the test phase, the prediction of a new instance is performed by voting of all models. The results are thus combined using the weights of the models, in case of binary classification a threshold of 0.5 is applied. We used the default configuration provided by the Weka toolkit [233], implementing NBBoosting, a model using Naive Bayes (NB) as *weak* learner. We used NB because, on the considered context, this classifier achieved better performance with respect to other classifiers.

Bootstrap Aggregating (Bagging). Bagging [117] combines the output of various models in a single prediction. During the training phase, m datasets with the same size as the original one are generated by performing sampling with replacement (Bootstrap) from the training set. Hence for each dataset a model is trained using a *weak* classifier. During the test phase, for each instance the composite classifier uses a majority voting rule to combine the output of the

models into a single prediction. We used the default configuration provided by the WEKA toolkit [233], implementing NBBAGGING, a model that use the same *weak* learner used for BOOSTING (*e.g.*, NB).

Validation and Voting. VALIDATION AND VOTING [118] (also called VOTING) is a weighting method. It combines the confidence scores obtained by the underlying classifiers. For each instance to predict, each classifier returns a confidence score ranging between 0 and 1. The scores are combined by an operator (*e.g.*, AVERAGE in case of AVGVOTING and MAXIMUM in case of MAXVOTING). A class is marked as buggy if the combination of the confidence scores is higher than 0.5, while it is predicted as clean otherwise. We configured VOTING as already done in previous work [33] using LOGISTIC REGRESSION, RADIAL BASIS FUNCTION, C4.5, DECISION TABLE, MULTI-LAYER PERCEPTRON, NAIVE BAYES, and SUPPORT VECTOR MACHINE.

CODEP. CODEP [33] is a technique based on STACKING [119] that uses a meta-classifier (*e.g.*, LOGISTIC REGRESSION) to infer the bugginess of classes [34]. During the training phase CODEP trains each of the base classifiers and create a new dataset by collecting the confidence scores assigned by the classifiers on each training instance. Finally, with the aim of combining the outputs of the base classifiers, a meta-classifier is built on such a new dataset. In our study, we configured the model by adding SUPPORT VECTOR MACHINE, a popular machine learner, to the set of base classifiers previously used by Panichella *et al.* [33] and used also for the VALIDATION AND VOTING ensemble methods. For the same reason we used LOGISTIC REGRESSION as meta-classifier.

Random Forest. RANDOM FOREST [120] is an ensemble of pruned decision trees. As showed for BAGGING, each decision tree is built by using BOOTSTRAP. The combination of the prediction of the decision trees is performed by using majority voting. In our experiments we used the default configuration provided by the WEKA toolkit [233].

We are aware of the possible impact of classifiers' configuration on the ability of finding bugs [241], however the identification of the ideal settings in the parameter space of a single classification technique would have been prohibitively expensive [242]. For this reason, we applied the classifiers using their default configuration.

4.4.3 *Validation Strategies and Evaluation Metrics*

A key decision in this context was the selection of an appropriate validation strategy [243].

In the context of *within-project* bug prediction (**RQ2.1**), we adopted the *10-Fold Cross Validation* [244]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (*e.g.*, each fold has the same proportion of bugs). A single fold is used as test set, while the remaining ones are used as training set. The process was repeated 10 times, using each time a different fold as test set. Then, the model performances were reported using the mean achieved over the ten runs. It is important to note that we repeated the 10-fold validation 100 times (each time with a different seed) to cope with the randomness arising from using different data splits [74].

In the contexts of *cross-project* bug prediction (**RQ2.2**), we could not adopt the *10-Fold Cross Validation* as validation strategy, as we could not use as test set data coming from the same system as the training set. Thus, we opted for the *Leave-One-Out Cross-Validation* [245]. In this strategy, the model is trained using the data of all the systems but one, which is retained as *test set*. The cross-validation has been then repeated 21 times, allowing each of the 21 systems to be the test set exactly once [245]. We used this validation strategy since it is among the least biased and most stable validation approaches, according to the findings reported by Tantithamthavorn *et al.* [243]. Furthermore, it is important to note that this strategy (i) allows all systems to be used for both training and test purpose, and (ii) has been widely used in the context of bug prediction [33, 42, 61, 246].

In the context of **RQ2.3**, we had to build local bug prediction models. To this aim, we exploited the Expectation Maximization (EM) clustering algorithm proposed by Dempster *et al.* [247]. The choice of this algorithm was driven by multiple factors. Firstly, it can automatically determine the number of clusters through an internal cross-validation process. Secondly, it is similar to the MCLUST algorithm used by Bettenburg *et al.* [248] and Menzies *et al.* [40]. Lastly, previous work [234] showed that the performance achieved by EM are close to those obtained by the algorithm originally proposed by Menzies *et al.* [40]. In the context of this study, we relied on the implementation of the algorithm available in the Weka toolkit [233]. Given a project P_i, the input of the clustering algorithm was represented by the data coming from all the

systems bug P_i, *i.e.*, we still worked in a cross-project setting by means of the *Leave-One-Out Cross-Validation* [245] where the test sets were represented by the data of P_i. We created a bug prediction model relying on ASCI for each of the clusters.

As evaluation metrics, we avoid the computation of the widely used accuracy and F-Measure, as they are threshold-dependent metrics that can bias the interpretation of bug prediction capabilities [74]. Conversely, to properly evaluate the ability of our approach to predict the bug-proneness of classes we relied on the Matthew's Correlation Coefficient (MCC), which is a measure indicating the extent to which the independent and dependent variables are well related to each other. Metric values close to 1 indicate higher performances. As shown by Hall *et al.* [74], this is the most reliable threshold-independent metric for the evaluation of bug prediction models.

As a final step of our analyses, we also statistically verified the validity of our findings. To this aim, we exploited the Scott-Knott ESD test [243]: this is an extension of the original Scott-Knott test [249] that (i) applies a hierarchical clustering algorithm to group together the performances of the cross-project bug prediction models experimented based on the statistically significance of the differences observed in terms of MCC, and (ii) refines the clusters by merging together groups whose differences are negligible in terms of effect size [210].

4.5 ANALYSIS OF THE RESULTS

In this section we present the results of our study, by discussing each research question independently.

4.5.1 *RQ2.1: Evaluation of Ensemble Techniques when Adopted for Within-Project Bug Prediction*

Figure 4.3 depicts the box plots of the MCC achieved on the 21 software systems in our dataset by ASCI and the baseline experimented within-project bug prediction models (white asterisks highlight the means).

As shown, ASCI has performances comparable with the other ensemble techniques: it has a median MCC slightly higher than all the baseline approaches but AvgVoting, however it does not provide major improvements in the identi-

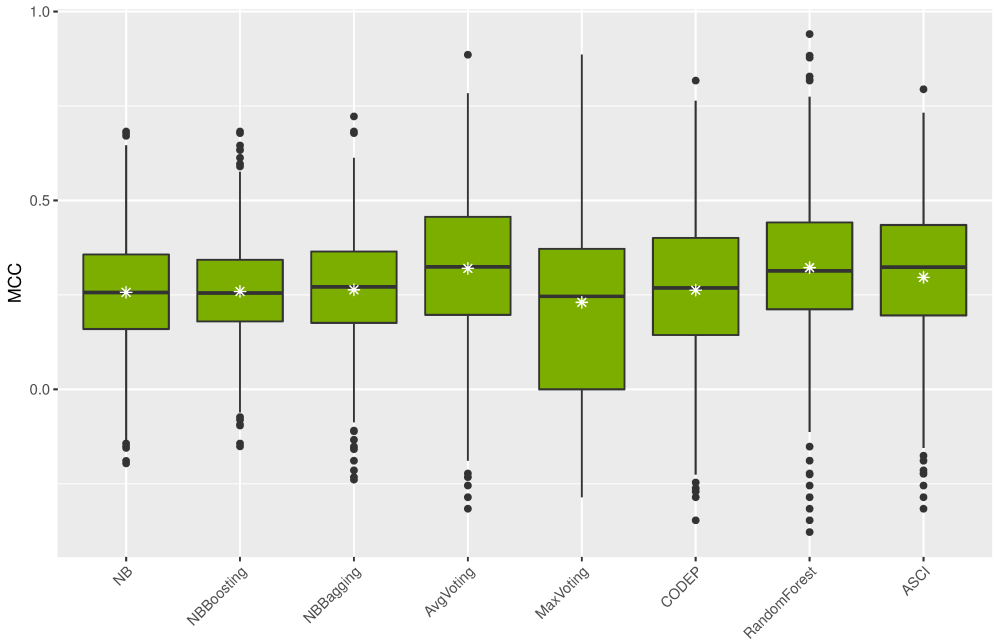


Figure 4.3: Boxplots of MCC achieved by the ensemble methods under study and NB in the within-project bug prediction context.

fication of buggy classes. Thus, our study confirms previous findings on the capabilities of ASCI [65].

A surprising finding concerns the performances of the model trained using Naive Bayes (NB), *i.e.*, the model built using the best stand-alone classifier. As it is possible to observe, not only its performances are generally similar to those achieved by models built using ensemble classifiers, but also in some cases it has better prediction capabilities. Indeed, in 4 out 7 cases (*e.g.*, NBBoosting, NBBagging, MaxVoting, CODEP), the performances achieved by the models based on the ensemble techniques are similar (*e.g.*, mean improvement smaller than 1% in terms of MCC) or worse that those based on NB (average MCC 25%). Thus, the *use of ensemble classifiers does not guarantee better prediction performances than stand-alone models*.

Looking to the other models, we found that AvgVoting is able on average to obtain the best results (*e.g.*, +7% with respect to NB) but unlike previous work [37, 65], we found the *average improvement* with respect to other ensemble techniques (*e.g.*, RandomForest and ASCI) *is quite limited or negligible*.

More in general, we noticed that *within-project models suffer of high performance variability* across different runs, meaning that variations in the training set lead to very different results. Thus, practitioners interested in setting up within-

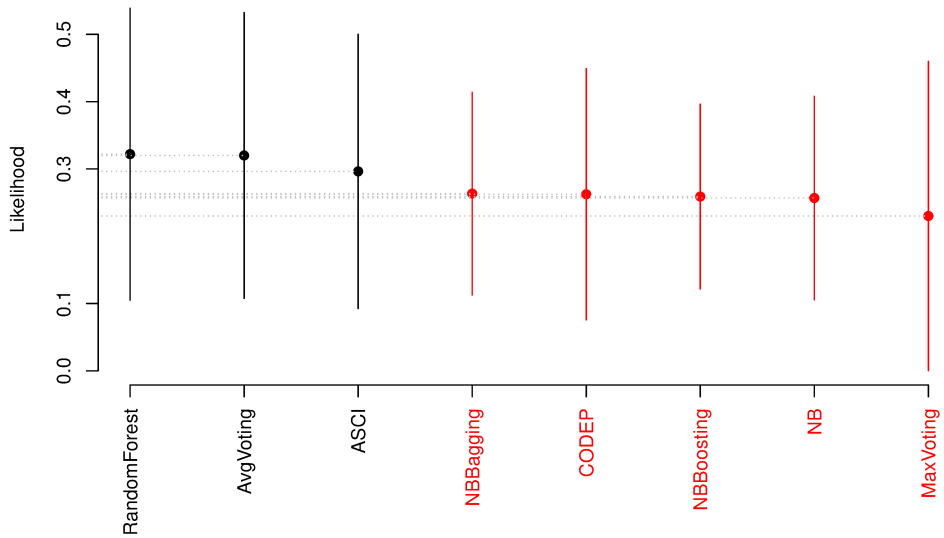


Figure 4.4: The likelihood of each technique in within prediction appearing in the top Scott-Knott ESD rank in terms of MCC. Circle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a classification technique appears at the top-rank for 50% of the studied datasets.

project models need to be careful when selecting the training set of machine learning techniques.

Figure 4.4 shows the likelihood of each analyzed ensemble techniques along with NAIVE BAYES to appear in the top Scott-Knott ESD rank. Interestingly, the statistical test showed that the differences observed among RANDOMFOREST, AVGVOTING, and ASCI are negligible in terms of effect size. Indeed, they were placed in the same cluster by the test, meaning that they are statistically equivalent in terms of prediction capabilities. From a practitioner's perspective, this result suggests the selection of one of these three techniques while setting up a prediction model based on ensemble classifiers.

Summary for RQ2.1. In the context of within-project bug prediction, the use of an ensemble classifier does not guarantee better prediction performances with respect to the best stand-alone classifier (*e.g.*, NAIVE BAYES). We confirm that the models based on VALIDATION AND VOTING are able to achieve slightly better results, but the obtained improvement is not statistically significant with respect to other ensemble techniques, such as RANDOM FOREST and ASCI.

4.5.2 *RQ2.2: Evaluation of Ensemble Techniques when Adopted for Cross-Project Bug Prediction*

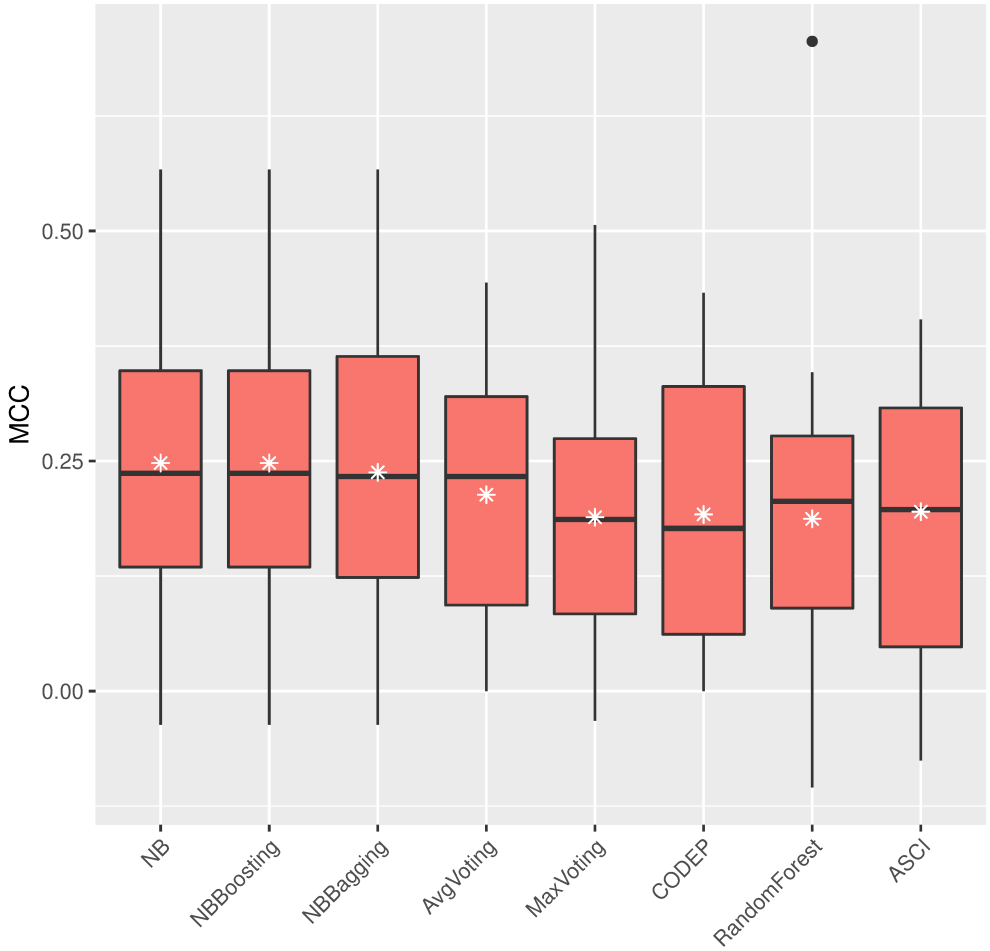


Figure 4.5: Boxplots of MCC achieved by the ensemble methods under study and NB in the cross-project bug prediction context.

Figure 4.5 depicts the box plots of the AUC-ROC and MCC achieved on the 21 software systems in our dataset by the experimented cross-project bug prediction models (white asterisks highlight the means).

Also in this context, we found Naive Bayes to have performances similar to those achieved by ensemble methods. Thus, we confirm again that the models based on ensemble classifiers do not necessarily provide improvements with respect to a well selected stand-alone model.

As for ASCI, our results clearly show how its performances are lower than Naive Bayes, NBBoosting and NBBagging. At the same time, AvgVoting

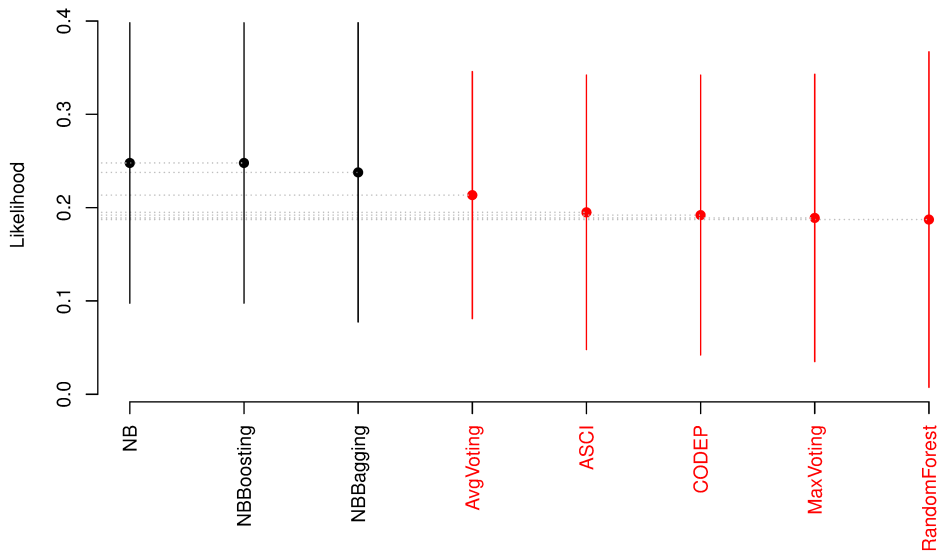


Figure 4.6: The likelihood of each technique in cross prediction appearing in the top Scott-Knott ESD rank in terms of MCC. Circle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a classification technique appears at the top-rank for 50% of the studied datasets.

shows a slightly better accuracy with respect to ASCI in terms of MCC (+2% on average). Thus, the application of our technique in a cross-project setting does not provide improvements in the prediction og bugs: possibly, this means that ASCI suffers the well-known problem of data heterogeneity.

In the second place, we noticed that the performances of the VALIDATION AND VOTING approach depend on the operator used to combine the outputs of different stand-alone classifiers: specifically, AVGVOTING tends to perform slightly better than MAXVOTING in terms of MCC (21% vs 18%). Thus, the setting of the technique has an impact on its performances by up to 3% in terms of MCC. Furthermore, unlike previous work [35, 61] we found that in some cases AVGVOTING performs worse than other ensemble techniques. For example, the average MCC achieved by NBBOOSTING and NBBAGGING is respectively 4% and 3% higher (25% and 24% vs 21%).

A third interesting finding regards the performances of CODEP. Our results confirm those reported by Zhang *et al.* [61]: in particular, we found that AVGVOTING outperforms CODEP (+2% in terms of MCC). Thus, *we can confirm that* AVGVOTING *tends to be more stable and accurate than* CODEP.

It is also interesting to discuss the results of the RANDOM FOREST technique. Previous work by Robnik *et al.* [250] and Jiang *et al.* [251] observed that it is one of the most reliable and accurate machine learners, however also in this case we discovered that it is not able to provide improved performances with respect to other ensemble techniques. In particular, it has worse performances than AVGVOTING in terms of MCC (*e.g.*, -2%).

As a more general observation, it is important to note that the performances of all the cross-project models experimented are quite low—on average they do not exceed 25% in terms of MCC. On the one hand, all the experimented models solely relied on code metrics as independent variables. As widely shown in literature [23, 64, 17] a combination of predictors of different natures (*e.g.*, process metrics) has an important effect on the overall performances of bug prediction models. On the other hand, our results still suggest that cross-project bug prediction is still far from being actually usable in practice. For this reason, the research community needs to investigate more the problem, trying to identify useful tools to make cross-project bug prediction actually effective.

The results discussed so far are also statistically significant: Figure 4.6 shows the likelihood of each analyzed ensemble techniques along with NAIVE BAYES to appear in the top Scott-Knott ESD rank. NAIVE BAYES and the ensemble using it as weak learner (*e.g.*, NBBOOSTING and NBBAGGING) are able to achieve the best performances in terms of MCC. All the other experimented ensemble methods, including ASCI are statistically worst than them and, therefore, they are placed in another cluster.

Summary for RQ2.2. None of the cross-project models experimented is able to exceed 25% of MCC (on average), meaning that the problem of identifying buggy classes using external sources of information is still far from being solved. Furthermore, the use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers. Indeed, the models based on NAIVE BAYES or using it as weak learner (*e.g.*, NBBOOSTING and NBBAGGING) are able to achieve the best performances. ASCI, instead, does not work properly in a cross-project setting.

4.5.3 *RQ2.3: Evaluation of Ensemble Techniques when Adopted for Local Cross-Project Bug Prediction*

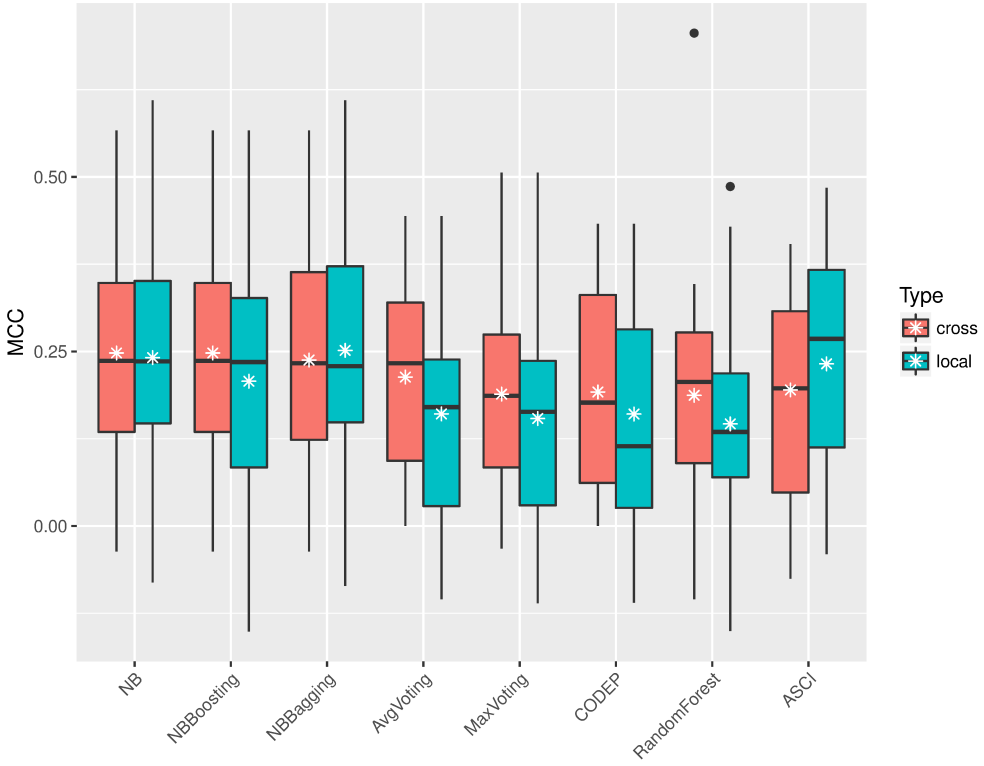


Figure 4.7: Boxplots of MCC achieved by the ensemble methods under study and NB in the global/local cross-project bug prediction context.

On the basis of the results achieved in **RQ2.2**, we verified whether the application of local learning—that was suggested as a promising way to reduce data heterogeneity—could improve the performances of ASCI. Figures 4.7 depicts the box plots reporting MCC achieved on the 21 subject systems when combining local learning and the ensemble techniques considered in our study, along with those achieved by the standard local bug prediction model that relies on Naive Bayes. To ease the comparison with the results of **RQ2.2**, we also report box plots for the global models built using the same set of classifiers.

As a first observation, *local models do not always achieve better performances with respect to global models*. While we could confirm previous findings [234], we also observed that *the use of ensemble techniques for local bug prediction can sometimes badly affect the capabilities of the resulting models*. An example of this behavior is represented by the ensemble methods exploiting Naive Bayes as weak learner

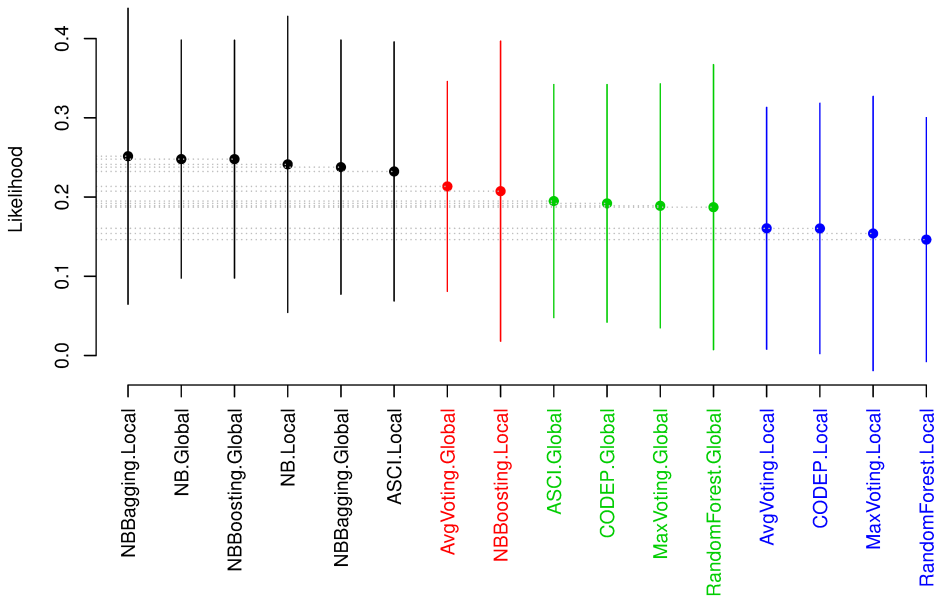


Figure 4.8: The likelihood of each technique in global/local cross prediction appearing in the top Scott-Knott ESD rank in terms of MCC. Circle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a classification technique appears at the top-rank for 50% of the studied datasets.

(*i.e.*, NBBoosting and NBBagging): we can notice that in case of NBBoosting, the average MCC significantly decreases with respect to the corresponding global models (-6% and -4%, respectively). Moreover, in the comparison with Naive Bayes, NBBoosting and NBBagging showed similar performance as the stand-alone one.

Another interesting observation concerns the results of ensemble techniques combining the output of different classifiers, *i.e.*, Validation and Voting and CODEP. In these cases local models tend to provide worse performances than global ones. As for the two Validation and Voting techniques experimented, a likely motivation for such result comes from the characteristics of the algorithms: as shown in previous research [41, 65], this technique fails in case of high variability among the predictions provided by different classifiers because the majority of the base classifiers might wrongly classify the bug-proneness of a class, thus negatively influencing the performances of techniques which combine the output of different classifiers. When applying Validation and Voting locally, we observed that specific classifiers act much better on some specific clusters than other machine learners, meaning that very few of them can correctly classify the bug-proneness of code entities. As a consequence, the

voting is often not useful, leading to a decreasing of the overall performances of local bug prediction: from a quantitative point of view, the local scenario reduces the average MCC achieved by AVGVOTING and MAXVOTING by up to 5% and 3%, respectively, when compared to global models.

Looking at RANDOM FOREST, we found that this technique was badly affected when applying the clustering technique. Specifically, local models exploiting such classifier obtained a median MCC 7% lower with respect to global cross-project models built using the same classifiers.

Besides the results discussed so far, we found ASCI to be the only classifier actually able to exploit the lower heterogeneity of data provided by local models. Indeed, it was able to boost its MCC of 7%, becoming much more effective than the baselines, both considering their global and local versions. In other words, *the local version of* ASCI *provides better performances with respect to all the other global and local models experimented.*

As previously done, we performed the Scott-Knott ESD test. Figure 4.8 shows the likelihood of each the analyzed ensemble techniques along with NAIVE BAYES to appear in the top Scott-Knott ESD rank. Also in this case, we report both *local* and *global* cross-project bug prediction models for sake of understandability of our findings.

We could notice that three local models, *i.e.,* NBBAGGING.LOCAL, NB.LOCAL, and ASCI.LOCAL, have a similar average likelihood as the best global models, *i.e.,* NB.GLOBAL, NBBOOSTING.GLOBAL, and NBBAGGING.GLOBAL. Thus, we can conclude that *local and global models are mostly equivalent from a statistical point of view*. Moreover, local bug prediction should be considered, when applying ASCI in the context of cross-project bug prediction.

Summary for RQ2.3 Local learning is often not able to improve the performances of bug prediction models. The only exception is represented by ASCI, which has better performances with respect to those achieved by global models. The statistical analysis conducted, however, highlighted how local and global models are mostly equivalent.

4.6 DISCUSSION

After having discussed the main results for our three research questions, in this section we provide an in-depth analysis and discussion on the different

performances achieved by the experimented models in within- and cross-project settings.

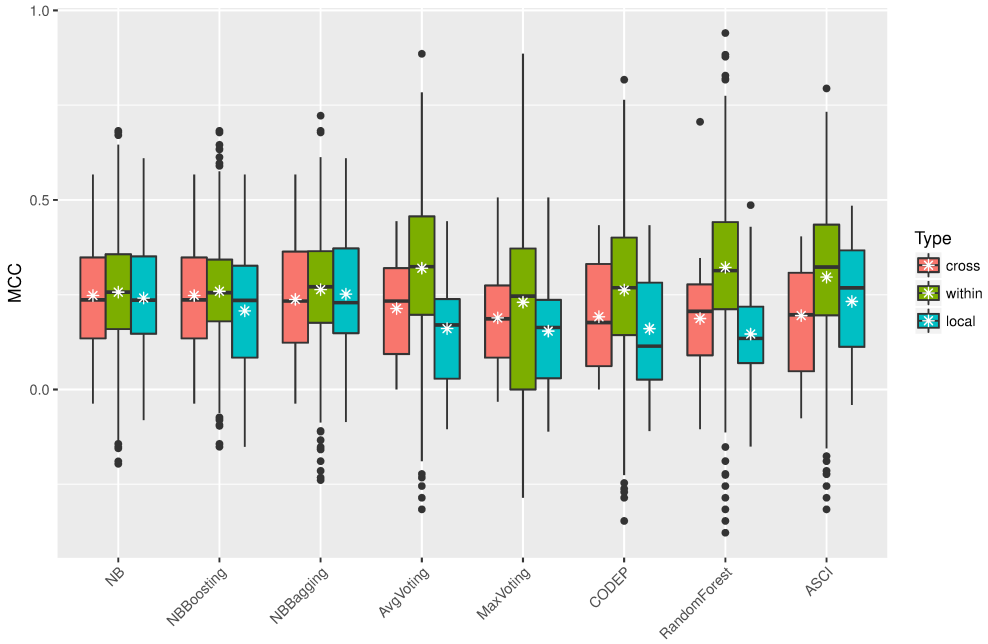


Figure 4.9: Boxplots of MCC achieved by the ensemble methods under study and NB in the global/local cross-project and within-project bug prediction contexts.

Figures 4.9 shows the box plots reporting MCC achieved by global and local cross-project models as well as by within-project models built using the ensemble techniques considered in our study.

Looking at Figure 4.9 we can conclude that *within-project models generally exhibit better performances with respect to cross-project ones*, thus confirming previous findings by Turhan *et al.* [42]. The result holds for all the experimented ensemble techniques. In general, we noticed that also in the within-project scenario the *use of ensemble classifiers does not guarantee better prediction performances with respect to stand-alone models*, *e.g.*, MaxVoting performs 1% worse than Naive Bayes.

Looking more in depth, we noticed that the model relying on Naive Bayes slightly improve its capabilities when trained using a within-project strategy (+1% and +2% with respect to global and local cross-project models, respectively). This result is confirmed also when applying Boosting (+1% and +5%) and Bagging (+2% and +1%).

Interesting is the case of models based on the Validation and Voting technique. Specifically, we found a significant performance decay when changing

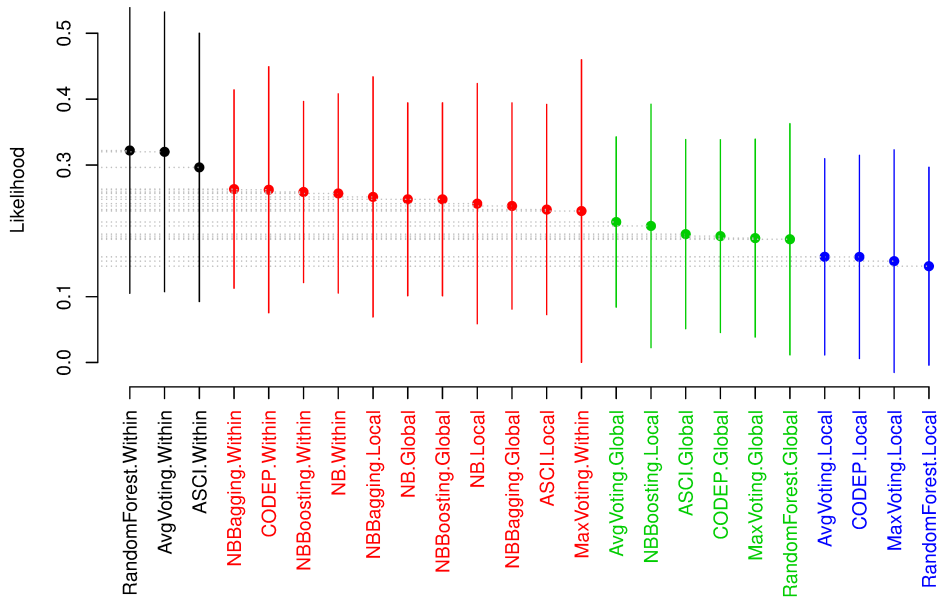


Figure 4.10: The likelihood of each technique in within and global/local cross prediction appearing in the top Scott-Knott ESD rank in terms of MCC. Circle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a classification technique appears at the top-rank for 50% of the studied datasets.

the training strategy, independently from the combination operation (*i.e.*, Average or Maximum): the MCC of AvgVoting and MaxVoting were 11% and 4%, respectively, lower than those achieved in the within-project context and even lower when looking at the local models (-15% and -8%, respectively).

Also in the case of classifiers based on the combination of multiple learners (*i.e.*, CODEP, ASCI, and RandomForest) we observed important differences when considering a within- or cross-project training. In particular, the average MCC reported by Random Forest dropped of 13%, while CODEP and ASCI dropped by 7% and 10%, respectively.

The Scott-Knott ESD test in Figure 4.10 shows the likelihood of each the analyzed ensemble techniques along with Naive Bayes in the contexts of within and cross-project bug prediction. The statistical analyses confirmed the results discussed so far: indeed, within-project models are able to achieve better performances in terms of MCC, thus being statistically more accurate than the others.

As an additional analysis aimed at measuring the robustness of the experimented models, we computed the Area Under the ROC Curve (AUC-ROC). This

metric, ranging between 0.5 and 1, reports the overall capabilities of a prediction model in discriminating buggy and non-buggy classes. A metric values close to 1 indicate higher performances. It is important to note that AUC-ROC and MCC are two complementary metrics: while MCC statistically measures the accuracy of the predictions obtained by the classifier, the AUC-ROC gives an indication on its robustness [252] (*i.e.,* how well the classifier separates the binary classes).

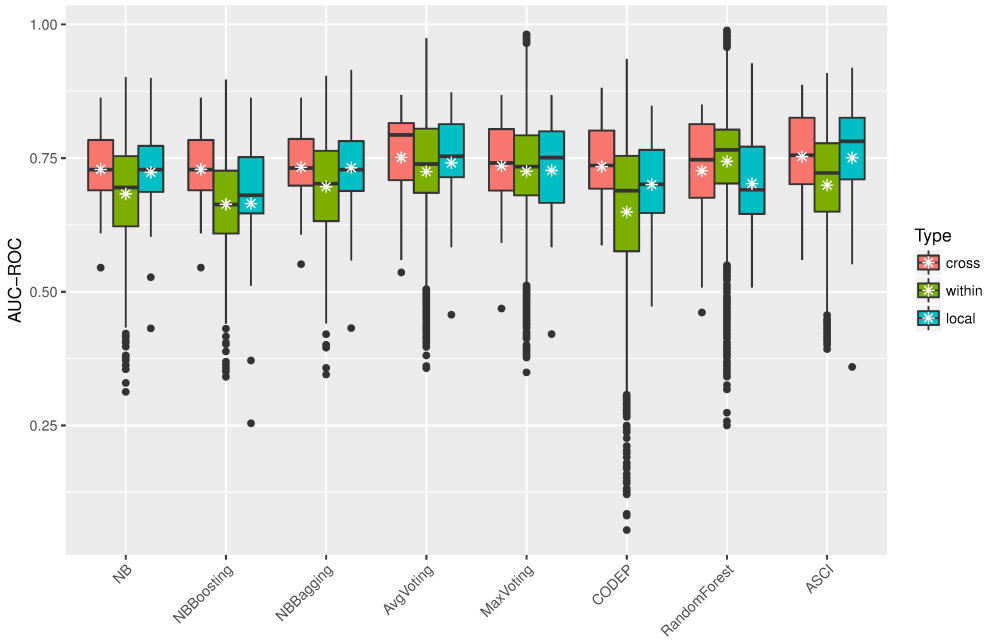


Figure 4.11: Boxplots of AUC-ROC achieved by the ensemble methods under study and NB in the global/local cross-project and within-project bug prediction contexts.

Figure 4.11 shows box plots representing the performances of global cross-, local cross-, and within-project models in terms of AUC-ROC. As shown, we observed that *cross-project models generally perform better than within-project ones*. In our opinion, this is a relevant result since it clearly shows the limitation of the data analysis conducted by previous work which only interpreted results by considering the F-Measure.

Looking more deeply into the results, we found that cross-project models behave similarly, if not better, than within-project ones when considering the AUC-ROC. From a practical point of view, this means that *cross-project models are more robust than within-project ones* (*i.e.,* their performances do not vary as much as the ones of within-projects models when ran on different data), while *within-project models are more precise than cross-project ones* (*i.e.,* their accuracy is higher than the one of cross-project models). This result seems to suggest

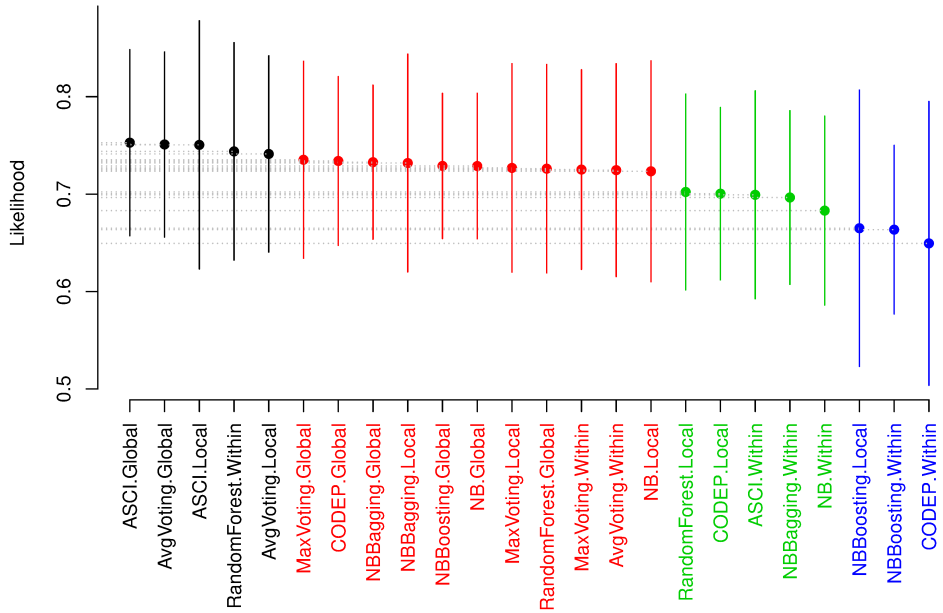


Figure 4.12: The likelihood of each technique in within and global/local cross prediction appearing in the top Scott-Knott ESD rank in terms of AUC-ROC. Circle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval. 50% of likelihood means that a classification technique appears at the top-rank for 50% of the studied datasets.

that within- and cross-project strategies have different pros and cons, being to some extent complementary: as part of our future agenda, we plan to further investigate the extent to which a smart mixture of both the strategies can lead to better bug prediction performances.

It is interesting to note that the local version of ASCI is the technique that performs better than all the others. This result confirms that such technique should be preferred in case the robustness is the main objective that a practitioners wants to achieve when running a bug prediction model.

The Scott-Knott ESD test in Figure 4.12 shows the likelihood of each the analyzed ensemble techniques along with NAIVE BAYES in the contexts of within and cross-project bug prediction. The statistical analyses confirmed the results discussed so far: indeed, cross-project bug prediction models are more reliable that within-project ones when considering the AUC-ROC (*i.e.*, of the top-5 models, only one is trained using a within-project strategy); on the other hand within-project models are able to achieve better performances in terms of MCC, thus being statistically more accurate than the others.

4.7 THREATS TO VALIDITY

In this section we discuss the threats that might affect the validity of the empirical study conducted in this chapter.

Threats to construct validity. Threats in this category regard the relationship between theory and observation. In our work, a threat is represented by the dataset we relied on. The dataset come from the PROMISE repository [204], which is widely considered reliable and, indeed, has been also used in several previous work in the field of bug prediction [60, 61, 37, 33, 32, 35, 114, 253]. Although we cannot exclude possible imprecisions and/or incompleteness of the data used in the study, we applied a formal data preprocessing recommended by Shepper *et al.* [73], which allowed us to reduce noise and remove erroneous entries present in the considered datasets. Moreover, it is important to note that to produce stable results we just considered software systems having less than 50% of buggy classes [75].

As for the experimented prediction models, we exploited the implementation provided by the WEKA framework [233], which is widely considered as a reliable source.

We are aware of the importance of parameter tuning for bug prediction models. To acknowledge that using the default parameters for each classifier is a threat to our study. However, we used them since finding the best configuration for all of the classifiers would have been too expensive [242]. As future goal, we plan to further analyze the impact of parameters' configuration to our findings.

Threats to conclusion validity. They are related to the relation between treatment and outcome. To reduce the impact of the adopted validation methodology, we relied on the *Leave-One-Out Cross-Validation* methodology [245]. This choice was driven by results recently reported that showed that such validation technique is among the ones that are more stable and reliable [243].

To ensure that the results would have not been biased by confounding effects due to data unbalance [239] or highly correlated independent variables [254], we adopted formal procedures aimed at (i) over-sampling the training sets [239] and (ii) removing non-relevant independent variables through feature selection [236].

As for the evaluation of the performances of the experimented models, we considered AUC-ROC and MCC, which have been highly recommended by Hall *et al.* [74] to correctly interpret the results. We excluded, instead, other

widely-used metrics such as precision, recall, and F-Measure [198] because they are threshold-dependent and possibly hide the real performances of bug prediction models [74].

Threats to external validity. These are threats concerned with the generalizability of the findings. We analyzed 21 different software projects coming from different application domains and having different characteristics (*i.e.*, developers, size, number of components, *etc.*). Of course, we cannot claim the generalizability with respect to industrial environments, however the replication of the study on industrial projects is part of our future research agenda.

Regarding the selected ensemble techniques, we considered a variety representative of the state of the art (*e.g.*, [33, 60]). Finally, it is important to note that we built models based on code metrics: as part of our future research agenda, we aim at analyzing the impact of process- (*e.g.*, the entropy of changes proposed by Hassan *et al.* [14]) and developer-related (*e.g.*, the number of developers working on a code component [15]) metrics on our findings.

4.8 CONCLUSION

In this chapter we proposed ASCI, an approach able to dynamically recommend the classifier to use to predict the bug-proneness of a class based on its structural characteristics.

To build our approach, we firstly performed an empirical study aimed at verifying whether five different classifiers correctly classify different sets of buggy components: as a result, we found that even different classifiers achieve similar performances, they often correctly predict the bug-proneness of different sets of classes.

We evaluated the usage of ASCI for within- and cross-project bug prediction. Specifically, we compared its performances with those achieved by other six ensemble techniques previously used in bug prediction on a set of 21 software projects from the PROMISE dataset. With respect to previous benchmark studies, we mitigated possible threats to validity applying some precautions with respect to the quality of data used.

We found that the problem of cross-project bug prediction is still far from being solved. The use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers, but in general the VALIDATION

AND VOTING and ASCI techniques should be preferred among other ensemble methodologies.

When turning our attention on the combination between local learning and ensemble classifiers, we did not observe major differences; indeed, the statistical analyses revealed that local and global models are mostly equivalent. However, we found ASCI to be the only technique that is effective in exploiting local learning for reducing data heterogeneity and improving its prediction capabilities.

Finally, we found some key findings when comparing cross- and within-project models. In the first place, the latter are more precise than cross-project models, independently from the training strategy (global vs local). Nevertheless, the use of ensemble classifiers in the context of within-project models does not guarantee better prediction performances with respect to models relying on stand-alone classifiers. On the other hand, we also observed that cross-project models are more robust than within-project ones, meaning that the two strategies might be complemented in order to take advantages of the pros of each strategy.

This observation is the main starting point for our future research agenda. Still, we plan to replicate the study in an industrial context, using a richer set of independent variables, and investigating the impact of classifiers configuration on our findings. More importantly, following the suggestions by Lanza *et al.* [255] we plan to perform a user study with developers aimed at evaluating the real usefulness of the suggestions provided by the different bug prediction models experimented. Finally, we plan to investigate how to combine cross-project and within-project strategies to achieve more accurate and robust models.

5

A TEST CASE PRIORITIZATION GENETIC ALGORITHM GUIDED BY THE HYPERVOLUME INDICATOR

5.1 INTRODUCTION

The goal of regression testing is to verify that software changes do not affect the behavior of unchanged parts [50]. Many approaches have been proposed in literature to reduce the effort of regression testing [50, 51], which remains a particular expensive post-maintenance activity [52]. One of these approaches is *test case prioritization* (TCP) [53, 54], whose goal is to execute the available test cases in a specific order that increases the likelihood of revealing regression faults earlier [55]. Since fault detection capability is unknown before test execution, most of the proposed techniques for TCP use coverage criteria [50] as surrogates with the idea that test cases with higher code coverage will have a higher probability to reveal faults. Once a coverage criterion is chosen, search algorithms can be applied to find the ordering maximizing the selected criterion.

Greedy Algorithms have been widely investigated in the literature for test case prioritization, such as simple greedy algorithms [50], additional greedy algorithms [53], 2-optimal greedy algorithms [51], or hybrid greedy algorithms [141]. Other than greedy algorithms, meta-heuristics have been applied as alternative search algorithms to test case prioritization. To allow the application of meta-heuristics, proper fitness functions have been developed [51], such as the Average Percentage Block Coverage (APBC), or the Average Percentage Statement Coverage (APSC). Each fitness function measures the Area Under Curve (AUC) represented by the cumulative coverage and cost scores achieved when incrementally executing the test cases according to a specific prioritization (or order). As such, multiple points in the cost-coverage space are condensed into a single scalar value that can be used as a fitness function for meta-heuristics, such as single-objective genetic algorithms. Later work on search-based TCP also employed multi-objective genetic algorithms considering different AUC-based metrics as different objectives to optimize [78, 79, 80, 81].

We observed that the AUC metric used in the related literature for TCP represents a simplified version of the well-known *hypervolume* [62], which is a metric used in many-objective optimization. Indeed, the problem of condensing multiple points in the objective space (*i.e.,* a Pareto front) has been already investigated in many-objective optimization using the more general concept of *hypervolume under manifold* [62], which is a generalization of the AUC-based metrics used in previous TCP studies but for the higher dimensional objective space. We argue that the *hypervolume* can be used to condense not only a single cumulative code coverage criteria (as done by previous AUC metrics used in TCP literature) but also multiple testing criteria, such as the test case execution cost or further coverage criteria (*e.g., branch, and past-fault coverage*), in only one scalar value.

We introduce a Hypervolume-based Genetic Algorithm (`HGA`) to solve the TCP problem with multiple testing criteria. We conduct an empirical study with the aim to answer the following questions: (1) *How does `HGA` perform compared to other state-of-the-art techniques for the TCP problem?* (2) *To what extent does `HGA` scale when dealing with more than three testing criteria?*

To answer the aforementioned open questions, in this chapter we provide an extensive evaluation of Hypervolume-based and state-of-the-art approaches for TCP when dealing with up to five testing criteria. In particular, we carried out two different case studies to assess the *cost-effectiveness*, the *efficiency*, and the *scalability* of the various approaches. In the first study, we compared `HGA` with respect to two state-of-the-art techniques: a cost cognizant additional greedy algorithm [53, 77]; and `NSGA-II`, a multi-objective search-based algorithm [78, 79, 80, 81]. The study is designed to answer the following research questions:

- **RQ3.1:** *What is the cost-effectiveness of HGA, compared to state-of-the-art test case prioritization techniques?*

- **RQ3.2:** *What is the efficiency of HGA, compared to state-of-the-art test case prioritization techniques?*

Our results suggest that the solution (test ordering) produced by `HGA` is more cost-effective than the solution generated by `Additional Greedy` and the Pareto optimal solution achieved by `NSGA-II`. In terms of efficiency, `HGA` is much faster than `NSGA-II` and `Additional Greedy`, and its efficiency does not decrease as the size of the software program and of the test suite increase.

To assess the scalability of HGA, we conducted a second case study by considering up to five testing criteria (*i.e.*, objectives). This further evaluation is needed since previous studies [82, 83] showed the benefits of optimizing multiple criteria in regression testing with respect to considering each criterion individually. Moreover, a well-known problem in many-objective optimization is that traditional multi-objective evolutionary algorithms (e.g., NSGA-II) do not scale when handling more than three criteria. This is because the number of non-dominated solutions increases exponentially with the number of objectives [84]. Therefore, we compared the performance of HGA with those achieved by two many-objective algorithms, namely GDE3 [256] and MOEA/D-DE [257], that are known to outperform NSGA-II when optimizing more than three criteria. Specifically, our second study is steered by the following research questions:

- **RQ3.3:** *What is the cost-effectiveness of HGA, compared to many-objective test case prioritization techniques?*
- **RQ3.4:** *What is the efficiency of HGA, compared to many-objective test case prioritization techniques?*

Our results show that HGA is not only more or equally effective than the state-of-the-art many-objective algorithms, but it is also up to 16 times more efficient.

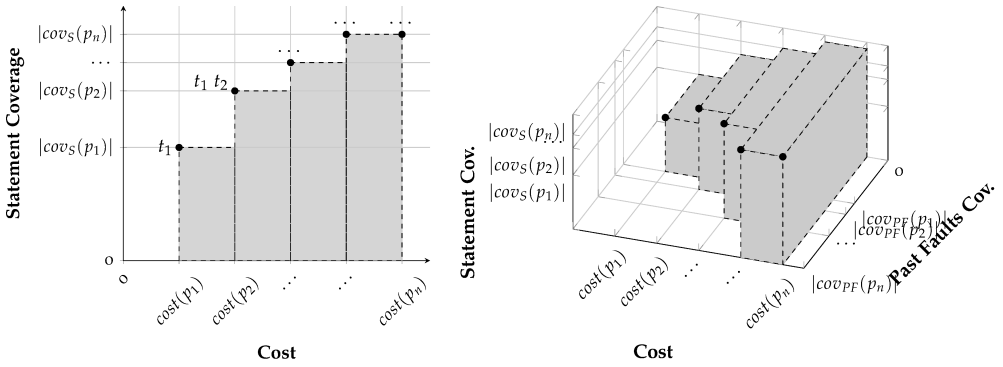
5.2 HYPERVOLUME GENETIC ALGORITHM FOR TEST CASE PRIORITIZATION

This section describes the proposed hypervolume metric for the multi-objective test case prioritization problem. It also highlights connections and differences with the AUC-based metrics used in previous work on search-based test case prioritization [51, 78, 79, 80, 81].

5.2.1 *Hypervolume indicator*

In many-objective optimization there is a growing trend to solve many-objective problems using *quality scalar indicators* to condense multiple objectives into a single objective [62]. Therefore, instead of optimizing the objective functions directly, indicator-based algorithms are aimed at finding a set of solutions that maximize the underlying quality indicator [62]. One of the most popular

indicators is the *hypervolume*, which measures the quality of a set of solutions as the total size of the objective space that is dominated by one (or more) of such solutions (combinatorial union [62]). For two-objective problems, the *hypervolume* corresponds to the area under curve, i.e., the proportion of the area that is dominated by a given set of candidate solutions, while for three-objective problems it is represented by a volume.



(a) Two testing criteria

(b) Three testing criteria

Figure 5.1: Cumulative points in two- and three-objective test case prioritization. The gray area (or volume) denotes the portion of objective space dominated by the cumulative points $P(\tau)$.

Hypervolume in two-objective TCP. To illustrate intuitively the proposed hypervolume metric, let us consider for simplicity only two testing criteria: (i) maximizing the statement coverage and (ii) minimizing the execution cost of a test suite. When considering the test cases in a specific order, the cumulative coverage and the cumulative execution cost reached by each test case draw a set of points within the objective space.

For example, let us consider the test suite $T = \{t_1, t_2, \ldots, t_n\}$ with the following statement coverage $Cov = \{cov_S(t_1), cov_S(t_2), \ldots, cov_S(t_n)\}$ and execution cost $Cost = \{cost(t_1), cost(t_2), \ldots, cost(t_n)\}$. As depicted in Figure 5.1-(a), if we consider the ordering $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ we can measure the cumulative scores as follows: the first test case t_1 covers a specific set of code statements $cov_S(p_1) = cov_S(t_1)$ with cost equal to $cost(p_1) = cost(t_1)$ (first cumulative point p_1); the second test case in the ordering t_2 reaches a new cumulative statement coverage $cov_S(p_2) = cov_S(p_1) \cup cov_S(t_2)$ with $cost(p_2) = cost(p_1) + cost(t_2)$ (second cumulative point p_2). In general, $cov_S(p_i) = cov_S(p_{i-1}) \cup cov_S(t_i)$ and $cost(p_i) = cost(p_{i-1}) + cost(t_i)$. Thus, each test case prioritization corresponds to a set of points in the two-objective space denoted by the two testing criteria, i.e., statement coverage and execution cost in our example (see Figure 5.1-(a)). These points are *weakly monotonically increasing* since cumulative cost increases,

while cumulative coverage is stable or increases when adding a new test case from the ordering, i.e., $cov_S(p_i) \subseteq cov_S(p_{i+1})$ and $cost(p_i) \leqslant cost(p_{i+1})$. Note that in Figure 5.1-(a) $|cov_S(p_i)|$ denotes the cardinality of the set $cov_S(p_i)$.

Given this set of points we can measure how quickly the given ordering τ optimizes the two objectives by measuring the proportion of the *area* dominated by the corresponding cumulative points $P(\tau)$, denoted by the gray area in Figure 5.1-(a). The *dominated area* is represented by all points in the objective space that are worse than the cumulative points according to the concept of *dominance* in the multi-objective paradigm in Definition 2.4. Notice that by definition [258], the area dominated by a given point $A = (x_a, y_a)$ within the bi-dimensional objective space $F = \{cost, |cov|\}$ (i.e., cumulative cost and cumulative coverage) is the rectangle (area) delimited by all points in F such that $cost \geq x_a$ and $|cov| \leq y_a$. For example, the area dominated by a cumulative point p_i in Figure 5.1-(a) is the rectangle (area) delimited by $cost \geq cost(p_i)$ and $|cov| \leq |cov_S(p_i)|$. Given a set of non-dominated points $P(\tau)$ within the bi-dimensional objective space $F = \{cost, |cov|\}$, the overall dominated area is given by the union of the area (rectangle) dominated by each single point $p_i \in P(\tau)$ [258].

Two different orderings correspond to two different sets of cumulative points and then two different dominated areas. Therefore, we can compare the corresponding fraction of dominated areas to decide whether one candidate test case ordering is better or not than another one (fitness function): larger dominated areas imply faster statement coverage rate. In this two-objective space the dominated area can easily be computed as the sum of the rectangles of width $[cost(p_{i+1}) - cost(p_i)]$ and height $|cov_S(p_i)|$ as reported in Figure 5.1-(a).

Hypervolume in three-objective TCP. Similarly, if we consider a third testing criterion (such as past faults coverage $|cov_{PF}(p_i)|$) each candidate prioritization corresponds to a set of points in a three-dimensional space and, in this case, the dominated proportion of the objective space is represented by a volume instead of an area, as depicted in Figure 5.1-(b). Since even in this three-objective space the cumulative points are always *weakly monotonically increasing*, the dominated volume can be computed as the sum of the parallelepipeds of width $[cost(p_{i+1}) - cost(p_i)]$, height $|cov_S(p_i)|$ and depth $|cov_{PF}(p_i)|$.

Hypervolume in N-objective TCP. For more than three testing criteria the objective space dominated by a set of cumulative points is called a *hypervolume* and represents a generalization of the *area* for a higher dimensional space.

Without loss of generality, let $T = \{t_1, t_2, t_3, \ldots, t_n\}$ be a test suite of size n and $F = \{cost, Cov_1, \ldots, Cov_m\}$ a set of testing criteria used to prioritize the test cases in T, where *cost* denotes the execution cost of each test case while $Cov_1, \ldots, Cov_m$ are the remaining m testing criteria to maximize. Given a permutation τ of test cases in T we can compute the corresponding set of cumulative points $P(\tau) = \{p_1, \ldots, p_n\}$ obtained by cumulating the scores $cost, Cov_1, \ldots, Cov_m$ achieved by each test case in τ.

Definition 5.1. The *hypervolume* dominated by a permutation $P(\tau)$ of test cases can be computed as follows:

$$I_H(\tau) = \sum_{i=1}^{n-1} \Big[[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)| \Big] \quad (5.1)$$

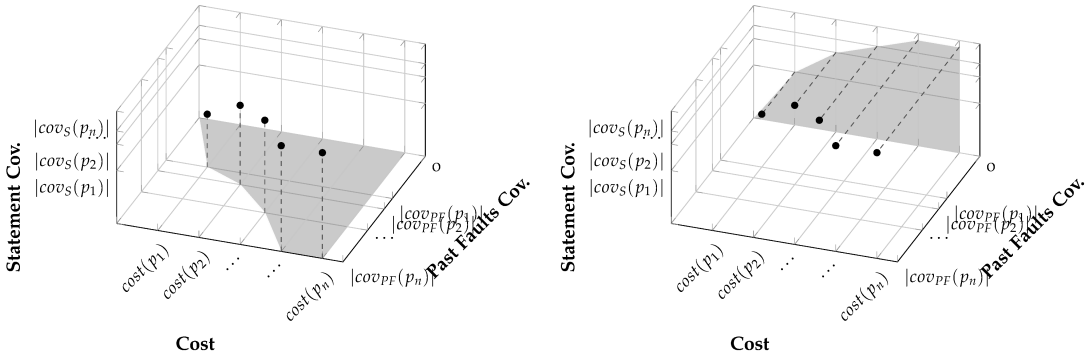
where $[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)|$ measures the hypervolume dominated by a generic cumulative point p_i, but non-dominated by the next point p_{i+1} in the ordering τ. Since in test case prioritization the maximum values of all the testing criteria are known (e.g., the maximum execution cost or the maximum statement coverage are already known), we can express the hypervolume as a fraction of the whole objective space as follows:

Definition 5.2. The fraction of the *hypervolume* dominated by a permutation $P(\tau)$ of test cases is:

$$I_{HP}(\tau) = \frac{\sum_{i=1}^{(n-1)} \Big[[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)| \Big]}{cost(p_n) \times |Cov_1^{max}| \times \ldots |Cov_m^{max}|} \quad (5.2)$$

where $cost(p_n)$ is the execution cost of the whole test suite T and $|Cov_i^{max}|$ denotes the maximum values for the i-th coverage criterion. Such a metric ranges in the interval $[0; 1]$. It is equal to +1 in the ideal case where the test case ordering allows to reach the maximum test criteria scores independently from the execution cost value $cost(p_i)$. A higher $I_{HP}(\tau)$ mirrors a higher ability of the prioritization τ in maximizing the testing criteria with lower cost.

Relation with AUC-based metrics. The $I_{HP}(\tau)$ metric proposed in this chapter can be viewed as a generalization of the AUC-based metrics (e.g., APSC) used in prior work on search-based test case prioritization. For example, the APSC metric measures the average cumulative fraction of statements coverage



(a) Branch Coverage

(b) Statement Coverage

Figure 5.2: Cumulative points in three-objective test case prioritization. The gray areas denote the Area Under Curve for the two projections of the cumulative points $P(\tau)$ onto planes [Cost $\times$ Past Fault Cov.] and [Cost $\times$ Statement Cov.].

as the Area Under Curve delimited by the test case ordering with respect to the cumulative statement coverage scores [51]. In light of the proposed hypervolume metric, APSC can be viewed as a simplified version of $I_{HP}(\tau)$ where all test cases have execution cost equal to one and only the statement coverage is considered as testing criterion. A similar consideration can be done for all the other cumulative fitness functions used in previous work on search-based test case prioritization [51, 78, 79].

An important difference between the AUC-based metrics (e.g., APSC) and $I_{HP}(\tau)$ lies in how they measure the area dominated by a given test case permutation/ordering $P(\tau)$. The AUC-based metrics provide an over-estimation of the area dominated by $P(\tau)$ using the trapezoidal rule [51]. Instead, $I_{HP}(\tau)$ uses the rectangular rule, thus, strictly satisfying the definition of *dominance* in multi- and many-objective optimization (see Definition 2.4).

Finally, despite the AUC metrics being strictly dependent on each other, they are calculated independently in test case prioritization based on multi-objective Genetic Algorithms. Indeed, these values are projections of a manifold of cumulative points (*e.g.*,, a projection of a volume into two areas). For example, let us consider the test suite $T = \{t_1, t_2, \ldots, t_n\}$ with the following statement coverage $Cov_S = \{cov_S(t_1), cov_S(t_2), \ldots, cov_S(t_n)\}$, past faults coverage $Cov_{PF} = \{cov_{PF}(t_1), cov_{PF}(t_2), \ldots, cov_{PF}(t_n)\}$, and execution cost $Cost = \{cost(t_1), cost(t_2), \ldots, cost(t_n)\}$. As depicted in Figure 5.2-(a), if we consider the ordering $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ we can independently measure the average percentages $APSC_c$ and $APPFC_c$, related to statement coverage and past faults coverage. Therefore, despite the AUC metrics being strictly dependent on each other, they are calculated independently.

Algorithm 1: Hypervolume Computation

Input:
Permutation of test cases τ
Execution cost vector *cost*
Testing criteria to maximize $F = \{Cov_1, \ldots, Cov_m\}$
Result: Hypervolume score for τ

```
1  begin
2      I_HP(τ) = 0
3      cumCost = 0, cumCov_1 = ∅, ..., cumCov_m = ∅
4      for each t in τ do
5          prevCost = cumCost
6          cumCost = cumCost + cost(t)
7          for each f_i ∈ F do
8              cumCov_i = cumCov_i ∪ Cov_i(t)
9          I_HP(τ) = I_HP(τ) + (cumCost − prevCost) × |cumCov_1| × ··· × |cumCov_m|
10         // terminating the loop if maximum coverage is reached
11         if ∀Cov_i ∈ F, cumCov_i == Cov_i^max then
12             break
13     // adding the remaining part of the hypervolume
14     I_HP(τ) = I_HP(τ) + (cost^max - cumCost) ×|Cov_1^max| × ··· × |Cov_m^max|
15     // normalizing the hypervolume
16     for each f_i ∈ F do
17         I_HP(τ) = I_HP(τ) / |Cov_i^max|
18     I_HP(τ) = I_H(τ) / cost^max
```

Hypervolume complexity. As pointed out by Auger *et al.* [62], the computation of the hypervolume indicator is usually not a trivial task and it is strongly impacted by the choice of the reference point and the distribution of solutions on the Pareto front. Despite this, it is worth noting that in the case of Test Case Prioritization a candidate test case ordering corresponds to a set of *monotonically* increasing cumulative scores. For this reason, we can use Equation 5.2 to compute the dominated hypervolume instead of the more expensive algorithm proposed by Auger *et al.* [62]. Indeed, the $I_{HP}(\tau)$ metric sums up the slices of dominated hypervolume delimited by two subsequent cumulative points. Thus, let m be the number of the testing criteria and let n be the number of cumulative points (corresponding to the size of the test suite), the $I_{HP}(\tau)$ requires to sum the n hypervolume slices, each one computed as the multiplication of m test criteria scores. Thus, the overall computational time is $O(n \times m)$. Conversely, in traditional many-objective optimization the points delimiting the non-dominated hypervolume are non-monotonically increasing and thus, the computation of the hypervolume metric requires a more complex algorithm which is exponential with respect to the number of objectives m [62], or testing criteria for TCP.

Efficient hypervolume computation. To speed-up the computation of the hypervolume metric, we use Algorithm 1. Given a permutation of test cases

τ, the corresponding execution cost array *cost*, and a set of testing criteria to maximize $Cov_1, \ldots, Cov_m$; the algorithm initializes the cumulative coverage scores (line 3 of Algorithm 1). Such scores are then incrementally updated for each test case in the given order τ (main loop in lines 4-12). In particular, for each test t in τ, the algorithm computes the cumulative cost (line 6) and cumulative coverage scores (lines 7-8), one cumulative coverage score for each testing criterion $Cov_i \in F$. Then, these values are used to compute the actual $I_{HP}(\tau)$ (line 9). If the maximum coverage is reached earlier for all $Cov_i \in F$ (i.e., before iterating over all $t \in \tau$), the loop is terminated (lines 11-12). The remaining portion of the $I_{HP}(\tau)$ metric is added in line 14 of Algorithm 1: it corresponds to the hypervolume of size $(cost^{max} - cumCost) \times |Cov_1^{max}| \times \cdots \times |Cov_m^{max}|$. Finally, $I_{HP}(\tau)$ is normalized in lines 15-18. The core idea of Algorithm 1 is to reduce the number of iterations needed to compute $I_{HP}(\tau)$ given the fact that the remaining portion of the hypervolume is known a priori when the maximum cumulative coverage is reached for all testing criteria in F.

5.2.2 *Hypervolume-based Genetic Algorithm*

In this chapter, we consider the $I_{HP}(\tau)$ metric as a suitable fitness function to guide search algorithms in finding the optimal ordering τ in multi-objective test case prioritization. In particular, we applied Genetic Algorithm (GA) [259], a stochastic search technique based on the mechanism of natural selection and natural genetics. We selected this algorithm because it has been used to solve a wide range of optimization problems that are not solvable in polynomial time. Moreover, with respect to other search algorithms, it is highly parallelizable [260].

GA starts with a random population of solutions. Each individual (i.e., chromosome) represents a solution of the optimization problem. The population evolves through subsequent generations where individuals are evaluated based on a fitness function to be optimized. At each generation, new individuals (i.e., offsprings) are created by applying three operators: (i) a selection operator, based on the fitness function, (ii) a crossover operator, that recombines two individuals from the current generation with a given probability, and (iii) a mutation operator, which modifies the individuals with a given probability.

We propose a new genetic algorithm named `HGA` (Hypervolume-based Genetic Algorithm), depicted in Algorithm 2. Despite, GAs are commonly used for

Algorithm 2: Hypervolume Genetic Algorithm

```
   Input:
   Solution representation: permutation of test cases
   Fitness function: I_HP(τ)
   Result: the best permutation of test cases according to I_HP(τ)
1  begin
2      initialize population with random candidate solutions
3      evaluate each candidate solution
4      while max # of generations has not been reached do
5          select best individuals based on I_HP(τ) using binary tournament selection
6          recombine pairs of individuals using PMX-Crossover
7          mutate individuals using SWAP-Mutation
8          evaluate each candidate solution
```

solving single-objective problems, using the *hypervolume* indicator as fitness function, it is possible to combine multiple objectives in a single one. Each solution is a permutation of integers in which each element represents a test case to be executed and the population is represented by a set of different test case permutations. The selection operator is the *binary tournament selection* (line 5), which randomly picks two individuals for the tournament and selects the one with the better fitness function. The crossover operator is the *PMX-Crossover* (line 6), which swaps the permutation elements at a given random crossover point. The mutation operator is the *SWAP-Mutation* (line 7) that randomly swaps two chosen permutation elements within each offspring. More details on the parameter settings are reported in Sections 5.3.1.4 and 5.4.1.3. The fitness function that drives the GA evolution is the *hypervolume* indicator described in Section 5.2.1. `HGA` can be briefly summarized as (i) generating test cases orderings, (ii) evaluating the permutations using the $I_{HP}(\tau)$ metric, and (iii) using this value to drive the GA evolution.

5.3 EVALUATING COST-EFFECTIVENESS AND EFFICIENCY OF HYPERVOLUME GENETIC ALGORITHM

This section discusses the empirical study we carried out to assess the performances of `HGA`.

5.3.1 *Design of the Empirical Study*

The *goal* of this study is to evaluate the Hypevolume-based Genetic Algorithm, with the *purpose* of improving the test case prioritization problem. The *context*

consists of ten programs from the Software-artifact Infrastructure Repository (SIR) [261] and one large open-source project from a previous study on regression testing [81]. In particular, we selected five programs from the GNU utilities, namely `bash`, `flex`, `grep`, `gzip`, and `sed`. From the Siemens suite, we selected further five programs, i.e., `printtokens`, `printtokens2`, `replace`, `schedule`, and `schedule2`. Finally, we considered `MySQL`, a large real-world system that has been previously studied by Epitropakis *et al.* [81].

The characteristics of these 11 programs are reported in Table 5.1, including their size (in terms of lines of code), test suite size, and type of faults. In total, the selected programs have a size ranging between 374 and 1,283,433 LOC, while the number of test cases varies between 214 and 5,542. We selected these programs since they have been used in previous work on regression testing [51, 81, 77, 262, 263, 264]. Moreover, they have different size, number of tests, and number and type of faults. It is worth noting that seeded faults were introduced by applying mutation testing following the SIR guidelines [261]. Trivial mutants, exposed by a high number of test cases, were removed as considered too easy to find. For the sake of this analysis, we always selected the largest *hard* matrices (i.e., matrices of faults that are killable by few tests) in case of multiple fault matrices available in the SIR repository.

Table 5.1: Programs used in the study.

Program	LOC	# Tests	# Faults	Fault Type	Description
bash	59,846	1,061	5	Seeded	Shell language interpreter
flex	10,459	567	15	Seeded	Fast lexical analyzer
grep	10,068	809	10	Seeded	Regular expression utility
gzip	5,680	214	11	Seeded	Compression tool
printtokens	726	4,130	200	Seeded	Lexical analyzer
printtokens2	570	4,115	200	Seeded	Lexical analyzer
replace	564	5,542	200	Seeded	A tool for pattern matching and substitution
schedule	412	2,650	90	Seeded	Priority scheduler
schedule2	374	2,710	200	Seeded	Priority scheduler
sed	14,427	360	5	Seeded	Non-interactive text editor
MySQL	1,283,433	2,005	20	Real	Relational database management system

5.3.1.1 *Research Questions*

The empirical evaluation is steered by the following research questions:

- **RQ3.1:** *What is the cost-effectiveness of HGA, compared to state-of-the-art test case prioritization techniques?* This research question aims at evaluating to what extent the test case ordering obtained by `HGA` is able to detect faults

(*effectiveness*) earlier (lower execution *cost*) in comparison with two state-of-the-art techniques: a cost cognizant additional greedy algorithm [53, 143] and a multi-objective search based algorithm namely `NSGA-II` [265] used in prior test case prioritization [77, 81]. This reflects the developers' needs to discover regression faults with minimum execution cost.

- **RQ3.2:** *What is the efficiency of HGA, compared to state-of-the-art test case prioritization techniques?* With this second research question, we are interested in comparing the running time (*efficiency*) required by `HGA` to find an optimal test ordering, in comparison with the two experimented state-of-the-art test case prioritization techniques.

5.3.1.2 *Testing Criteria*

To answer our research questions, we considered different testing criteria widely used in previous test case prioritization work [51] [266] [81]:

Statement coverage criterion. For the programs from SIR, we measured statement coverage achieved by each test case using the `gcov` tool part of the GNU C compiler (`gcc`). For `MySQL`, we used the statement coverage matrices provided by Epitropakis *et al.* [81] built using the software profiling tool `Valgrind`.

Execution cost criterion. To compute the execution cost, we could just measure the test case execution time. However, this measure depends on several external factors such as different hardware, application software, operating system, etc. In this chapter, we addressed this issue by counting the number of executed instructions in the production code, instead of measuring the actual execution time. To this aim, we used the `gcov` tool to measure the execution frequency of each source code instruction for the programs from the GNU, and the Siemens Suite. For `MySQL`, we used the cost execution estimations provided by Epitropakis *et al.* [81] and based on `Valgrind`'s measurements. Notice that approximating the execution cost as the number of executed instructions is a standard procedure in the literature [50, 83].

Past faults coverage criterion. For the programs from GNU and the Siemens Suite, we considered the versions of the programs with seeded faults available in the SIR repository [261]. SIR also specifies whether or not each test case is able to reveal these faults. Such information can be used to assign a past faults coverage value to each test case, computed as the number of known past faults that each test is able to reveal in the previous version. For `MySQL`, Epitropakis *et*

al. [81] collected 20 real faults from the `MySQL` bug-tracking system [267]. These faults had "closed" status and were already been fixed by patches.

Δ-coverage criterion. In addition to the aforementioned criteria, the dataset by Epitropakis *et al.* [81] includes for `MySQL` the Δ-coverage criterion. This metric represents the difference of statement coverage between two consecutive versions of a program. The conjecture behind the use of this information is that changed lines of code are more likely to introduce faults in the system. It was calculated applying the `diff` program between two consecutive coverage matrices.

Using the testing criteria described above, we examined three different formulations of the TCP problem:

Two-criteria. The goal is to find an optimal ordering of test cases which (i) minimizes the *execution cost*, and (ii) maximizes the *statement coverage*. We applied this formulation for the ten programs from SIR [261].

Three-criteria. For this formulation, we considered the *past faults coverage* as a third criterion to be maximized. We applied this formulation on the same programs already used for the 2-objective formulation.

Four-criteria. We replicated the study performed by Epitropakis *et al.* [81] on `MySQL` considering *statement coverage*, Δ*-coverage*, *past faults coverage*, and *execution cost*. This experiment allows us to verify how the test case prioritization algorithms perform on a software system of considerable size.

5.3.1.3 *Evaluated Algorithms*

We compared the results of `HGA` instantiated with two, three, and four criteria with the results of (i) the `Additional Greedy` [53, 77, 143] and (ii) the multi-objective genetic algorithm, `NSGA-II` [77, 265].

Additional Greedy. The `Additional Greedy` algorithm instantiated for the TCP problem [53, 77] considers at the same time coverage and cost by maximizing the coverage per unit of time of the selected test cases (cost cognizant additional greedy). Similarly, for what concerns the three-objective formulation of the test case prioritization problem, we used the algorithm proposed by Yoo and Harman [77], which conflates code coverage, execution cost and past coverage in one objective function to be minimized.

Additional greedy is an iterative deterministic search algorithm that starts with an empty order of test cases $\tau_0 = \langle\rangle$; then, it selects the test case t_{max} having

the highest value of code coverage per time unit (greedy step), i.e., $\tau_1 = \langle t_{max} \rangle$. In each of the subsequent iterations, it selects the test case yielding the largest (additional) increment of code coverage per time unit compared to the order τ_i built in the last previous iteration of the algorithm. The loop ends when the highest coverage per time unit is reached, i.e., when adding any un-selected test does not lead to an increment in coverage. To complete the test ordering, the un-prioritized test cases that do not contribute to the additional coverage could be ordered using any strategy (e.g., using a random order). In this work, we recursively re-applied the `Additional Greedy` algorithm to the un-prioritized tests until all are ordered, as done in previous work [51].

When multiple coverage criteria are used (as for the three-objective formulation), the additional coverage per unit time of each test t is computed using the following equation:

$$g(t) = \frac{1}{m} \times \frac{1}{cost(t)} \times \sum_{i=1}^{i=m} f_i(t) \tag{5.3}$$

where $F = \{f_1, \ldots, f_m\}$ is the set of coverage criteria to consider and $cost(t)$ denotes the execution cost of the test t.

NSGA-II. The Non-dominated Sorting Genetic Algorithm II [81] is a computationally fast and elitist multi-objective evolutionary algorithm based on a non-dominated sorting approach. As any population-based evolutionary algorithms, NSGA-II starts with a set of solutions (test case orderings in our case) randomly generated within the solution space. At each generation, *offsprings* are generated by combining pairs of fittest individuals through three genetic operators: *selection*, *crossover* and *mutation*. To form the population for the next generation, parents and offsprings are ordered using the non-dominated sorting algorithm, which assigns to each candidate solution a fitness score that combines the *non-dominance relation* (see Equation 2.6) and the *crowding distance*. The best individuals according to the sorting algorithm are selected to form the new population. The process is repeated until a maximum number of iterations (also called *generations*) is reached.

When applying NSGA-II to the TCP problem [77], the objectives functions to optimize are the AUC-based metrics. Therefore, each coverage criterion is translated to a corresponding AUC-based metric by applying Equation 2.5. For

example, the AUC-based metric to optimize for statement coverage is the cost cognizant variant of Average Percentage of Statements Coverage ($APSC_c$):

$$APSC_c = \frac{\sum_{i=1}^{m}(\sum_{j=TS_i}^{n} t_j - \frac{1}{2}t_{TS_i})}{\sum_{j=1}^{n} t_j \cdot m} \tag{5.4}$$

where $T = \{t_1, t_2, \ldots, t_n\}$ is the test suite to be optimized, with cost $C = \{c_1, c_2, \ldots, c_n\}$, TS_i is the first test case in an ordering T' of T that is able to cover the statement i.

In this chapter, we selected NSGA-II because it has been widely used in literature and for regression testing in particular [81, 77, 263]. Moreover, our choice was guided by the fact that `NSGA-II` has been proven to be particularly suited for prioritization problems [80, 268, 269].

5.3.1.4 *Implementation Details and Parameter Setting*

All the algorithms have been implemented using *JMetal* [270], a Java-based framework for multi-objective optimization with meta-heuristics. To reduce the execution time needed to perform the experiments, we pre-processed the coverage data using the lossless *coverage compaction algorithm* proposed by Epitropakis *et al.* [81]. This technique improves the performance of all the algorithms reducing the size of the coverage matrices by a factor between 7 and 488 [81].

For both NSGA-II and `HGA` we used the same parameters values used in previous chapter on TCP [77, 81]:

- **Population size**: We used a population of 250 individuals for all software projects in our study;

- **Selection**: The selection operator is the *binary tournament selection*, which randomly picks two individuals for the tournament and selects the one with the best fitness function. For `HGA`, the winner of each tournament is the solution with the best $I_{HP(\tau)}$ score (Equation 5.2). Instead, for NSGA-II, the winner of the tournament is the test case with the best *non-dominance rank*, or with the highest *crowding distance* at the same level of *non-dominance rank*.

- **Crossover**: we used the *PMX-Crossover*, which swaps the permutation elements at a given random crossover point. The applied crossover probability is $p_c = 0.90$.
- **Mutation**: As mutation operator, we used the *SWAP-Mutation* that randomly swaps two chosen permutation elements within each offspring.
- **Stopping criterion**: The evolutionary algorithms end when reaching 100 generations, corresponding to 25,000 fitness evaluations. Only for `MySQL`, we ran the algorithm for 200 generations (i.e., 50,000 fitness evaluations) to replicate the experiment made by Epitropakis *et al.* [81].

To account for the inherent random nature of search based algorithms [271], we performed 30 independents runs for each program and for each search algorithm in our study.

5.3.1.5 *Evaluation Metrics*

To address **RQ3.1** we used the *cost-cognizant average fault detection percentage* metric ($APFD_c$) proposed by Elbaum *et al.* [143]. This metric measures the effectiveness of a given test case ordering by summing up the costs of the first test cases that are able to reveal the faults [143]. The higher the $APFD_c$ value, the lower the average cost needed to detect the same number of faults. Since we performed 30 independent runs, we report the mean and the standard deviation of the $APFD_c$ scores achieved for each program under study and for each TCP problem. It is worth noting that for `NSGA-II` we report the mean and the standard deviation of all the solutions on the Pareto set. The *average cost cognizant percentage of faults detected per unit cost* can be computed as follows:

$$APFD_c = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \cdot m} \tag{5.5}$$

where $T = \{t_1, t_2, \ldots, t_n\}$ is the test suite to be optimized, with cost $C = \{c_1, c_2, \ldots, c_n\}$ and TF_i is the first test case in an ordering T' of T that reveals fault i.

We statistically analyzed the obtained results, to check whether the differences between the $APFD_c$ scores produced by the compared algorithms over different independent runs are statistically significant or not. To this aim we used two

different statistical tests: (i) *Welch's t-test*, and (ii) *Wilcoxon t-test* [209]. In particular, we used the *Welch's t-test* to compare `HGA` with `Additional Greedy`, while we applied the *Wilcoxon t-test* to compare `HGA` with `NSGA-II`. In particular, we used *Welch's t-test* to test two groups with unequal variance, while we used *Wilcoxon t-test* in case of equal variance. In both cases, we considered a p-value threshold of 0.05. Significant p-values indicate that the corresponding null hypothesis can be rejected in favor of the alternative ones.

Other than testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistical test [272] to measure the effect size, i.e., the magnitude of the difference between the $APFD_c$ achieved with different algorithms. The effect size is considered `small` for $0.56 \leq d < 0.64$, `medium` for $0.64 \leq d < 0.71$ and large for $d \geq 0.71$ [272].

To address **RQ3.2** we compared the average running time required by each algorithm for each software program used in the empirical study. Moreover, in order to check if results produced by the algorithms were statistically significant, we used *Welch's t-test*, *Wilcoxon's t-test* [209] and the Vargha-Delaney ($\hat{A}_{12}$) statistical test [272] as for $\mathbf{RQ}_1$. The execution time was measured using a machine with Intel Core i7 processor running at 2.40GHz with 12GB RAM.

5.3.2 *Empirical Results*

5.3.3 *RQ3.1: What is the Cost-Effectiveness of HGA, Compared to State-of-the-Art Test Case Prioritization Techniques?*

Table 5.2 reports the $APFD_c$ values obtained by (i) `Additional Greedy`, (ii) `NSGA-II`, and (iii) `HGA` for the three formulations of the test case prioritization problem. In particular, for the `HGA` we show the mean and standard deviation of the $APFD_c$ values achieved over 30 independent runs, while for `NSGA-II` we show the average number of solutions achieved over 30 independent runs as well as the mean and standard deviation of the $APFD_c$ values of all solutions. In the following, we discuss the obtained results for each problem formulation separately.

Two-criteria. In this formulation, we compared the different algorithms on the ten programs from the Software-artifact Infrastructure Repository (SIR) [261]. From the comparison between `HGA` and `Additional Greedy` in Table 5.2, we observe that the former achieves statistically higher $APFD_c$ scores than the latter

Table 5.2: Test case prioritization problem: $APFD_c$ achieved by Additional Greedy, NSGA-II, and HGA in the 2-criteria, 3-criteria formulations and the 4-criteria formulation for *MySQL*. The table reports also Welch's test p-values (*Greedy*) and Wilcoxon test p-values (*NSGA-II*), along with numeric and verbal effect size ($\hat{A}_{12}$) values. $p - values$ that are statistically significant (i.e., $p - value < 0.05$) are reported in bold face. $\hat{A}_{12} > 0.5$ means HGA is better than the state-of-the-art algorithm; $\hat{A}_{12} < 0.5$ means state-of-the-art algorithm is better than HGA; and $\hat{A}_{12} = 0.5$ means they are equal. Column # *Sol.* reports the number of solutions achieved by NSGA-II.

Program	**Add. Greedy**	**NSGA-II**			**HGA**		**HGA ≠ Add. Greedy**			**HGA ≠ NSGA-II**		
		# Sol.	**Mean**	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	$\hat{A}_{12}$	**Effect Size**	**p-value**	$\hat{A}_{12}$	**Effect Size**
Two criteria												
bash	**0.948**	1	0.915	0.040	0.920	0.036	< **0.01**	0.23	Large	0.51	0.55	Negligible
flex	0.453	1	**0.698**	0.001	**0.698**	0.001	< **0.01**	1.00	Large	0.52	0.45	Negligible
grep	0.476	1	0.483	0.011	**0.486**	0.009	< **0.01**	0.93	Large	0.28	0.58	Small
gzip	0.119	88	**0.569**	0.116	0.535	0.108	< **0.01**	1.00	Large	0.50	0.45	Negligible
printtokens	0.937	137	0.944	0.012	**0.947**	0.011	< **0.01**	0.77	Large	0.30	0.58	Small
printtokens2	0.972	228	**0.974**	0.004	0.973	0.003	**0.03**	0.73	Medium	0.85	0.31	Small
replace	0.976	142	**0.984**	0.005	0.982	0.005	< **0.01**	0.83	Large	**0.02**	0.32	Medium
schedule	**0.989**	224	0.959	0.016	0.962	0.013	< **0.01**	0.00	Large	0.46	0.56	Negligible
schedule2	0.830	250	**0.939**	0.017	0.901	0.020	< **0.01**	1.00	Large	< **0.01**	0.08	Large
sed	0.989	2	**0.995**	0.001	**0.995**	0.001	< **0.01**	1.00	Large	**0.02**	0.60	Small
Three criteria												
bash	**0.948**	12	0.916	0.033	0.923	0.033	< **0.01**	0.27	Medium	0.37	0.57	Negligible
flex	0.453	13	0.695	0.004	**0.698**	0.001	< **0.01**	1.00	Large	< **0.01**	0.79	Large
grep	0.476	9	0.484	0.009	**0.487**	0.008	< **0.01**	0.87	Large	**0.05**	0.65	Small
gzip	0.118	50	**0.595**	0.131	0.562	0.081	< **0.01**	1.00	Large	**0.03**	0.34	Small
printtokens	0.922	8	**0.947**	0.009	0.946	0.013	< **0.01**	1.00	Large	0.96	0.50	Negligible
printtokens2	0.968	97	0.973	0.002	**0.974**	0.004	< **0.01**	0.90	Large	0.11	0.62	Small
replace	0.971	4	0.984	0.005	**0.986**	0.005	< **0.01**	1.00	Large	**0.04**	0.65	Small
schedule	**0.981**	58	0.968	0.013	0.963	0.014	< **0.01**	0.10	Large	0.21	0.41	Small
schedule2	0.925	26	**0.940**	0.017	0.919	0.017	**0.04**	0.43	Negligible	< **0.01**	0.17	Large
sed	0.989	14	**0.994**	0.001	0.993	0.001	< **0.01**	1.00	Large	< **0.01**	0.26	Large
Four criteria												
mysql	0.475	52	**0.654**	0.047	0.650	0.046	< **0.01**	1.00	Large	0.65	0.47	Negligible

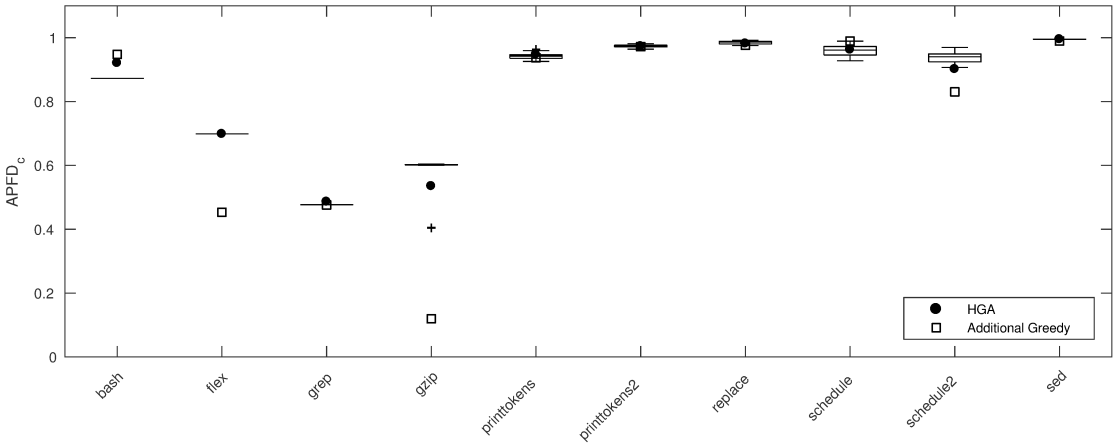


Figure 5.3: APFD*c* scores achieved by Additional Greedy (□), NSGA-II (boxplots), and HGA (•) on the two-criteria formulation of Test Case Prioritization problem.

in 8 out of 10 programs (*i.e.*, $\hat{A}_{12} > 0.5$ and p-value < 0.05). Moreover, according to the $\hat{A}_{12}$ statistics the effect size is `large` in seven cases. The improvements range between a minimum of +1% and a maximum of +42% achieved for `printtokens2` and `gzip`, respectively. On the other hand, `Additional Greedy` produced a significantly higher APFD$_c$ scores in the remaining two cases, *i.e.*, for `bash` and `schedule`. However, we notice that in these cases the difference are quite small, being -2.8% and -2.7% for `bash` and `schedule`, respectively.

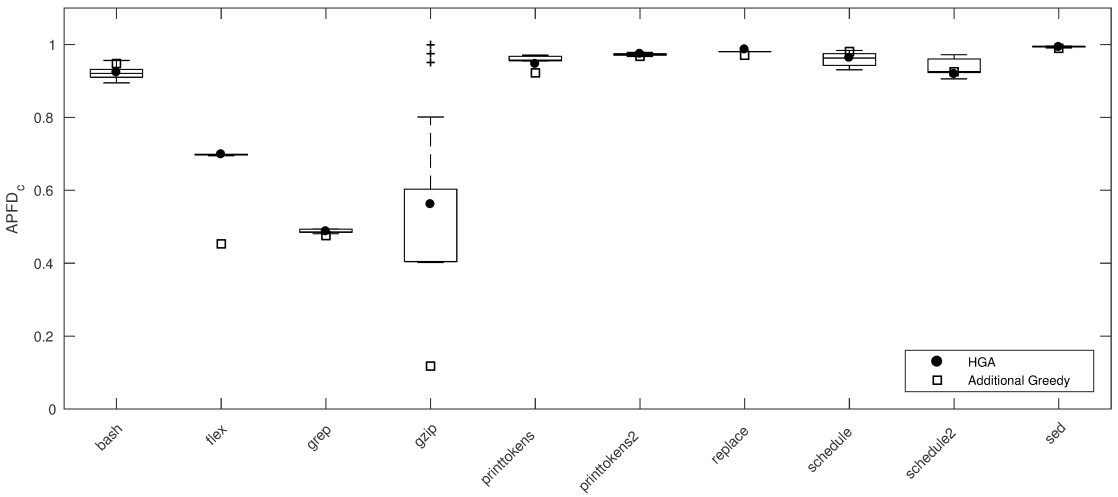


Figure 5.4: APFD*c* scores achieved by Additional Greedy (□), NSGA-II (boxplots), and HGA (•) on the three-criteria formulation of Test Case Prioritization problem.

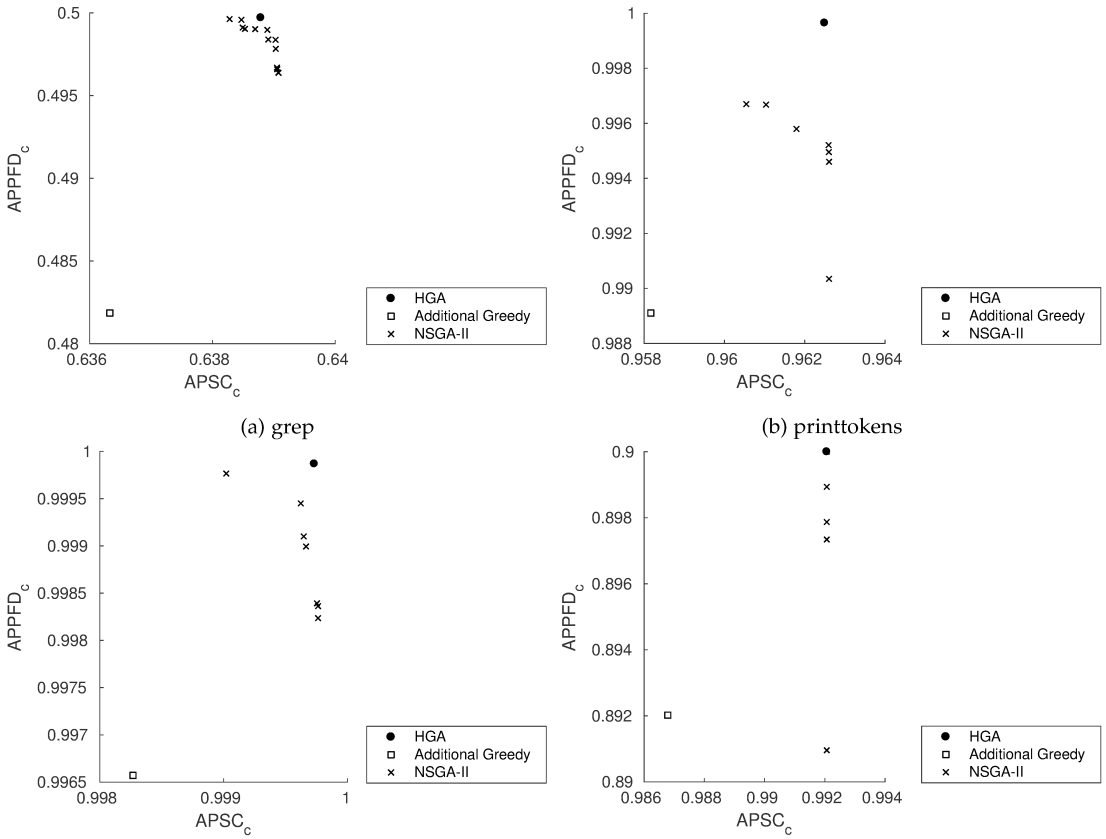
From the comparison between `HGA` with `NSGA-II`, we notice that in only three cases we can reject the null hypothesis for the *Wilcoxon t-test*. Looking at the Vargha-Delaney ($\hat{A}_{12}$) statistics, in two programs `NSGA-II` outperforms `HGA`, in one case with `medium` effect size and in the other with `small` effect size,

while in only one case HGA outperforms NSGA-II with small effect size. For the remaining seven projects, no statistical significant difference is observed among the two evolutionary algorithms. This means that despite using only one fitness function (*i.e.,* the hypervolume indicator), HGA is able to produce test permutations that are competitive with those generated NSGA-II, which instead uses Pareto optimality and multiple objectives.

These results are also confirmed by the boxplots in Figure 5.3, which show the distributions of $APFD_c$ scores achieved by the different algorithms, by considering for NSGA-II and HGA a particular run which is representative of the aggregate scores reported in Table 5.2. We can notice that in most cases there is not a huge difference between the set of solutions in the Pareto front provided by NSGA-II and the single solution by HGA in terms of $APFD_c$ scores. Indeed, in only two cases (*e.g.,* gzip and schedule2) the majority of the Pareto optimal solutions provided by NSGA-II are better than the one produced by HGA. In the remaining cases, the solution provided by HGA is better than at least 50% of the many solutions (up to 250) provided by NSGA-II.

For example, for schedule the single solution generated by HGA is equivalent (in terms of $APFD_c$) to the median solution in the boxplot for NSGA-II. It is worth noting that for this system NSGA-II provides on average 58 non-dominated solutions (*i.e.,* Pareto efficient test permutations) that a software tester could choose for regression testing. Among those 58 solutions, 29 have a better $APFD_c$ scores than the single solution by HGA, while the remaining 29 solutions are worse than those by HGA. We notice that the solutions in the Pareto front that are better than HGA are unknown until all test permutations are executed. To the best of our knowledge, no guideline is available in literature to help testers choosing which solution to pick from the Pareto front (e.g., the ones with the highest likelihood of detecting more faults). Hence, although NSGA-II on average performs similar to HGA, its practical cost-effectiveness strongly depends on which permutation testers select from the Pareto front.

Three-criteria. Results for this formulation are quite similar to those observed for the two-criteria formulation. In particular, HGA outperforms Additional Greedy in 7 out of 10 cases with a large effect size. In the other three cases (*e.g.,* bash, schedule, and schedule2), Additional Greedy produces better test permutations with respect to HGA, but in only one case (schedule) with a large effect size. For bash and schedule2 the effect size is medium and negligible, respectively. The permutations produced by HGA improve the $APFD_c$ up to +44%



(a) grep (b) printtokens

(c) printtokens2 (d) schedule2

Figure 5.5: Pareto frontiers achieved for the three-criteria formulation of Test Case Prioritization.

with respect to `Additional Greedy`, while in the opposite case the difference is low (*e.g.*, -2.8% on bash).

Looking at the results obtained when running `NSGA-II`, we notice that in 6 cases out of 10 we can reject the null hypothesis. In three of those cases, `HGA` outperforms `NSGA-II`: in one case with a `large` effect size and in other two cases with a `small` effect size. Instead, in the remaining three cases, `NSGA-II` is better than `HGA`: in two cases with `large` effect size and in one case with a `small` effect size.

Given its multi-objective nature, `NSGA-II` returns many Pareto efficient solutions (between 4 and 97). However, different solutions in the Pareto fronts provide different $APFD_c$ scores as reported in the boxplots of Figure 5.4, which shows for the three objective formulation the distributions of $APFD_c$ scores achieved by the different algorithms, by considering for `NSGA-II` and `HGA` a particular run which is representative of the aggregate scores reported in Table 5.2. As we can observe, in eight systems the $APFD_c$ score of the single solution provided by `HGA` is higher or equal to the median $APFD_c$ score achieved by

all solutions of NSGA-II. For gzip, we observe a huge variation in the $APFD_c$ distribution yielded by NSGA-II: it ranges between 0.40 and 1.00, with median value of 0.40. For this project, choosing a proper solution from the Pareto front is very critical since not all its solutions have better $APFD_c$ scores than HGA. In particular, only 26% of the Pareto front is more cost-effective than the single solution achieved by HGA. Once again, no guideline exists that helps the testers choosing the most cost-effective solutions in the Pareto front as the $APFD_c$ scores can be computed only by executing all test permutations.

To better understand how the three algorithms optimizes the selected testing criteria, Figure 5.5 plots —for some programs considered in our study, namely grep, printtokens, printtokens2, and sed— the Pareto front produced by NSGA-II, and the single solutions generated by Additional Greedy and HGA with respect to the objectives optimized by NSGA-II ($APSC_c$ and $APPFD_c$). The complete set of plots for all the programs in our study is available in our online appendix [273]. This comparison allows to understand whether the solutions generated by one algorithm dominate (*i.e.*, are better than) the solutions produced by an alternative algorithm in the space of the AUC-metrics. It is possible to notice that in all the cases the solutions generated by NSGA-II and HGA always dominate the solutions produced by Additional Greedy.

From the comparison between NSGA-II and HGA, we observe that the single solution yielded by the latter is never dominated by the Pareto front generated by the former. Vice versa, the single solution by HGA always dominates the large majority of the Pareto fronts produced by NSGA-II. For example, for printtokens the single solution generated by HGA has $APPFD_c = 1$ with $APSC_c = 0.96$. Instead, at the same level of $APSC_c$ the Pareto front by NSGA-II achieves a lower $APPFD_c$ score of 0.99. For schedule2, HGA dominates the whole Pareto front generated by NSGA-II. These results are very unexpected considering that NSGA-II explicitly optimizes $APSC_c$ and $APPFD_c$ as two contrasting objectives. On the other hand, HGA optimizes the hypervolume indicator, which generalizes and combines $APSC_c$ and $APPFD_c$ as discussed in Section 5.2.1. Moreover, this demonstrates that the better results of the solutions produced in some cases by NSGA-II in Figure 5.4 (in terms of ability to discover new faults) are not always directly related to the extent the different objective functions (AUC metrics) are optimized. Considering that there are no guidelines to select the best solution (test case ordering) among a set of solutions produced by multi-objective genetic

algorithms, this result poses a question on whether it is worth at all to use multi-objective algorithms for the test case prioritization problem.

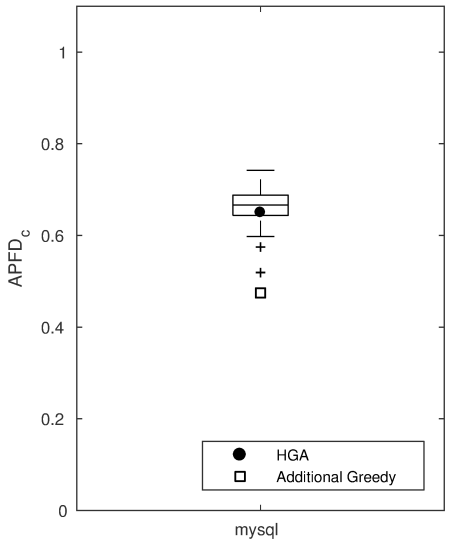


Figure 5.6: APFDc scores achieved by Additional Greedy (□), NSGA-II (boxplots), and HGA (•) for `MySQL`.

Four-criteria. In this formulation, we consider only the large system `MySQL`, thus, replicating the experiment performed by Epitropakis *et al.* [81]. The average and the standard deviation of the APFD$_c$ achieved by the three algorithms in comparison are reported in Table 5.2 (last row), while Figure 5.6 depicts the distributions of the APSC$_c$ scores yielded in a single run representative of the scores in Table 5.2. From Table 5.2, we observe that `HGA` reaches an average APFD$_c$ = 0.65, in comparison to 0.48 of `Additional Greedy`. This difference is statistically significant (p-value$<$ 0.05) and the effect size is `large`. Instead, there is no statistically significant difference between `HGA` and `NSGA-II`. In other words, `HGA` generates solutions (test permutations) that are competitive with the Pareto efficient solutions produced by `NSGA-II` in terms of APFD$_c$.

Since `NSGA-II` produces multiple solutions, Figure 5.6 compares the distributions of the APFD$_c$ scores achieved by all solutions in its Pareto front, with the solutions generated by `HGA` and `Additional Greedy`. As we can observe, the single solution produced by `Additional Greedy` is worse than all solutions generated by `NSGA-II` as well as the one produced by `HGA`. The distribution of APFD$_c$ score by `NSGA-II` presents a large variation, namely the cost-effectiveness ranges between 0.55 and 0.76, with a median value of 0.65.

Finally, Figure 5.7 plots on the same run showed in Figure 5.6, the Pareto front produced by `NSGA-II` and the solutions generated by `Additional Greedy` and `HGA` with respect to APSC$_c$, APDC$_c$ and APPFD$_c$, which are the objectives

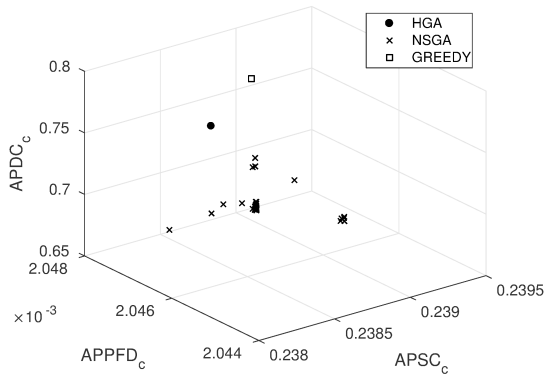


Figure 5.7: Pareto Frontiers on the 4-criteria formulation of Test Case Prioritization for MySQL.

optimized by NSGA-II. We notice that the solution provided by HGA is not dominated by any other solution (generated either by NSGA-II or Additional Greedy). Instead, the single solution generated by HGA dominates the whole Pareto front of NSGA-II. Therefore, thanks to the hypervolume indicator, HGA better optimizes the AUC-metrics, although those metrics ($APSC_c$, $APDC_c$ and $APPFD_c$) are explicitly used by NSGA-II as objective to optimize. Finally, the two solutions achieved by Additional Greedy and HGA do not dominate each other. Also in this case, the better results produced by NSGA-II in Figure 5.6 (in terms of ability to discover new faults) are not always directly related to the extent the different objective functions (AUC metrics) are optimized, thus posing the question of whether it is worth using multi-objective genetic algorithms for the test case prioritization problem.

Summary for RQ3.1. In terms of cost-effectiveness, HGA outperforms Additional Greedy in most of the cases. HGA and NSGA-II produce test permutations that are comparable in terms of $APFD_c$. However, the Pareto front generated in each run by NSGA-II presents a large variance for the $APFD_c$ scores. Thus, its cost-effectiveness strongly depends on how software testers pick a non-dominated solution from the front. Finally, most of the solutions generated by NSGA-II are dominated by HGA with respects to the AUC-based metrics.

Table 5.3: Test case prioritization problem: execution time needed by Additional Greedy, NSGA-II, and HGA in the 2-criteria, 3-criteria formulations and the 4-criteria formulation for *MySQL*. The table reports also Welch's test p-values (*Greedy*) and Wilcoxon test p-values (*NSGA-II*), along with numeric and verbal effect size ($\hat{A}_{12}$) values. $p-values$ that are statistically significant (i.e., $p-value < 0.05$) are reported in bold face. $\hat{A}_{12} > 0.5$ means HGA is slower than the state-of-the-art algorithm; $\hat{A}_{12} < 0.5$ means HGA is faster than state-of-the-art algorithm; and $\hat{A}_{12} = 0.5$ means they are equal.

Program	**Add. Greedy**	**NSGA-II**		**HGA**		**HGA ≠ Add. Greedy**			**HGA ≠ NSGA-II**		
		Mean	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	$\hat{A}_{12}$	**Magnitude**	**p-value**	$\hat{A}_{12}$	**Magnitude**
2 criteria											
bash	**2s**	25s	1s	17s	2s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
flex	**1s**	10s	<1s	5s	1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
grep	**1s**	16s	1s	5s	<1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
gzip	**<1s**	1s	<1s	**<1s**	<1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
printtokens	49s	10s	1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens2	49s	13s	1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
replace	1min 23s	24s	1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule	33s	5s	<1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule2	28s	5s	<1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
sed	**<1s**	3s	<1s	1s	<1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
3 criteria											
bash	**2s**	21s	2s	15s	1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
flex	**<1s**	9s	<1s	4s	<1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
grep	**1s**	14s	1s	6s	1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
gzip	**<1s**	1s	<1s	**<1s**	<1s	**<0.01**	1.00	Large	**<0.01**	0.02	Large
printtokens	46s	12s	1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens2	43s	14s	1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
replace	1min 23s	35s	4s	**3s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule	28s	6s	**<1s**	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule2	28s	6s	1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
sed	**<1s**	3s	<1s	1s	<1s	**<0.01**	1.00	Large	**<0.01**	0.00	Large
4 criteria											
mysql	**9s**	2min 37s	4s	2min 21s	5s	**<0.01**	1.00	Large	**<0.01**	0.00	Large

5.3.4 *RQ3.2: What is the Efficiency of HGA, Compared to State-of-the-Art Test Case Prioritization Techniques?*

We compared `Additional Greedy`, `NSGA-II`, and `HGA` on the ten programs from the Software-artifact Infrastructure Repository (SIR) [261] and on `MySQL`. Table 5.3 reports the mean execution time required by each algorithm for each software program used in the empirical study along with standard deviations, and the results of the statistical tests.

The comparison between `HGA` and `Additional Greedy`, shows that on 5 out of 10 programs from the SIR repository and on `MySQL`, the `Additional Greedy` algorithm is statistically faster (*i.e.*, $\hat{A}_{12} > 0.5$ and p-value < 0.05) than `HGA`, despite being less cost-effective as demonstrated in $\mathbf{RQ}_1$. In all these cases, according to the $\hat{A}_{12}$ statistics the effect size is `large`. The improvements range between a minimum of 2 times and a maximum of 8.5 times achieved for `gzip` and `bash`, respectively. On the other hand, `Additional Greedy` needs a significant higher execution time on the remaining programs. In these cases, the effects size is `large` and the improvements range between 28 and 49 times (respectively on `schedule2` and `printtokens`). It is worth noting that the performances of `Additional Greedy` are strongly influenced by the number of test cases. Indeed, in the five programs on which `Additional Greedy` performs better, the number of test cases ranges between 214 for `gzip` and 1,061 for `bash`. As the number of test cases increases (*e.g.*, on the remaining projects the number of test cases ranges between 2,650 and 5,542) the greedy algorithm is not able to scale up. Instead, `HGA` is significantly more efficient for very large test suites.

To verify whether the (positive and negative) differences between the execution time of the two algorithms significantly interact with the test suite size, we applied the *permutation test* [274]. It corresponds to a non-parametric version of the Analysis of Variance (ANOVA) test and, thus, it does not require that the distributions under analysis are normally distributed. For the test, we used the implementation available in `R`, and its package `lmPerm` in particular, with a large number of iterations (10^8) to have stable results [84]. The permutation test revealed that there is a statistically significant interaction between execution time of the two algorithms and the number of the test cases to prioritize (p-value=4.14×10^{-4}). In other words, the larger the test suite, the larger is the improvement obtained by `HGA` over `Additional Greedy` in terms of execution time.

From the comparison between HGA and NSGA-II, we can notice that for all the programs and problem formulations we can reject the null hypothesis for the *Wilcoxon t-test*. Looking to the Vargha-Delaney ($\hat{A}_{12}$) statistics, in all programs HGA outperforms (is more efficient than) NSGA-II with large effect size. Indeed, NSGA-II requires between 1.11 (*e.g.*, MySQL) to 14 times (*e.g.*, the Three-criteria formulation on printtokens2) the execution times required for HGA. On average HGA is 5 times faster than NSGA-II. As we already noticed in the comparison with Additional Greedy, the number of test cases strongly influences the performance of NSGA-II. Indeed, as the number of test cases increases the ratio between the time required by NSGA-II and HGA increases. These observations are also confirmed by the permutation test: the differences (improvements/worsening) between the execution time of HGA and NSGA-II significantly interacts with the test suite size (p-value=2.90×10^{-5}) Furthermore, we notice that adding another test criteria to the problem formulation, affects much more negatively the performance achieved by NSGA-II than the one achieved by HGA. For example, on the project replace NSGA-II requires 24 seconds when considering two testing criteria; while its running time for the same project but with three criteria increases to 35 seconds. Instead, the running time of HGA increases of only one second when considering three testing criteria instead of two.

Summary for RQ3.2. The efficiency of Additional Greedy is strictly related to the number of test cases. Indeed, when the number of test cases is high, HGA is much faster than Additional Greedy. Looking at NSGA-II, HGA improves the efficiency of test case prioritization in all the considered programs up to 14 times.

5.4 EVALUATING THE SCALABILITY OF HYPERVOLUME GENETIC ALGORITHM ON MANY OBJECTIVES FORMULATIONS

In Study I, we assessed both the cost-effectiveness and the efficiency of HGA when optimizing up to four testing criteria. Our results showed that HGA is more effective and/or more efficient than state-of-the art techniques previously used in the literature when solving the multi-objective TCP problem. However, we still need to assess the scalability of HGA when considering a larger number

of testing criteria, i.e., when moving from the multi-objective formulation to the many-objective formulation of TCP (up to five criteria).

Therefore, in this section we compare HGA with two well-established many-objective evolutionary algorithms in the context of the many-objective formulation of the TCP problem.

5.4.1 *Design of the Empirical Study*

The *goal* of this study is to evaluate the scalability of the Hypevolume-based Genetic Algorithm, with the *purpose* of improving the test case prioritization problem when the number of objectives to consider grows up. The study has been conducted in the *context* presented in Section **??**, namely ten programs from the Software-artifact Infrastructure Repository (SIR) [261]. More details are provided in Section **??**.

The empirical evaluation is steered by the following research questions:

- **RQ3.3:** *What is the cost-effectiveness of HGA, compared to many-objective test case prioritization techniques?* This research question aims at evaluating to what extent the test cases ordering obtained by HGA is able to detect faults (*effectiveness*) earlier (lower execution *cost*) in comparison to two state-of-the-art many-objective algorithms, namely GDE3 and MOEA/D-DE. This reflects the developers' needs to discover regression faults with minimum execution cost.

- **RQ3.4:** *What is the efficiency of HGA, compared to many-objective test case prioritization techniques?* With this second research question, we are interested in comparing the running time (*efficiency*) required by HGA to find an optimal test ordering compared to the alternative many-objective algorithms.

5.4.1.1 *Testing Criteria*

For this case study, we selected two further testing criteria in addition to those already used in Study I (Section **??**), i.e., **Branch and Function coverage.** Similarly to statement coverage, we use the tool gcov to determine the branches covered and the functions called by each test case.

Using all available test criteria, we examined the following many-objective formulations of the TCP problem:

Four-criteria. The goal is to find an optimal ordering of test cases which (i) minimizes the execution cost, (ii) maximizes the statement coverage, (iii) maximizes the past faults coverage, and (iv) maximizes the branch coverage. We applied this formulation for the ten programs from SIR [261].

Five-criteria. For this formulation, we considered the *function coverage* as a fifth criterion to be maximized. We applied this formulation on the same programs already used for the four-objective formulation.

Notice that the goal of our analysis is not to determine which are the coverage criteria that have the higher likelihood of revealing regression faults. Therefore, we selected those that have been widely-used in prior studies (*e.g.*, [51, 81, 83, 266]). Nevertheless, it is possible to formulate other criteria by just providing a clear mapping between tests and coverage-based requirements. The criteria used in this study serve to illustrate how the Hypervolume-based metric can be applied to any number and kind of testing criteria to be satisfied, where further criteria just represent additional axes to be considered when computing the fitness function $I_{HP}(\tau)$.

5.4.1.2 *Evaluated Algorithms*

We compared HGA with two many-objective search algorithms, namely:

GDE3. GDE3 [256] is an extension of the Differential Evolution (DE) algorithm for global optimization that supports an arbitrary number of objectives and constraints. In the context of many-objective optimization, GDE3 improves the earlier GDE versions by generating better distributed solutions. It has been tested on different types of problems showing an improved diversity of the solutions with respect to NSGA-II [265]. Furthermore, according to previous studies [256], this algorithm needs a lower number of function evaluations to converge.

MOEA/D-DE. In general, MOEA/D [257] attempts to optimize at the same time each single objective optimization sub-problem, instead of solving the whole problem directly. With the aim of making the search effective and efficient, MOEA/D exploits the neighborhood relationship among sub-problems for making its search effectively and efficiently. It has been designed for solving many-objective problems with complicated Pareto sets, such as TCP. MOEA/D-DE uses a differential evolution (DE) operator for producing new solutions. Moreover, it uses two extra parameters (*e.g.*, the crossover control parameter CR and the mutation factor F) to maintain the population diversity, which is needed for

exploring the search space effectively, particularly at the early stage of the search.

When applying GDE3 and MOEA/D-DE to the TCP problem [77], the objective functions to optimize are the AUC-based metrics, one for each coverage criterion to optimize (see Section **??**).

5.4.1.3 *Implementation Details and Parameter Setting*

All the algorithms have been implemented using *JMetal* [270], a Java-based framework for multi-objective optimization with metaheuristics. As in the previous case study, we pre-processed the coverage data using the lossless coverage compaction algorithm proposed by Epitropakis *et al.* [81].

For all the considered algorithms, we used the same parameters values used in previous papers on TCP [81, 77] and already applied in Study I:

- **Population size**: We used a population of 250 individuals for all software projects in our study;
- **Selection**: As selection operator HGA uses the *binary tournament selection*, the winner of each tournament is the solution with the best $I_{HP(\tau)}$ score (Equation 5.2). In GDE3 and MOEA-D/DE the fittest individuals are selected using the *differential evolution selection operator*.
- **Crossover**: we used the *PMX-Crossover*, which swaps the permutation elements at a given random crossover point. For HGA, the applied crossover probability is $p_c = 0.90$, while for GDE3 and MOEA-D/DE we applied the default value (*e.g.*, 1.00). GDE3 and MOEA-D/DE need also CR. This parameter indicates how single sub-problems are separable (*i.e.*, the lower the value, the more the problems are separable). We applied the default values (*e.g.*, 0.50 for GDE3 and 1.00 MOEA-D/DE).
- **Mutation**: As mutation operator, we used the *SWAP-Mutation* that randomly swaps two chosen permutation elements within each offspring. GDE3 and MOEA-D/DE need an additional parameter F. This scaling factor controls the speed and robustness of the search (*i.e.*, with a lower value the algorithm converges faster, but it has a higher risk of stacking in a local optimum). Also in this case, we applied the default value (*e.g.*, 0.50).
- **Stopping criterion**: The evolutionary algorithms end when reaching 100 generations, corresponding to 25,000 fitness evaluations.

To account for the inherent random nature of search based algorithms [271], we performed 30 independents runs for each program and for each search algorithm in our study.

5.4.1.4 *Evaluation Metrics*

To address **RQ3.3** we used the *cost-cognizant average fault detection percentage* metric ($APFD_c$) proposed by Elbaum *et al.* [143] and presented in Section **??**. Since we performed 30 independent runs, we report the mean and the standard deviation of the $APFD_c$ scores achieved for each program under study and for each TCP problem. It is worth noting that for `GDE3` and `MOEA/D-DE` we report the mean and the standard deviation of all the solutions on the Pareto set.

We statistically analyzed the obtained results, to check whether the differences between the $APFD_c$ scores produced by the compared algorithms over different independent runs are statistically significant or not. To this aim we used the *Wilcoxon t-test* [209] with a p-value threshold of 0.05 for both the TCP problems. Significant p-values indicate that the corresponding null hypothesis can be rejected in favor of the alternative ones. Other than testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistical test [272] to measure the effect size, i.e., the magnitude of the difference between the $APFD_c$ achieved by different algorithms.

To address **RQ3.4** we compared the average running time required by each algorithm for each software program used in the empirical study. Moreover, in order to check if results produced by the algorithms are statistically significant, we used *Wilcoxon t-test* [209] and the Vargha-Delaney ($\hat{A}_{12}$) statistical test [272] as for $\mathbf{RQ}_1$. The experiment was conducted in the same environment as the one presented in Section **??**, that is a machine with Intel Core i7 processor running at 2.40GHz with 12GB RAM.

5.4.2 *Empirical Results*

5.4.3 *RQ3.3: What is the Cost-Effectiveness of HGA, Compared to Many-Objective Test Case Prioritization Techniques?*

Table 5.4 reports the mean and the standard deviation of the $APFD_c$ scores obtained by (i) `GDE3`, (ii) `MOEA/D-DE`, and (iii) `HGA` on the four- and five-criteria

Table 5.4: Test case prioritization problem: $APFD_c$ achieved by GDE3, MOEA/D-DE, and HGA in the 4-criteria and 5-criteria formulations. The table reports also Wilcoxon test p-values, along with numeric and verbal effect size ($\hat{A}_{12}$) values. $p - values$ that are statistically significant (i.e., $p - value < 0.05$) are reported in bold face. $\hat{A}_{12} > 0.5$ means HGA is better than the state-of-the-art algorithm; $\hat{A}_{12} < 0.5$ means state-of-the-art algorithm is better than HGA; and $\hat{A}_{12} = 0.5$ means they are equal. Column # *Sol.* reports the number of solutions achieved by GDE3 and MOEA/D.

Program	**GDE3**			**MOEA/D-DE**			**HGA**		**HGA > GDE3**			**HGA > MOEA/D-DE**		
	# Sol.	**Mean**	**St. Dev.**	**# Sol.**	**Mean**	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	$\hat{A}_{12}$	**Effect Size**	**p-value**	$\hat{A}_{12}$	**Effect Size**
4 criteria														
bash	22	0.870	0.021	250	**0.916**	0.030	0.903	0.045	< **0.01**	0.73	Medium	0.30	0.42	Small
flex	38	0.684	0.005	250	**0.698**	0.001	**0.698**	0.001	< **0.01**	1.00	Large	0.83	0.48	Negligible
grep	25	**0.485**	0.003	250	**0.485**	0.009	0.484	0.011	0.76	0.52	Negligible	0.87	0.49	Negligible
gzip	20	**0.602**	0.074	250	0.530	0.097	0.595	0.122	**0.02**	0.32	Medium	0.32	0.58	Small
printtokens	67	0.945	0.003	250	0.950	0.008	**0.952**	0.011	< **0.01**	0.71	Medium	0.31	0.58	Small
printtokens2	56	**0.974**	0.001	250	**0.974**	0.003	**0.974**	0.004	0.81	0.48	Negligible	0.92	0.49	Negligible
replace	51	0.983	0.002	250	**0.986**	0.005	0.984	0.004	0.91	0.51	Negligible	0.14	0.39	Small
schedule	91	0.962	0.003	250	0.967	0.015	**0.970**	0.016	< **0.01**	0.69	Medium	0.63	0.54	Negligible
schedule2	12	**0.949**	0.007	250	0.946	0.021	0.926	0.020	< **0.01**	0.16	Large	< **0.01**	0.24	Large
sed	32	0.981	0.004	250	0.992	0.001	**0.993**	0.001	< **0.01**	1.00	Large	< **0.01**	0.84	Large
5 criteria														
bash	61	0.849	0.017	250	**0.911**	0.044	0.897	0.045	< **0.01**	0.83	Large	0.22	0.41	Small
flex	73	0.684	0.005	250	0.697	0.002	**0.698**	0.001	< **0.01**	1.00	Large	0.06	0.64	Small
grep	98	**0.484**	0.003	250	0.482	0.006	**0.484**	0.010	0.21	0.59	Small	0.11	0.62	Small
gzip	144	0.532	0.084	250	0.591	0.118	**0.601**	0.147	**0.01**	0.69	Medium	0.84	0.52	Negligible
printtokens	218	0.945	0.003	250	**0.950**	0.011	**0.950**	0.011	**0.01**	0.69	Medium	0.89	0.51	Negligible
printtokens2	190	0.973	0.001	250	**0.974**	0.003	**0.974**	0.004	0.23	0.59	Small	0.87	0.51	Negligible
replace	153	0.983	0.001	250	**0.986**	0.005	**0.986**	0.004	< **0.01**	0.73	Medium	0.88	0.49	Negligible
schedule	195	0.961	0.003	250	**0.966**	0.013	0.965	0.013	**0.05**	0.65	Small	0.60	0.46	Negligible
schedule2	12	0.949	0.006	250	**0.951**	0.017	0.930	0.022	< **0.01**	0.21	Large	< **0.01**	0.23	Large
sed	114	0.981	0.004	250	0.991	0.001	**0.992**	0.002	< **0.01**	1.00	Large	0.13	0.62	Small

formulation of the test case prioritization problem. For GDE3 and MOEA/D-DE the table reports also the average number of solutions achieved over the different runs. In the next paragraphs, we discuss the achieved results for each formulation separately.

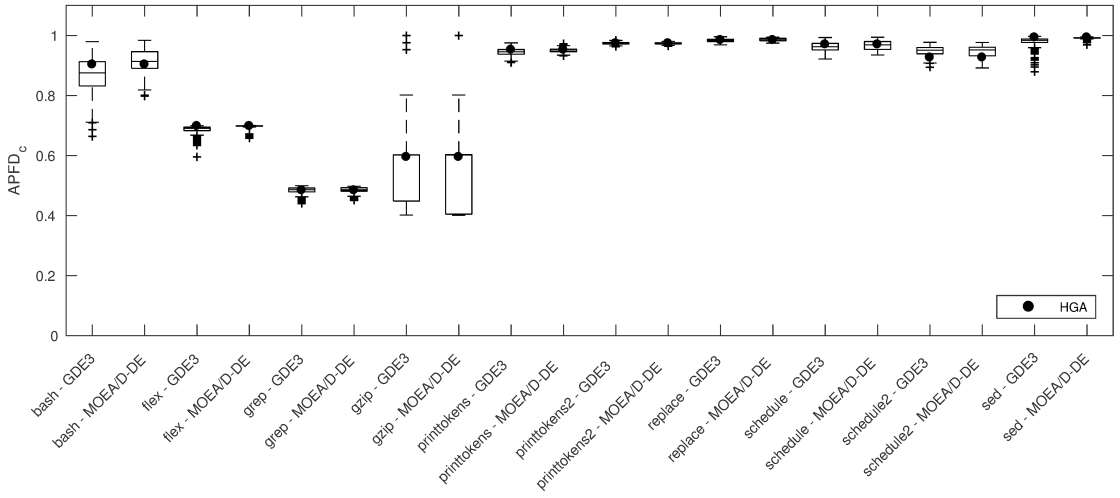


Figure 5.8: Boxplots of the APFDc achieved by GDE3, MOEA-D/DE, and HGA on the 4 criteria formulation of Test Case Prioritization problem.

4-criteria. In general, we observe that HGA achieves equal or better APFD$_c$ values with respect to GDE3 and MOEA-D/DE. In particular, the *Wilcoxon t-test* revealed that in 7 out of 10 projects the differences in terms of APFD$_c$ values between HGA and GDE3 are statistically significant. In five projects, HGA achieves better APFD$_c$ scores than GDE3 (two cases with large effect size and three cases with medium effect size). In the remaining two projects, GDE3 is better than HGA with a large effect for schedule2 and a medium effect size for gzip. Notice that GDE3 returns multiple Pareto efficient solutions (test permutations), whose number ranges between 12 (for schedule2) and 91 (for schedule). Figure 5.8 shows the distribution of the APFD$_c$ scores on a representative run: the APFD$_c$ scores achieved by GDE3 may vary when considering different solutions in the Pareto fronts. In nine projects, the APFD$_c$ score of the single solution generated by HGA is higher or equal to the median score yielded by all Pareto optimal solutions of GDE3. For bash, the APFD$_c$ distribution yielded by GDE3 presents a large variation since it ranges between 0.65 and 0.98, with a median value of 0.85. For the same project, the single solution by HGA outperforms the 70% of the Pareto optimal solutions obtained with GDE3.

From the comparison between HGA and MOEA-D/DE, we notice that in most cases the solution provided by the two algorithms are statistically comparable.

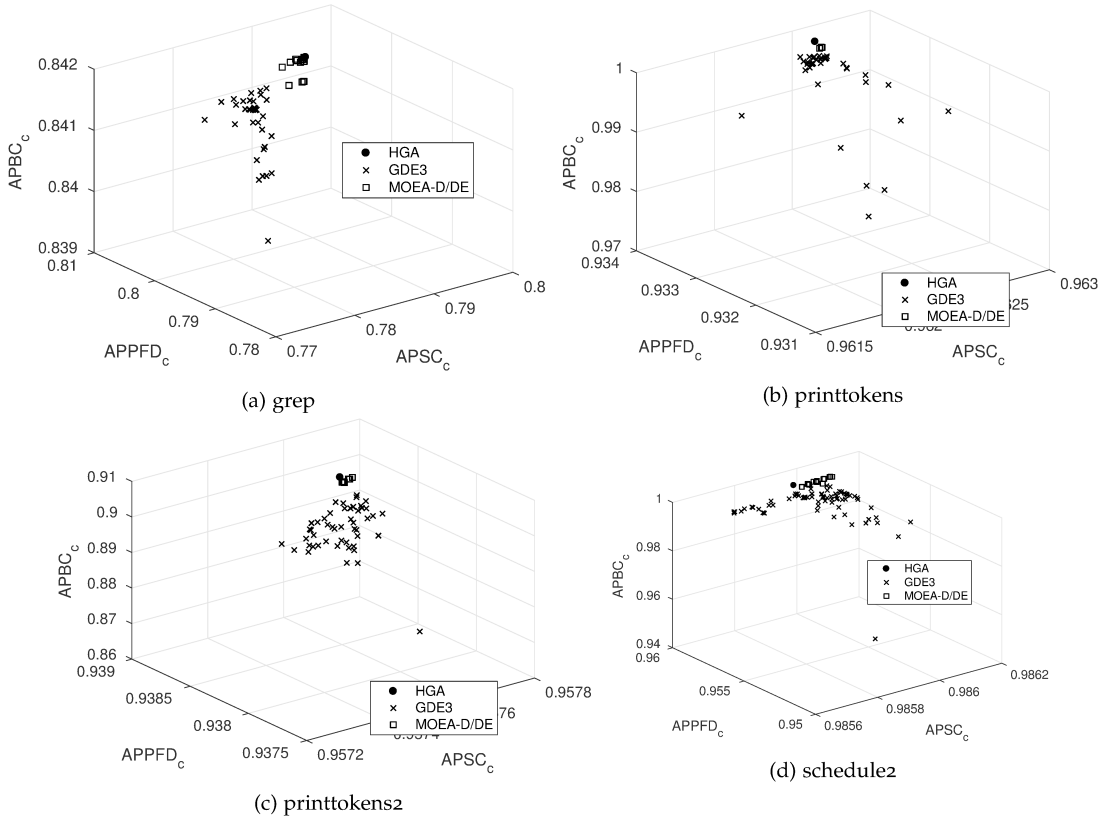


Figure 5.9: Pareto frontiers on the 4-criteria formulation of Test Case Prioritization.

Indeed in 8 out of 10 cases the null hypothesis cannot be rejected according to the *Wilcoxon test*. In the remaining projects, in one case (i.e., `schedule2`) `HGA` is better than `MOEA-D/DE` with a `large` effect size while in another one (i.e., `sed`) `MOEA-D/DE` is better than `HGA` with a `large` effect size. However, not all Pareto efficient solutions yielded by `MOEA-D/DE` achieve the same $APFD_c$ scores as shown in Figure 5.8. One exemplary case is observable for `gzip`: for this project, the $APFD_c$ scores of the Pareto optimal solutions by `MOEA-D/DE` range between 0.40 and 1.00, with mean value of 0.50. Instead, the single solution by `HGA` is better than 75% of Pareto optimal solutions generated by `MOEA-D/DE`.

Figure 5.9 plots (for the same run considered in Figure 5.8 and for some of the programs) the Pareto fronts produced by `GDE3` and `MOEA-D/DE` as well as the single solutions generated by `HGA` with respect to the objectives optimized by the many-objective algorithms (i.e., the AUC-based metrics). As we can observe, the fronts provided by `GDE3` and `MOEA-D/DE` do not dominate the solution generated by `HGA`. Vice versa, the solution generated by `HGA` dominates a large portion of the Pareto fronts produced by the two many-objective algorithms. These results demonstrate the better ability of `HGA` to optimize the AUC-based metrics (thanks

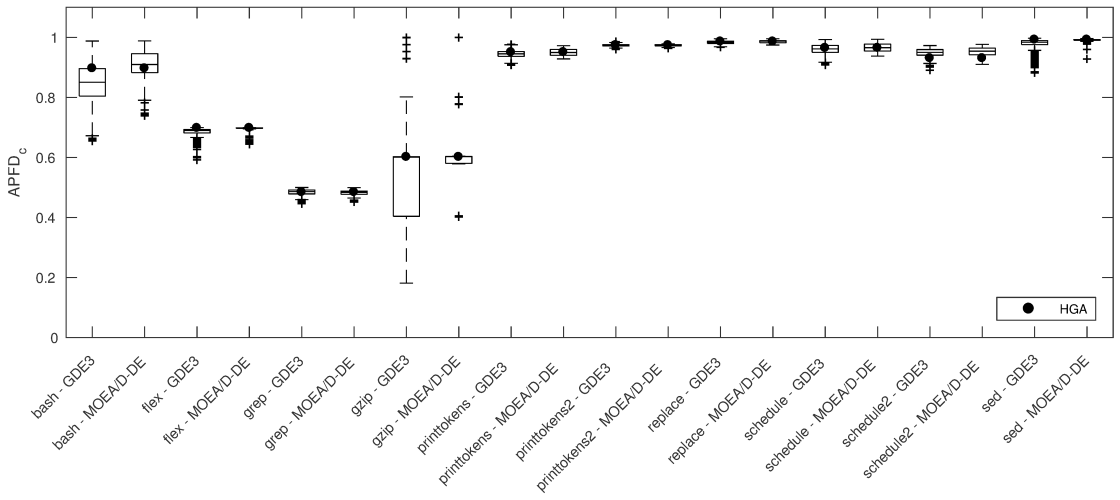


Figure 5.10: Boxplots of the APFDc achieved by GDE3, MOEA-D/DE, and HGA on the 5 criteria formulation of Test Case Prioritization problem.

to the hypervolume indicator) even if the alternative algorithms (i.e., GDE3 and MOEA-D/DE) explicitly use such metrics as objectives to optimize. As already discussed in Section **??**, the better results achieved by some of the solutions produced by the many-objective algorithms (in terms of ability to discover new faults) are not always directly related to the extent the different objective functions (AUC metrics) are optimized, thus posing the question of whether it is worth using multi-objective or many-objective algorithms for the test case prioritization problem, given the difficulty to discriminate the best solution among the different solutions produced by these algorithms. The complete set of plots of all the programs is available in our online appendix [273].

5-criteria. For this formulation, we observe that the achieved results are very similar to those achieved in the previous formulation, despite the usage of one additional testing criterion. Concerning the comparison between HGA and GDE3, we notice that for 8 out of 10 programs the former algorithm achieves statistically higher APFD$_c$ scores than the latter as indicated by the *Wilcoxon t-test*. In 7 out 10 projects, HGA performs better than GDE3: three times with large effect size, three times with medium effect size, and once with small effect size. In the remaining case, GDE3 achieves a significantly higher average APFD$_c$ than HGA with a large effect size. Looking at the results for MOEA-D/DE, we observe that in 9 out of 10 cases we cannot reject the null hypothesis and, thus, the results achieved by HGA and MOEA-D/DE are comparable in terms of APFD$_c$. In the remaining project (i.e., schedule2), MOEA/D-DE is better with a large effect size.

It is important to highlight that for both GDE3 and MOEA-D/DE, Table 5.4 reports the mean $APFD_c$ achieved from all Pareto optimal solutions and across all the independent runs. However, the table does not describe the distributions of the $APFD_c$ scores achieved when considering different solutions in the Pareto fronts. To this aim, Figure 5.10 compares the distributions (boxplots) of the $APFD_c$ scores for GDE3 and MOEA-D/DE with the single solution generated by HGA. As we can notice, the solution by HGA achieves an $APFD_c$ higher than the median scores achieved by the Pareto optimal solutions produced with GDE3 and MOEA-D/DE. In terms of $APFD_c$, the performance of the two many-objective algorithms presents a large variation for various projects in our study. For gzip, the $APFD_c$ scores for GDE3 varies from 0.20 and 1.00 while for MOEA-D/DE they vary between 0.40 and 1.00. With such huge variation, choosing a proper solution from the Pareto fronts is very critical since not all the solutions have better $APFD_c$ scores than HGA. Indeed, for gzip only 4 out of 144 solutions in the Pareto front generated by MOEA-D/DE are more cost-effective than the single solution achieved by HGA. However, again there is no guideline that helps choosing the most cost-effective solutions in the Pareto front as the fault detection capability can be computed only after the execution of all test permutations.

Summary for RQ3.3. In terms of $APFD_c$, HGA outperforms GDE3 and it is competitive with respect to MOEA-D/DE in most cases. The single solution provided by HGA is better than many of the Pareto optimal solutions generated by the many-objective algorithms, providing better compromise in the objectives space.

5.4.4 *RQ3.4: What is the Efficiency of HGA, Compared to Many-Objective Test Case Prioritization Techniques?*

To answer **RQ3.4**, we compared GDE3, MOEA-D/DE, and HGA on the ten programs from the Software-artifact Infrastructure Repository (SIR) [261]. Table 5.5 reports the mean execution time required by each algorithm for each software program used in the empirical study along with standard deviations, and the results of the statistical tests.

The comparison between HGA and GDE3 shows that, in all the programs under study, the former algorithm is statistically faster (*i.e.*, $\hat{A}_{12} < 0.5$ and

Table 5.5: Test case prioritization problem: execution time needed by GDE3, MOEA/D-DE, and HGA in the 4-criteria, and 5-criteria formulations. The table reports also Wilcoxon t-test p-values and numeric and verbal effect size ($\hat{A}_{12}$) values of the hypothesis HGA $<$ GDE3, HGA $<$ MOEA/D-DE. $p-values$ that are statistically significant (i.e., $p-value < 0.05$) are reported in bold face. $\hat{A}_{12} > 0.5$ means HGA is slower than the state-of-the-art algorithm; $\hat{A}_{12} < 0.5$ means HGA is faster than the state-of-the-art algorithm; $\hat{A}_{12} = 0.5$ means they are equal.

Program	**GDE3**		**MOEA/D-DE**		**HGA**		**HGA < GDE3**			**HGA < MOEA/D-DE**		
	Mean	**St. Dev.**	**Mean**	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	$\hat{A}_{12}$	**Effect Size**	**p-value**	$\hat{A}_{12}$	**Effect Size**
4 criteria												
bash	1min 25s	6s	1min 7s	4s	**51s**	5s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
flex	16s	1s	13s	1s	**7s**	1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
grep	37s	2s	32s	1s	**19s**	1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
gzip	1s	<1s	1s	<1s	**<1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens	25s	2s	26s	6s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens2	39s	6s	35s	6s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
replace	57s	<1s	54s	1s	**4s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule	13s	<1s	11s	<1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule2	12s	<1s	11s	<1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
sed	6s	<1s	6s	<1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
5 criteria												
bash	1min 7s	2s	1min 6s	2s	**45s**	1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
flex	13s	<1s	12s	<1s	**6s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
grep	52s	5s	37s	3s	**26s**	3s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
gzip	1s	<1s	1s	<1s	**<1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens	24s	1s	23s	<1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
printtokens2	28s	<1s	26s	<1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
replace	1min 3s	1s	1min	1s	**4s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule	15	<1s	13	<1s	**1s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
schedule2	14s	<1s	13s	<1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large
sed	7s	<1s	6s	<1s	**2s**	<1s	**<0.01**	0.00	Large	**<0.01**	0.00	Large

p-value < 0.05) than the latter with `large` effect size. The improvements range between a minimum of 1.65 times and a maximum of 27 times, achieved on `bash` (five-criteria formulation) and on `printtokens2` (five-criteria formulation), respectively. On average, `HGA` is 9 times faster than `GDE3`. Similarly to the results achieved for `NSGA-II` in Section **??**, the number of test cases strongly influences the performance of `GDE3`. Indeed, the ratio between the execution time needed by `GDE3` and the execution time required by `HGA` increases as the number of test cases grows. This insight is further confirmed by the permutation test: the differences (improvements/worsening) between the execution time of `HGA` and `GDE3` significantly interacts with the test suite size (p-value=7.9×10^{-5}).

Comparing `HGA` and `MOEA-D/DE`, we notice that the results are very similar to those achieved when comparing `HGA` and `GDE3`. Indeed, for all the projects we can reject the null hypothesis for the *Wilcoxon t-test*. Moreover, according to the Vargha-Delaney ($\hat{A}_{12}$) statistics, in all cases `HGA` is more efficient than `MOEA-D/DE` with `large` effect size. `MOEA-D/DE` requires between 1.30 (*i.e.*, on `bash` with the four-criteria formulation) to 25 times (*i.e.*, on `printtokens2` with the five-criteria formulation) the execution times required for `HGA`. On average `HGA` is 8 times faster than `MOEA-D/DE`. To assess the interaction between the number of test cases and the performance gap between `HGA` and `MOEA-D/DE`, we performed the permutation test achieving a p-value equal to 2.20×10^{-16}.

> **Summary for RQ3.4.** `HGA` improves, on average, the efficiency of test case prioritization with respect to `GDE3` and `MOEA/D-DE` by 8 and 9 times, respectively. Moreover, `HGA` is able to scale up better as the number of test cases to prioritize increases.

5.5 THREATS TO VALIDITY

This section discusses the threats to the validity of our empirical evaluation, classifying them into *construct*, *internal*, *external*, and *conclusion* validity.

Construct Validity. In this study, they are mainly related to the choice of the metrics used to evaluate the characteristics of the different test case prioritization algorithms. To evaluate the optimality of the experimented algorithms (`HGA` and `Additional Greedy`) we used the $APFD_c$ [143], a well-known metric used in previous work on multi-objective test case prioritization [125, 275]. Another construct validity threat involves the correctness of the measures used as test

criteria: statement coverage, fault coverage and execution cost. To mitigate such a threat, the code coverage information was collected using two open-source profiler/compiler tools (GNU `gcc` and `gcov`). The execution cost has been measured by counting the number of source code blocks expected to be executed by the test cases [50, 83], while the original fault coverage information has been extracted from the SIR repository [261].

Internal Validity. To address the random nature of the GAs themselves [271], we run `HGA`, `NSGA-II`, `GDE3`, and `MOEA-D/DE` 30 times for each subject program (as done in previous work [51, 77, 262]), and considered the mean $APFD_c$ scores. The tuning of the EA's parameters is another factor that can affect the internal validity of this work. In this study, we use the same genetic operators and the same parameters used in previous work on test case prioritization [51, 79].

External Validity. We consider 11 open source and proprietary programs, that were used in previous work on regression testing [51, 81, 262, 263, 264, 276]. We firstly compared `HGA` on three different formulations of the test case prioritization problem, with respect to two state-of-the-art algorithms for test case prioritization (*e.g.,* `Additional Greedy` and `NSGA-II`). Secondly, we look at two new formulations of the problem considering more criteria and comparing with two many-objective meta-heuristic algorithms (*i.e.,* `GDE3` and `MOEA/D-DE`).

Conclusion Validity. We interpret our findings using appropriate statistical tests. In particular, to test the significance of the differences we used (i) *Welch's t-test* [209] and (ii) *Wilcoxon t-test* [209], while to estimate the magnitude and the effect size of the observed differences we used the Vargha-Delaney statistic [272]. Conclusions are based only on statistically significant results.

5.6 CONCLUSION

This chapter proposed a hypervolume-based genetic algorithm (`HGA`) to improve multi-criteria test case prioritization. Specifically, we use the concept of *hypervolume* [62], which is widely investigated in many-objective optimization, to generalize the traditional Area Under Curve (AUC) metrics used in previous work on test case prioritization [51, 78, 79, 80, 81]. Indeed, the *hypervolume* metric condenses multiple testing criteria through the proportion of the objective space, while AUC based metrics can manage only one cumulative code coverage criterion per time [51].

We performed a first empirical study with the aim of evaluating the cost-effectiveness and efficiency of HGA, compared to two state-of-the-art algorithms for the Test Case Prioritization problem, namely Additional Greedy [53] and NSGA-II [77, 265]. Our results show that HGA is more or equally cost-effective than the state-of-the-art approaches in most cases. The single solution provided by the algorithm is able to dominate most of the solutions provided by NSGA-II in terms of cost-effectiveness. Moreover, the performance of HGA does not decrease when larger programs and more objectives are considered. Looking at the execution time we note that the efficiency of Additional Greedy is strictly related to the number of test cases, while HGA is faster than NSGA-II in all the considered programs and formulations.

Afterwards we compared HGA with two many-objective evolutionary algorithms, *i.e.*, GDE3 [256] and MOEA/D-DE [257] to analyze the scalability of the proposed algorithm with respect to two many-objective state-of-the-art algorithms. We show that, in terms of cost-effectiveness, HGA is equivalent or better than GDE3 and MOEA/D-DE, while being much more efficient in terms of execution time.

As future work, we plan to incorporate diversity measures proposed in previous studies on multi-objective test case selection [83, 262] to improve the performance of HGA for software systems with highly redundant test suites, where greedy algorithms are particularly competitive. We plan to apply the proposed HGA also for other test case optimization problems, such as *Test Suite Minimization* and *Test Case Selection*. Finally, starting from the considerations made in Section **??**, we plan to perform a new empirical study to investigate which testing criteria are more capable to discover new faults.

6

ON THE IMPACT OF CODE SMELLS ON THE ENERGY CONSUMPTION OF MOBILE APPLICATIONS

6.1 INTRODUCTION

Testing is not only related to the functional properties of the software system, but also on its non-functional properties. Among these, energy efficiency is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. Although the problem is mainly concerned with hardware efficiency, in the recent past researchers successfully demonstrated how even software may be at the root of energy leaks [56]. The problem is even more evident in the context of mobile applications (*a.k.a.*, "apps"), where billions of customers rely on smartphones every day for social and emergency connectivity [57].

Green mining is the branch of software engineering responsible for the identification of factors causing energy leaks, as well as for the definition of practical solutions to deal with them. In this context, recent research has ranged from the definition of approaches to measure the power profile of mobile apps [56, 162] to the analysis of the impact of programming solutions on the energy consumption [170, 85, 176, 172].

As well as functional testing, also testing the energy efficiency of mobile apps is particularly expensive. For this reason, it would be reasonable to focus the testing on those software components that are more likely to consume energy. In the context of mobile apps development, a set of new peculiar bad programming practices of Android developers has been defined by Reimann *et al.* [58]. These Android-specific smells may threat several non-functional attributes of mobile apps, such as security, data integrity, and source code quality [58]. As highlighted by Hetch *et al.* [59], these type of smells can also lead to performance issues.

Although several important research steps have been made and despite the ever-increasing number of empirical studies aimed at understanding the reasons behind the presence of energy leaks in the source code, little knowledge

is available in literature on the potential impact on energy consumption of the Android-specific code smells defined by Reimann *et al.* [58]. These smells are detectable through static analysis, hence they could be used to efficiently detect energy leaks. Unfortunately, while the impact of these smells on energy consumption has been theoretically supposed by Reimann *et al.* [58], there exists only a few empirical evidence on it. For this reason, we aim at conducting a large empirical study to analyze the impact of the Android-specific smells on the energy consumption of mobile apps. Indeed, only the work by Carette *et al* [180] initially explored the relationship between smells and energy efficiency. However, they analyzed the behavior of just three code smell types on five mobile apps, finding that the removal of such code smells has a limited effect on energy efficiency (quantified as a 4% improvement of the overall energy consumption).

In this chapter, we provide a deeper investigation to determine (i) to what extent code smells affecting source code methods of mobile applications influence energy efficiency, and (ii) whether refactoring operations applied to remove them directly improve the energy efficiency of refactored methods. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann *et al.* [58] in the context of 60 Android apps belonging to the dataset provided by Choudhary *et al.* [87]. To the best of our knowledge, this is up to date the largest study aimed at practically investigating the actual impact of such code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency.

To conduct our analyses, we built upon two tools that we previously developed and evaluated, *i.e.*, aDoctor (see Section 6.2) and PETrA (see Section 6.3). The former is a novel *Android-specific* code smell detector that has been evaluated in our prior study [68] using 18 apps and it is very accurate, with a precision of 98% and a recall of 98%. The latter is a software-based tool that estimates the energy profile of mobile applications [69]. It has been evaluated using 54 apps [69] and provides an estimation error within 5% of the actual values measured with a hardware-based tool [85] in 95% of the cases.

Results of our study highlight that methods affected by code smells consume up to 385% more energy than methods not affected by any smell. A *fine-grained* analysis reveals the existence of four specific *energy-smells*, namely *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Finally, we

also shed light on the usefulness of refactoring as a way for improving energy efficiency by code smell removal. Specifically, we found that it is possible to improve the energy efficiency of source code methods by up to 900% through refactoring code smells.

6.2 ADOCTOR: LIGHTWEIGHT DETECTION OF ANDROID-SPECIFIC CODE SMELLS

In this section, we introduce ADOCTOR (ANDROID Code smell detecTOR), a novel code smell detector that identifies 15 Android-specific code smells. ADOCTOR is built on top of the Eclipse Java Development Toolkit (JDT)[1]. While the catalogue by Reimann *et al.* [58] proposes a set of 30 design flaws related to both implementation and UI design, we focus our attention solely on the smells characterizing a problem in the source code. Therefore, our tool supports the identification of 15 Android-specific code smells. In the following, we present the detection rules applied by ADOCTOR, as well as the underlying architecture supporting the identification.

6.2.1 *Code Smells detected by aDoctor*

This section reports, the definition of each smell supported by ADOCTOR as well as the rule followed for its detection.

- **Data Transmission Without Compression (DTWC).** The smell arises when a method transmits a file over a network infrastructure without compressing it, causing an overhead of communication [58]. ADOCTOR detects the smell if a method performs an `Http` request involving an instance of the class `File` without using a compression library such as ZLIB[2] or the APACHE HTTP CLIENT[3].

- **Debuggable Release (DR).** In Android, the attribute `android:debuggable` of the `AndroidManifest` file is set during the development for debugging an app. Leaving the attribute `true` when the app is released is a major security threat since every external app can have full access to the source

1 http://www.eclipse.org/jdt/
2 http://www.zlib.net
3 https://hc.apache.org

code. In this case, the detector simply parses the `AndroidManifest` file looking for the `android:debuggable` properties. If it is explicitly set to `true`, the smell is detected.

- **Durable Wakelock (DW).** A Wakelock is the mechanism allowing an app to keep the device on in order to complete a task. However, when such task is completed, the lock should be released to reduce battery drain [58]. In Android, the class `PowerManager.WakeLock` is in charge to define the methods to acquire and release the lock. If a method using an instance of the class `WakeLock` acquires the lock without calling the `release`, a smell is identified.

- **Inefficient Data Format and Parser (IDFP).** When analyzing XML or JSON files, the use of `TreeParser` slows down the app, and thus it should be avoided and replaced with other more efficient parsers (*e.g.*, `StreamParser`) [58]. In this case, aDoctor identifies the smell by evaluating whether a method uses the `TreeParser` class.

- **Inefficient Data Structure (IDS).** The mapping from an integer to an object through the use of a `HashMap<Integer, Object>` is slow, and should be replaced by other efficient data structures, such as the `SparseArray` [58]. Therefore, methods using an instance of `HashMap<Integer, Object>` are identified by aDoctor as smelly.

- **Inefficient SQL Query (ISQLQ).** In Android, the use of a `SQL` query is discouraged as it introduces overhead, while other solutions should be preferred (*e.g.*, using webservices) [58]. If a method defines a `JDBC` connection and sends an `SQL` query to a remote server, the smell is identified.

- **Internal Getter and Setter (IGS).** In Android development, the use of accessors methods (*i.e.*, getters and setters) are expensive and, thus, internal fields should be accessed directly [58]. All the methods accessing other objects using getters and/or setters are identified by aDoctor as affected by this smell.

- **Leaking Inner Class (LIC).** Reimann *et al.* defined this smell as a *"non-static nested class holding a reference to the outer class"* [58]. This could lead to a memory leak. Analyzing the files having nested classes, aDoctor identifies this smell by counting the relationships that the outer class has

with the nested classes. If the counter is higher than 1, a *Leaking Inner Class* is detected.

- **Leaking Thread (LT).** In Android programming a thread is a garbage collector (GC) root. The GC does not collect the root objects and, therefore, if a thread is not adequately stopped it can remain in memory for all the execution of the application, causing an abuse of the memory of the app. If an `Activity` starts a thread and does not stop it this is considered a design flaw [58]. aDoctor detects this smell if a method of an `Activity` class starts a thread without stopping it through the `stop` method.

- **Member Ignoring Method (MIM).** Non-static methods that do not access any internal properties of the class should be made `static` in order to increase their efficiency [58]. In this case, our detector exploits the references of a method, and if it does not reference any internal fields, a smell is identified.

- **No Low Memory Resolver (NLMR).** An Android developer can define the behavior of the app when it runs in background overriding the method `Activity.onLowMemory` [58]. This method should be used to clean caches or unnecessary resources. If it is not defined, the app can lead to abnormal memory use. Consequently, if a mobile app does not contain the method `onLowMemory`, aDoctor detects a smell.

- **Public Data (PD).** This smell arises when private data is kept in a store that is publicly accessible by other applications, possibly threatening the security of the app [58]. In Android, this is done by setting the context of the class as private, using the `Context.MODE_PRIVATE` command. Classes that do not define the context or define the context as non-private are detected by aDoctor as smelly.

- **Rigid Alarm Manager (RAM).** The `AlarmManager` class allows to execute operations at specific moments. Obviously, an Alarm Manager-triggered operation wakes-up the phone, possibly threatening the energy and memory efficiency of the app. It is recommended to use the `AlarmManager.setInexactRepeating` method, which ensures that the system is able to bundle several updates together [58]. Therefore, a code smell is identified by our detector if a class using an instance of `AlarmManager` does not define the method `setInexactRepeating`.

- **Slow Loop (SL).** The standard version of the `for` loop is slower than the `for-each` loop [58]. Therefore, Android developers should always use an enhanced version of the loop to improve the efficiency of the app. Our detector identifies smelly instances as all the methods using the `for` loop.
- **Unclosed Closable (UC).** A class that implements the `java.io.Closeable` interface is supposed to invoke the `close` method to release resources that an object is holding [58]. If the class does not call such a method, aDoctor identifies a smell.

6.2.2 *RQ4.1: aDoctor Evaluation*

We evaluate aDoctor with the *goal* to quantify the ability of aDoctor in recommending portions of source code affected by a design flaw, with the *purpose* of investigating its effectiveness during the detection of Android-specific code smells in Android applications. Specifically, our research question is the following:

> **RQ4.1** *What are the precision and recall scores of* aDoctor *in detecting Android-specific code smells?*

The *context* of the study consists of a set of 18 Android apps belonging to different categories, and having different scope and size. The complete list of apps considered in the study is available on the aDoctor website [277].

6.2.2.1 *Empirical Study Design*

We ran aDoctor on the apps in our context. To evaluate its precision and recall, we needed an oracle reporting the actual code smell instances contained in the considered Android apps. Since there is not an annotated set of Android-specific code smells available in literature, we built our own oracle. To this aim, we asked a Master's student from the University of Salerno to manually analyze the apps taken into account in order to extract the methods affected by each of considered smells. Starting from the definition of the 15 smells, the student manually analyzed the source code of the latest version of the apps, looking for instances of those smells. This process took approximately 180 man-hours of work. Then, a second Master's student (still from the University of Salerno) validated the produced oracle, to verify that all affected code components

Table 6.1: Performance of ADOCTOR on the apps object of the empirical study

Code Smell	Precision	Recall	F-Measure
DTWC	87%	89%	88%
DR	100%	100%	100%
DW	100%	100%	100%
IDFP	100%	100%	100%
IDS	100%	100%	100%
ISQLQ	85%	88%	86%
IGS	100%	100%	100%
LIC	100%	100%	100%
LT	100%	100%	100%
MIM	100%	100%	100%
NLMR	100%	100%	100%
PD	100%	100%	100%
RAM	100%	100%	100%
SL	100%	100%	100%
UC	100%	100%	100%
Average	98%	98%	98%

identified by the first student were correct. Just 14 of the instances classified as smelly by the first student were classified as false positives by the second student. After a discussion performed between the two students, 8 of these 14 instances were definitively classified as false positives (and, therefore, removed from the oracle). Note that we cannot ensure about the completeness of the oracle. Moreover, to avoid bias the students were not aware of the experimental goals and of specific algorithms used by ADOCTOR to identify smells. The oracle defined is available on the ADOCTOR website.

Once the set of actual smells was ready and the set of candidate smells identified by ADOCTOR was available, we compared the two sets using two widely adopted Information Retrieval (IR) metrics, *i.e.*, precision and recall [198].

To have an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall.

We report the overall precision and recall obtained analyzing each smell type on the 18 apps. The results achieved on the single apps are available on the ADOCTOR website [277].

6.2.2.2 *Analysis of the Results*

Over all the 18 apps considered, ADOCTOR detects 1,444 code smell instances (on average, 80 per app). The most frequent ones are the *Member Ignoring Method* (467 instances), Slow Loop (378 instances), and Data Transmission Without Compression (266 instances) smells. Since the analyzed apps contain on average 121 classes, our results reveal that the Android-specific smells are quite diffused

and, thus, the phenomenon is worth investigating. Note that the complete results on the distribution of code smells are available on the aDoctor website [277].

Table 6.1 reports, for each Android-specific smell, the results achieved over the set of 18 apps taken into account. The results clearly show that aDoctor is able to correctly identify almost all the code smell instances present in the Android apps. Only in two cases the results do not reach 100% precision and recall, *i.e.*, *Data Transmission Without Compression* and *Inefficient SQL Query*. We manually analyzed these cases in order to understand the reasons behind the results, finding that the detector missed some instances because the classes affected by such smells used different compression libraries with respect to the ones considered in the detection rules. Indeed, both smells are related to the communication with remote servers. To do so, Android apps usually rely on some widely spread libraries such as Zlib or the Apache HTTP Client. However, there are some cases where other libraries are employed and, therefore, the detector is not able to correctly identify the design flaws. For instance, aDoctors identifies a false positive *Data Transmission Without Compression* instance in the class `AndroidomaticKeyerActivity`, belonging to the package `com.templaro.opsiz.aka` of the Androidomatic Keyer app. This class relies on the SiliCompressor library[4] to compress files before sending them, but aDoctor does not recognize the compression because the method calls done by the class do not refer to the libraries it consider.

While in this case aDoctor fails in the identification of the smell, it is worth noting that we configured our detector in order to work with the most common libraries used by Android developers. Moreover, the issue reveals a potential way to improve the detection accuracy of the tool. Indeed, as the support to other libraries will be implemented, the performances of the tool will be higher.

The discussion is different for the other smells, since aDoctor always reaches 100% of F-Measure. This is due to the fact that the detection rules are effective in capturing all the small programming issues applied by Mobile developers. In conclusion, we can affirm that the proposed tool is efficient in terms of accuracy of the recommendations.

4 https://github.com/Tourenathan-G5organisation/SiliCompressor

6.3 PETRA: SOFTWARE-BASED ENERGY PROFILING OF ANDROID APPS

This section presents our novel software-based approach, coined PETrA (**P**ower **E**stimation **T**ool fo**r** **A**ndroid), suitably developed to measure the power consumption of mobile apps at a method-level granularity.

6.3.1 *PETrA Workflow*

As depicted in Listing 3, its main process is composed of three main blocks: (i) app preprocessing, (ii) energy profile computation, and (iii) output generation. In the following paragraphs, we detail each part independently.

6.3.1.1 *App Preprocessing*

In the first step, PETrA needs to set the software environment before measuring the energy consumed when executing a mobile app. To this aim, it uses as input an executable version of the app under analysis in the form of an apk file. The app is identified by the apk location and the name of the app to profile, which correspond to `apkLocation`, and `appName` in Algorithm 3 respectively. Then, PETrA installs the apk on a mobile phone able to run it (*e.g.*, a smartphone having an arbitrary version of the Android operating system) and enables the `debuggable` option. Enabling debugging is mandatory, because otherwise the instrumentation of the app, needed to profile it, would not be possible.

6.3.1.2 *Energy Profile Computation*

Once the app is properly set up, PETrA exercises the app under consideration using a test case given as input, *et al.* `tCase` in Listing 3. This test case can be created with automated tools (*e.g.*, Monkeyrunner or Monkey) or with manual operations performed by the software engineer. Once the test case is run, the *core* process behind PETrA starts.

For the profiling phase, we leverage the `Project Volta` Android tools, which are based on the self-modeling paradigm proposed by Dong and Zhong [278], *i.e.*, the definition of a mobile system that automatically generates its en-

Algorithm 3: PETrA workflow

```
   Input:
   apkLocation: the location of the apk
   appName: the name of the application
   tCase: the test case to run
   nRuns: the number of executions
   Result: the consumption in Joule of each method call
1  begin
2      install the apk in apkLocation
3      for run=0
4      run¡nRuns
5      run++ do
6          clear the appName cache
7          reset Batterystats
8          start the profiler
9          exercise appName with tCase
10         stop the profiler
11         collect BatteryStats data
12         collect SysTrace data
13         collect dmtracedump data
14         load PowerProfile.xml
15         for each method call in trace file do
16             compute the energy consumption for the method call
17         save the results
18         stop appName
19     uninstall appName
```

ergy model without any external assistance. Such tools are `dmtracedump`[5], `Batterystats`[6], and `Systrace`[7]. Specifically:

- `dmtracedump` provides an alternate way to show trace log files. The files generated by `dmtracedump` are easy to parse and allow the developers to establish precisely, at microseconds granularity, when a method call has been invoked and when it returned. PETrA relies on this component in order to store the execution traces of the app under analysis. For each method call `dmtracedump` provides the entry and the exit time. The final output is a list of the executed method calls during the run.

- `BatteryStats` is an open source tool of the Android framework able to collect battery data from the device under evaluation. In particular, it is able to show which processes are consuming battery energy and which tasks should be modified in order to improve battery life. It is executable via the command line. The data collected can be analyzed as a log file or can be converted to an HTML visualization that can be viewed in a browser using `Battery Historian`[6]. PETrA uses the `Batterystats` log in

5 https://developer.android.com/studio/profile/traceview.html
6 https://developer.android.com/studio/profile/battery-historian.html
7 https://developer.android.com/studio/profile/systrace-commandline.html

order to retrieve the active smartphone components and their status in a specific time window. Furthermore, it can provide the information about the device voltage. Given this information, it is then possible to calculate the energy consumed by the smartphone during a time window.

- `Systrace` is a tool that can be used to analyze application performance. It captures and displays the execution times of the active processes of a smartphone, combining data from the Android kernel, *i.e.*, the CPU scheduler, disk activity, and application threads. The data can be viewed as an HTML report that shows the overview of the processes in a given time window In PETRA, the information provided by `Systrace` is used to capture the frequency of the CPU in a given time window. Considering that CPUs have different consumptions as their frequency varies, this information completes the one provided by `Batterystats` improving the estimations.

After gathering the information related to the active components with their status, the CPU frequencies and the method call invocations, the `power profile` file is loaded. The `power profile` values define the current consumption for a component along with an approximation of the battery drain caused by each component over time. For instance, it specifies how many MilliAmperes of current are required to run the CPU at a certain frequency. Every smartphone has its own power profile. It is worth noting that each device manufacturer must provide this information and that this info can be found in a defined location in the device[8].

Given the previous data, it is possible to compute the energy consumed for every method call invocation. First of all, given a method call invocation and its termination we can calculate the overall time window T_w as the arithmetic difference between the two time instants when these two events occurred. However, the energy consumed within one single time window is not constant but may change because of a CPU frequency variation or a component state change happened. Therefore, we divided the time windows in smaller time units, *et al.* data frames T_Δ. When the entry to a method is registered, a new time window T_w and a new time frame T_Δ start. Whenever a component changes its state, the existing time frame T_Δ is terminated and a new one (for the new state) is started. When the exit point to a method is registered, then the corresponding

8 https://source.android.com/devices/tech/power/values.html

time window T_w is terminated as well as the latest time frame T_Δ. In this way, each data frame T_Δ is characterized by coherent component states (e.g., CPU frequency) and by a coherent (constant) energy drain. For example, if the CPU is working at the maximum frequency and none of the components change their state, the time windows T_w will be composed by only one time frame T_Δ of the same duration, *et al.* be the difference between the method entry and exit. Therefore, we can calculate the current power intensity at each time frame T_Δ as follows:

$$I_\Delta = \sum_{\forall c \in C} I_{\Delta,c,s} \tag{6.1}$$

where C is the set of smartphone hardware components, $I_{\Delta,c,s}$ is the current intensity of the component c with the state s within the current time frame T_Δ. For example, 92.6 is the number of MilliAmpere consumed in one second by a Nexus 4 when the CPU frequency is fixed to 384Mhz.

After calculating the current intensity, it is possible to calculate the energy consumed in a time frame, as follows:

$$J_\Delta = I_\Delta \times V_\Delta \times T_\Delta \tag{6.2}$$

where J_Δ is the consumed energy in Joule, I_Δ is the current intensity in Ampere, V_Δ is the device voltage in Volt and T_Δ is the length of the time frame in seconds.

Finally, the energy consumed by a method call can be calculated by summing up the energy consumed in each time frame in which the method call was active:

$$J = \sum_{T_\Delta \in T_w} (I_\Delta \times V_\Delta \times T_\Delta) \tag{6.3}$$

6.3.1.3 *Output Generation*

The final output provided by PETRA is a csv file, containing the energy estimation for each method call. More precisely, it provides the signature of each executed method call, along with the consumption in Joule and the execution time in seconds.

By default, PETRA uses `Monkey` to generate test cases, which are used to exercise the mobile app under analysis and to store the energy drain information for every exercised method call. It is important to highlight that the usage of

`Monkey` is not mandatory, as other tools can be also used with PETrA. However, in order to have a fair comparison with the oracle data, we used `Monkeyrunner` (*et al.* we used the same test cases).

Finally, PETrA relies on the `Android Activity Manager`[9], so the apk must be enabled for debugging. Furthermore, in order to provide a better estimation, PETrA exercises the app multiple times (`nRuns` in Algorithm 3). Note that in our experiments `nRuns` is fixed to 10 and that in order to avoid any bias due to multiple runs, at the start of each run the app cache is cleaned and `Batterystats` is reset (lines 4 and 5 in Algorithm 3).

6.3.2 *RQ4.2: PETrA Evalutation*

The *goal* of the study is to analyze the accuracy of PETrA in providing energy consumption estimations of mobile apps at method-level granularity with the *purpose* of investigating whether the proposed approach can be used as a valid alternative to hardware-based solutions. More specifically, the study aims at addressing the following research question:

> **RQ4.2:** *How close are the estimations from* PETrA *to a hardware-based tool?*

6.3.2.1 *Context Selection and Oracle Extraction*

The *context* of the study consisted of a set of 54 Android apps from the `Google Play Store` having different categories and scope. The complete list of apps considered in the study is available on the PETrA replication package [279]. The choice of using these apps is not random, but rather guided by the need of having a set of applications for which an oracle reporting the consumption measured at the method level with hardware-based tools is publicly available. Indeed, since we had no hardware-based tool available to perform measurements we had to look for alternative solutions.

Some available datasets provide data about the energy consumption of software changes [280] or system calls [281]. However, these datasets are not suitable for our purpose because they (i) do not provide detailed measures for source code at the method level and (ii) contain data from desktop and web applications (*e.g.*, Firefox) rather than mobile apps. For this reason, we relied

9 https://developer.android.com/studio/command-line/shell.html

on the dataset provided by Linares-Vasquez *et al.* [85], that reports the actual power consumption of the methods belonging to the APIs used by the 54 mobile apps considered in the study. The authors computed the measurements relying on the Monsoon toolkit [86]. Note that the dataset also contains the test data needed to exercise the app in the same manner as done by Linares-Vasquez *et al.* [85] (more details on the measurement process come later in this section). A direct consequence of our choice to rely on this dataset is that we had to limit the focus of our analysis to methods belonging to APIs. However, we still took into account the energy consumption of 414.899 API calls belonging to the 321 APIs used by the considered apps. Moreover, according to recent findings achieved by Li and Gallagher [282], method invocations represent the more influencing energy consuming operation and, therefore, it is particularly interesting the analysis of the context of API calls.

6.3.2.2 *Test Environment Setup and Energy Profiles Extraction*

As PETrA is a software-based approach, it requires a simple test environment composed only of a smartphone and a PC. While seemingly simple, we need to ensure a well-isolated test environment in order to avoid biases. To this aim, we carefully followed the guidelines from previous work in the field [151, 85, 162, 283]. The subsequent subsections detail each setup choice.

Choice of the Smartphone. We selected a factory resetted `LG Nexus 4` having Android 5.1.1 Lollipop as the operating system, and equipped with a 1.5 GHz quad-core Snapdragon S4 Pro processor with 2 GB of RAM, and having a 2100 mAh, 3.8V battery. The choice is guided by the need to have the same smartphone used in the paper by Linares-Vasquez *et al.* [85] in order to conduct a fair evaluation. Moreover, it is worth noting that this particular hardware allows to be connected via a data cable, namely a cable where the USB charging can be disabled[10]. Thus, during the experiment no energy is transferred over the cable, allowing more stable measurements.

Isolating the execution of an app. To isolate the behavior of an application being executed on the smartphone, we adopted a number of precautions. In particular, we firstly disabled all the unnecessary apps and processes (*e.g.*, Google Services) running on the phone to avoid race conditions. Then, we avoided asynchronous events, such as incoming messages or calls by removing

10 http://android.stackexchange.com/questions/54902/disable-usb-charging

the sim card from the phone. Finally, we held the phone steady to avoid energy measurements by sensors and WiFi signal changes.

Extraction of the Energy Profiles of APIs. To extract the energy profiles of the apps in our dataset, we have to enable the debug mode. This entails manually adding `android:debuggable="true"` in the file `AndroidManifest.xml` and regenerating the `apk` file for each app (*i.e.* the executable), using ANDROID STUDIO [284].

Once we created these debuggable and executable versions of the apps, we applied PETRA to them. As explained in Section **??**, our approach receives as input a set of test cases for exercising the app under consideration and measuring the energy consumption at method-level granularity. In the context of this experiment, we exercised the apps in our dataset by using exactly the same `Monkeyrunner`[11] test cases used by Linares-Vasquez *et al.* [85]. This allows a fair comparison between the energy profiles extracted using our approach and the oracle provided using the MONSOON toolkit [86].

The output of this step consisted of a set of files reporting the execution traces of each app, accompanied by the information on the energy consumed by each method during that execution. It is important to note that in this stage we collected the information for all the methods belonging to an application. However, to compare the energy profiles extracted by PETRA with the ones extracted using MONSOON [86], we needed to select only the methods belonging to an API. To this aim, we selected from the final output produced by PETRA only the Android public methods, removing also the calls to other Java APIs.

Moreover, to be more confident about the energy profiles built by PETRA, we repeated the measurements 10 times. Similarly to Linares-Vasques *et al.* [85], we aggregated the results of the 10 runs (*i.e.*, the joules consumed by the methods in each run) using the mean operator. Therefore, the final output consisted of a unique value representing the average energy consumed by the method belonging to an API exercised during the test execution.

6.3.2.3 *Data Analysis and Metrics*

Once extracted the energy profiles using PETRA, we answered $\mathbf{RQ}_1$ by comparing the energy profiles computed using PETRA with the oracle data [85]. To evaluate to what extent the energy consumption provided by our approach is close to the values measured with a hardware based tool, we employed a set of

11 https://developer.android.com/studio/test/monkeyrunner/

metrics widely used in the area of cost estimation [285] [286]. Specifically, we used the *Mean Magnitude Relative Error* (MMRE) [285] defined as follow:

$$MMRE = \frac{1}{N}\sum_{i=1}^{n} MRE_i \tag{6.4}$$

where n is the number of energy estimations computed by PETrA on each app (i.e., number of methods), and MRE indicates the *Magnitude Relative Error* [285] and has values in the range defined by the following formula:

$$MRE_i = \frac{|J_i - \hat{J}_i|}{J_i} \tag{6.5}$$

where J_i and $\hat{J}_i$ are the energy estimations produced by Monsoon and PETrA respectively for the method i.

Besides determining the mean error in the estimations provided by our approach, we also computed the $PRED(x)$ metric, namely the *Relative Error Deviation Within x* [287]. This measure gives an indication of how many methods have estimation errors with our approach within x of the estimation values provided by Monsoon. In particular, $PRED(x)$ is defined as the average fraction of the *MREs* off by no more than x as defined by Jorgensen [288].

$$PRED(x) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases} 1 \; if \; MRE_i \leq x \\ 0 \; otherwise \end{cases} \tag{6.6}$$

In the field of cost estimation, the parameter x is usually set to 25, *i.e.,* the estimated cost is within 25% of the actual cost of a project [286]. However, in our context an estimation error of 25% could be very large. For instance, a variation of 25% is very large when estimating the energy consumed by a data structure used in the source code [172]. An analysis done in this way would be too coarse-grained. Thus, we verified whether PETrA can achieve a lower estimation error, by setting x=2; 5; 7; 10; 20. In this way, we were able to control how the estimation errors of our approach are distributed.

Obviously, the estimation errors in $PRED(5)$ are also included in $PRED(2)$, the estimation errors in $PRED(5)$ in $PRED(7)$ and so on. This means that the distribution over the different $PRED(x)$ measures is cumulative. For instance, if $PRED(2)$ is equal to 0.91 and $PRED(5)$ is 0.96, then 5% of the estimation errors are between 2% and 5%.

Finally, we performed a *fine-grained* analysis aimed at understanding the types of errors achieved by PETrA during the energy profile estimations. To this aim, we (i) measured the ratio of over/under estimations provided by our approach, and (ii) provided motivations behind the estimation errors.

Table 6.2: MMRE, PRED(x), over estimations, and under estimations computed for the apps under evaluation

#	MMRE	PRED(2)	PRED(5)	PRED(7)	PRED(10)	PRED(20)	Over Estimations	Under Estimations
1	0.04	0.71	1.00	1.00	1.00	1.00	1.00	0.00
2	0.01	0.92	0.94	0.94	0.95	0.95	0.91	0.09
3	0.01	0.89	0.99	0.99	0.99	0.99	0.99	0.01
4	<0.01	0.82	0.92	1.54	0.92	0.92	0.92	0.08
5	<0.01	0.91	1.00	1.00	1.00	1.00	1.00	0.00
6	0.01	0.78	1.00	1.00	1.00	1.00	0.89	0.11
7	0.02	0.75	0.99	0.99	1.00	1.00	0.89	0.11
8	0.01	0.83	1.00	1.00	1.00	1.00	0.84	0.16
9	0.01	0.89	1.00	1.00	1.00	1.00	1.00	0.00
10	<0.01	0.69	0.89	0.90	0.91	0.94	0.87	0.13
11	0.01	0.93	1.00	1.00	1.00	1.00	0.94	0.06
12	<0.01	0.71	0.98	0.98	0.99	0.99	0.94	0.06
13	0.02	0.91	0.95	0.95	0.96	0.97	0.85	0.15
14	<0.01	0.84	0.99	0.99	1.00	1.00	0.88	0.12
15	<0.01	0.89	0.99	0.99	0.99	1.00	0.99	0.01
16	0.02	0.91	0.99	0.99	1.00	1.00	1.00	0.00
17	<0.01	0.87	0.99	0.99	0.99	0.99	0.99	0.01
18	<0.01	1.00	1.00	1.00	1.00	1.00	1.00	0.00
19	0.02	0.96	0.97	0.97	1.00	1.00	0.95	0.05
20	0.02	0.87	1.00	1.00	1.00	1.00	0.82	0.18
21	<0.01	0.78	0.81	0.87	1.00	1.00	0.92	0.08
22	<0.01	1.00	1.00	1.00	1.00	1.00	1.00	0.00
23	<0.01	0.88	0.98	0.98	0.98	0.99	0.98	0.02
24	0.01	0.77	0.99	0.99	0.99	0.99	0.93	0.07
25	0.01	0.91	0.96	0.97	0.97	0.98	0.96	0.04
26	<0.01	0.30	0.32	0.32	0.32	0.38	0.28	0.72
27	0.02	0.95	0.99	1.00	1.00	1.00	0.76	0.24
28	<0.01	1.00	1.00	1.00	1.00	1.00	1.00	0.00
29	0.01	0.96	0.98	0.98	0.98	0.99	0.79	0.21
30	0.02	0.96	0.97	0.97	1.00	1.00	0.89	0.11
31	0.01	0.96	1.00	1.00	1.00	1.00	1.00	0.00
32	0.01	0.99	1.00	1.00	1.00	1.00	0.91	0.09
33	<0.01	0.10	0.11	0.12	0.13	0.17	0.11	0.89
34	0.01	0.88	0.98	0.99	1.00	1.00	0.78	0.22
35	0.01	0.91	0.94	0.95	0.96	0.97	0.82	0.18
36	0.01	0.63	0.94	0.94	1.00	1.00	0.91	0.09
37	0.02	0.93	1.00	1.00	1.00	1.00	0.97	0.03
38	<0.01	0.95	1.00	1.00	1.00	1.00	0.81	0.19
39	0.01	0.92	1.00	1.00	1.00	1.00	0.78	0.22
40	<0.01	0.92	0.99	0.99	0.99	0.99	0.99	0.01
41	0.02	0.94	0.99	0.99	0.99	0.99	0.85	0.15
42	<0.01	0.83	0.92	0.92	0.92	0.92	0.92	0.08
43	0.01	0.99	1.00	1.00	1.00	1.00	1.00	0.00
44	0.01	0.95	1.00	1.00	1.00	1.00	0.91	0.09
45	0.02	0.97	0.99	0.99	0.99	0.99	0.94	0.06
46	0.01	0.97	0.98	0.98	0.98	0.99	0.98	0.02
47	0.03	0.09	0.95	0.95	0.95	0.95	0.95	0.05
48	0.02	0.68	1.00	1.00	1.00	1.00	0.82	0.18
49	<0.01	0.91	0.97	0.97	0.97	0.97	0.96	0.04
50	0.01	0.91	1.00	1.00	1.00	1.00	0.82	0.18
51	0.02	0.77	1.00	1.00	1.00	1.00	0.89	0.11
52	0.01	0.92	1.00	1.00	1.00	1.00	1.00	0.00
53	0.02	0.95	1.00	1.00	1.00	1.00	1.00	0.00
54	<0.01	0.97	1.00	1.00	1.00	1.00	1.00	0.00
Average	**0.01**	**0.85**	**0.95**	**0.96**	**0.96**	**0.96**	**0.89**	**0.11**

6.3.2.4 *Analysis of the Results*

For each app considered in the empirical study, Table 6.2 shows the $MMRE$ and the distribution of the $PRED(x)$ achieved when comparing the estimations provided by PETrA with those reported by the oracle data [85]. Table 6.2 also reports the percentage of over/under estimations given by PETrA. Finally, the row *Average* shows the results obtained when considering all the estimations as a unique dataset.

A first point to discuss is the mean estimation error provided by our tool (column $MMRE$). From Table 6.2 we observe that for all 54 mobile apps the *mean relative error* ($MMRE$) is always lower than 0.05, being 0.01 on average. To have a practical idea of the magnitude of the error, an $MMRE = 0.01$ corresponds to a percentage of battery discharge of $3.1 \times 10^{-6}\%$, and thus we can claim that PETrA misses the correct energy consumption by a small factor. For 39 out of the 54 apps that we considered, the mean estimation error (MMRE) is equal to or less than 0.01. For the remaining 15 apps, we experienced an average error of 0.022.

> **Observation 1.** In 72% of mobile apps the *mean estimation error* provided by PETrA is at most 0.01. Although in the other cases the difference is slightly larger, it still only reaches at most 0.04. Thus, the proposed software-based solution provides energy estimations that are quite close to the actual values.

Observation 1 only provides a partial view on the performance of our tool as we also need a better understanding of the relative difference between the estimation of PETrA and the hardware based solution. This is why we use the $PRED(x)$ metric, which indicates the percentage of methods in each app with an error (MRE) lower than x. Table 6.2 shows that in 95% of the methods our tool is able to provide an estimation error within 5% of the actual values measured using the Monsoon toolkit. Moreover, in 85% of the methods our tool provides estimation within 2% of error. These data confirm what we found when analyzing the $MMRE$ metric: a software-based solution built using public Android APIs performs similarly to a hardware-based solution that computes the energy consumption of Android apps. The result is particularly evident on 23 apps of our dataset (43%), where we observed that all the estimations of PETrA falls back into 5% of the actual energy consumption (*i.e.*, $PRED(5) = 1.00$). Moreover, despite the results for $PRED(5)$ already revealed the effectiveness of

PETrA, it is also important to mention that only 2% of methods have estimation errors outside the 50% of the actual energy consumption. Finally, while PETrA generally works well and provides estimations very close to the actual values, there are a few methods where such errors are higher than the ones discussed above. In particular, for 5% of methods the corresponding estimations are not within 5% of the oracle values. Further analyzing the factors behind such deviations, we observed that these are due to the usage of sensors (*i.e.*, motion, environmental, and position sensors). Unfortunately, the measurement of the power consumption of sensors is still an open issue in software-based energy measurement [289]. In the case of PETrA these components are not analyzed by Android tools and as such, we inherit this weakness. We plan to analyze this aspect as a future work, in order to improve the performance of our tool.

> **Observation 2.** In 95% of the methods PETrA provides an *estimation error* within 5% of the actual values measured with the Monsoon toolkit, confirming the good performances of our tool. Errors in measurement mainly occur in cases where there is significant use of network capabilities or when the sensors are used.

Regarding the types of estimations provided by PETrA, we observed that our tool rarely underestimates the energy consumption (overall, in 11% of the cases), while 89% of the estimations are slightly higher than the actual values. This is mainly due to the fact that in each energy estimation some noise is summed up during the measurement. Thus, the estimation results are typically higher than the actual values. However, as previously shown, our tool is able to achieve a good compromise between the errors committed and the accuracy of the evaluations. On the other hand, the few underestimations are generally due to the usage of sensors.

> **Observation 3.** In 89% of the methods PETrA overestimates the energy consumed. This is mainly due to the noise accumulated over the different APIs estimations. In the remaining 11% of the methods, our tool underestimates the actual energy consumption because of the presence of sensors.

Once we evaluated the three different aspects described above (*i.e.*, $MMRE$, $PRED(x)$, and over/under estimations), we can provide an answer to our **RQ**. First of all, we can conclude that PETrA *provides estimations close to the actual ones*:

indeed, the *mean relative error* (*MMRE*) produced by our tool is always lower than 0.05 (0.01 on average). At the same time, *95% of the estimation errors are within 5% of the actual values computed using a hardware-based tool*: this confirms that a software-based approach can actually perform similarly to the alternative hardware solution. Finally, our analyses highlighted some potential drawbacks of our tool. Indeed, the significant use of network capabilities as well as the usage of sensors can lead to higher estimation errors.

6.4 EMPIRICAL STUDY DEFINITION AND DESIGN

The *goal* of the study is to analyze the source code of mobile apps with the *purpose* of investigating whether (i) the presence of code smells influences energy consumption, and (ii) the removal of such design flaws through refactoring actually reduces the energy consumption of mobile applications. More specifically, the study aims at addressing the following three research questions:

Table 6.3: The Code Smells considered in our Study

Abbreviation	Name	Description	Refactoring
DTWC	Data Transmission Without Compression	A method transmitting a file over network without compressing it	Add Data Compression
DWL	Durable Wakelock	A method acquiring a wakelock and not releasing it	Aquire WakeLock with Timeout
IDS	Inefficient Data Structure	A method using an Hashmap<Integer, Object>	Use Efficient Data Structure
ISQLQ	Inefficient SQL Query	A method using a SQL query over JDBC to a remote server	Use JSON query
IDFAP	Inefficient Data Format And Parser	A method using a TreeParser	Use Efficient Data Parser and Format
IGS	Internal Getter/Setter	Internal fields are accessed via getters and setters	Direct Field Access
LT	Leaking Thread	A method using a thread that will never be stopped	Introduce Run Check Variable
MIM	Member-Ignoring Method	Non-static methods that don't access any property	Introduce Static Method
SL	Slow Loop	A slow version of a for-loop is used	Enhanced For-Loop

- **RQ4.3:** *What is the impact of code smells on the energy consumption of mobile apps?*
- **RQ4.4:** *Which code smells have a higher negative impact on the energy consumption of mobile apps?*
- **RQ4.5:** *Does the refactoring of code smells positively impact the energy consumption of mobile apps?*

It is important to note that the goal of **RQ4.3** is not demonstrating that a smelly method consumes more than an equivalent non-smelly one, but rather to understand whether it is actually worth studying these types of smells from

an energy perspective. Similarly, with **RQ4.4** we aim at understanding which smell types are more relevant from an energy consumption perspective. Finally, **RQ4.5** assesses the actual gain provided by refactoring in terms of energy consumption.

6.4.1 *Context Selection*

The *context* of the study consists of 60 open source Mobile Android apps publicly available in the F-Droid repository[12]. Specifically, we selected all the apps from the benchmark dataset provided by Choudhary *et al.* [87], which collects a subset of apps used in previous studies [290, 291, 292, 293] having different size and scope and that are still maintained by their own developers. The complete list of the apps used in this study is available in our online appendix [294].

Table 6.3 reports the set of code smells investigated in the study, together with a brief explanation and the corresponding refactoring operations. In particular, we analyzed the behavior of 9 *Android-specific* code smells extracted from the catalogue defined by Reimann *et la.* [58]. This catalogue reports a set of poor design/implementation choices applied by Android developers that are believed to impact non-functional attributes of mobile apps, such as software quality, user experience, performance, and energy consumption. However, it is important to point out that the actual impact of the defined smells on non-functional attributes has only been conjectured by the authors of the catalogue, and no empirical evaluation has been directly conducted to verify and measure such an impact.

Among the 30 types of Android-specific design flaws available in the catalogue, we selected only 9 code smells for three main reasons. First of all, we selected the design flaws that directly affect the source code, while the catalogue also includes problems related to poor user interface design choices, *e.g., Nested Layout*. Secondly, we included the code smells supposed to be directly connected with the energy consumption of the app rather than the ones related to violations of other non-functional aspects, such as data security and privacy. For instance, we have not considered the *Public Data* code smell, which appears when private data is stored in a location publicly accessible by other applications [58]: even if this problem affects the source code of an app, it does not seem directly connected with its energy consumption. Finally, we

12 https://f-droid.org/

focused on method-level code smells only, since for them we can isolate the energy consumption for each single method execution. On the other hand, the analysis of class-level code smells (*e.g.*, most of the Fowler's smells [177]) are particularly challenging because objects (instances of a class) can remain in memory during the execution of the app[13] and, hence, isolating their behavior is more difficult. Therefore, while the analysis of other class-level code smells could be worthwhile, it requires specialized tools and methodologies able to adequately deal with them. It is, therefore, part of our future research agenda.

6.4.2 *Data Extraction*

In this section, we provide an overview of the data extraction process to (i) detect code smell instances, and (ii) measure the energy consumption of the considered apps. For these purposes we relied on two tools that we previously developed and evaluated, *i.e.*, aDoctor (see Section 6.2) and PETrA (see Section 6.3). it is worth noting that we followed a well-defined process already used in previous work [162, 85, 151, 283] to extract the energy profile of the 60 Mobile apps:

- The phone used in the experiment is a factory re-setted `LG Nexus 4` having Android 5.1.1 Lollipop as operating system, and equipped with a 1.5 GHz quad-core Snapdragon S4 Pro processor with 2 GB of RAM, and having a 2100 mAh, 3.8V battery. The choice of the phone is guided by previous research in the field [162, 85, 151, 283] but also because this particular hardware allows to be connected via a data cable, namely a cable where the USB charging can be disabled[14]. Therefore, no energy is transferred over the cable, allowing more stable measurements. Before starting the experiment, we completely re-set the phone in order to avoid bias in the power measurements. Moreover, to limit noise (i) we disabled all the unnecessary apps and processes running on the phone to avoid race conditions, (ii) we did not insert any sim card to avoid asynchronous events, such as incoming messages or calls, and (iii) to avoid energy measurements by sensors and WiFi signal changes we held the phone steady. This setup, available in our online appendix [294], was needed in order to allow PETrA to work in an adequate test environment.

13 https://developer.android.com/reference/android/app/Activity.html
14 http://tinyurl.com/jg7q3lf

- As for the test cases to give as input to PETrA, we automatically generated them using *Monkey*[15], a tool belonging to the Android SDK that produces pseudo-random streams of user events (*i.e.,* clicks, touches, gestures). The choice of Monkey has been guided by recent results [87, 295] showing that this tool achieves the better compromise between coverage and effort needed for the set up. In the experiment, we used the configuration of Monkey suggested by Choudhary *et al.* [87]. Since Monkey may produce events which have the effect of testing external parts of the app under test (*e.g.,* a click may open the status bar), we properly configured Monkey to focus only on the app under analysis. Moreover, in order to not improperly enable/disable smartphone functionality (*e.g.,* WiFi, bluetooth, GPS), we hid the status bar[16].

- The measurements provided by PETrA were repeated 10 times in order to have a more reliable estimation of the energy profiles. Each run costs around five minutes since, as reported by Choudhary *et al.* [87], this is the time needed by Monkey to achieve code coverage convergence. The results achieved after 10 runs (*i.e.,* the joules consumed by the methods in each run) were aggregated using the mean operator. In our case, the mean can be considered significant because the energy consumption of each exercised method tends to remain similar over the 10 runs and, therefore, the distribution of the energy consumption of each method does not contain outliers. In particular, to verify the normality of the distribution of the energy consumptions we adopted the following process: (i) firstly, we normalized the data of each run in the interval $[0, 1]$ using the mix-max algorithm [296] and (ii) we applied the Shapiro-Wilk test [297] with an α threshold for significance set to 0.05. It is important to remark that for this test the null-hypothesis represents the normality of the distribution. In our case, the $\rho - value$ assumed value equals to 0.4921 and thus we could not reject the hypothesis that the sample comes from a population which has a normal distribution, meaning that the variance of the distribution is not large and, therefore, the mean operator can be considered.

  Thus, the final output consisted of a unique value representing the average energy consumed by the methods exercised during the test execution.

15 http://tinyurl.com/gvnxdd3
16 http://tinyurl.com/jxkor7a

Overall, the data extraction process (*i.e.,* the smell detection and the extraction of the energy profiles of 60 apps) took eight weeks.

6.4.3 *Data Analysis*

Once we extracted the code smell instances affecting the apps with aDoctor and the energy profiles using PETrA, we answered **RQ4.3** by comparing the energy consumed by methods with and without code smells. We used the boxplots to provide a graphical comparison between the distributions of the energy consumed by the two groups of methods, i.e., with and without Android-specific smells. To test the statistical significance of the differences (if any) between such distributions we used the Mann-Whitney test [209]. The results are intended as statistically significant at $\alpha = 0.05$. We estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [210] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [210]. Note that the energy consumption of methods might be influenced by other factors such as size, complexity, or level of coverage achieved by the test cases. However, we carefully excluded this hypothesis by performing an additional analysis on the effect of these source code characteristics on the energy consumed by each method (see Section 6.6 for further details).

In the context of **RQ4.3** we do not discriminate the specific type of smell affecting a method (*i.e.,* a method is considered smelly if it contains any type of code smell). A fine grained analysis of the impact of the different smell types on energy consumption is investigated in **RQ4.4**.

To this aim, we split the set of smelly methods identified in the context of the previous research question in 9 subsets, one subset for each code smell type. Then, we compared the distributions of energy consumed by such subsets in order to find statistically significant differences (if any). Note that since our goal is to evaluate the impact of single code smell types, the subsets built in this stage do not contain methods affected by more than one code smell type.

Finally, the goal of **RQ4.5** was to evaluate whether refactoring operations (originally targeted to remove the smells) are actually useful for reducing the energy consumption of the smelly methods. To perform this analysis, we manually analyzed the source code of the methods involved in the design

problem and performed refactoring operations according to the guidelines provided by Reimann *et al.* [58]. In particular, the set of methods to analyze and refactor (4,631) have been distributed among two of the authors, who were responsible for refactoring 2,315 instances each. Each of the involved authors independently refactored the methods assigned to him, by relying on (i) the definitions of refactoring and (ii) the examples provided by Reimann *et al.* [58]. This step required approximatively 450 man-hours.

The output of this phase consisted of the source code where code smells were removed. Then, the two authors involved in the task discussed their activities in order to double-check the consistency of their individual refactoring applications. It is worth remarking that these types of smells can be removed by applying simple program transformations that do not impact the external behavior of the source code. For instance, the previously mentioned *Inefficient Data Structure* can be refactored by replacing the `HashMap<Integer, Object>` with a `SparseArray<Bitmap>` [58]. To be confident that the process did not alter the behavior of the app under analysis, we also re-executed the test cases generated by Monkey (and used to answer our previous RQs) at the end of each refactoring. Once refactored the source code, we repeated the energy measurements using the same design as for **RQ4.3**. Then, we compared the energy consumption obtained using the smelly version of the app with the energy consumption obtained by its corresponding refactored version. Also in this case, we used boxplots and statistical tests for significance (Mann-Whitney test) and effect size (Cliff's Delta).

To have a practical view of the results achieved in the study, we also computed the percentage of the battery charge consumed by methods affected and not affected by code smells. In particular, given the characteristics of the phone used in the experiments (*i.e.*, 2,100 mAh and 3.8V battery), the percentage of battery discharge can be computed using the following formula [298]:

$$f(n) = \left(\frac{V \cdot S \cdot I}{V} \cdot \frac{1}{I \cdot S} \right) \% \tag{6.7}$$

where V is the voltage, I represents the current intensity, and S the time. Our measures are performed by considering the joules consumed by a method. Formally, a joule represents the work required to move an electric charge of one coulomb through an electrical potential difference of one volt ($V \cdot C$). Since a coulomb is the charge transported by a constant current of one ampere in

one second ($I \cdot S$), the numerator of the first part of Equation 6.7 (*i.e.*, $V \cdot S \cdot I$) is exactly the amount of joules consumed by a method. Hence, a method consuming 0.01 J will consume $3 \cdot 10^{-5}\%$ of the total battery charge because Equation 6.7 is instantiated as follow:

$$\left(\frac{0.01}{3.8} \cdot \frac{1,000}{2,100 \cdot 3,600}\right)\% = 3 \cdot 10^{-5}\% \tag{6.8}$$

where the value of 1,000 at the numerator is because the charge is expressed in mAh and not in Ah, and 3,600 is used to convert hours to seconds.

6.5 ANALYSIS OF THE RESULTS

In this section we describe the results achieved to answer our three research questions.

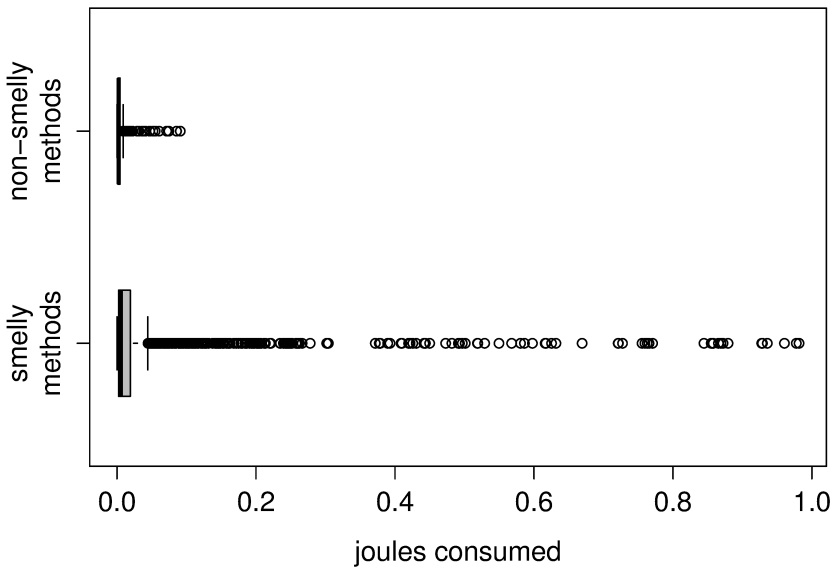


Figure 6.1: Energy Consumption of methods affected and not affected by code smells

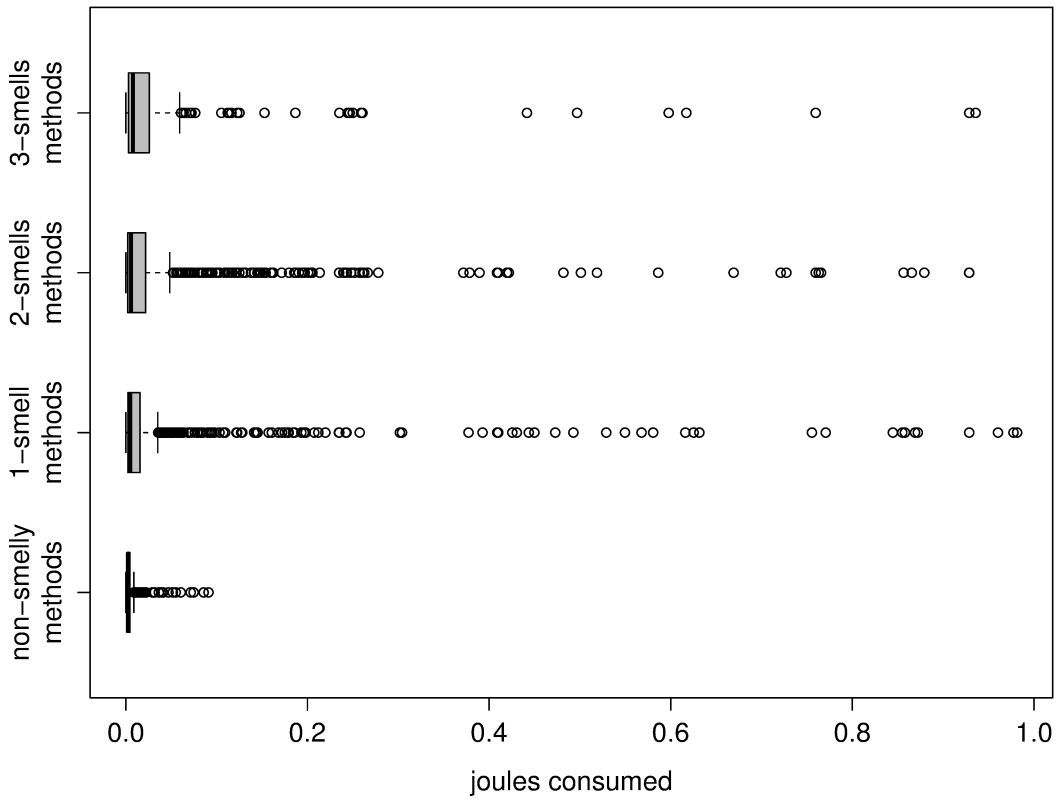


Figure 6.2: Energy Consumption of methods having one, two, and three code smell types

6.5.1 *RQ4.3: What is the Impact of Code Smells on the Energy Consumption of Mobile Apps?*

In the entire dataset, aDoctor detected 6,196 code smell instances. The most frequent ones are: *Member Ignoring Method* (3,104 instances), *Slow Loop* (1,288 instances), and *Data Transmission Without Compression* (564 instances) code smells. The detailed results about the distribution of the studied smells over the 60 apps are reported in our online appendix [294]. Figure 6.1 depicts the boxplot reporting the distribution of the energy consumption of smelly and non-smelly methods. Note that for sake of clarity, we aggregate the results of all the 60 apps. As we can see, the difference in the energy consumed by the methods in the two sets is quite large and, therefore, it seems there exists a relationship between the presence of smells and the energy consumption of methods. Indeed, the median energy consumption of methods affected by code smells (0.0053*J*) is

almost 3 times higher with respect to the median energy consumption of the other methods (0.002*J*). More importantly, the mean energy consumption of smelly methods is 7 times higher than the one of smell-free methods (0.05*J* vs 0.007*J*).

In general, from the boxplot we observe that the methods not affected by any design flaw have an extremely low energy variability, unlike those affected by smells. A representative example can be found in the `a2dpvolume` project[17], an app able to automatically adjusts the media volume when the phone is connected to Bluetooth devices. The method `onCreate` of the `Vol.AppChooser` class creates a thread without closing it and, therefore, it is affected by a *Leaking Thread* smell. In ten runs, PETrA estimates its energy consumption around 0.77 joules on average, namely 385% more than the median consumption of the non-smelly methods (110% more when considering the mean). The results observed above are also confirmed by the Mann-Whitney and Cliff's Delta tests, which reveal that the differences between the two sets of methods are statistically significant (p-value$<$0.001) with a large effect size (d=0.61).

While the analysis carried out until now clearly highlighted a trend in terms of energy consumption for smelly and non-smelly methods, it is important to note that a smelly method may be affected by multiple smell types. For this reason, we performed an additional analysis to verify how the energy consumption of methods varies when considering methods affected by zero, one, two, and three code smell types. Note that we have not found cases where more than three smells co-occurred in the same method. Figure 6.2 reports the results of this analysis. As shown, the higher the number of code smells in a method the higher the median energy consumption. In particular, we found that the median energy consumed by methods affected by three smell types is 0.008*J*, which is twice the energy consumed by methods affected by only one smell (0.004*J*), and 39% times higher than methods affected by two smells (0.0057*J*). The differences are statistically significant (p-value$<$0.001 in both cases) with a small effect size (d=0.14 and d=0.11, respectively).

From a practical point of view, these results lead to the battery discharge percentages shown in Table 6.4. We can observe that each single execution of a method affected by a smell discharges the battery at least $1.3 \cdot 10^{-6}$%, while methods not affected by design flaws normally consume $6.9 \cdot 10^{-8}$% of the battery (*i.e.*, the battery discharges 19 times faster by executing smelly methods).

17 https://play.google.com/store/apps/details?id=a2dp.Vol

Table 6.4: Battery Discharge of Methods Affecting by Different Number of Code Smells

Number of Smells	% of Battery Discharge
0-smells	$6.9 \cdot 10^{-8}\%$
1-smell	$1.3 \cdot 10^{-6}\%$
2-smells	$1.9 \cdot 10^{-6}\%$
3-smells	$2.7 \cdot 10^{-6}\%$

Moreover, methods affected by more than one smell have a higher impact on the battery discharge, up to $2.7 \cdot 10^{-6}\%$ which means 40 times faster battery discharges than non-smelly methods.

Further analyses on the interaction between different smells revealed that most of the times there are four specific smells that co-occur together in methods having higher energy consumption, namely *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Indeed, in 84% of the cases the methods having three smells are affected by a combination of these four specific design flaws. Moreover, the frequent co-occurrence of such smells is also visible when analyzing methods affected by two smells (*Member Ignoring Method* and *Slow Loop* co-occur in 33% of the methods, *Leaking Thread* and *Member Ignoring Method* in 28%, *Leaking Thread* and *Slow Loop* in 11%, *Internal Getter and Setter* and *Slow Loop* in 8%). It is important to note that methods where other types of smells co-occur tend to be more efficient than methods affected by a combination of the four smells mentioned above. From a practical point of view, this result tells us that not all the code smells influence the phenomenon, but rather particular smell types and their combinations can be a cause of energy leaks.

Summary for RQ4.3. Overall, from our *coarse-grained* preliminary analysis we can conclude that the presence of code smells can result in a strong increment of the energy consumption (up to 385% with respect to smell-free methods, with a battery discharge of up to 40 times faster). Moreover, we found that methods affected by more smells consume more than methods not affected by design flaws. However, we observed four code smells, *i.e.*, *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, that generally co-occur together and that may be the cause of energy leaks.

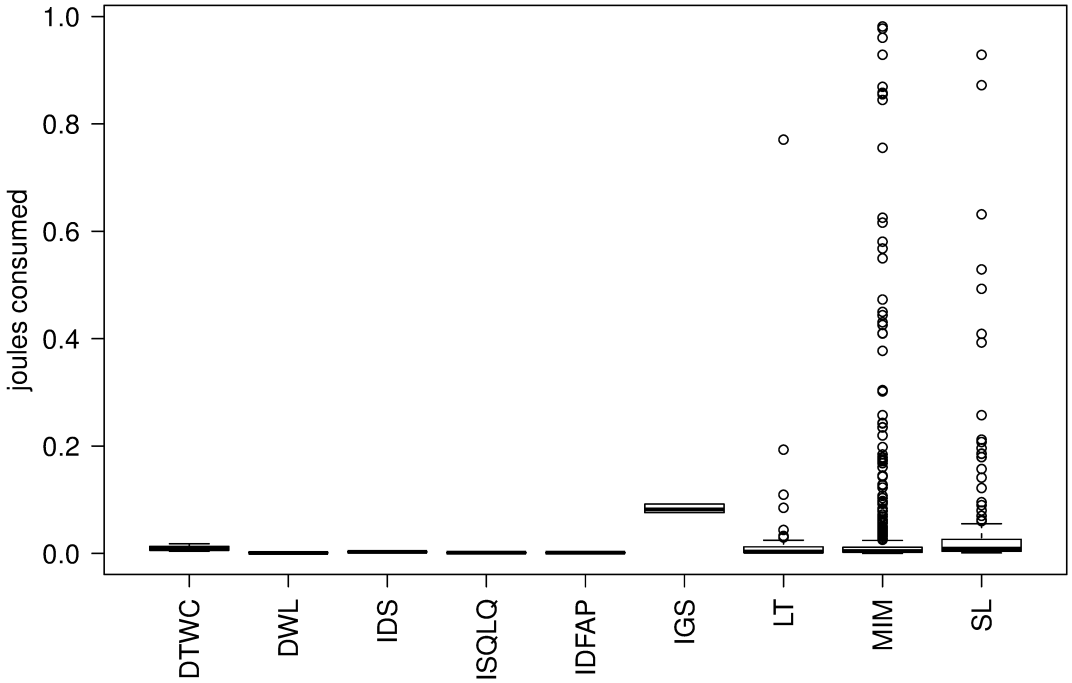


Figure 6.3: Energy Consumption of methods affected by single code smell types

6.5.2 *RQ4.4: Which Code Smells have a Higher Negative Impact on the Energy Consumption of Mobile Apps?*

From **RQ4.3** we have learned that (i) methods affected by smells consume more energy than non-smelly methods, and (ii) methods having a specific combination of smell types tend to be less energy-efficient. To have a *fine-grained* view of the impact of single code smell types on the energy consumption, we further analyzed the methods affected by each code smell. Figure 6.3 reports the boxplots comparing the distribution of the joules consumed by the different smell types considered in this study on the apps in our dataset. The first thing that leaps to the eye is that the four code smells co-occurring more frequently are those with higher impact on the energy consumption of the methods, *i.e.*, *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. This aspect confirms our conjecture that the energy consumption is influenced by specific types of design flaws. The claim is also supported by the percentage of battery discharged by methods affected by different smell types (Table 6.5), where we observed that methods affected by these four code smells

Table 6.5: Average Battery Discharge of Different Types of Code Smells

Smell Type	% of Battery Discharge
DTWC	$8.7 \cdot 10^{-8}\%$
DWL	$7.7 \cdot 10^{-8}\%$
IDS	$7.7 \cdot 10^{-8}\%$
ISQLQ	$7.3 \cdot 10^{-8}\%$
IDFAP	$7.4 \cdot 10^{-8}\%$
IGS	$2.8 \cdot 10^{-6}\%$
LT	$1.1 \cdot 10^{-6}\%$
MIM	$1.4 \cdot 10^{-6}\%$
SL	$2.1 \cdot 10^{-6}\%$

have a much higher discharge rate than other methods (*e.g.*, each execution of a method affected by *Slow Loop* discharges the battery 30 times faster than non-smelly methods). As for the other smells, despite some of them being highly diffused (*e.g., Data Transmission Without Compression*) there is no clear relationship between their presence and energy consumption. In the following we report the detailed results, discussing each of the *most impacting* smells independently.

Internal Getter and Setter. According to the set of best practices for Android programming provided by Google[18], direct field access is up to 7 times faster than invoking a virtual method. During our experiments, we clearly experienced differences in methods affected and not affected by this smell. Indeed, we observed an evident increment of the energy consumption when a method performs a call to getters and setters. On average, such methods consume 40 times more energy than smell free methods (0.08J vs 0.002J). A representative example is the method `AcalDateTime.applyLocalTimeZone` of the `aCal` app[19], which sets the local time zone by reading information about the actual zone using the method `getUTCInstance`, and then sets a new time using the method `setTimeZone`. The energy consumption of this method is, on average, 46% higher than non-smelly methods. The example is generalizable to the other methods affected by this smell and, indeed, the differences between the two distributions (smelly vs non smelly methods) are statistically significant (p-value$<$0.005) with a large effect size (d=0.97).

Leaking Thread. In Android programming a thread is a garbage collector (GC) root. The GC does not collect the root objects and, therefore, if a thread is not adequately stopped it can remain in memory for the entire execution of

18 http://tinyurl.com/j9wucev
19 http://tinyurl.com/8hg67fk

the application, causing an abuse of the memory of the app. As an indirect consequence, the methods involved in this smell have a higher energy consumption with respect to the others. This smell implies a consumption of energy 15 times higher with respect to smell-free methods (0.03J vs 0.002J considering the median). Other than the already mentioned class `a2dp.Vol.AppChooser` (see Section 6.5.1), we found several other cases where this type of smell is involved in energy leaks. For instance, the method `onResume` of the `FillupListActivity` class contained in the `Mileage` app[20] opens a thread in order to model the behavior of the application when it is restarted. The missing closure of the thread implies an energy consumption more than 100% higher than classes not affected by any smell in the app. When applying the Mann-Whitney test, we compared the distributions of energy consumed by methods involved in *Leaking Thread* and methods that correctly close threads, finding the differences to be statistically significant (p-value$<$0.005) with a medium effect size (d=0.35).

Member Ignoring Method. This smell arises when a method does not access any field of the class it belongs to. It should be made `static` in order to speed up the invocations of such method [58]. In our context, we observed that in cases where the method is called several times the increment of energy consumed is 20 times higher with respect to non-smelly methods (0.04J consumed on average in each call compared to the 0.002J consumed by non-smelly methods). An example appeared in the `Alarm Klock` app[21]. The method `onItemClick` of the `alarmclock.ActivityAlarmClock` class is responsible for capturing the taps of the user on the screen when using the app. This method is therefore called in action several times during each app execution. We observed an energy consumption 244% higher compared with other methods. We also observed a similar trend for all the other instances of this type of smell. Indeed, the Mann-Whitney test revealed that the differences are statistically significant (p-value$<$0.005) and the Cliff test reported a medium effect size (d=0.34).

Slow Loop. Methods affected by this smell iterate over collections using the standard version of the `for` loop, while the `for-each` loop can provide a better efficiency [58]. Our results show that methods involved in this type of smell, on average, consume almost 30 times more energy than non-smelly methods (0.06J vs 0.002J). As an example, the `onOptionsItemSelected` method, belonging to

20 http://tinyurl.com/cw3uttu
21 http://tinyurl.com/ngzkv3v

the `BlinkenlightsBattery` class of the `Battery Circle` app[22] is responsible for analyzing all the possible configuration options provided as input by the user. In our experiment, we found the energy consumption of this method to be 80% higher than the other non-smelly methods of the same app. Also in this case, the differences between the two distributions (smelly vs non smelly methods) are significant (p-value$<$0.005) with a small effect size (d=0.25).

Other smells. Besides the smells mentioned above, we observed that other smell types do not have a relevant impact on the energy consumption of mobile applications. The result is quite surprising since previous research by Hecht *et al.* [59] showed that other smells (*e.g., IDFAP*) negatively impact properties such as CPU usage, that are supposed to be somehow related to energy consumption. In our experiment, we found that methods affected by other types of smells (not discussed above) do not statistically differ from non-smelly methods in terms of energy consumption as confirmed by the Mann-Whitney test (p-values always higher than 0.27). Among all other design flaws, it is worth discussing the case of *Inefficient Data Structure*: recent findings by Hasan *et al.* [172] demonstrated how the application of a wrong type of data structure increases the energy consumption by up to 300%. This seems to be in contrast with respect to our findings. However, it is important to note that Hasan *et al.* have conducted their experiments in the context of larger and more complex applications, such as Java libraries, while in the context of mobile apps we observed that this type of problem is generally poorly diffused (*i.e.*, only 0.6% of the methods analyzed are affected by this smell) and, therefore, not particularly relevant.

Summary for RQ4.4 Our analysis reveals that there exist four *energy-smells*, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*, which significantly impact the energy consumption of methods in a mobile app.

6.5.3 *RQ4.5: Does the Refactoring of Code Smells Positively Impact the Energy Consumption of Mobile Apps?*

The results of the previous research question pointed out the existence of four specific Android-specific smells having a higher impact on the energy

22 http://tinyurl.com/o33ms7d

Table 6.6: Energy consumed by methods before and after the application of refactorings. "R-" prefix stands for Refactored.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
DTWC	0.004	0.006	0.008	0.010	0.013	0.018
R-DTWC	0.004	0.006	0.008	0.010	0.013	0.018
DWL	0.001	0.001	0.001	0.001	0.002	0.003
R-DWL	0.001	0.001	0.001	0.001	0.002	0.003
IDS	0.002	0.002	0.003	0.003	0.003	0.004
R-IDS	0.002	0.002	0.003	0.003	0.003	0.004
ISQLQ	0.001	0.001	0.001	0.001	0.001	0.002
R-ISQLQ	0.001	0.001	0.001	0.001	0.001	0.002
IDFAP	0.001	0.001	0.001	0.001	0.001	0.002
R-IDFAP	0.001	0.001	0.001	0.001	0.001	0.002

Table 6.7: Energy consumed by methods before and after the application of refactorings. "R-" prefix stands for Refactored.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
IGS	0.076	0.076	0.082	0.083	0.092	0.092
R-IGS	0.001	0.001	0.009	0.016	0.024	0.024
LT	0.01	0.02	0.03	0.077	0.013	0.77
R-LT	0.001	0.001	0.003	0.009	0.009	0.019
MIM	0.001	0.002	0.004	0.04	0.028	0.981
R-MIM	0.0001	0.002	0.018	0.02	0.019	0.038
SL	0.001	0.006	0.0119	0.056	0.026	0.929
R-SL	0.001	0.004	0.0114	0.010	0.014	0.018

consumption, *i.e.*, *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, while the remaining five do not seem to influence the phenomenon. To further verify the validity of the findings achieved so far, we evaluated to what extent the refactoring of the considered smells has an effect on the reduction of the energy consumption of the smelly methods. To this aim, we manually refactored the 4,631 smelly methods affected by only one of the smells considered. Note that in this research question we have not considered the methods affected by more smells since we are interested in understanding the effect of the single refactoring operations on the energy consumption of such methods.

Table 6.6 reports the statistics on the energy consumption of the methods affected by *Data Transmission Without Compression*, *Durable Wakelock*, *Inefficient Data Structure*, *Inefficient SQL Query*, and *Inefficient Data Format and Parser*, respectively, before and after the refactoring. As it is possible to observe, the refactoring operations applied do not have any effect on the resulting energy consumption: this definitively confirms the previous findings, showing that (i) these smells poorly influence the phenomenon, and (ii) the application of

refactoring is not worthwhile to improve the energy efficiency of such smelly methods.

On the other hand, Table 6.7 reports the statistics on the energy consumption of methods affected by the four smells identified as related to the phenomenon in the previous research questions, *i.e., Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, respectively, before and after the application of the refactorings. In all the comparison between smelly and refactored versions a recurring pattern can be observed: when the code smells affecting the methods are removed, the energy consumption of such methods decrease.

For methods affected by *Internal Getter and Setter* we can observe that the associated solution, namely the *Direct Field Access* refactoring [58], reverses the negative effect of the smell by reducing the energy consumption by almost 9 times with respect to the smelly methods (the median energy consumption of smelly methods is 0.082 joules, while the median of the distribution of the refactored methods is $0.009J$). Another important observation is that the energy consumed by these methods after the refactoring is comparable with the energy consumed by methods that were not affected by any smell. This means that in this case the refactoring is able to remove the negative effects of the smell. The result is even more evident when considering the case of the previously mentioned `applyLocalTimeZone` method of the `aCal` app. By directly using the fields reporting actual zone and current time, the method consumes, on average, 0.012 joules, namely 683% less than the non-refactored version ($0.094J$). The magnitude of the differences between smelly and refactored methods is large (d=0.67) and statistically significant (p-value$<$0.005).

Similar results can be observed for the *Leaking Thread* smell, with the median energy consumption equal to $0.03J$ when methods are affected by the smell, compared to $0.003J$ when the methods are refactored. Hence, the usefulness of the *Introduce Run Check Variable* refactoring [58] is quantifiable in reducing 900% of the energy consumption of methods previously affected by a *Leaking Thread*. Also in this case, we discuss the differences between the smelly and the refactored version of the `AppChooser.onCreate` method of the `a2dpvolume` class, mentioned in the context of $\mathbf{RQ}_1$. From an average of 0.77 joules previously obtained, we observed that the energy consumed by the refactored version is $0.01J$, namely 77 times lower. The differences are statistically significant (p-value$<$0.001) with a medium effect size (d=0.35). Furthermore, it is important to note that the refactored method consumes considerably less than most of the

other non-smelly methods. Indeed, since the median energy consumption of non-smelly methods is 0.02J, the method, when refactored, falls into the first quartile of the distribution of non-smelly methods.

The *Introduce Static Method* refactoring [58] needed to remove the *Member Ignoring Method* smell turned out to be strongly impacting the energy efficiency of source code methods. Indeed, all the outliers present in previously shown smelly distribution (Figure 6.3) disappear after the removal of the smells. On average, the energy consumption of methods changes from a mean of 0.04J to 0.02J, leading to an improvement of exactly 50%. Although the differences are statistically significant (p-value$<$0.001) with a medium effect size (d=0.46), it is worth considering that in this case the median energy consumption of the distribution of the refactored methods is higher than the one related to the smelly version of methods (0.018J vs 0.004J). This result appears to be not in line with the discussion up until now. However, this is just a reflection of the large number of outliers previously affecting the distribution. Indeed, analyzing this aspect more in depth, we observed that most of the outliers, once they are refactored, have an energy consumption that is slightly higher than the median of the non-smelly methods: as a consequence, the resulting distribution is more condensed and the statistical descriptors are higher than before. One example is the `ActivityAlarmClock.onItemClick` method of the `Alarm Klock` app, described before as a method consuming 244% more than non-smelly methods. Once refactored, we observed a strong improvement of its energy efficiency (in the ten runs the consumption decreases from 0.97 to 0.03 joules, namely 33 times lower), leading to an average consumption slightly higher than non-smelly methods.

The discussion about the *Slow Loop* smell is similar to the one of the other energy smells. Indeed, the application of the *Enhanced For-Loop* refactoring improves, on average, the energy efficiency by 180% with respect to energy consumed by the smelly methods. Also in this case, all the outliers disappear after applying the refactoring and generally the distribution is much less scattered than before. The differences are statistically significant (p-value$<$0.005), yet, with small effect size (d=0.12). Finally, it is worth observing that the consumption of refactored methods remains higher than the energy consumption of non-smelly methods (0.0114J vs 0.002J). While the motivations behind this aspect may be related to several other factors that are not the object of this study, in our context it is important to point out the huge benefit provided by the

refactoring to the energy efficiency of such methods. The example discussed in $\mathbf{RQ}_2$ about the method `BlinkenlightsBattery.onOptionsItemSelected` of the `Battery Circle` app is significant in order to understand the effect of refactoring *Slow Loop* instances. Indeed, once refactored the method passed from a consumption of 0.87 joules to 0.01J (-99%), *i.e.,* it completely changed its energy profile.

Table 6.8: Average Battery Discharge of Methods Before and After Refactored

Smell Type	% of Battery Discharge Before	% of Battery Discharge After
IGS	$2.8 \cdot 10^{-6}$%	$3.1 \cdot 10^{-7}$%
LT	$1.1 \cdot 10^{-6}$%	$1.1 \cdot 10^{-7}$%
MIM	$1.4 \cdot 10^{-6}$%	$6.2 \cdot 10^{-7}$%
SL	$2.1 \cdot 10^{-6}$%	$3.9 \cdot 10^{-7}$%

To broaden the scope of the discussion, the quantity of source code that needs to be refactored to remove the code smells is small. For instance, the *Introduce Static Method* only requires the addition of the keyword `static` to the signature of the method, together with other small changes to adapt method calls over all the project (*i.e.,* modifying external method calls in a way they call a static method). However, we observed that such *small changes in the source code result in a big change in the energy consumed* by the methods involved. The results are confirmed by the analysis of the percentage of battery discharge of methods before and after being refactored, as reported in Table 6.8. As we can see, all the types of refactoring result in a significantly lower energy drain. In our opinion, this is a key result since it reveals the actual cost-effectiveness of refactoring of *Android-specific* code smells

Summary for RQ4.5. Refactoring code smells has a key role in improving the energy efficiency of source code methods. On average, we found that the refactored versions of methods previously affected by the four smells actually influencing the energy efficiency, *i.e., Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, consume almost 90% less than methods affected by smells. We also found several cases where refactoring helps in reducing the energy consumption up to 870%. Based on these results, we observe that refactoring is a powerful activity that should be applied by mobile developers.

6.6 THREATS TO VALIDITY

The main threats related to the relationship between theory and observation (*construct validity*) are due to imprecisions in the measurements we performed. Above all, we relied on the aDoctor tool to detect candidate code smell instances. We are aware that our results can be affected by the presence of false positives and false negatives. However, the performance of the tool has been evaluated on 18 apps considered in the study, finding that aDoctor has a precision of 98% and a recall of 98%. These results allow us to be confident about the code smell instances found over all the considered apps. In addition, we replicated all the analysis performed to answer our research questions by just considering the 18 apps where the smells have been validated. The results achieved in this analysis (available in our replication package [294]) are in line with those obtained in our chapter, confirming all our findings.

On the other hand, we measured energy consumption using our tool PETrA. As explained in Section 6.3, we empirically evaluated the accuracy of the approach on 54 mobile apps comparing the power estimation of the tool with the oracle provided by Linares-Vasquez *et al.* [85]. The validation revealed that in 95% of the cases the estimations of our tool are within 5% of the actual values. Therefore, we believe that the data provided by the tool are consistent and close enough to the actual energy consumption. Moreover, in case of differences between the estimations provided by PETrA and the hardware-based tool, these would be consistent for both methods affected by smells and refactored methods, so that the error would not invalidate our findings. We acknowledge, however, that some measures could have been more suitable to evaluate the estimations provided by PETrA[299, 300, 301, 302, 303, 304]. We aggregated the results given by PETrA using the mean operator. As highlighted in section 6.4, in our case, the mean can be considered significant because the energy consumption of each exercised method tends to remain similar over the 10 runs and, therefore, the distribution of the energy consumption of each method does not contain outliers. Despite this, in order to be more confident about our findings, we repeated the experiment by aggregating the energy consumption using the sum, *i.e.,* the final output was a unique value representing the sum of the energy consumption of the methods exercised during the 10 runs. The results achieved from this analysis are available in our online appendix [294] and similar to those reported in Section 6.5. Furthermore, it is worth noting that

to measure the consumption of each method we analyzed the energy consumed by the application within a certain time frame. Although this is clearly an approximation (*e.g.,* the presence of more threads running simultaneously might bias the measurements), the same procedure has been performed in several previous research papers [163, 164, 165, 166, 168], which, however, also include additional approximations due to tail energy: in this sense, we believe that our measurement process is still more precise than previous work, and thus we are confident about the results provided in this chapter.

To study the effect of refactoring on energy efficiency ($\mathbf{RQ}_3$), we manually refactored the source code. The procedure involved two of the authors who carefully followed the guidelines provided by Reimann *et al.* [58]. At the end of the first stage of refactoring, the authors involved in the task opened a discussion aimed at double checking the refactoring operations individually performed. While we cannot exclude imprecision and some degree of subjectiveness (mitigated by the discussion) in the way we refactor the source code, it is important to note that we re-executed the same test cases generated by Monkey to (i) double-check the validity of the refactoring operations applied, and (ii) control that the external behavior of the refactored methods was not changed after the refactoring.

Threats related to the relationship between the treatment and the outcome (*conclusion validity*) are represented by the analysis methods exploited in our study. We discuss our results by presenting descriptive statistics and using proper statistical tests in order to assess the significance and the magnitude of our findings. In addition, the practical relevance of the differences observed in terms of energy consumption is highlighted by effect size measures.

Threats to *internal validity* concern factors that could influence our observations. We are aware that in principle we cannot claim a direct cause-effect relation between the presence of code smells and the energy consumption of methods. However, on the one hand the impact of code smells is firstly demonstrated by the fact that we observed a strong improvement of the energy efficiency when such smells were refactored (**RQ3**). On the other hand, we performed an additional analysis to verify whether other factors could have influenced our results. Specifically, we re-run our analysis by considering the energy consumption of methods having different (i) size, (ii) complexity, and (iii) level of test coverage. In particular:

1. We grouped together methods with similar size by considering their distribution in terms of size. Specifically, we computed the distribution of the lines of code of methods. This step results in the construction of (i) the group composed of all the methods having a size lower than the first quartile of the distribution of all the size of the methods, *i.e., small* size; (ii) the group composed of all the methods having a size between the first and the third quartile of the distribution, *i.e., medium* size; and (iii) the group composed of the methods having a smell and having a size larger than the third quartile of the distribution of all the method sizes, *i.e., large* size;

2. We computed the energy consumption for each method belonging to the three groups, in order to investigate whether larger methods consume more with respect to smaller methods.

The experiment has been repeated considering the McCabe cyclomatic complexity [305] and the coverage obtained by the test cases ran over the methods analyzed [306] as measures to split methods in *small*, *medium*, and *large* sets. We reported the achieved results in our online appendix [294]. We observed that such characteristics are not strongly correlated with the energy consumption of the methods.

Finally, regarding the generalization of our findings (*external validity*) we evaluated the impact of 9 code smell types on the energy consumption of 60 mobile applications. However, further studies aiming at replicating our work on larger datasets are desirable and part of our future agenda.

6.7 CONCLUSION

This chapter presents a large scale empirical investigation taking into account the role of 9 *Android-specific* code smells [58] on the energy consumption of mobile apps. The goal is to assess whether it is possible to focus the energy testing on components containing *Android-specific* code smells. Starting from a *coarse-grained* analysis in which we analyzed if source code methods affected by smells have higher energy consumption than non-smelly methods, we then refined our analyses in order to investigate which are the specific code smell types that influence the phenomenon more. Finally, we evaluated whether refactoring operations applied to remove code smells help in substantially

improving the efficiency of mobile applications. The achieved results provide valuable findings and insights for the research community.

Lesson 1. *Code smells have a strong impact on the energy efficiency of source code methods.* Specifically, methods affected by smells consume up to 385% more with respect to smell-free methods. Even if the interaction between them results in lower efficiency, we found four particular smell types that frequently co-occur and that impact energy consumption more than others, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*. These aspects highlight the importance of investing (1) in studying more in depth the dynamics behind *Android-specific* code smells and (2) in developing tools that prevent their introduction.

Lesson 2. *Refactoring code smells is a key activity to improve energy efficiency.* We found that the energy consumption of refactored methods is reduced by up to 900% with respect to smelly methods. Therefore, we empirically demonstrated how small changes applied to remove code smells result in applications that are much more efficient in terms of energy consumption. Approaches and tools able to support mobile developers in automatically refactoring the source code represent a *must* for future research in the field.

These lessons represent the main input for our future research agenda on this topic, mainly focused on designing and developing a new generation of code quality-checkers and refactoring tools, other than corroborating our results by studying the impact of other smells, *e.g.,* Fowler's smells [177] on energy efficiency.

7

CONCLUSIONS, LESSONS LEARNT, AND OPEN ISSUES

Software testing is widely recognized as an essential part of any software development process, representing however an extremely expensive activity. Indeed, new methods and tools able to better allocating the developers effort are needed in order to increment the system reliability and to reduce the testing costs. The resources available should be allocated effectively upon the portions of the source code that are more likely to contain bugs. In this thesis we focus on three activities able to focus and prioritize the testing focus, specifically (i) bug prediction, (ii) test case prioritization, and (iii) detection of code smell able to fix energy smell. Indeed, despite the effort devoted by the research community in the last decades through the conduction of empirical studies and the devising of new approaches led to interesting results, in the context of our research we highlighted some aspects that might be improved and proposed empirical investigations and novel approaches.

In the context of bug prediction, we devised two novel metrics (Chapter 3), namely the DEVELOPER'S STRUCTURAL AND SEMANTIC SCATTERING. The conjecture behind the proposed metrics is that high levels of structural and semantic scattering make the developer more error-prone. The results of our empirical study show the superiority of our model with respect to (i) the model exploiting the number of developers working on a code component as a predictor, (ii) the model using the developers' focus metric by Posnett *et al.*, (iii) the model built on top of product metrics. The two scattering metrics are highly complementarity with the metrics used by the alternative prediction models. Thus, we devised a "hybrid" model providing an average boost in prediction accuracy.

We proposed a novel adaptive prediction model, coined as ASCI (Chapter 4), which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class. Then we experimented ASCI in the contexts of within-project and cross-project bug prediction. We analyzed seven ensemble classifiers belonging to six categories on 21 software systems. We found that the problem of cross-project bug

prediction is still far from being solved. The use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers, but in general the Validation and Voting and ASCI techniques should be preferred among other ensemble methodologies. When turning our attention on the combination between local learning and ensemble classifiers, we did not observe major differences; indeed, the statistical analyses revealed that local and global models are mostly equivalent. However, we found ASCI to be the only technique that is effective in exploiting local learning for reducing data heterogeneity and improving its prediction capabilities

With respect to the test case prioritization problem, we proposed (Chapter 5) HGA, a genetic algorithm based on the hypervolume indicator. We provided an extensive evaluation of Hypervolume-based and state-of-the-art approaches when dealing with up to five testing criteria. Our results suggest that the solution (test ordering) produced by HGA is more cost-effective than the solution generated by Additional Greedy and the Pareto optimal solution achieved by NSGA-II. In terms of efficiency, HGA is much faster than NSGA-II and Additional Greedy, and its efficiency does not decrease as the size of the software program and of the test suite increase. Moreover, when dealing with more than three criteria, our results show that HGA is not only more or equally effective than the state-of-the-art many-objective algorithms, but it is also up to 16 times more efficient.

To cope with energy efficiency issues of mobile applications and thus reducing the related effort, we developed PETrA and aDoctor. PETrA is a novel tool for extracting the energy profile of mobile applications, while aDoctor is a code smell detector able to identify 15 Android-specific code smells. We provided a deeper investigation (Chapter 6) to determine (i) to what extent code smells affecting source code methods of mobile applications influence energy efficiency, and (ii) whether refactoring operations applied to remove them directly improve the energy efficiency of refactored methods. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann *et al.* in the context of 60 Android apps belonging to the dataset provided by Choudhary *et al.*. The results of our study highlight that methods affected by code smells consume up to 385% more energy than methods not affected by any smell. A fine-grained analysis reveals the existence of four specific energy-smells, namely *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Finally, we also shed

light on the usefulness of refactoring as a way for improving energy efficiency by code smell removal. Specifically, we found that it is possible to improve the energy efficiency of source code methods by up to 900% through refactoring code smells.

Finally, we ensure the technological transfer and the replication of the results by making a number of tools, replication packages, and datasets publicly available.

7.1 CONCLUSIONS

The answers to the high-level research questions are summarizable as follow.

- **Chapter 3**: *To what extent developer's scattering metrics are able to improve a bug prediction model based on state-of-the-art metrics?* Developer's scattering metrics showed quite high accuracy in identifying buggy classes. Among the considered systems their accuracy ranges between 53% and 98%, while the F-measure between 47% and 98%. Moreover, DCBM performs better than the baseline approaches, demonstrating their superiority in correctly predicting buggy classes. By combining the eleven predictors exploited by the five prediction models subject of our study it is possible to obtain a boost of prediction accuracy up to +5% with respect to the best performing model (*i.e.*, DCBM) and +9% with respect to the best combination of baseline predictors. Also, the top five "hybrid" prediction models include at least one of the predictors proposed in this work (*i.e.*, the structural and semantic scattering of changes) and the best model includes both. By combining the eleven predictors exploited by the five prediction models subject of our study it is possible to obtain a boost of prediction accuracy up to +5% with respect to the best performing model (*i.e.*, DCBM) and +9% with respect to the best combination of baseline predictors. Also, the top five "hybrid" prediction models include at least one of the predictors proposed in this work (*i.e.*, the structural and semantic scattering of changes) and the best model includes both.

- **Chapter 4**: *To what extent a technique able to select a classifier based on the characteristics of the class is able to out-perform state-of-the-art classifiers?* In the context of within-project bug prediction, the use of an ensemble classifier does not guarantee better prediction performances with respect to the

best stand-alone classifier (*e.g.,* NAIVE BAYES). We confirm that the models based on VALIDATION AND VOTING are able to achieve slightly better results, but the obtained improvement is not statistically significant with respect to other ensemble techniques, such as RANDOM FOREST and ASCI. None of the cross-project models experimented is able to exceed 25% of MCC (on average), meaning that the problem of identifying buggy classes using external sources of information is still far from being solved. Furthermore, the use of ensemble techniques does not provide evident benefits with respect to stand-alone state-of-the-art classifiers. Indeed, the models based on NAIVE BAYES or using it as weak learner (*e.g.,* NBBOOSTING and NBBAGGING) are able to achieve the best performances. ASCI, instead, does not work properly in a cross-project setting. Local learning is often not able to improve the performances of bug prediction models. The only exception is represented by ASCI, which has better performances with respect to those achieved by global models. The statistical analysis conducted, however, highlighted how local and global models are mostly equivalent.

- **Chapter 5**: *What are the cost-effectiveness, the efficiency, and scalability of a genetic algorithm algorithm based on the hypervolume, compared to state-of-the-art test case prioritization techniques?* In terms of cost-effectiveness, `HGA` outperforms `Additional Greedy` in most of the cases. `HGA` and `NSGA-II` produce test permutations that are comparable in terms of $APFD_c$. However, the Pareto front generated in each run by `NSGA-II` presents a large variance for the $APFD_c$ scores. Thus, its cost-effectiveness strongly depends on how software testers pick a non-dominated solution from the front. Finally, most of the solutions generated by `NSGA-II` are dominated by `HGA` with respects to the AUC-based metrics. Looking at the efficiency, `Additional Greedy` is strictly related to the number of test cases. Indeed, when the number of test cases is high, `HGA` is much faster than `Additional Greedy`. Looking at `NSGA-II`, `HGA` improves the efficiency of test case prioritization in all the considered programs up to 14 times. Regarding the scalability, `HGA` outperforms in most cases two well known many-objective algorithm (*e.g.,* `GDE3` and `MOEA-D/DE`). Moreover, The single solution provided by `HGA` is better than many of the Pareto optimal solutions generated by the many-objective algorithms, providing better compromise in the objectives space. `HGA` is much more efficient, on average, with respect these algorithms by 8

and 9 times, respectively. Moreover, HGA is able to scale up better as the number of test cases to prioritize increases.

- **Chapter 6**: *To what extent Android-specific code smells can be used to focus energy testing of mobile apps?* Our *coarse-grained* preliminary shows that the presence of code smells can result in a strong increment of the energy consumption (up to 385% with respect to smell-free methods, with a battery discharge of up to 40 times faster). Moreover, we found that methods affected by more smells consume more than methods not affected by design flaws. Our analysis reveals that there exist four *energy-smells*, *i.e.*, *Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*, which significantly impact the energy consumption of methods in a mobile app. Refactoring code smells has a key role in improving the energy efficiency of source code methods. On average, we found that the refactored versions of methods previously affected by the four smells actually influencing the energy efficiency, *i.e.*, *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, consume almost 300% less than methods affected by smells. We also found several cases where refactoring helps in reducing the energy consumption up to 870%. Based on these results, we observe that refactoring is a powerful activity that should be applied by mobile developers.

7.2 LESSONS LEARNT

The main lessons learnt from this thesis can be summarizable as follow.

- **Chapter 3 - Lesson 1.** *Developer factors should be exploited to achieve better bug prediction models.* This thesis shows in an effective way how developer factors can improve bug prediction models. Specifically, we defined two metrics that consider the amount of code components a developer modifies in a given time period and how these components are spread structurally (structural scattering) and in terms of the responsibilities they implement (semantic scattering). The achieved results showed the superiority of our model and its high level of complementarity with respect to the considered competitive techniques. Indeed, developer factors should be widely considered when building bug prediction models.

- **Chapter 4 - Lesson 1.** *Structural characteristics of the classes should be considered to choose the best classifier in bug prediction.* In this thesis we presented ASCI, an ensemble technique that analyzes the structural characteristics of classes to decide which classifier to use. In the context of within-project bug prediction, our model achieves higher performances than the ones achieved by the best stand-alone model. Moreover, in case of high variability among the predictions provided by the classifiers the majority of them might wrongly classify the bug-proneness of a class, negatively influencing the performances of techniques which combine the output of different classifiers (*i.e., Validation and Voting*).

- **Chapter 4 - Lesson 2.** *Cross-project bug prediction is still impracticable.* The problem of identifying buggy classes using external sources of information is still far from being solved. Furthermore, the use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers. However, among the considered ensemble techniques we confirm that the Validation and Voting and ASCI techniques should be preferred.

- **Chapter 4 - Lesson 3.** *Local learning does not improve bug prediction models.* When turning our attention on the combination between local learning and ensemble classifiers, we did not observe major differences; indeed, the statistical analyses revealed that local and global models are mostly equivalent. However, we found ASCI to be the only technique that is effective in exploiting local learning for reducing data heterogeneity and improving its prediction capabilities.

- **Chapter 4 - Lesson 4.** *Cross-project models are less accurate but more robust with respect to within-project models.* We found some key findings when comparing cross- and within-project models. In the first place, the latter are more precise than cross-project models, independently from the training strategy (global vs local). Nevertheless, the use of ensemble classifiers in the context of within-project models does not guarantee better prediction performances with respect to models relying on stand-alone classifiers. On the other hand, we also observed that cross-project models show an higher ability than within-project ones to discriminate the buggy components when the discrimination threshold is varied. This means that the two strategies might be complemented in order to take advantages of the pros of each of them.

- **Chapter 5 - Lesson 1.** *The hypervolume indicator can be used as an effective and efficient fitness function for genetic algorithms.* We proposed a hypervolume-based genetic algorithm to improve multi-criteria test case prioritization. Specifically, we use the concept of hypervolume [13], which is widely investigated in many-objective optimization, to generalize the traditional Area Under Curve (AUC) metrics used in previous work on test case prioritization.

- **Chapter 6 - Lesson 1.** *Code smells have a strong impact on the energy efficiency of source code methods.* Specifically, methods affected by smells consume up to 385% more with respect to smell-free methods. Even if the interaction between them results in lower efficiency, we found four particular smell types that frequently co-occur and that impact energy consumption more than others, i.e., *Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*. These aspects highlight the importance of investing (i) in studying more in depth the dynamics behind *Android-specific* code smells and (ii) in developing tools that prevent their introduction.

- **Chapter 6 - Lesson 2.** *Refactoring code smells is a key activity to improve energy efficiency.* We found that the energy consumption of refactored methods is reduced by up to 900% with respect to smelly methods. Therefore, we empirically demonstrated how small changes applied to remove code smells result in applications that are much more efficient in terms of energy consumption. Approaches and tools able to support mobile developers in automatically refactoring the source code represent a must for future research in the field.

7.3 OPEN ISSUES

Despite the effort devoted by the research community and despite the advances proposed in this thesis, reducing the testing effort still proposes a number of open issues and challenges that need to be addressed in the future.

- **Chapter 3** *Analysis of developers' bug inducing behaviors.* The role of developer-related factors in the bug prediction field is still a partially explored area. A deeper investigation of the factors causing scattering to developers, and negatively impacting their ability of dealing with code change tasks is

needed. Hence, we plan to reach such an objective by performing a large survey with industrial and open source developers.

- **Chapter 4** *Combining cross-project and within-project bug prediction.* Our results suggest that a combination of cross-project and within-project strategies might be exploited to further improve bug prediction capabilities.

- **Chapter 5** *Solving other test case optimization problems using the hypervolume indicator.* The proposed HGA could be applied also for other test case optimization problems, such as Test Suite Minimization and Test Case Selection. Moreover, we plan to perform a new empirical study to investigate which testing criteria are more capable to discover new faults.

- **Chapter 6** *Design and development of new tools for improving energy efficiency.* Lessons 6 and 7 represent the main input for our future research agenda on energy efficiency, mainly focused on designing and developing a new generation of code quality-checkers and refactoring tools, other than corroborating our results by studying the impact of other smells, *e.g.*, *Fowler's smells* on energy efficiency.

BIBLIOGRAPHY

[1] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[2] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 179–190, ACM, 2015.

[3] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[4] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[5] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 751–761, Oct 1996.

[6] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 897–910, 2005.

[7] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switchess," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, p. 886–894, 1996.

[8] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, (New York, NY, USA), pp. 580–586, ACM, 2005.

[9] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, (Washington, DC, USA), pp. 9–, IEEE Computer Society, 2007.

[10] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 33, 2014.

[11] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu, "Mutation-aware fault prediction," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 330–341, ACM, 2016.

[12] A. N. Taghi M. Khoshgoftaar, Nishith Goel and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Software Reliability Engineering*, pp. 364–371, IEEE, 1996.

[13] J. S. M. Todd L. Graves, Alan F. Karr and H. P. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[14] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*, (Vancouver, Canada), pp. 78–88, IEEE Press, 2009.

[15] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

[16] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, (New York, NY, USA), pp. 19:1–19:10, ACM, 2010.

[17] R. Moser, W. Pedrycz, and G. 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*, ESEM '08, (New York, NY, USA), pp. 309–311, ACM, 2008.

[18] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?," in *Proceedings of the 7th International Conference*

on Predictive Models in Software Engineering, Promise '11, (New York, NY, USA), pp. 2:1–2:8, ACM, 2011.

[19] T. Lee, J. Nam, D. Han, S. Kim, and H. P. 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*, pp. 311–321, ACM, 2011.

[20] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Developer micro interaction metrics for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1015–1035, 2016.

[21] W. P. Raimund Moser and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *International Conference on Software Engineering (ICSE)*, ICSE '08, pp. 181–190, 2008.

[22] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), pp. 452–461, ACM, 2006.

[23] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, p. 531–577, 2012.

[24] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't program on fridays! how to locate fix-inducing changes," in *Proceedings of the 7th Workshop Software Reengineering*, May 2005.

[25] L. T. Jon Eyolfso and P. Lam, "Do time of day and developer experience affect commit bugginess?," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pp. 153–162, 2011.

[26] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 491–500, 2011.

[27] E. J. W. J. Sunghun Kim and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[28] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pp. 4–14, ACM, 2011.

[29] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pp. 104–113, 2012.

[30] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 452–461, IEEE Press, 2013.

[31] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[32] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the International Conference on Software Engineering*, pp. 789–800, IEEE, 2015.

[33] A. Panichella, R. Oliveto, and A. D. Lucia, "Cross-project defect prediction models: L'union fait la force," in *Proceedings of IEEE CSMR-WCRE*, pp. 164–173, 2014.

[34] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1, pp. 1–39, 2010.

[35] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Transactions on Software Engineering*, vol. 36, pp. 852–864, Nov 2010.

[36] A. T. Mısırlı, A. B. Bener, and B. Turhan, "An industrial case study of classifier ensembles for locating software defects," *Software Quality Journal*, vol. 19, no. 3, pp. 515–536, 2011.

[37] T. Wang, W. Li, H. Shi, and Z. Liu, "Software defect prediction based on classifiers ensemble," *Journal of Information & Computational Science*, vol. 8, no. 16, pp. 4241–4254, 2011.

[38] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of International Conference on Software Engineering*, pp. 481–490, IEEE, 2011.

[39] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[40] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2013.

[41] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. 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*, p. 46, ACM, 2016.

[42] A. B. B. Burak Turhan, Tim Menzies and J. S. D. Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[43] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th ACM SIGSOFT ESEC/FSE*, pp. 91–100, ACM, 2009.

[44] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[45] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Transactions on Software Engineering*, 2017.

[46] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Advances in neural information processing systems*, pp. 841–848, 2002.

[47] F. Ferrucci, E. Mendes, and F. Sarro, "Web effort estimation: the value of cross-company data set compared to single-company data set," in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, pp. 29–38, ACM, 2012.

[48] L. Minku, F. Sarro, E. Mendes, and F. 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*, pp. 1–10, IEEE, 2015.

[49] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro, "From function points to cosmic-a transfer learning approach for effort estimation," in *International Conference on Product-Focused Software Process Improvement*, pp. 251–267, Springer, 2015.

[50] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, pp. 67–120, Mar. 2012.

[51] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[52] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[53] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[54] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pp. 102–112, ACM, 2000.

[55] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," *Software Engineering Notes*, vol. 25, pp. 102–112, 2000.

[56] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Softw. Engg.*, vol. 20, pp. 374–409, Apr. 2015.

[57] A. Smith, "Smartphone ownership," 2013.

[58] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," 2014.

[59] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, (New York, NY, USA), pp. 59–69, ACM, 2016.

[60] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?," *Software Quality Journal*, pp. 1–28, 2017.

[61] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Proceedings of the IEEE Annual Computer Software and Applications Conference*, vol. 2, pp. 264–269, IEEE, 2015.

[62] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler, "Theory of the hypervolume indicator: optimal μ-distributions and the choice of the reference point," in *Proceedings of SIGEVO workshop on Foundations of Genetic Algorithms*, pp. 87–102, ACM, 2009.

[63] D. Di Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia, "On the role of developer's scattered changes in bug prediction," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 241–250, IEEE, 2015.

[64] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, 2017.

[65] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[66] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Hypervolume-based search for test case prioritization," in *Proceeding of the Symposium on Search-Based Software Engineering*, vol. 9275 of *Lecture Notes in Computer Science*, pp. 157–172, Springer, 2015.

[67] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Prioritization genetic algorithm guided by the hypervolume indicator,"

[68] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491, Feb 2017.

[69] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 103–114, Feb 2017.

[70] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: a software-based tool for estimating the energy profile of android applications," in *Proceedings of the 39th International Conference on Software Engineering Companion*, pp. 3–6, IEEE Press, 2017.

[71] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Submitted to the Information Software Technology*, 2017.

[72] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[73] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *Software Engineering, IEEE Transactions on*, vol. 39, pp. 1208–1215, Sept 2013.

[74] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "Developing fault-prediction models: What the research can show industry," *IEEE Software*, vol. 28, no. 6, pp. 96–99, 2011.

[75] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 321–332, 2016.

[76] D. Di Nucci, P. Fabio, and A. De Lucia, "Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study,"

[77] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. International Symposium on Software Testing and Analysis*, pp. 140–150, ACM, 2007.

[78] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on gpu," in *Proceeding of the Symposium on Search-Based Software Engineering*, vol. 8084 of *Lecture Notes in Computer Science*, pp. 111–125, Springer Berlin Heidelberg, 2013.

[79] M. Islam, A. Marchetto, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases based on latent semantic indexing," in *Proceedings of European Conference on Software Maintenance and Reengineering*, pp. 21–30, IEEE, 2012.

[80] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 918–940, 2016.

[81] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 234–245, ACM, 2015.

[82] S. Sampath, R. Bryce, and A. Memon, "A uniform representation of hybrid criteria for regression testing," *Software Engineering, IEEE Transactions on*, vol. 39, no. 10, pp. 1326–1344, 2013.

[83] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia *IEEE Transactions on Software Engineering*.

[84] A. Panichella, F. Kifetew, and P. Tonella *IEEE Transactions on Software Engineering*.

[85] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th*

Working Conference on Mining Software Repositories, MSR 2014, (New York, NY, USA), pp. 2–11, ACM, 2014.

[86] "Moonsoon-solutions. power monitor." http://www.msoon.com/LabEquipment/PowerMonitor/.

[87] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, (Washington, DC, USA), pp. 429–440, IEEE Computer Society, 2015.

[88] W. M. Khaled El Emam and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, p. 63–75, 2001.

[89] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, p. 297–310, 2003.

[90] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," in *Proceedings of the 9th IEEE International Symposium on Software Metrics*, pp. 338–349, IEEE CS Press, 2003.

[91] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, 2010.

[92] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 284–292, IEEE, 2005.

[93] A. E. Hassan and R. C. Holt, "Studying the chaos of code development," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[94] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005*, ICSM '05, pp. 263–272, IEEE Computer Society, 2005.

[95] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, pp. 489–498, IEEE Computer Society, 2007.

[96] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pp. 309–318, IEEE, 2010.

[97] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 61–72, ACM, 2006.

[98] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, pp. 476–493, June 1994.

[99] M. E. Bezerra, A. L. Oliveira, P. J. Adeodato, and S. R. Meira, *Enhancing RBF-DDA algorithm's robustness: Neural networks applied to prediction of fault-prone software modules*, pp. 119–128. Springer, 2008.

[100] M. E. Bezerra, A. L. Oliveira, and S. R. Meira, "A constructive rbf neural network for estimating the probability of defects in software modules," in *2007 International Joint Conference on Neural Networks*, pp. 2869–2874, IEEE, 2007.

[101] M. O. Elish, "A comparative study of fault density prediction in aspect-oriented systems using mlp, rbf, knn, rt, denfis and svr models," *Artificial Intelligence Review*, vol. 42, no. 4, pp. 695–703, 2014.

[102] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, pp. 675–686, Oct 2007.

[103] R. Malhotra, "An empirical framework for defect prediction using machine learning techniques with android software," *Applied Soft Computing*, vol. 49, pp. 1034–1050, 2016.

[104] A. Panichella, C. V. Alexandru, S. Panichella, A. Bacchelli, and H. C. Gall, "A search-based training algorithm for cost-aware defect prediction," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pp. 1077–1084, ACM, 2016.

[105] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proceedings of 6th IEEE International Conference On Software Testing, Verification and Validation*, pp. 252–261, 2013.

[106] S. Herbold, A. Trautsch, and J. Grabowski *IEEE Transactions on Software Engineering*.

[107] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, 2017.

[108] S. H. Walker and D. B. Duncan, "Estimation of the probability of an event as a function of several independent variables," *Biometrika*, vol. 54, no. 1-2, pp. 167–179, 1967.

[109] D. D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *European conference on machine learning*, pp. 4–15, Springer, 1998.

[110] D. S. Broomhead and D. Lowe, "Radial basis functions, multi-variable functional interpolation and adaptive networks," tech. rep., Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[112] J. R. Quinlan, "Simplifying decision trees," *International journal of man-machine studies*, vol. 27, no. 3, pp. 221–234, 1987.

[113] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[114] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, pp. 485–496, July 2008.

[115] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.

[116] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Icml*, vol. 96, pp. 148–156, 1996.

[117] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[118] J. Kittler, M. Hatef, R. P. Duin, and J. Matas, "On combining classifiers," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 3, pp. 226–239, 1998.

[119] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[120] T. K. Ho, "Random decision forests," in *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, vol. 1, pp. 278–282, IEEE, 1995.

[121] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[122] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 460–463, IEEE Computer Society, 2009.

[123] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 523–534, IEEE, 2016.

[124] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *15th International Symposium on Software Reliability Engineering*, pp. 113–124, IEEE Computer Society, 2004.

[125] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, pp. 159–182, Feb. 2002.

[126] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A framework of greedy methods for constructing interaction test suites," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 146–155, 2005.

[127] M. Cohen, M. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, pp. 633–650, 2008.

[128] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proceedings of International Symposium on Empirical Software Engineering*, 2005.

[129] Z. Q. Zhou, A. Sinaga, and W. Susilo, "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites," in *System Science (HICSS), 2012 45th Hawaii International Conference on*, pp. 5584–5593, IEEE, 2012.

[130] Y. Cao, Z. Q. Zhou, and T. Y. Chen, "On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions," in *Quality Software (QSIC), 2013 13th International Conference on*, pp. 153–162, IEEE, 2013.

[131] M. Marré and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 974–984, 2003.

[132] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[133] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, pp. 1–7, ACM, 2007.

[134] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 26–36, ACM, 2013.

[135] E. Rogstad, L. Briand, and R. Torkar, "Test case selection for black-box regression testing of database applications," *Information and Software Technology*, vol. 55, no. 10, pp. 1781–1795, 2013.

[136] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to

generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[137] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon, "Assessing software product line testing via model-based mutation: An application to similarity testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pp. 188–197, IEEE, 2013.

[138] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 6, 2013.

[139] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[140] M. Papadakis, C. Henard, and Y. Le Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pp. 1–10, IEEE, 2014.

[141] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering Methodology*, vol. 24, no. 2, pp. 10:1–10:31, 2014.

[142] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 35th International Conference on Software Engineering*, pp. 192–201, IEEE, 2013.

[143] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceeding of the 23rd International Conference on Software Engineering*, pp. 329–338, IEEE, 2001.

[144] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," tech. rep., Department of Computer Science and Engineering, 2006.

[145] G. Rothermel, R. Untch, C. Chu, and M. Harrold in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*.

[146] K. E. Atkinson, *An introduction to numerical analysis*. John Wiley & Sons, 2008.

[147] E. Hughes, "Evolutionary many-objective optimisation: many once or one many?," in *IEEE Congress on Evolutionary Computation*, vol. 1, pp. 222–227, Sept 2005.

[148] I. Hur and C. Lin, "A comprehensive approach to dram power management," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 305–316, Feb 2008.

[149] P. Choudhary and D. Marculescu, "Power management of voltage/frequency island-based systems using hardware-based methods," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, pp. 427–438, March 2009.

[150] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *WMCSA'99*, pp. 1–9, 1999.

[151] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 12–21, ACM, 2014.

[152] "Arduino." https://www.arduino.cc.

[153] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "Powerapi: A software library to monitor the energy consumed at the process-level," in *PoweERCIM News*, 2013.

[154] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "Runtime monitoring of software energy hotspots," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, (New York, NY, USA), pp. 160–169, ACM, 2012.

[155] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the sixth conference on Computer systems*, pp. 153–168, ACM, 2011.

[156] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 29–42, ACM, 2012.

[157] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *NSDI'13*, pp. 43–56, 2013.

[158] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, (New York, NY, USA), pp. 280–293, ACM, 2009.

[159] F. Ding, F. Xia, W. Zhang, X. Zhao, and C. Ma, "Monitoring energy consumption of smartphones," in *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pp. 610–613, IEEE, 2011.

[160] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 105–114, ACM, 2010.

[161] T. Do, S. Rawshdeh, and W. Shi, "ptop: A process-level power profiling tool," in *in Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower'09*, 2009.

[162] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 92–101, IEEE Press, 2013.

[163] N. Amsel and B. Tomlinson, "Green tracker: A tool for estimating the energy consumption of software," in *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, (New York, NY, USA), pp. 3337–3342, ACM, 2010.

[164] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia, "The power of system call traces: Predicting the software energy consumption impact of changes," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, (Riverton, NJ, USA), pp. 219–233, IBM Corp., 2014.

[165] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[166] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran, "Mining energy traces to aid in software development: An empirical case study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, (New York, NY, USA), pp. 40:1–40:8, ACM, 2014.

[167] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, (New York, NY, USA), pp. 78–89, ACM, 2013.

[168] G. Procaccianti, H. Fernández, and P. Lago, "Empirical evaluation of two best practices for energy-efficient software development," *J. Syst. Softw.*, vol. 117, pp. 185–198, July 2016.

[169] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How does code obfuscation impact energy usage?," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, (Washington, DC, USA), pp. 131–140, IEEE Computer Society, 2014.

[170] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Proceedings of the First International Workshop on Green and Sustainable Software*, GREENS '12, (Piscataway, NJ, USA), pp. 55–61, IEEE Press, 2012.

[171] A. Noureddine and A. Rajan, "Optimising energy consumption of design patterns," in *Proceedings of the 37th International Conference on Software*

Engineering - Volume 2, ICSE '15, (Piscataway, NJ, USA), pp. 623–626, IEEE Press, 2015.

[172] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), pp. 225–236, ACM, 2016.

[173] C. Bunse, H. Höpfner, E. Mansour, and S. Roychoudhury, "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments," in *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pp. 600–607, May 2009.

[174] N. Hunt, P. S. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pp. 63–70, Feb 2011.

[175] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps: A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 143–154, ACM, 2015.

[176] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, (New York, NY, USA), pp. 36:1–36:10, ACM, 2014.

[177] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[178] J.-J. Park, J.-E. Hong, and S.-H. Lee, "Investigation for software power consumption of code refactoring techniques," in *Proceedings of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering*, SEKE '14, 2014.

[179] D. Li and W. G. J. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings*

of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, (New York, NY, USA), pp. 46–53, ACM, 2014.

[180] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 115–126, Feb 2017.

[181] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, pp. 181–190, IEEE Computer Society, 2011.

[182] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[183] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[184] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, (Piscataway, NJ, USA), pp. 403–414, IEEE Press, 2015.

[185] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[186] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *Software Engineering, IEEE Transactions on*, vol. 41, pp. 462–489, May 2015.

[187] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, May 2016.

[188] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[189] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, pp. 671–694, July 2014.

[190] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, MOBILESoft '16, (New York, NY, USA), pp. 225–234, ACM, 2016.

[191] F. Palomba, D. A., G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[192] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*, pp. 387–419. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[193] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE transactions on software engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[194] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, 2017.

[195] V. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Paradigm*. John Wiley and Sons, 1994.

[196] D. Di Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model - replication package - https://figshare.com/articles/A_Developer_Centered_Bug_Prediction_Model/3435299," 2016.

[197] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[198] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[199] L. M. Y. Freund, "The alternating decision tree learning algorithm," in *Proceeding of the Sixteenth International Conference on Machine Learning*, pp. 124–133, 1999.

[200] R. Kohavi, "The power of decision tables," in *8th European Conference on Machine Learning*, pp. 174–189, Springer, 1995.

[201] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[202] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1961.

[203] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Eleventh Conference on Uncertainty in Artificial Intelligence*, (San Mateo), pp. 338–345, Morgan Kaufmann, 1995.

[204] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The promise repository of empirical software engineering data," June 2012.

[205] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*, ACM, 2005.

[206] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 13–22, 2001.

[207] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artif. Intell.*, vol. 97, pp. 273–324, Dec. 1997.

[208] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proccedings of 6th International Workshop on Program Comprehension*, (Ischia, Italy), IEEE CS Press, 1998.

[209] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 3rd edition ed., 1998.

[210] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition ed., 2005.

[211] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[212] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pp. 121–130, 2013.

[213] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 280–289, Sept 2013.

[214] I. Jolliffe, *Principal Component Analysis*. John Wiley & Sons, Ltd, 2005.

[215] P. A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*. 1982.

[216] A. Tosun, B. Turhan, and A. Bener, "Ensemble of software defect predictors: a case study," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 318–320, ACM, 2008.

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

[218] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute selection and imbalanced data: Problems in software defect prediction," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, vol. 1, pp. 137–144, 2010.

[219] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, 2011.

[220] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, no. C, pp. 170–190, 2015.

[221] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, pp. 19–24, ACM, 2008.

[222] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method - replication package - http://figshare.com/articles/Dynamic_Selection_of_Classifiers_in_Bug_Prediction_an_Adaptive_Method/4206294," 2017.

[223] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Int J Data Warehousing and Mining*, vol. 2007, pp. 1–13, 2007.

[224] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[225] S. Chidamber, D. Darcy, and C. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Transactions on Software Engineering (TSE)*, vol. 24, no. 8, pp. 629–639, 1998.

[226] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, p. 23, IBM, 2008.

[227] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 57–72, Oct. 2001.

[228] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[229] C. Perlich, F. Provost, and J. S. Simonoff, "Tree induction vs. logistic regression: A learning-curve analysis," *J. Mach. Learn. Res.*, vol. 4, pp. 211–255, Dec. 2003.

[230] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, (New York, NY, USA), pp. 9:1–9:10, ACM, 2010.

[231] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476–493, Jun 1994.

[232] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 382–391, IEEE Press, 2013.

[233] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[234] S. Herbold, A. Trautsch, and J. Grabowski, "Global vs. local models for cross-project defect prediction," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1866–1902, 2017.

[235] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[236] M. A. Hall, "Correlation-based feature selection for machine learning," tech. rep., 1998.

[237] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[238] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah *IEEE Transactions on Software Engineering*.

[239] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, pp. 321–357, June 2002.

[240] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Online appendix of: Mining energy-greedy

API usage patterns in Android apps: an empirical study." www.cs.wm.edu/semeru/data/MSR14-android-energy.

[241] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[242] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[243] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, 2017.

[244] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society. Series B (Methodological)*, pp. 111–147, 1974.

[245] C. Sammut and G. I. Webb, eds., *Leave-One-Out Cross-Validation*, pp. 600–601. Boston, MA: Springer US, 2010.

[246] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, (Washington, DC, USA), pp. 252–261, IEEE Computer Society, 2013.

[247] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[248] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pp. 60–69, IEEE, 2012.

[249] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[250] M. Robnik-Šikonja, "Improving random forests," in *European conference on machine learning*, pp. 359–370, Springer, 2004.

[251] Y. Jiang, B. Cukic, and T. Menzies, "Can data transformation help in the detection of fault-prone modules?," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, pp. 16–20, ACM, 2008.

[252] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve.," *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[253] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pp. 244–255, IEEE, 2016.

[254] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM Comput. Surv.*, vol. 27, pp. 326–327, Sept. 1995.

[255] M. Lanza, A. Mocci, and L. Ponzanelli, "The tragedy of defect prediction, prince of empirical software engineering research," *IEEE Software*, vol. 33, no. 6, pp. 102–105, 2016.

[256] S. Kukkonen and J. Lampinen, "Gde3: The third evolution step of generalized differential evolution," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, vol. 1, pp. 443–450, IEEE, 2005.

[257] H. Li and Q. Zhang, "Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 284–302, 2009.

[258] J. M. Bader, *Hypervolume-Based Search for Multiobjective Optimization: Theory and Methods*. Paramount, CA: CreateSpace, 2010.

[259] H. John, "Holland, adaptation in natural and artificial systems: An introductory analysis with applications to biology, control and artificial intelligence," 1992.

[260] G. Luque and E. Alba, *Parallel genetic algorithms: Theory and real world applications*, vol. 367. Springer, 2011.

[261] S. G. E. Hyunsook Do and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering*, vol. 10, pp. 405–435, 2005.

[262] A. De Lucia, M. Di Penta, R. Oliveto, and A. Panichella, "On the role of diversity measures for multi-objective test case selection," in *Proceedings of the International Workshop on Automation of Software Test*, pp. 145–151, 2012.

[263] S. Yoo and M. Harman, "Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689–701, 2010.

[264] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, pp. 135–141, Nov. 1996.

[265] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast elitist multi-objective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[266] Y.-C. Huang, C.-Y. Huang, J.-R. Chang, and T.-Y. Chen, "Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history," in *Proceedings of the Annual Computer Software and Applications Conference*, pp. 413–418, IEEE, 2010.

[267] "Mysql, online bug repository." http://bugs.mysql.com/.

[268] A. Marchetto, C. Di Francescomarino, and P. Tonella, "Optimizing the trade-off between complexity and conformance in process reduction," in *Proceeding of the Symposium on Search-Based Software Engineering*, pp. 158–172, Springer, 2011.

[269] Y. Zhang and M. Harman, "Search based optimization of requirements interaction management," in *Proceeding of the Symposium on Search-Based Software Engineering*, pp. 47–56, IEEE, 2010.

[270] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011.

[271] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of*

the 33rd International Conference on Software Engineering, pp. 1–10, ACM, 2011.

[272] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[273] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "A test case prioritization approach using genetic algorithm guided by the hypervolume indicator - replication package - https://figshare.com/articles/A_Test_Case_Prioritization_Approach_Using_Genetic_Algorithm_Guided_by_the_Hypervolume_Indicator/5235427," 2017.

[274] R. D. Baker, "Modern permutation test software," in *Randomization Tests* (E. Edgington, ed.), Marcel Decker, 1995.

[275] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[276] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, pp. 219–249, 2002.

[277] "adoctor website." http://tinyurl.com/hnm2sla.

[278] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 335–348, ACM, 2011.

[279] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: a software-based tool for estimating the energy profile of android applications - replication package - https://figshare.com/articles/PETrA_a_Software-based_Tool_for_Estimating_the_Energy_Profile_of_Android_Applications/4233767," 2017.

[280] C. Zhang and A. Hindle, "A green miner's dataset: Mining the impact of software change on energy consumption," in *Proceedings of the 11th*

Working Conference on Mining Software Repositories, MSR 2014, (New York, NY, USA), pp. 400–403, ACM, 2014.

[281] C. Zhang, J. Campbell, and A. Hindle, "Green trace: The impact of software change on system calls and energy consumption," in *in submission to Mining Software Repositories (MSR), 2014 11th IEEE Working Conference on*, IEEE, 2014.

[282] X. Li and J. P. Gallagher, "Fine-grained energy modeling for the source code of a mobile application," tech. rep., PeerJ PrePrints, 2016.

[283] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of android applications," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 121–130, Sept 2014.

[284] "Android studio." https://developer.android.com/studio/index.html.

[285] L. C. Briand and I. Wieczorek, "Resource estimation in software engineering," *Encyclopedia of software engineering*, 2002.

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

[287] D. P. U. V. Nguyen and T. M. WVU, "Studies of confidence in software cost estimation research based on the criterions mmre and pred," 2009.

[288] M. Jorgensen, "Experience with the accuracy of software maintenance task effort prediction models," *IEEE Transactions on software engineering*, vol. 21, no. 8, pp. 674–681, 1995.

[289] P. Hurni, B. Nyffenegger, T. Braun, and A. Hergenroeder, "On the accuracy of software-based energy estimation techniques," in *Proceedings of the 8th European Conference on Wireless Sensor Networks*, EWSN'11, (Berlin, Heidelberg), pp. 49–64, Springer-Verlag, 2011.

[290] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated*

Software Engineering, ASE 2012, (New York, NY, USA), pp. 258–261, ACM, 2012.

[291] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, (New York, NY, USA), pp. 59:1–59:11, ACM, 2012.

[292] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 224–234, ACM, 2013.

[293] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), pp. 623–640, ACM, 2013.

[294] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications - replication package - https://figshare.com/articles/On_the_Impact_of_Code_Smells_on_the_Energy_Consumption_of_Mobile_Applications/3759489," 2017.

[295] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 94–105, ACM, 2016.

[296] L. Al Shalabi and Z. Shaaban, "Normalization as a preprocessing engine for data mining and the approach of preference matrix," in *Dependability of Computer Systems, 2006. DepCos-RELCOMEX'06. International Conference on*, pp. 207–214, IEEE, 2006.

[297] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[298] R. H. Romer, *Energy : an introduction to physics*. San Francisco: Freeman, 1976.

[299] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion mmre," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 985–995, 2003.

[300] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd, "What accuracy statistics really measure," *IEE Proceedings-Software*, vol. 148, no. 3, pp. 81–85, 2001.

[301] D. Port and M. Korte, "Comparative studies of the model evaluation criterions mmre and pred in software cost estimation research," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 51–60, ACM, 2008.

[302] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[303] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman, "Exact mean absolute error of baseline predictor, marpo," *Information and Software Technology*, vol. 73, pp. 16–18, 2016.

[304] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619–630, ACM, 2016.

[305] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[306] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58–63, Feb. 1963.