Empirical Software Engineering (2024) 29:68
https://doi.org/10.1007/s10664-024-10455-7

Just-in-Time crash prediction for mobile apps

Chathrie Wimalasooriya[1] · **Sherlock A. Licorish**[1] · **Daniel Alencar da Costa**[1] · **Stephen G. MacDonell**[1]

Accepted: 7 February 2024

Abstract
Just-In-Time (JIT) defect prediction aims to identify defects early, at commit time. Hence, developers can take precautions to avoid defects when the code changes are still fresh in their minds. However, the utility of JIT defect prediction has not been investigated in relation to crashes of mobile apps. We therefore conducted a multi-case study employing both quantitative and qualitative analysis. In the quantitative analysis, we used machine learning techniques for prediction. We collected 113 reliability-related metrics for about 30,000 commits from 14 Android apps and selected 14 important metrics for prediction. We found that both standard JIT metrics and static analysis warnings are important for JIT prediction of mobile app crashes. We further optimized prediction performance, comparing seven state-of-the-art defect prediction techniques with hyperparameter optimization. Our results showed that Random Forest is the best performing model with an AUC-ROC of 0.83. In our qualitative analysis, we manually analysed a sample of 642 commits and identified different types of changes that are common in crash-inducing commits. We explored whether different aspects of changes can be used as metrics in JIT models to improve prediction performance. We found these metrics improve the prediction performance significantly. Hence, we suggest considering static analysis warnings *and* Android-specific metrics to adapt standard JIT defect prediction models for a mobile context to predict crashes. Finally, we provide recommendations to bridge the gap between research and practice and point to opportunities for future research.

Keywords Android apps · Reliability · Crash · Just-in-Time prediction

Communicated by: Yasutaka Kamei

✉ Chathrie Wimalasooriya
chathrie.wimalasooriya@postgrad.otago.ac.nz

Sherlock A. Licorish
sherlock.licorish@otago.ac.nz

Daniel Alencar da Costa
danielcalencar@otago.ac.nz

Stephen G. MacDonell
stephen.macdonell@otago.ac.nz

[1] Department of Information Science, University of Otago, Dunedin, New Zealand

Published online: 08 May 2024

1 Introduction

With over 6.3 billion smartphone users worldwide[1] and users spending most of their mobile time (i.e., 88% of mobile time) on apps, it is no surprise that the mobile app market is thriving today. Smartphones are bundled with various types of apps, and users rely on these and other installed apps for a diverse range of purposes, from making basic phone calls to more sophisticated tasks such as banking, shopping and managing their healthcare. Hence, ensuring the reliability of these apps is highly important. *Reliability* is defined here as how well a system, product or component performs specified functions under specified conditions for a specified period of time (ISO/IEC 25010:2011 2011).

Notwithstanding the need for app reliability, new apps are published continuously to satisfy users' demand. The most popular mobile platform, Android (S. Inc. 2012), publishes more than 50,000 apps each month.[2] To gain and retain competitive advantage, developers strive to constantly deliver highly reliable apps, with the main reliability concern being one of avoiding fail-stop errors such as crashes. However, many released apps still suffer from crashes: a previous survey comprising 3,534 respondents reported that most of the participants had experienced problems while using apps, with 62% of respondents reporting crashes, freezes or other errors, 47% experiencing slow launch times, and 40% reporting an app that would not launch (Tan et al. 2018; Dickerson 2016). Indeed, frequent or seemingly random crashes of an app could easily make users frustrated, leading to them abandoning certain apps and moving to alternatives (Zarif et al. 2020). Such negative user experiences could also affect a company's reputation. Furthermore, developers typically need to frequently update their apps given continuous requests from users for new functionalities and enhancements (McIlroy et al. 2016; Nayebi et al. 2016). This frequent update (evolution) process may introduce new defects that can cause crashes in the future, leading to expensive maintenance costs (McIlroy et al. 2016; Xia et al. 2016).

Accordingly, enabling developers to better identify and mitigate the causes of crashes early and often is important to reduce potential crashes that users might face. For this purpose, developers perform quality assurance (QA) activities such as code inspection and unit testing to identify defects (including crashes) prior to release (Kamei et al. 2012). However, to optimise their limited QA resources, developers must allocate those resources wisely. For this reason, prioritizing QA activities is important. To aid this process, various defect prediction techniques have been proposed in software engineering research (Shin et al. 2021; Li et al. 2019; Okutan and Yıldız 2014). These approaches support prioritizing QA tasks by identifying potentially defective software modules (e.g., file, package), so that practitioners can allocate resources to those modules most likely to cause problems. As shown by Kamei et al. (Kamei et al. 2012), such prediction approaches are more useful if they provide feedback early in the software development process, and soon after developers make changes to code. To address this challenge, Kim et al. (Kim et al. 2008) proposed that solutions should predict defects (or potential defectiveness) at the change level (i.e., at commit time) (Kim et al. 2008), referred to as "Just-In-Time (JIT) defect prediction" by Kamei et al. (Kamei et al. 2012), and later the same term was used by other studies (Catolino et al. 2019; Yang et al. 2015; Hoang et al. 2019; Cheng et al. 2022). JIT prediction

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[2] https://www.appbrain.com/stats/number-of-android-apps

techniques based around code commits have been held to provide many advantages over other approaches. For example, JIT predictions are fine-grained, helping the developer to identify defect-inducing changes mapped to small areas of the code. This means developers' focus is needed only for a small region of the code. Furthermore, developers can be more effectively assigned to fixing a defect or it causes, since each change can be mapped to the developer who committed the change. Another advantage is that defects can be predicted early, at the time of submitting a commit. As a result developers can seek to fix them when the changes in a commit are still fresh in their mind (Kamei et al. 2012). If the detection of a problem is delayed, it becomes more difficult to fix the problem subsequently. The effort for fixing software problems increases exponentially with time (Boehm 1987). Hence, JIT defect prediction models help with testing and debugging by saving manual effort when compared to later, coarse-grained predictions (Kamei et al. 2012).

Despite the benefits of JIT defect prediction models (Kamei et al. 2012; Pascarella et al. 2019; Trautsch et al. 2020), there has been minimal research investigating the relationship between JIT models and crashes. Specifically considering mobile apps, the different practices (e.g., continuous release, or newcomers publishing their own apps) involved in mobile app development might have an impact on the performance of the JIT models since the models were originally designed for non-mobile software and so did not take mobile-related factors into account. While there have been a few previous attempts to apply JIT models to mobile apps (Catolino et al. 2019; Kaur et al. 2016; Catolino 2017), all of these works focus on general defects rather than crashes, which is a specifically critical type of defect. Since a "crash bug" is different when compared to general defects, based on its severity and complexity, and previous research lacks evidence on whether defect prediction models can be adapted for crash prediction, we investigated this issue. Also, predicting crashes is important to prioritize crash bugs (among all defects), since such bugs negatively affect apps users. More generally, we had previously surveyed the space in noticing that "reliability" and by extension, "robustness", as a quality dimension is rarely examined in the mobile apps literature (Wimalasooriya et al. 2022). Thus, this encouraged us to focus on investigating or adapting traditional defect prediction model for predicting crashes in mobile apps. Although predicting crashes in mobile apps has been investigated by Xia et al. (Xia et al. 2016), their predictions were at a coarse-grained release-level. An et al. (An and Khomh 2015) also investigated JIT crash prediction for Firefox which is a browser app, but not mobile. Hence, we are the first to explore JIT prediction of crashes for mobile apps.

In this work, we bridge existing gaps exploring JIT crash prediction of mobile apps by contributing to three major areas:

- **Selection of metrics**: Most of the metrics used for defect prediction of non-mobile software have been adopted into mobile context (Catolino et al. 2019). However, it is still uncertain to what extent these adopted metrics are effective and whether they are applicable for predicting crashes in mobile apps. To the best of our knowledge, this is the first study to address the research gap concerning JIT crash predictions for mobile apps. In particular, we combine the approach of Kamei et al. (Kamei et al. 2012), comprising JIT defect prediction based on commit-related metrics (such as the number of lines added and developer experience) with static source code metrics and static analysis warnings to perform experimentation on a breadth of metrics to check their relevance in JIT prediction of crashes in mobile apps. We use tools including Commit Guru and PMD to collect metrics at the commit level and use RA-SZZ (Neto et al. 2019) algorithm to identify crash-inducing commits for the purpose of labelling commits as "clean" or "crash-inducing," and then

apply feature selection techniques (e.g., correlation analysis) to understand the most influential metrics.

- **Impact of classification technique**: Various classification techniques have been proposed in the literature for defect prediction. However, classifiers perform differently depending on the nature of the datasets considered, such as datasets of general defects or crashes, datasets from different domains (e.g., non-mobile/ mobile) or different levels of predictions (e.g., commit-level, release-level). Hence, it is still unclear which classifier can achieve the best performance for predicting crashes in mobile apps without experimental evidence. We address this second gap by evaluating the performance of state-of-the-art JIT defect prediction models in the context of mobile apps for predicting crashes. This provides insights on selection of a particular classifier providing information on what technique outperforms others.
- **The need for new metrics:** We study whether new metrics are required beyond those available in existing literature related to JIT models for mobile apps. In addition, we study whether these new metrics have more impact than existing metrics when it comes to predicting crashes in mobile apps. We investigate these aspects because existing metrics do not specifically consider mobile contexts. For example, mobile apps are developed and run on special platforms such as Android, and Android upgrades occur in shorter time periods compared to non-mobile software. A previous work by Trautsch et al. (Trautsch et al. 2020) considered programming rules such as Java rule violations combined with metrics by Kamei et al. (Kamei et al. 2012) for JIT defect prediction, but they did not consider Android-specific programming since their focus was not mobile apps. Hence, it is not clear whether existing metrics are sufficient to represent the cause of crashes in mobile context and to therefore serve to predict crashes. This is the third gap addressed in our study, where we engineer new metrics for supporting JIT crash prediction for mobile apps. In this regard, we perform a qualitative study by manually analysing commits, leading to a classification of changes that happen in crash-inducing commits. This classification of changes provides an understanding of what types of changes are more likely to introduce crashes and thus, provide insights on potential new mobile-specific metrics that can be employed in future defect prediction models to enhance prediction performance.

We conducted an empirical study to achieve the abovementioned objectives, investigating more than 30,000 commits. In doing so we also provide a mechanism to build a dataset suitable to train JIT crash prediction models, along with a dataset containing 113 reliability-related metrics calculated for those 30,000+commits for 14 open-source Android apps from 12 different categories (e.g., communication, finance, games, and so on).

The remainder of the paper is organized as follows. Section 2 summarises the background and previous related work on this topic, Section 3 presents the empirical study setup and Section 4 presents our research approach and results. In Section 5, we further discuss our findings, implications and recommendations for research and practice. Section 6 examines the threats to validity of the study and the way we mitigated them. We conclude the study and outline future work in Section 7.

2 Background and Related Work

To the best of our knowledge, there are no previous studies in the literature investigating Just-in-Time crash predictions for mobile apps. The only study investigating predictions of mobile app crashes is the study by Xia et al. (2016), where predictions are performed

at a coarse-grained release level. At this level these may be deemed as (too) long term (i.e., a little tardy when it comes to developing mobile apps). Therefore, below we review works related to Just-in-Time defect prediction more generally that have guided and inspired our work.

2.1 Just-In-Time defect prediction for non-mobile apps

Defect prediction has been an active area of software engineering research for 50 years (Li et al. 2020a; Fenton and Neil 1999) so is regarded as an enduring challenge. Studies have focused on different granularity levels: coarse-grained levels such as, class-level (Di Nucci et al. 2017; Gyimóthy et al. 2005), module-level (Munson and Khoshgoftaar 1992; Kamei et al. 2007; Hall et al. 2011) and finer-grained levels such as commit-level (JIT techniques). Coarse-grained level predictions typically predict a *location* in the software (e.g., probable defect-prone files/modules in future releases) using metrics such as complexity (e.g., McCabe's Cyclomatic complexity), size (e.g., LOC) or object-oriented metrics (e.g., coupling between objects) (Okutan and Yıldız 2014; Fenton and Neil 1999). Thus, even though developers know the particular defective file or module, still they may spend time and effort to find the defective code snippet in the files or modules. In contrast, JIT techniques predict whether a *commit* induce a defect at commit time, i.e., they provide more granular level predictions by using commit related metrics (e.g., number of modified files, number of added lines, and so on) (Kamei et al. 2012).

JIT defect prediction originated when Mockus and Weiss (2000) built a model to predict the risk of software changes. The model is built using historic data of a commercial software project which is a network switch project. Changes to the switch system initiated through Initial Modification Requests (IMR). IMR consists of multiple modification requests (MR) where one MR may contain multiple changes. A number of change metrics such as size in lines of code added, deleted, modified; diffusion of the change as reflected in the number of files, subsystems touched, modified; and several measures related to developer experience were used to predict the probability of a MR being defect inducing. They found that change diffusion and developer experience are the most important metrics to predict defective changes. The authors showed the usefulness of JIT prediction from an industrial perspective by implementing the model as a web-based tool and deploying it in the same company. The model allowed the company to make important quality assurance decisions with regards to testing and delivery.

Sliwerski et al. (Śliwerski et al. 2005) investigated defect-inducing changes in the Mozilla and Eclipse projects. The authors found some distinct patterns in the defect-prone changes with respect to the size of the change – the number of files touched was higher in defect-inducing changes – and the day of the week that the change was applied – defect-fixing and other changes made on Fridays had a higher chance of introducing defects. Kim et al. (Kim et al. 2008) used numerous factors when studying 12 open-source projects, where metrics extracted from different sources such as change metadata, source code, change log messages were used to build models to predict defect-inducing changes. Their model achieved performance of 78% accuracy, 60% recall and 61% precision on average. They further analysed the combinations of different metrics groups, where results showed that these combinations performed better than a single group and there was no metrics combination that worked best across all projects, but each project had a specific combination of metrics groups that delivered optimal performance. Kim

et al. (Kim et al. 2008) were the first to use insights from Mockus and Weiss (Mockus and Weiss 2000) combining with SZZ to formalize the technique to predict at commit time what is called as 'Just-in-Time' today, following Kamei et al. (Kamei et al. 2012) who were the first to use the term 'Just-in-Time' in the context of software prediction.

Interest in JIT defect prediction has increased in the past decade. The studies conducted by Kamei et al. (Kamei et al. 2012) (Kamei et al. 2016) have played a key role in JIT defect prediction research and formed a source of inspiration for other researchers who were interested in the topic. The authors introduced a set of metrics which we refer to as *Kamei* et al. in our study. These are change-based metrics that incorporate diffusion, size, purpose, history of the change and developer experience. We include these metrics in our work and used the same name to refer to these metrics. Their model was trained and evaluated on six open-source and five commercial projects, where they were able to achieve a performance of 34% precision and 64% recall despite using a highly imbalanced dataset. The authors also mentioned that the performance of their model is useful in practice by checking the average number of defect-inducing changes per day that were incorrectly flagged as defective (average false positives per days). It was only 2.1 changes per day as false positives that developers need to unnecessarily check. Compared to Mockus and Weiss (Mockus and Weiss 2000) (where predictions are provided at the level of Modifications Requests which are made of multiple changes), predictions by Kamei et al. (Kamei et al. 2012) are at a more granular level (i.e., at individual change level).

Building on the first study by Kamei et al. (Kamei et al. 2012), several other studies investigated JIT defect predictions by analysing further aspects and proposing new techniques to improve JIT predictions. Pascarella et al. (Pascarella et al. 2019) proposed a novel fine-grained JIT defect prediction model. They start from the basis of the approach by Kamei et al. to detect defective commits and further improved this approach to filter only the files in a commit that are defect-prone, so that only those files should be reviewed thoroughly. Trautsch et al. (Trautsch et al. 2020) conducted a study investigating 38 Java apps and their file changes and found that features such as static analysis warnings, static source code metrics and warnings density-derived metrics could further improve performance and cost reduction of fine-grained JIT prediction models. Rahman et al. (Rahman et al. 2011), Yang et al. (Yang et al. 2015), and Barnett et al. (Barnett et al. 2016) have also proposed alternative techniques to improve and complement existing JIT defect prediction models by employing cached history, deep learning techniques, ensemble techniques, unsupervised models and textual analysis, where promising results were reported.

Importantly, however, the above studies focus only on defects, not on crashes. An et al. (An and Khomh 2015) studied JIT prediction of crashes for Mozilla Firefox. Their focus is thus more closely related to ours since we both focus on crashes instead of general defects at commit time. The authors found that commits introducing crashes are mostly made by developers with less experience than others, and they perform more additions and deletions of lines of codes in such commits. Their predictive models achieved precision of 61.4% and recall of 95.0%. However, our work is different from theirs in that Firefox is a browser app and we consider mobile apps.

2.2 Just-In-Time defect prediction for mobile apps

The topic of software defect prediction has been extensively studied for decades in the context of non-mobile software (Okutan and Yıldız 2014; Fenton and Neil 1999; Hall et al. 2011), but few works have investigated defect prediction in the mobile context. The first work

studying JIT techniques for defect prediction in mobile apps is that by Kaur et al. (Kaur et al. 2016) in 2015, which analysed process metrics (i.e., change measures) combined with static source code metrics and showed that a combined model based on process and code metrics is better than a model solely based on code metrics. In general, research investigating mobile apps mostly adopted metrics used in non-mobile software, as opposed to proposing new metrics or guidelines specific for mobile apps. Catolino (Catolino 2017) sought to adapt metrics from Kamei et al. (Kamei et al. 2012) in a mobile context; they applied feature selection techniques to select the most significant metrics to predict defective commits and built a logistic regression (LR) model. Results showed that the model could detect only a limited number of defects (Recall = 25%), and the authors conclude that further analysis is required with regards to more independent variables and projects. This was the baseline for the subsequent work done by Catolino et al. (Catolino et al. 2019), in which they conducted a large scale empirical study deeply analysing commits extracted from 14 apps. They explored metrics from Kamei et al. that may be adapted for mobile apps by using different feature selection techniques. The authors also ascertained whether different classifiers impacted the performance of JIT defect prediction models and whether ensemble techniques improved model performance. Outcomes from the work showed that Naïve Bayes achieved the best performance compared to the other classifiers, and in some cases Naïve Bayes outperformed some well-known ensemble techniques (e.g., Voting (Dietterich 2000) and AdaBoost (Rokach 2010)).

Most recent studies in this space have demonstrated interest in proposing new deep learning models, considering feature representation learning issues. The study by Zhao et al. (Zhao et al. 2021a) is the first study to analyse JIT defect prediction considering feature representation learning issues for mobile apps. Zhao et al. (Zhao et al. 2021b) then proposed an imbalanced deep learning model to learn the high-level feature representation towards mitigating class imbalance issues. In other work Xu et al. (Xu et al. 2021) proposed a feature learning method using a cross-project strategy due to scarcity of data, focusing on effort-aware evaluation, i.e., Effort-Aware Recall (EARecall) and Effort-Aware F-measure (EAF-measure). Separately, Cheng et al. (Cheng et al. 2022) proposed an effort aware framework for JIT cross-project prediction considering techniques to obtain representative features. Results of all these studies showed that their models performed significantly better than the baseline models they used. However, as we mentioned earlier in this section, these previous works have all analysed defects generally, and not crashes specifically, which is the focus of this study.

2.3 Defect-inducing commits

JIT defect prediction models are trained using defect-inducing (bug-inducing) commits i.e., the commits where a defect is introduced (Kamei et al. 2012). The most practical and well-known technique for automatic identification of defect-inducing commits is SZZ (Rodríguez-Pérez et al. 2018; Fan et al. 2019; Williams and Spacco 2008), which was initially proposed and implemented by Zimmermann et al. (Śliwerski et al. 2005). SZZ has been cited in 1150[3] studies as of January 2023. To identify defect-inducing commits, SZZ first identifies defect-fixing commits, i.e., commits that fix defects. Defect-fixing commits contain references to the identifiers of these defects in their logs (Bug IDs). In the defect-fixing commits, SZZ locates the modified lines to remove the bug and these lines are referred to

[3] https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q=When+do+changes+induce+fixes%3F&btnG=

as *buggy lines* (Fan et al. 2019). Next, SZZ traces back the commit history (starting from the *buggy lines*) to find the commit that introduced the *buggy lines* in the first place, i.e., the defect-inducing commits or crash-inducing commits (when the focus is on crashes (An and Khomh 2015)). Several improvements over the SZZ algorithm (e.g., RA-SZZ or MA-SZZ) have been proposed by prior studies (Neto et al. 2019). Among them, RA-SZZ is an improved version of SZZ that considers refactoring changes in the algorithm and outperforms other variants (Neto et al. 2018). Also, note that SZZ assumes bugs have been fixed and bug-fixing commits are available. Recent alternative approaches proposed by Wu et al. (Wu et al. 2018) and Wen et al. (Wen et al. 2016) aim to locate bug-inducing commits *before* a bug is fixed. However, these techniques are not yet as widely tested as SZZ.

2.4 Building defect prediction models

In the field of software defect prediction, two main classes of classification techniques have been used: statistical and machine learning techniques. For example, logistic regression is a well-known statistical method (Gyimóthy et al. 2005) widely used in defect predictions due to its simplicity (i.e., it is comparatively easy to interpret the relationships among variables, making the models easy to understand) despite its advanced performance. In fact, logistic regression has the potential to outperform some advanced machine learning classifiers (Ghotra et al. 2015). On the other hand, machine learning techniques have been used with increasing frequency. Shihab (Shihab 2012) and Hall et al. (Hall et al. 2011) showed that logistic regression and random forest were the two most widely used classification techniques in the software defect prediction literature due to their model explainability and built-in within-model explanation techniques (e.g., variable importance analysis for random forest). Tantithamthavorn et al. (Tantithamthavorn et al. 2016) selected three types of classifiers that are often used for defect prediction in their comparison study: probability-based (i.e., naive bayes), regression based (i.e., logistic regression) and machine learning based (i.e., random forest) classifiers. Xia et al. (Xia et al. 2016) studied four classifiers in their work to predict crashing releases of mobile apps: naive bayes, decision tree, k-nearest neighbour and random forest. Shu et al. (Shu et al. 2019) selected random forest, naive bayes, logistic regression, multilayer perceptron and k-nearest neighbour approaches in their study, since such classifiers are widely used for software engineering classification problems. Based on this previous literature, we considered all these classification techniques: logistic regression, naive bayes, random forest, decision tree and multilayer perceptron in our comparative analysis (refer to Section 4.2.1). We did not consider k-nearest neighbour since it rarely appeared as a best learner in the studies of Xia et al. (Xia et al. 2016) and Shu et al. (Shu et al. 2019). We also considered two more ensemble learning algorithms in addition to random forest in our analysis, i.e., Adaboost (Li et al. 2019) and XGBoost (Laradji et al. 2015) classifier, since ensemble learning has become popular in the field of machine learning and are held to present several benefits (such as their robustness to irrelevant and redundant features) (Catolino et al. 2019; Laradji et al. 2015). Next, we explain our study design.

3 Study setup

We conducted this multi-case study using both quantitative and qualitative analyses to answer our research questions. This section describes the research questions and the study design.

3.1 Research questions

This section describes our research questions. We formed four granular research questions to answer our main research question, the latter being: "*How well can we predict crash-inducing commits?*". The four sub-questions are: RQ1: What existing metrics are most important for JIT crash prediction of mobile apps?, RQ2: What is the best modelling approach for JIT crash prediction of mobile apps?, RQ3: What types of changes occur in crash-inducing commits? and RQ4: How can we further improve JIT crash predictions of mobile apps?

RQ1: What existing metrics are most important for JIT crash prediction of mobile apps?

Several commit-related metrics (e.g., # lines added and developer experience) that have been proposed by Kamei et al. (Kamei et al. 2012) have played a key role in the JIT defect prediction literature (Catolino et al. 2019; Pascarella et al. 2019; Kamei et al. 2016). It is important to understand which (if any) metrics of Kamei et al. (Kamei et al. 2012) are most important to indicate crash-inducing commits in the context of mobile apps. Furthermore, the nature of the relationship between those metrics and the increased likelihood of a commit causing a crash is also important to explore. This research question will investigate such relationships, such as whether a metric shares a negative or positive correlation with crashes and which metrics have the highest contribution to crash prediction. Based on these relationships, developers can be made more aware of the commits that have a higher chance of causing a crash and so pay more attention to them.

RQ2: What is the best modelling approach for JIT crash prediction of mobile apps?

Previous literature on this topic focuses heavily on general software defects (Catolino et al. 2019; Kaur et al. 2016; Catolino 2017). Since our focus is on crashes (and not general defects), in this RQ, we investigate whether traditional JIT prediction models are applicable for predicting crashes. Also, we explore whether it is possible to use these models in a mobile context. To answer this question, we explore recent state-of-the-art ML classifiers and metrics that are commonly used for JIT defect prediction of traditional software systems. We investigate these metrics and techniques to predict crashes at commit time in a mobile context.

RQ3: What types of changes occur in crash-inducing commits?

We are interested in understanding the actual reasons for crashes, since defect prediction models do not reflect on any root causes of defects or crashes (Shin et al. 2021; Kim et al. 2008). Therefore, we manually analyse crash inducing commits by selecting a sample representing all the mobile apps in our dataset. Some of these crash-inducing commits fix a previous crash where developers provide reasons for crashes. Answering this research question, we analyse what types of changes happen in crash-inducing commits, provide a classification of changes and identify any specific types of changes tend to introduce crashes.

RQ4: How can we further improve JIT crash predictions of mobile apps?

Since the metrics analysed in RQ1 are those that are typical of non-mobile software (Kamei et al. 2012; Pascarella et al. 2019; Trautsch et al. 2020), we believe taking Android-specific development aspects into account to build models may improve JIT prediction in a mobile context. Hence, we are further interested in investigating additional metrics representing Android development contexts. Therefore, to answer this RQ, we explore whether different types of changes identified in RQ3 can be used as features to optimise prediction models' performance. This evidence provides contextual improvement for JIT crash prediction for mobile apps.

3.2 Data collection

We built a dataset by collecting over 30,000 commits from 14 different Android apps since a suitable curated dataset for our work was not available. Below, we present the different steps we followed to build the dataset used in our study. All data is available in our online replication package.[4] An overview of our study's analysis process is provided in Fig. 1.

3.2.1 App subjects

To perform our study, we needed to obtain Android apps based on the following criteria: Android apps that have recorded crashes; a variety of apps covering different app categories (e.g., communications and games) to ensure that our study covers a broad range of crashes; and apps that provide access to their source code and issue repositories as we need commit related information. We focus on Android apps because Android apps hold the majority (71%) of the market share,[5] Android apps are popular in the domain of mobile apps research, and also due to Android's open-source software (OSS) nature. We only consider Java apps hosted on GitHub since some of the analysis tools we used support only Java and Git repositories. Since there are previously performed crash-related studies (Tan et al. 2018; Xia et al. 2016), we also use these studies to inform app selection in our study, in part for study comparability. Based on previous studies (i.e., (Tan et al. 2018; Xia et al. 2016; Su et al. 2020)) which focus on crashes, we selected 14 apps (see Table 1). None of the selected apps were included in the related works described in Section 2.2. This might be due to the fact that those studies focused on general defects rather than crashes. All the selected apps fulfil our selection criteria explained above and they are available in the F-Droid repository. F-Droid is widely used in software engineering research (Xia et al. 2016) due to many reasons: it is the largest repository of open-source Android apps (Su et al. 2020), it provides apps' meta data such as links to the relevant source code repository including release date, version number, and so on, and it is a source of real-world apps.

3.2.2 Identifying crash-inducing commits

Next, we need to identify crash-inducing commits for the above apps since we need that data to train our defect prediction models, as described in Section 2.3. Therefore, our next focus is to identify crash-inducing commits and label each commit as either crash-inducing

[4] https://tinyurl.com/2p3ntuet

[5] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

or not. Following previous literature, we used RA-SZZ (Neto et al. 2019) to identify crash-inducing commits (refer to Section 2.3). We prepared the inputs required by RA-SZZ which are crash-fixing commits (refer to Section 2.3); in other words, commit IDs/hashes that fixed crashes.

To collect crash-fixing commit IDs, we needed to start from closed (resolved) crash reports, which are issue reports that describe crash incidents. Hence, we developed a Python script (Crawler 2, see Fig. 1: Phase (a)) using git commands and search queries,[6][7] to extract crash issues from the GitHub issue tracker. Following previous studies, we used a keyword search approach to identify crash issues (Tan et al. 2018; Su et al. 2020). Our crawler searches for the keywords ("crash" or "crashing" or "exception" or "freeze" or "hang") in issue titles to find crash issues. The terms 'crash' and 'exception' are commonly used in keyword search approaches (Tan et al. 2018; Su et al. 2020). We added the keywords 'freeze' and 'hang' since such terms are also used in the previous literature (Thitichaimongkhol and Senivongse 2016; Phong et al. 2015) as well as in developers' commits logs and issue descriptions to indicate fail-stop errors such as crashes. As such, we exclude other types of issues such as non-crash defects, enhancements, and feature additions. Using GitHub search queries, we filtered for closed issues only. Once we have collected closed crash issues for each app, we retrieve the fixing commits for each of the crash issues. Developers normally leave hints and links/references about their fixes in their commit logs, which can be identified by commonly used heuristics (Asaduzzaman et al. 2012; Wu et al. 2011). Following previous studies, we use keyword-based search using regular expressions to match a given format of keywords and issue ID (the closed crash issue IDs in the previous step) such as "fixes #256" and "solves #35" to match any parts in commit logs. We also checked whether the same issue title appears in commit messages. Furthermore, in the same way, we checked whether the fixing commit ID (commit hash) appears within the comments of crash issues (e.g., "fixed in b2f66ac"). Based on these heuristics, we mine the links between commit logs and crash issues (i.e., we extract the fixing commits). This way we collected 610 of crash fixing commits and ran RA-SZZ on these commits to retrieve their crash-inducing commits. We then added this information (labels) to our dataset. The commit IDs returned by RA-SZZ are labelled as crash-inducing, whereas the remaining commit IDs are labelled as non-crash-inducing commits. Thus, the dependent variable used in our models is whether a commit is crash-inducing ($Y = 1$) or whether a commit is non-crash-inducing ($Y = 0$).

3.2.3 Validation of data

Ensuring the reliability of the collected data and the techniques we used is important in enabling us to draw valid conclusions meaningful insights from our empirical study. To ensure reliability, we used apps only from the F-Droid repository (see Section 3.2.1) which contains only real-world mobile apps that are available on the Google Play store. Next, in order to identify crash-inducing commits, following previous JIT prediction research, we rely on the RA-SZZ algorithm (see Section 3.2.2) to collect crash-inducing changes since it is the most effective algorithm available in the literature (Neto et al. 2018). However, inputs to RA-SZZ (i.e., crash-fixing commits) depend on the quality of the links between

[6] https://docs.github.com/en/rest/search?apiVersion=2022-11-28#constructing-a-search-query

[7] https://docs.github.com/en/search-github/getting-started-with-searching-on-github/understanding-the-search-syntax

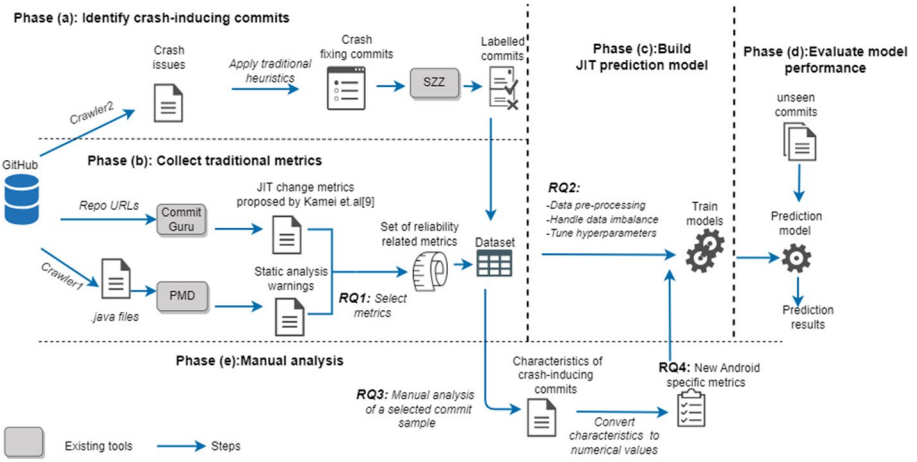

Fig. 1 Overview of our study's analysis process

crash issues and their fixing commits, which we extracted using our own implementation. Previous studies have manually verified the links and fixing commits (inputs to SZZ algorithm) to confirm that SZZ was finding the correct bug-inducing commits (Williams and Spacco 2008; Kim et al. 2006). Following those studies, we also performed a manual analysis to ensure correct inputs to SZZ. The first author manually checked the linked data starting with checking all crash issues. This involved first manually checking all the issue title/descriptions to guarantee that all issues are crash related. Then the linked fixing commits were checked to ensure the IDs of those issues appear in the fixing commit logs to ensure those fixes are of same crash issues. We found three cases out of 613 fixing commits with invalid links, and we filtered them out before running SZZ on the fixing commits. Additionally, we manually verified whether crash-inducing changes (output from SZZ) are indeed the correct crash-inducing changes. Following Williams and Spacco (Williams and Spacco 2008) who manually analysed a sample of 25 commits out of 37,000, we followed a similar approach. Considering the time required for extensive analysis of large samples, the first author randomly selected a sample of 30 crash-inducing commits (that were identified by RA-SZZ) and checked whether they contained the changes that induced fixes later in fixing commits. We found that most of the changes made in selected crash-inducing commits (27 of 30 i.e., 90%) appear to induce actual fixes in crash-fixing commits. Another author reviewed a random sample of 10 of those 30 crash-inducing commits and agreed with all the decisions. We have removed the three false positives from the dataset. The number of false positives is relatively small and false positives can be expected given the limitations of the RA-SZZ algorithm (see Section 6).

4 Study approach and results

This section presents the results of our empirical study. The sub-sections below from 4.1 to 4.4, each describe the approach followed and results obtained for research questions RQ1, RQ2, RQ3 and RQ4, respectively. Python and the Scikit-learn ML libraries were used for implementation (where further details are provided below).

Table 1 Details of the 14 mobile apps considered in the study

Project	URL of Repository	Category	Size	Downloads	# of commits	
					Crash-inducing	non-crash inducing
K-9 Mail	https://github.com/k9mail/k-9	Communication	4.7 MB	5 M+	61	5117
ConnectBot	https://github.com/connectbot/connectbot	Communication	1.7 MB	1 M+	3	995
My Expenses	https://github.com/mtotschnig/MyExpenses	Finance	7.3 MB	1 M+	20	5018
AntennaPod	https://github.com/AntennaPod/AntennaPod	Music & Audio	5.7 MB	500 K+	154	3637
Open Sudoku	https://github.com/ogarcia/opensudoku	Game	681 KB	10 K+	2	42
Anki-Android	https://github.com/ankidroid/Anki-Android	Education	16 MB	5 M+	82	5673
Twidere	https://github.com/TwidereProject/Twidere-Android	Social	16 MB	500 K+	154	1853
AmazeFileManager	https://github.com/TeamAmaze/AmazeFileManager	Tools	4.9 MB	1 M+	75	2303
FastHub	https://github.com/k0shk0sh/FastHub	Productivity	6.6 MB	500 K+	131	634
TuCanMobile	https://github.com/Tyde/TuCanMobile	Education	5.6 MB	10 K+	23	246
Transdroid	https://github.com/erickok/transdroid	Tools	2.2 MB	100 K+	49	518
AnyMemo	https://github.com/helloworld1/AnyMemo	Education	3.9 MB	100 K+	3	1479
KISS	https://github.com/Neamar/KISS	Personalization	199 KB	100 K+	47	1373
Poet Assistant	https://github.com/caarmen/poet-assistant	Books & Reference	20 MB	100 K+	33	388
Total number of commits collected from all apps:					837	30,113

4.1 RQ1: existing metrics are most important for JIT crash prediction of mobile apps?

The purpose of this RQ is to enable us to understand the most important metrics in the mobile context for crash prediction at the commit-level. Thus, we needed to understand the relationships between the metrics and crash-inducing commits. To answer this RQ, we employed the following approach.

4.1.1 RQ1-approach

Metrics Collection We collected the required metrics for over 30,000 commits for the selected apps. Table 2 presents the metrics selected for this analysis. As shown in Table 2, the metrics are of two sets: one is the metrics set proposed by Kamei et al. (Kamei et al. 2012), comprising 14 metrics grouped into five dimensions: diffusion, size, purpose, history and experience; the other set comprises reliability-related object-oriented metrics, and static analysis warnings by PMD.[8] The full list of 99 PMD metrics is provided in Appendix A. PMD analyses source code to find common programming flaws, for example, unused variable, empty catch block, avoid catching null pointer exceptions, and so on, which make programs more complicated as well as prone to runtime errors. As noted by Trautsch et al. (Trautsch et al. 2020), some object-oriented metrics and static analysis warnings can improve the performance of just-in-time defect prediction models. PMD defines a broad range of Java rules (i.e., 220 rules)[9](Meldrum et al. 2020), and among them we selected 99 warnings relevant to our purpose. These 99 metrics include 82 warnings that are identified as reliability-related in a recent study by Meldrum et al. (Meldrum et al. 2020) (where 'reliability' deals with system faults or errors including crashes (Sommerville 2011)), two exception-related warnings (83–84 in Table 2) since handling exceptions is important to avoid runtime errors, and a further eight warnings related to multithreading (85–92 in Table 2) since asynchronous programming errors can also introduce defects that may lead to crashes (Fan et al. 2018a). Further, another seven warnings (93–99 in Table 2) reflective of object-oriented metrics related to nesting levels, coupling and numbers of parameters were included (since metrics such as NL (Nested Levels), CBO (Coupling Between Objects), and NUMPAR (Number of Parameters) were among the top 10 features for JIT defect prediction in the study conducted by Trautsch et al. (Trautsch et al. 2020)). Altogether, 113 metrics were selected based on their relevance to our context, which are further analysed to identify the most important metrics for predicting crash-inducing commits.

We used two existing tools to extract the selected metrics: Commit Guru (Rosen et al. 2015) and PMD (see Fig. 1: Phase (b)). Both tools' source code are publicly available. The tool Commit Guru was specifically developed to analyse and present the metrics proposed and used by Kamei et al. (Kamei et al. 2012). We downloaded the backend system of

[8] https://pmd.github.io/

[9] https://github.com/codecop/pmd-rules

Commit Guru,[10] set it up locally and ran it on our apps to collect the metrics recommended by Kamei et al. (Kamei et al. 2012). Commit Guru uses repository URLs as inputs to calculate metrics. The PMD tool requires inputs as Java source files (.java files) or a directory/zip/jar file containing the source files. To utilize PMD for commit-level analysis, our implementation (Crawler 1, see Fig. 1) traversed the commits and extracted modified.java files from each commit and then downloaded these files to separate directories (i.e., one directory is created for each commit containing its modified source files).

Another Python script was then written to utilize PMD locally, given that PMD does not provide APIs. We used commands provided in the PMD documentation in our Python script to run PMD on the directories containing the downloaded.java files at a specific commit snapshot. This way we collected PMD metrics at the commit-level. We then combined data from PMD and Commit Guru to form the dataset.

Data preparation

(a) *Remove missing values:* Once metrics were collected, we prepared our dataset by removing records with missing values. In some cases, Commit Guru and PMD produced null values. For example, metric AGE (average time interval between last and current change) cannot be measured when there are no change histories (e.g., if the project in question is a new development project) (Fukushima et al. 2014). Following previous studies(Fukushima et al. 2014; Lomio et al. 2022), we have removed records with missing values and 29,665 commits remained. This is as large or larger than the size of other commit datasets used in previous research (e.g.,(Kim et al. 2008; Mockus and Weiss 2000; He et al. 2017)) or close to them at the upper limit (e.g.,(Hoang et al. 2019)). Therefore, we consider the size of our dataset is suitable for our study.

(b) *Resolve data imbalance*: Not unexpectedly for working real-world apps our dataset is highly imbalanced. From 29,665 commits only 2.8% (i.e., 829 commits) are crash-inducing, and all others (28,836 commits) are non-crash-inducing. A highly imbalanced dataset is a common issue in defect prediction research since few defective components and many non-defective components are a common occurrence (Song et al. 2018). For example, the ratio of defect-inducing changes to non-defect-inducing changes was 2.5% in the study by Kamei et al. (Kamei et al. 2012). Since data imbalance can lead to performance degradation (in that prediction models tend to be biased towards non-defective instances, being the majority class) (Kamei et al. 2012), we apply data balancing techniques to balance the dataset. There are four types of data balancing techniques (Song et al. 2018; Rodriguez et al. 2014):

1) under-sampling methods that eliminate instances from the majority class towards a balanced distribution; however, a main drawback of this method is that it can discard useful information which could be important for training ML models (Zhihao et al. 2019).
2) over-sampling methods that randomly replicate minority class instances, which could lead to overfitting problems (Song et al. 2018; Zhihao et al. 2019).
3) SMOTE (Synthetic Minority Oversampling Technique), a well-known, widely used and more intelligent over-sampling method that creates new minority class instances

[10] https://github.com/CommitAnalyzingService/CAS_CodeRepoAnalyzer

Table 2 Selected metrics in our study

Group	Metric Name and Definition
JIT metrics by Kamei et al. (Kamei et al. 2012)	*Diffusion*: 1. NS: Number of modified subsystems, 2. ND: Number of modified directories, 3. NF: Number of modified files, 4. Entropy: Number of modified codes across each file *Size*: 5.LA: Lines of code added, 6. LD: Lines of code deleted, 7. LT: Lines of code in a file before the change *Purpose*: 8. FIX: Whether or not the change is a bug fix *History*: 9. NDEV: Number of developers working on the files, 10. AGE: Average number of days since the last change, 11. NUC: Number of unique changes to modified files *Experience*: 12. EXP: Developer experience, 13. REXP: Recent developer experience, 14. SEXP: Developer experience on a subsystem
*PMD	1. ATNFS, 2. AAAI, 3. ABSALIL, 4. ACF, 5. ACNPE, 6. ACT, 7. ADLBDC, 8. ADL, 9. AEAI, 10. AICCC, 11. ALEI, 12. AMUO, 13. AUOV, 14. BC, 15. BMSS, 16. BNC, 17. CSF, 18. CSL, 19. CSR, 20. CCEWTA, 21. CMMBP, 22. CMMIC, 23. CMRTMMCN, 24. CTC-NSE, 25. CR, 26. COWE, 27. CCOM, 28. DNCSE, 29. DNHCSC, 30. DNTEIF, 31. DUFTFLI, 32. ECB, 33. EF, 34. EFB, 35. EIS, 36. EI, 37. ESB, 38. ESNIL, 39. ESS, 40. ESB, 41. ETB, 42. EWS, 43. EIN, 44. FDNCSF, 45. FOCSF, 46. FOL, 47. FSBP, 48. IO, 49. UNCIE, 50. UCEL, 51. UETCS, 52. IFSP, 53. ITGC, 54. JI, 55. JUS, 56. JUSS, 57. LINSF, 58. MNC, 59. MBIS, 60. MSVU, 61. MSMINIC, 62. MTOL, 63. NCLISS, 64. NSI, 65. OBEAH, 66. PCI, 67. PL, 68. REARTN, 69. RFFB, 70. SDFNL, 71. SMS, 72. SCRNI, 73. SEFSBF, 74. SBIWC, 75. TCWTC, 76. UIS, 77. UBA, 78. UCC, 79. UCT, 80. UOOI, 81. ULWCC, 82. UPCL, 83. UTWR, 84. ACGE, 85. ASAML, 86. ATG, 87. DNUT, 88. DCTR, 89. DCL, 90. NTSS, 91. USF, 92. UNAON, 93. LC, 94. CBO, 95. EI, 96. ADNIS, 97.CSB, 98. CC, 99. EPL

*The full names of the PMD metrics are provided in Appendix A

based on the k-nearest neighbour method, which was proposed by Chawla et al. (Chawla et al. 2002) to avoid the overfitting issues of the random over-sampling method. While SMOTE mitigates the overfitting problem, in some cases it may not be effective depending on the dataset since it creates synthetic data, not real data (Zhihao et al. 2019).

4) hybrid methods that combine under-sampling and oversampling methods (Rodriguez et al. 2014; Wang 2014). Previous defect prediction research used under-sampling (Kamei et al. 2012), over-sampling techniques such as SMOTE (Trautsch et al. 2020; Chawla et al. 2002), or a hybrid approach (Xia et al. 2016). However, all of these methods can lead to significantly different performance depending on the nature of the dataset and the classifier used (Song et al. 2018; Koziarski 2021).

Therefore, we explored all four types of balancing techniques. We split the dataset into training and testing subsets: 80% of the dataset was used for model building and training and the remaining 20% is used for testing i.e., to evaluate the model performance. A common best practice during model training is not to leak knowledge of the test set to the training set (data leakage) which would result in poor generalization of models ("Data Leakage And Its Effect On The Performance of An ML Model" 2022; "How to Avoid Data Leakage When Performing Data Preparation." (n.d.)). If the model performs well on our test set yet not perform well against new unseen data after deployment, one reason for this could be data leakage, where accidentally, knowledge between the test and train sets are shared during the training stage ("Data Leakage And Its Effect On The Performance of An ML Model" 2022). This way the model is already exposed to the test data, thus offering very impressive performance when we test the model. Therefore, carefully handling model building steps from the beginning such as data balancing, feature scaling) is important to avoid data leakage (Zhao 2022). We tried all four balancing techniques on the training set and found that the hybrid method combining SMOTE with under-sampling performed better than any individual sampling technique. A number of instances were removed randomly from the majority class i.e., non-crash-inducing commits (under-sampling) and then instances in the minority class i.e., crash inducing commits were increased using SMOTE (oversampling). Chawla et al. (Chawla et al. 2002) showed that combining SMOTE with an under-sampling method outperforms the classifiers built using plain sampling techniques by mitigating the overfitting issue.

(iii) *Address skewness:* Since some algorithms such as logistic regression require features to be normalized, we performed feature scaling through standardization on each feature except 'Fix', which is a binary variable. Standardization involves re-scaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one.[11] Here, first only the training set is standardized using the mean and standard deviation of the train set, and then the test set was also standardized using the same training set mean and standard deviation (i.e., both the train set and test set are normalized using information only from the train set, hence the test set is unseen to training (Zhao 2022), thereby avoiding data leakage).

[11] https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/

Dependent and Independent variables Supervised learning models require labelled data, i.e., in each record we must know the value of the *dependent variable* representing the object of study (Immaculate et al. 2019), which, in our case, is whether a code change introduces a crash or not. As noted in Section 3.2.2, with the help of RA-SZZ, we labelled whether our commits are crash inducing (i.e., 829 commits out of 29,665) or not, which is used as the dependent variable in this study. The above collected metrics are the *Independent variables* that are used in models as explanatory factors (predictors) of the dependent variable (Pascarella et al. 2019). As noted at the beginning of this section, at this point we have 113 metrics for further analysis towards the selection of appropriate metrics as independent variables. Selecting metrics before performing prediction is useful to understand whether standard traditional JIT metrics are valuable in a mobile context, and whether metrics originally proposed to predict *defect-inducing commits* are valuable when employed for predicting *crash-inducing commits*. Furthermore, previous studies showed that high dimensionality datasets, i.e., datasets with excessive metrics, including redundant and irrelevant metrics, degrade models' performance due to overfitting, and also result in extensive computational cost of prediction (Xu et al. 2016). To mitigate these issues, and to understand appropriate metrics in the mobile context, we applied feature selection techniques as in the next step.

Feature selection In machine learning, feature selection is the process of selecting a subset of metrics to replace the original metrics set to avoid the issue of model overfitting (Catolino et al. 2019). Following common practices, we applied feature selection only on the training set (Zhao 2022). Given the high dimensional dataset, we adapted a hybrid approach by combining two feature selection techniques as proposed in previous research (Jović et al. 2015; Ding and Fu 2018). The hybrid approach first uses a filter method (e.g., information gain) to obtain a candidate subset of features by reducing feature dimensions and then applies a wrapper method to consider feature subsets based on the impact/degree of influence on a modelling algorithm. Hence, following the hybrid approach, we first employs a filtering technique i.e., information gain (IG). IG, which is also known as mutual information, indicates the gain of each metric in the context of the dependent variable. It is a non-negative value that measures the dependency between two variables: if the IG value between dependent variable and an independent variable is equal to zero or is very low (e.g., < 0.01) (Catolino et al. 2019; Zhu 2021), knowledge gain from such variables in the context of the dependent variable is none or very limited, thus the usefulness of such variables is very limited (Catolino et al. 2019). Following the study by Catolino et al. (Catolino et al. 2019), we considered that metrics with IG $= 0.2$ or higher make a significant contribution towards the prediction. We removed all metrics where the IG value between them and the dependent variable was zero or less than 0.01. This way, we selected 49 from 113 metrics for further analysis (See Appendix B for a full list of IG values). Next, feature forward selection techniques were applied for the 49 metrics left after filtering based on IG above. In detail, we used Sequential Feature Selector as the feature forward selection technique. Sequential feature selection is a greedy procedure that adds features to form a feature subset by choosing the best feature at each iteration based on a cross-validated score of a desired classifier. This is a commonly used wrapper-based feature selection technique, and a previous study that compares feature selection techniques shows that a wrapper-based selector combined with logistic regression is one of the techniques that showed best performance during modelling (Xu et al. 2016). Therefore, we applied the sequential feature selection method with a preferred classifier (i.e., logistic regression) and a preferred

evaluation measure (i.e., AUC measure) following the study by Xu et al. (Xu et al. 2016) to further filter the metrics, also noting that sequential feature selection allows us to have full control of the model-building procedure. During sequential feature selection, we ignored metrics if adding them no longer improved the model performance. We have therefore chosen 14 metrics as the features (see Table 3) that are useful in a mobile context for JIT crash prediction. It is also worth to note that, IG and LR are chosen based on the nature of our metrics; that is, we need to analyse the relationship between continuous (all metrics are continuous except one metric 'Fix' which is binary) and binary (i.e., dependent variable) variables.

In addition, we checked for multi-collinearity using Spearman correlation. (Spearman correlation does not require data to be normally distributed, hence it works well with ordinal data.[12]) The metric pairs Entropy and ND, ND and NF, and EXP and SEXP are highly correlated based on a threshold of 0.8 (where higher than 0.8 means a very strong relationship exists[13]). The feature forward selection algorithm that we applied above filtered out the metrics ND and SEXP, mitigating the multi-collinearity issues since such algorithms usually remove redundant metrics (Xu et al. 2016). Hence, with regard to multi-collinearity, the feature selection algorithm we used inform our decisions as to which metrics should be removed or retained to avoid multi-collinearity.

Finally, we checked the predictive power of the selected metrics using the AUC score and checked how well the model fits our data using the R^2 value of the logistic regression model. R^2 is widely used in linear regression to estimate how the model fits its data, although there is no commonly accepted measure to assess the fit of a logistic regression (R-Documentation 2022; DeMaris 2002). Therefore, variants of "pseudo R squared" statistics (e.g., *Cox & Snell* R^2, *McFadden's* R^2) have been proposed for logistic regression (R-Documentation 2022). Following Allison (Allison 2013), we used *McFadden's* R^2 (Vang 2019) to check how well the model fits data.

4.1.2 RQ1-results

Following the above approach, we have selected 14 important metrics for JIT prediction of crashes, answering our first RQ (RQ1). Table 3 displays the selected 14 metrics including their type (JIT by Kamei et al.(Kamei et al. 2012) and PMD), coefficients, odds ratio and information gain values. Odds ratio corresponds to the ratio by which the probability of a commit being crash inducing increases when the feature increases by one unit. If an odds ratio for a metric is higher than one (odds ratio > 1), it indicates that the metric increases the probability of a commit to introduce a crash, hence, such metrics are risk-increasing (showing a positive relationship). On the other hand, an odds ratio that is less than one indicates a decrease of the probability (showing a negative relationship), thus they are risk-decreasing factors (Kamei et al. 2012). For example, in the case of lines added (LA), its odds ratio of 2.13 indicates that adding one line in the change increases the odds of that commit being crash-inducing by 2.13.

[12] https://ademos.people.uic.edu/Chapter22.html
[13] http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf

Table 3 Importance of selected metrics for prediction

Metric group	Metrics name	Coefficients (β)	Odds ratio (e^{β})	IG
JIT	LA	0.75	2.13	0.48
JIT	Entropy	0.70	2.02	0.18
PMD	ControlStatementBraces (CSB)	0.28	1.33	0.40
PMD	DoNotUseThreads(DNUT)	0.17	1.20	0.11
PMD	UseProperClassLoader(UPCL)	0.09	1.09	0.09
JIT	NDEV	-0.04	0.96	0.42
PMD	BeanMembersShouldSerialize (BMSS)	-0.05	0.95	0.40
JIT	NUC	-0.29	0.75	0.45
JIT	AGE	-0.33	0.71	0.08
JIT	LT	-0.75	0.47	0.22
JIT	EXP	-0.80	0.45	0.09
JIT	REXP	-0.83	0.44	0.04
JIT	NF	-0.87	0.41	0.44
JIT	FIX	-5.95	0.002	0.02

Furthermore, whether a metric is risk-increasing/-decreasing is also shown by the signs of the coefficients: negative coefficients are risk-decreasing whereas positive coefficients are risk increasing. Overall, according to the odds ratio values (see 'Odds ratio' column, Table 3), we observed that the number of lines added in a change (LA) and Entropy (spread of a change across each file) are having the strongest relationship with the dependent variable. It is revealing to see that some PMD metrics (e.g., CSB, DNUT) are more important than some commonly used JIT metrics proposed by Kamei et al. (Kamei et al. 2012). Most of the PMD metrics are risk-increasing except the metric related to serialization, i.e., BMSS. It is surprising to see that BMSS is a risk-decreasing factor since violations of PMD rules makes the code error-prone. This is probably related to the discussion among developers about the noisiness of the rule BMSS,[14] i.e., it doesn't include criteria to determine which classes are bean classes (even though the rule is concerned with bean classes), which means that the rule treats all classes as bean classes. Because of this, PMD triggers BMSS violations despite the fact that classes are not serializable. Some developers disable this rule due to this false trigger for warnings[18]. Hence, if this rule is used by developers and is to be used in future research, it should be treated with caution. Furthermore, most of the JIT metrics by Kamei et al. (Kamei et al. 2012) are risk-decreasing, which is consistent with the findings of a previous study (Trautsch et al. 2020); only 3 out of 10 (LA, Entropy, NDEV) are risk-increasing. We discuss these factors further in Section 5.1.

With regard to the IG values, some metrics from both Kamei et al. (Kamei et al. 2012) and PMD showed a significant contribution ($IG >= 0.2$) (Catolino et al. 2019) towards the prediction. For example, LA and NUC provide the strongest contribution to the model (i.e., 0.48 for the former and 0.45 for the latter). On the other hand, PMD metrics such as CSB returned an IG value of 0.40, which shows a contribution stronger than some JIT metrics by Kamei et al. (Kamei et al. 2012), such as REXP, EXP and AGE.

[14] https://github.com/pmd/pmd/issues/1668

Overall our results indicate that metrics sourced from both Kamei et al. (Kamei et al. 2012) and PMD static analysis warnings are important in the context of JIT crash prediction for mobile apps. McFadden's R^2 of these metrics is 0.26 which represents a good fit since McFadden's R^2 is within the range from 0.2 to 0.4 (Vang 2019; Hemmert et al. 2018). The AUC score of the logistic regression is 0.79 which shows that the selected metrics have reasonable predictive power (Kamei et al. 2012).

4.2 RQ2: What is the best modelling approach for JIT crash prediction of mobile apps?

In the previous RQ, our purpose is only to investigate the relationships between individual metrics and the prediction variable to understand the importance of existing metrics for JIT prediction in mobile context. Here in RQ2, our focus is on optimizing prediction performance using the selected metrics in RQ1. Hence, we performed a comparison of JIT prediction models to understand to what extent we can improve prediction performance by employing different classification approaches and different hyper parameter settings to answer RQ2 (Fig. 1: Phase c). The sections that follow describe the approach and results for RQ2.

4.2.1 RQ2-approach

Baseline Model Similar to previous research (Kamei et al. 2012; Gyimóthy et al. 2005; Cataldo et al. 2009), we use a logistic regression model as our baseline model. This is the same model which was built and evaluated during feature selection process in RQ1. Since logistic regression is a binary classifier, it outputs a probability value, i.e., value between 0 and 1 for each input commit. We use the default threshold value (i.e., 0.5) following previous research by Kamei et al.(Kamei et al. 2012) and Pascarella et al. (Pascarella et al. 2019)), which means that if the model predicted output as greater than 0.5, the commit is classified as crash-inducing otherwise it is classified as non-crash-inducing.

Classifiers We used below classifiers for the comparison of different JIT prediction models:

- *Logistic Regression*. Logistic regression is a statistical model which explains the relationship between one target/dependent variable and one or more independent variables (Catolino et al. 2019). It is widely used for classification in industry (e.g., weather forecasting) (Shu et al. 2019). Logistic regression models a probability estimation and a threshold can be set to predict the class a data belongs to. Based on the threshold, the estimated probability is classified into classes. This can be binomial (binary), ordinal and multinomial (Kamei et al. 2012; Catolino et al. 2019). We use binary logistic regression since our problem is a binary classification problem (i.e., whether a particular commit is crash inducing or not) (Kamei et al. 2012).
- *Naive Bayes.* Naive Bayes (Rish 2001) is a probabilistic classifier based on Bayes theorem. As logistic regression, this model returns the probability of a data belonging to a class. Naïve Bayes classifiers are likely to perform well on relatively small datasets

and are applicable only for classification problems,[15] unlike many other ML algorithms which are applicable for both classification and regression tasks.

- *Decision Tree*. A decision tree algorithm starts with a single node which branches into possible outcomes (Quinlan 1986). Each outcome leads to more nodes, which branch off into further possible outcomes giving a tree like shape (Quinlan 1986). The algorithm classifies data points by comparing their measures with all possible combinations of conditions captured in the nodes (Xia et al. 2016).
- *Random Forest*. Random forest combines multiple decision trees for prediction (Catolino et al. 2019). It builds a large number of decision trees during training time. The term "Random" is used because each decision tree is trained on a random subset of the training data (Breiman 2001). This technique is known as "Bagging" (also known as Bootstrap aggregation) (Breiman 2001). When it is to perform classification on a datapoint, each tree in the forest gives a predicted class label; then the final prediction is the class label which was predicted by the majority of the trees in the forest (Breiman 2001). Random forest algorithms tend to perform significantly better than individual decision tree, since there is always a limit on how accurately an individual decision tree can perform.[16] Even though some trees predict incorrectly, the final output is the label predicted by the greatest number of trees, hence accuracy is likely to be high as most of the trees are more likely to give the right prediction (Breiman 2001).
- *Multilayer Perceptron.* Multilayer perceptron is an artificial neural network which consists of interconnected neurons that transfer information to each other (Shu et al. 2019). Each neuron has an assigned value. The network has three types of layers: the input layer receives signals and output layer makes the classification/prediction about the output (Shu et al. 2019). In between input and output layers there should be at least one hidden layer (the third layer). The hidden layer(s) performs computation on the input data to produce output (Shu et al. 2019; Gardner and Dorling 1998).
- *Adaptive Boosting (AdaBoost).* AdaBoost is a well-known ensemble ML algorithm that uses a boosting algorithm (Catolino et al. 2019). Boosting algorithms helps to combine multiple base learners (weak classifiers) into a single strong classifier (Catolino et al. 2019). The main idea of the algorithm is to add weak classifiers sequentially while each new classifier attempts to correct the incorrect predictions made by the previous classifier. This is achieved by assigning weights to each instance of the training set while assigning more weights to misclassified instances, so the new model added put more focus on these instances to predict them correct (Catolino et al. 2019). Finally, all classifiers are combined through a voting mechanism to predict a new instance (Catolino et al. 2019; Rokach 2010).
- *XGBoost.* XGBoost (eXtreme Gradient Boosting) is also another ensemble learning algorithm that uses a gradient boosting framework (Chen and Guestrin 2016). Since a boosting algorithm is used, same as AdaBoost, XGBoost also converts a set of weak learners into a single strong learner and iteratively create a weak learner to add to the

[15] https://devopedia.org/naive-bayes-classifier

[16] https://towardsdatascience.com/from-a-single-decision-tree-to-a-random-forest-b9523be65147

strong learner (Chen and Guestrin 2016; Aljamaan and Alazba 2020). This is different to AdaBoost in the way of creating weak leaners. XGBoost also use computationally advanced mechanisms to utilize the computer's hardware to speed up the algorithm (Aljamaan and Alazba 2020).

Comparison of JIT models For the comparison of JIT models, we implemented a *pipeline*[17] in Python using the Scikit-learn library which allowed us to automate the complete workflow of training and tuning ML models. We optimized the models using hyperparameter tuning during training. Tuning hyperparameters is important since hyperparameters have direct influence over the behaviour of ML algorithms and they significantly affect model performance (Wu et al. 2019). For hyperparameter tuning, following mainly a previous study by Shu et al. (Shu et al. 2019) and guidelines from practitioners,[18][19] we selected a group of parameters for each classifier that impact classifiers' performance. Appendix C lists all the hyperparameters we selected for optimization. We started with parameter values including default values based on the above mentioned literature and expanded the group of values by exploring how much gain in model performances can be achieved by each parameter value. We chose the best performing parameter set based on the AUC score for each classifier. We used *RandomizedSearchCV*[20] with *stratified k-fold cross validation*[21] for parameter tuning. RandomizedSearchCV was chosen over GridSearchCV due to its efficiency.

Validation We employed two validation settings; tenfold cross validation and timewise validation. First, we use tenfold cross validation (where k = 10) with RandomizedSearchCV since tenfold cross validation is recommended and widely used as a training strategy, including for JIT defect prediction research (Kamei et al. 2012; Pascarella et al. 2019; Trautsch et al. 2020). In a tenfold cross validation approach, the training set is further split into 10 smaller sets ("folds"). Then for each of the 10 folds, a classifier is trained using the first nine folds while the remaining one-fold is held-out as the test split to evaluate the model performance. Afterwards, each of other nine folds becomes a testing split, and a new model is built and evaluated. The test split is unseen to training in each round. Since we used stratified strategy, the class distribution in the training and testing splits was kept the same as the original dataset to simulate real-world usage of the algorithm (scikit-learn 2022). A total of 10 models were built and evaluated on the 10 hold-out test splits in each round and the mean performance is reported as the cross-validated model performance, following the practices reported in previous studies (Li et al. 2020b).

Next, we further analysed the models in terms of time sensitivity. As McIntosh and Kamei (McIntosh and Kamei 2018) suggested, it is important to validate JIT prediction models on a timely basis since, random train-test sampling does not reflect the timely nature of software development. Hence, to test the model in terms of time sensitivity, we

[17] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
[18] https://towardsdatascience.com/how-to-tune-hyperparameters-of-machine-learning-models-a82589d48fc8
[19] https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/
[20] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
[21] https://scikit-learn.org/stable/modules/cross_validation.html

adapted a fivefold walk-forward approach following previous studies (Falessi et al. 2020; Zhao et al. 2019). Here we used fivefold instead of tenfold due to the dataset being imbalanced. In a time-series split, chronological order has to be preserved (Falessi et al. 2020). This requirement means that a random shuffle split in K-fold cross-validation is not applicable for time-wise validation. Consequently, time-series splits face a challenge, as the training set in some folds may contain data only from one class, given the absence of an identical distribution of original data as in shuffle splits. Therefore, the most optimal and feasible arrangement was fivefold configuration for time-wise validation. We used the TimeSeriesSplit[22] library from scikit-learn for implementation. First the dataset was chronologically ordered based on commit date. Next, the dataset was split into 5 sets. Beginning with the earliest frame (n) we used n as training data and n + 1 as test data. In the 5th split, the first 4 folds are returned as the training set and the 5th fold as the test set. In each split, the test set included newer commits than in the training set.

Note that in each validation settings we use only the training set for parameter tuning/model training while the test is separately held out for final evaluation, which is the common recommended practice to reduce the overfitting issue of hyperparameter optimization (Feurer and Hutter 2019). It is obvious that the use of cross-validation for parameter tuning substantially increases the robustness of the models against overfitting, therefore, some studies evaluate and report model performance on the validation set used for hyperparameter tuning irrespective of a hold-out test set. However, it is not sufficient to conclude that models generalize well for unseen data (without overfitting) which can only be diagnosed by using a separate test set (Feurer and Hutter 2019). Hence, in addition to the performance reported by cross validations, we report the performance of the models tested against the held-out test set (held out test set was never involved in training). Refer to the Section 4.2.2 for details on the performances of the models.

Evaluation The prediction outcome of the model for a commit are four ways (Trautsch et al. 2020): a commit is classified as crash-inducing when it is truly crash-inducing (true positive, TP); it can be classified as crash-inducing when it is actually not (false positive, FP); it can be classified as a non-crash-inducing when it is actually crash-inducing (false negative, FN); or it can be correctly classified as non-crash-inducing (true negative, TN).

To evaluate model performance (Fig. 1, Phase d), we used precision, recall, F-measure, AUC and MCC measures:

Precision: is the proportion of correctly predicted crash-inducing commits to all commits predicted as crash inducing, i.e., Precision = TP/ (TP + FP).
Recall: is the proportion of correctly predicted crash-inducing commits to the actual crash-inducing commits, i.e., Recall = TP/ (TP + FN).
F-measure: is the harmonic mean of recall and precision, i.e., F-measure = (2 × Precision x Recall)/ (Precision + Recall).
AUC: Beyond the above measures, we use AUC (Area Under the Receiver Operating Characteristic Curve) since this measure is not impacted by highly imbalanced datasets (Trautsch et al. 2020). The larger the AUC is, the better the performance of the classification algorithm (Xia et al. 2016). AUC score is used to get an overall idea of the performance of the model across thresholds since precision, and recall depend on the

[22] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

particular threshold (i.e., the default threshold of 0.5) used in the classifier (Trautsch et al. 2020).

MCC: following Falessi et al. (Falessi et al. 2020), we report the Matthews Correlation Coefficient (MCC) as an additional measure. It represents a measure that takes all four values in the confusion matrix into account and another recommended measure for imbalanced data. MCC is defined as:

$$(TP * TN - FP * FN)/\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}$$

4.2.2 RQ2-results

Table 4 reports the results of our baseline model, i.e., 8%, 69%, 14% and 79% for precision, recall, F- measure and AUC respectively.

We observe that baseline model achieves a high recall value, but with low precision and f-measure. The recall value of 69% tells us that the model can correctly classify more than half of the crash-inducing commits during app development. The high recall with low precision value is commonly reported in existing defect prediction models (Xia et al. 2016; Kamei et al. 2012), but still the precision value (i.e., 0.08) of baseline model is poor and not better than the existing defect prediction models. However, the AUC score (0.79) shows that the model is more effective than a random predictor (AUC = 0.5) and it implies that we can fine-tune our default threshold (0.5) for a better performance (i.e., better discrimination power between crash-inducing and non-crash-inducing commits). Figure 2 shows the Area Under Curve (AUC) of the Receiver Operating Characteristic (ROC) which plots the false positive rate and true positive rate across different thresholds.

Next, we report the results of the comparison of seven ML algorithms which were tuned based on AUC score under different hyper parameter settings. We followed the approach explained in Section 4.2.1 for comparison and evaluation of the models. Table 5 shows the model performance scores; the columns "Train" report the training scores for tenfold cross validation and Time-wise validation during model training, and the columns "Test" report the results of evaluating the built models against the held-out test set.

As shown in Table 5, considering AUC and MCC test scores (given suitable measures for imbalanced data), we observe that the best performing model is random forest (highlighted in grey) in both validation settings. Note that for the time-wise validation, we selected only the ML algorithms that showed relatively higher performance in tenfold cross validation (according to AUC and MCC test scores) since we rely on both validation settings. Even though AUC scores are high for all models, MCC scores showed low performance for some models (MCC score < 0.2 is considered as low performance (Carka et al. 2022)). Hence, the algorithms that returned < 0.2 for the MCC score for the tenfold cross validation when tested were excluded for the time-wise validation in Table 5. It is important to note that the results presented in Table 5 are for two separate validation settings. The models were built from scratch for each validation setting since each validation needs be carried out independently. As such, there is no data leakage between the two validation settings. Although the best performing model, random forest, showed improved performance compared to the baseline model in some cases (for instance, the AUC score was improved from 79 to 83% based on the tenfold cross validation results, as shown in Table 4 and 5), its performance was very low when considering other evaluation measures. In particular, the F-measure, precision and MCC scores were consistently low in both validation settings. Reviewing previous studies,

Table 4 Performance of the baseline model

Metrics	Score
Precision	0.08
Recall	0.69
F-measure	0.14
MCC	0.18
AUC	0.79

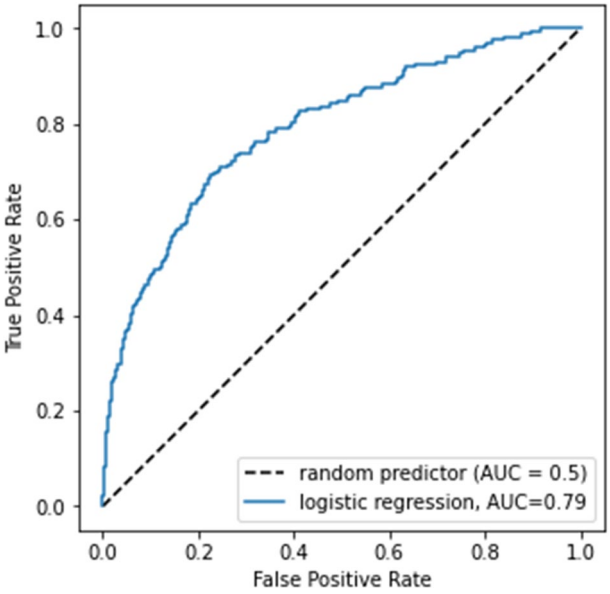

Fig. 2 ROC-AUC curve for the baseline model

Kamei et al. (Kamei et al. 2012) argued that a high recall value returned by their work (i.e., 0.67) was valuable, even with low precision values. Menzies et al. (Menzies et al. 2012) noted that, prediction models with high-recall and low-precision are extremely useful in many situations in industry. This is because practitioners can manage with the idea of checking more files or commits than needed as long as they can avoid the worst situations such as crashes. However, developers might not tolerate if there are high rates of false alarms. Previous studies (Christakis and Bird 2016; Bessey et al. 2010) recommended that less than 20% of false positives would be ideal for developers to continue using the code analysis tools. For example, results of a survey (Christakis and Bird 2016) that analysed developers experience with static analysis tools shows that 90% of developers are willing to accept up to a 5% false positive rate, 47% of developers are willing to accept a false positive rate up to 15%, and only 24% of developers accept a false positive rate as high as 20%. Therefore, models trained with traditional metrics are not ideally suited for practical use due to their high rates of false positives. Consequently, we extend our investigation towards enhancing prediction performance in answering the other research questions, RQ3 and RQ4.

Table 5 Precision, Recall, F-measure and AUC scores for different classifiers

Model	tenfold cross validation									
	Precision		Recall		F-measure		AUC		MCC	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
Logistic Regression (LR)	0.07	0.08	0.66	0.68	0.14	0.14	0.79	0.80	0.16	0.17
Naïve Bayes (NB)	0.07	0.08	0.66	0.67	0.14	0.14	0.77	0.77	0.17	0.16
Decision Tree (DT)	0.09	0.1	0.79	0.63	0.19	0.17	0.79	0.80	0.19	0.17
Random-forest (RF)	0.12	0.12	0.70	0.66	0.21	0.18	0.84	0.83	0.30	0.23
Multilayer Perceptron (MLP)	0.10	0.12	0.50	0.62	0.20	0.17	0.79	0.82	0.30	0.20
AdaBoost (AB) classifier ensemble with DT	0.12	0.16	0.30	0.29	0.20	0.20	0.81	0.82	0.21	0.22
AdaBoost classifier ensemble with LR	0.05	0.06	0.77	0.65	0.12	0.11	0.77	0.79	0.14	0.18
XGBoost classifier	0.23	0.24	0.38	0.39	0.30	0.30	0.84	0.83	0.97	0.29
Time-wise validation										
Random-forest (RF)	0.17	0.07	0.65	0.34	0.25	0.12	0.83	0.77	0.20	0.15
Multilayer perceptron (MLP)	0.22	0.10	0.81	0.28	0.35	0.10	0.93	0.59	0.39	0.10
AdaBoost (AB) classifier ensemble with DT	0.12	0.08	0.79	0.46	0.19	0.14	0.87	0.71	0.25	0.13
XGBoost classifier	0.98	0.14	0.99	0.15	0.99	0.14	0.98	0.72	0.99	0.11

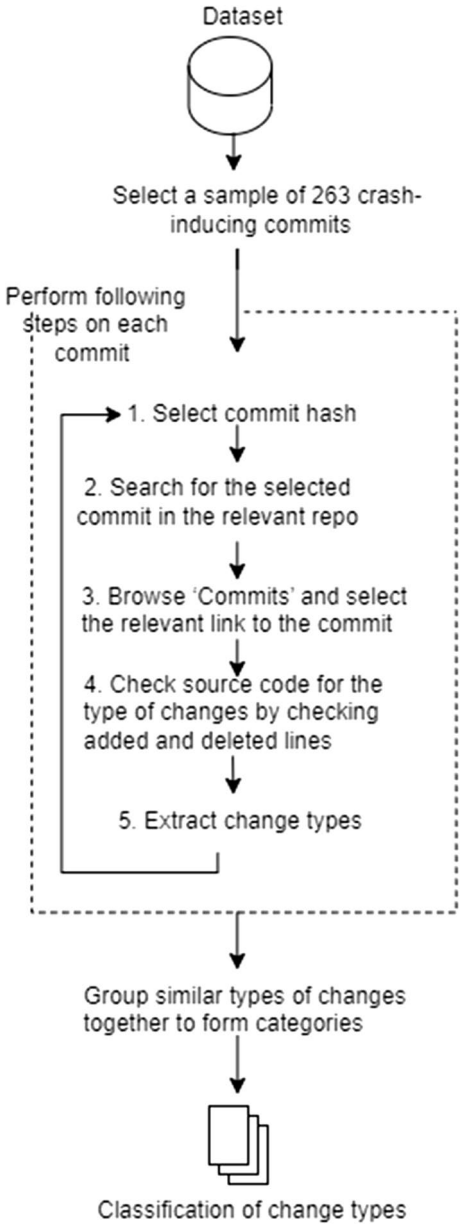

Fig. 3 Manual analysis process for RQ3

4.3 RQ3: What types of changes occur in crash-inducing commits?

Defect prediction models (same as in crash prediction) do not reflect root causes of defects (Kim et al. 2008). Therefore, we performed a manual analysis (Fig. 1: Phase e) to understand the developers' commits in depth, uncovering what types of changes they make in commits and to identify the change types that might introduce crashes. Figure 3 shows the steps of the analysis perform to answer RQ3.

4.3.1 RQ3-aproach

As shown in Fig. 3, first we selected a sample of crash-inducing commits from our dataset for the manual analysis. To select an appropriate sample of crash-inducing commit, the sample size of 263 was determined (from a total of 829 crash-inducing commits) using the Creative Research Systems' Sample Size Calculator[23] while achieving a confidence level of 95% with a 5% confidence interval following a previous study (Shin et al. 2021). Once the sample size was determined, we randomly selected the 263 crash-inducing commits covering all the projects using stratified random sampling (Arnaoudova et al. 2014). The advantage of using stratified random sampling is it ensures that all projects are represented (Black 1999). This means that we first calculated the proportion of commits for each project with respect to the total population of crash-inducing commits and we use the same proportion for the sample. For example, if the percentage of commits for project K-9 Mail in the dataset is 7% of the total population of crash inducing commits, 7% of the sample must be from project K-9 Mail. This way, we selected the appropriate sample representing each project. Next, we manually analysed the source code of the 263 selected commits to understand what kind of changes happen in these commits and extracted the types of changes.

Below we list the steps of the analysis process for each commit (refer to Fig. 3 for visualisation of the process).

1. First, select the commit hash to be analysed.
2. Open the relevant Git project repository of the commit to be analysed (the URLs for the related projects are provided in Table 1) and search for the commit hash.
3. Browse 'Commits' and select the relevant link to the commit. Mostly this link is directly linked to the commit, but there can be indirect situations depending on different contexts (e.g., when commits are merged) and due to different developers' practices. For example, in some projects like Anki-Android some commit hashes appear only in developers' conversations: link to commit 03f68bd is here.[24] Such situations may require browse 'Issues' instead "Commits". Also, there may be multiple commits under one issue, in that case select the relevant commit hash to directly link to the commit.
4. Then, to analyse and understand the changes made in each commit, analyse the source code, focusing on added and deleted lines to understand the type of change. In general, if a commit is only about fixing a previous bug/crash, it is easier to understand those changes than the changes made to add new features or feature extensions. Usually, the commits adding new features are spread across several files and such commits mostly involve multiple types of changes.

   Alternatively, analyse the commit title/commit messages or linked issues (when a previous bug is fixed in the commit), since such sources may also contain information related to the type of changes. However, such information is not provided or not detailed enough from a technical perspective in most of the cases. For, example the commit title "Cleanup and update of Decks.java."[25] is very abstract and not providing any details

[23] www.surveysystem.com/sscalc.htm

[24] https://github.com/ankidroid/Anki-Android/pull/8640

[25] https://github.com/ankidroid/Anki-Android/pull/3861/commits/c0830de6c00eb4db88010a57540416a5c5ef1f5d

from a technical perspective with regards to the source code changes, whereas the commit title "Trim prefixing spaces in FileUtils.getPath.."[26] provide sufficient details to understand technically that it is related to handling a file and some changes in the file path.

5. Extract change type and repeat step 1 to 5 for each commit to understand the types of changes that occurred in the commits.

Changes were then abstracted by grouping similar types of changes together. To form these abstract categories, we referred to Android documentation, development tutorials and blogs. The classification is explained with more details in Section 4.3.2 and it consists of eight categories as explained below:

- ***App core components:*** changes involve core components (e.g., Activities) of an Android app and components' life cycles.
- ***Android UI components and app utilities***: changes involve UI related components such as layout classes (e.g., ViewGroup and ListView), menu, widgets and so on.
- ***External resources***: changes related to handling external resources such as files, images, database or hardware (e.g., camera and hard disk).
- ***List and arrays***: changes related to Lists and Arrays.
- ***Third-party dependencies***: changes related to third-party services/libraries (e.g., HTTP communication).
- ***Threads and Runnable***: change related to threads.
- ***Deprecated API***: changes related to handling deprecated APIs.
- ***Other:*** changes related to general Java programming errors.

Since above changes are common and frequent in crash-inducing commits, such changes might have the potential to introduce crashes. To verify that, we reviewed these change types against the reasons of previous crashes provided by developers in their commits logs. Developers usually mention the reasons for crashes when they attempt to fix a previous crash. As such there were 26 out of 263 commits that contained fixes to previous crashes with reasons provided for the crashes. Thus, by looking at these commits and relevant issue titles, we can gauge the different types of reasons for crashes. This evidence is very useful and helps us understanding what caused the crashes.

Reliability check Once the above analysis is performed by the first author, two additional authors independently reviewed the commits, extract changes and grouped them. The first author first developed the scheme in a bottom up manner based on a sample of 80 commits (it is 30% of 263 commits following the practice of previous studies (Pascarella et al. 2018)), which was reviewed against the previous literature and blogs and it was then reviewed by the second author, where input was provided for its refinement. Then the first author analysed the data with the revised scheme, before the second and third authors performed reliability tests. Based on the review process we improved the details

[26] https://github.com/TeamAmaze/AmazeFileManager/commit/0b80beba760e7827ecc26eac2d64c72bac07bbf2

Table 6 Distribution of different characteristics of crash-inducing commits

App	# commits	1.Core comp	2.UI comp. & utilities	3. External resources	4 Lists & Arrays	5 Third party	6 Threads	7 Deprecated API	8. Other
K-9 Mail	19	5(26%)	2(11%)	7(37%)	2(11%)	4(21%)	4(21%)	-	3(16%)
ConnectBot	1	-	1(100%)	-	1(100%)	-	-	-	-
My Expenses	6	-	4(67%)	2(33%)	-	-	-	-	2(33%)
Anki-Android	24	13(54%)	7(29%)	7(29%)	7(29%)	4(17%)	5(21%)	2(8%)	3(13%)
Twidere	47	33(70%)	21(45%)	24(51%)	16(34%)	11(23%)	14(30%)	7(15%)	7(15%)
AntennaPod	47	18(38%)	21(45%)	9(19%)	9(19%)	14(30%)	6(13%)	5(11%)	8(17%)
AmazeFileManager	22	7(32%)	2(9%)	9(41%)	5(23%)	10(45%)	11(50%)	1(5%)	-
FastHub	40	33(83%)	17(43%)	13(33%)	14(35%)	12(30%)	6(15%)	3(8%)	2(5%)
TuCanMobile	7	4(57%)	2(29%)	-	5(71%)	3(20%)	5(71%)	4(57%)	-
Transdroid	15	11(73%)	11(73%)	-	3(20%)	6(40%)	1(7%)	2(13%)	1(7%)
AnyMemo	1	1(100%)	1(100%)	1(100%)	-	-	-	1(100%)	-
KISS	14	1(7%)	2(14%)	1(7%)	3(21%)	1(7%)	4(29%)	4(29%)	5(36%)
Poet Assistant	20	11(55%)	7(35%)	9(45%)	3(15%)	2(10%)	10(50%)	3(15%)	1(5%)
TOTAL	263	137(52%)	98(37%)	82(31%)	68(26%)	67(25%)	66(25%)	32(12%)	32(12%)

of instructions which are provided in the five (1–5) steps of analysis above and discussed the categories in the classification to achieve consensus. Finally, a formal reliability test was performed by the second and third authors on a sample of 26 crash inducing commits i.e., 10% (similar to a sample size of a previous work's reliability check, e.g., (Zein et al. 2016)) of the total crash inducing commits (263 commits). The authors independently characterized the commits and inter-rater agreement was calculated using Cohen's Kappa (Cohen 1960) which returned 0.7, indicating good agreement between the authors (Landis and Koch 1977). When there were disagreements, the final characterization was performed based on discussions until consensus was reached among all authors, and thus we achieved 100% agreement.

4.3.2 RQ3-results

Based on the manual analysis performed above, below we answer RQ3. We noted seven different types of changes that focus on seven different artefacts in the changes made in crash-inducing commits. These changes are related to: (1) App core components, (2) Android UI components and app utilities, (3) External resources, (4) List and arrays, (5) Third party, (6) Threads and Runnable, (7) Deprecated APIs and (8) Other. Table 6 shows the distribution of each type of changes of different apps. Since multiple types of changes can occur in one commit percentages do not add up to 100. We noted that developers tend to do mistakes while performing these types of changes and such mistakes could lead to crashes.

Next, we explain these categories of the classification. All eight categories are analysed, and contextual evidence also provided referring to development tutorials, blogs and Android documentation (URLs of these sources are provided).

- **App core components**. Android apps have four main components known as essential building blocks of an Android app: Activities, Services, Broadcast receivers and Content providers.[27] Each component serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.[28] From our selected sample of crash inducing commits, most of the commits (137 commits, see Table 6) include changes related to Android components (mostly Activity and Fragments) and their lifecycles. The Activity class provides six core callbacks for transitions between the six stages of the Activity life cycle. Activity hosts Fragments, and Fragments have a more complicated life cycle which has 12 core callbacks. Crash-inducing commits appear to often involve Activity/Fragment lifecycle related changes, indicating that developers do not properly handle these types of changes. For example, in the app KISS, commit 6266019 is a fix to a crash caused by improper use of intent. Here developers ignored the fact that when calling onNewIntent(); onResume()will also be called after onNewIntent(), so that some operations were moved from onNewIntent() to onResume() since those operations have to wait until the Activity starts. Intent is mostly used to launch activity, send broadcast receiver, start services and communicate between two activities.

[27] https://developer.android.com/reference/android/app/package-summary
[28] https://developer.android.com/guide/components/activities/activity-lifecycle

In addition to components' lifecycle, developers seem to forget other important Android rules related to these components such as configuration and initialization rules. In Tucan, commit af82713 is fixing a crash related to the changing of orientation. This was fixed by adding 'android:screenOrientation="portrait"' to Activity declaration in the Manifest file. In the app Ankidroid, commit 995e0a1 fixed a crash caused by incorrect instantiate of fragments (Android can't instantiate fragments since they are anonymous classes).

- **Other Android UI components and app utilities.** Apart from the above core components of the Android application model, there are other UI components such as layout classes (RecyclerView, ViewGroup, ListView), menu, widgets and other application utilities such as dialogs, notifications, action bar and preference in which developers commit changes in the codes very often. These types of changes are the second most frequent in in crash inducing commits (98 commits). In particular, developers seem to make mistakes while defining event listeners, passing arguments and calling methods for objects of these classes which cause exceptions such as NullPointerException and IllegalArgumentException. For example, improper calling of the dismiss() method on ProgressDialog (i.e., to remove the dialog from the screen) caused an IllegalArgumentException in the app Tucan commit 02e046e. This was fixed by adding a condition to check whether the dialog is not null. Another common type of change in this category involves accessing app resources. App's resources are the additional files and static content that code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and so on. These resources should be placed inside the project's 'res/' directory and accessed using resource IDs that are generated in the project's R class. When a requested resource cannot be found and there are invalid referencing to these resources, the resource API throws a Resources.NotFoundException.[29] For example, issue #676[30] in the Twidere-Android app is one such instance.
- **External resources**. Changes related to handling external resources are involved in 81 commits. Based on the selected sample, these resources are mostly files, images, database or hardware (e.g., camera and hard disk). Developers need to be more attentive while dealing with these resources since there is a chance for crashes to be introduced due to a lack of permission to the resources, inexistence of requested resource, invalid file handling/file write operations (e.g., invalid URLs to file paths) and incorrect database queries. For example, mistakes in string operations (splitting, substring, toString), ignore case, invalid spaces in URI prefix and suffix in URLs can lead to crashes. In the app AmazeFileManager (bug #2156),[31] developers were unable to fix a StringIndexOutOfBoundsException in one go and went through several commits (e.g., 0b80beb) to fix the crash while applying several string operations. With regards to database accesses, in the Poet-Assistant app, a crash occurred due to too many records returned by a single SQL query. This was fixed by splitting the query into smaller queries and aggregating the results later.[32] Handling images cause crashes due to failures in compressing, decoding, rotating and so on. For example, the app Anki-droid crashed due to compression failures,[33][34]. This is because it is unable to perform a compression opera-

[29] https://developer.android.com/reference/android/content/res/Resources.NotFoundException
[30] https://github.com/TwidereProject/Twidere-Android/issues/676
[31] https://github.com/TeamAmaze/AmazeFileManager/issues/2156
[32] https://github.com/caarmen/poetassistant/commit/7221c2e889e7e0afd3f605de4bd2bdd9f81a26f6
[33] https://github.com/ankidroid/Anki-Android/pull/5849
[34] https://github.com/ankidroid/Anki-Android/issues/5513

tion when the image is null. Also, BitmapFactory.decodeFile returns null when there is an invalid file paths to the image or when Android cannot decode the bitmap image (e.g., if the image is too large[35]).

Android recommends using image libraries such as Glide to load images in the most optimized manner[36]. Some apps follow this guideline to handle images efficiently. App AmazeFileManager integrated Glide to fix crashes (e.g., in commit edbc931). Glide provides a flexible API that supports fetching, decoding and displaying video stills, images, and animated GIFs. That said, the commit edbc931 is labelled as crash inducing. This is due to developers making mistakes while dealing with third party libraries (further explained in a next category) since they are not fully aware or familiar with such libraries.

- **List and arrays.** Another common type of crash-inducing change that happened frequently in commits is related to Lists and Arrays (68 commits). Developers seem to make mistakes while defining, iterating and referring to indexes of lists. Such mistakes are indexing errors when Lists are associated with UI layout components such as ListView/TextView via Adapter (Adapter is a bridge between UI component and underlying data for that component). For example, ListAdapter is an extended Adapter that is the bridge between a ListView and the data that backs the list. In app AmazeFileManager, commit 726ead1fixed resulted in a crash on item selection due to the current item's index being not valid with regards to RecyclerView adapter size. Conversations among the development team while fixing this issue show that they lack a proper understanding on why the index is greater than the recyclerview adapter size which is a value returned by Android. Other mistakes also can be seen such as missing conditional checks (e.g., null checks) while accessing lists. In app Tucan, commit 49120bd resulted in the throwing of an exception due to the missing of an equal check to compare dates.
- **Third-party dependencies.** Using well-known third-party services/libraries is a preferable choice for developers compared to writing code from scratch, especially for tasks such as HTTP communication, image loading and so on. Different apps depend on different third parties libraries based on the app's context. Attempting to deal with communication protocols such as POP3, SSH/SFTP, SMB and so on (AmazeFilemanager,K9), RSS feed (Antennapod), org.json library (Twidere, Ankidroid), Ngram Viewer dataset (PoetAssistant), and Glide image loading library (AmazeFileManager) are such third-party issues. Since these services are developed by other organizations, developers lack knowledge on their correct usage (Huang et al. 2019), and they make mistakes with regards to these services. Usually commits included in this category (67 commits) cause crashes due to passing incorrect parameter values/formats, using invalid URLs of services or forgetting to validate availability of specific services before use. For example, not validating the availability of RSS feed before use[37] and incorrect json formats (commit c25e6e5 in Twidere-Android and commit 1afeb08 in Anki-Android), caused NullPointerExceptions. In fact, org.json is known as one of the most exception-prone libraries (Su et al. 2020). Incorrect URLs to access RSS feed (e.g., http:////example.

[35] https://stackoverflow.com/questions/8442316/bitmap-is-returning-null-from-bitmapfactory-decodefile filename.

[36] https://developer.android.com/topic/performance/graphics/load-bitmap

[37] https://github.com/AntennaPod/AntennaPod/pull/2753

com, commit bde86e0 in AntennaPod) or no "http" at the beginning[38] due to improper leading and trailing white space in URLs caused URLsyntaxException. Some crashes can also be due to the API incompatibility between the library and the Android SDK/API version and lack of required hardware support and permission (Su et al. 2020).

Furthermore, it seems that developers lack knowledge about handling third party dependencies with regards to Android components. For example, during the commit 91194ef in app AmazeFileManager, developers' conversations questioned the need to append a trailing slash to file loading in SMB server and also focussed on where exactly to include the relevant code segment with regards to Android AsyncTask and Fragments[39].

- **Threads and Runnable.** The next most frequent type of crash-inducing Android app change is related to threads (66 commits). A single GUI thread model is usually used in Android, i.e., UI thread is the main thread which is in charge of handling UI updates and dispatching the events to the appropriate widgets. Each app owns one UI thread and should offload intensive long-running tasks to background threads to keep the app responsive. Loading long-running tasks in UI thread causes app freezes, unresponsiveness and crashes. Thus, developers should avoid lengthy operations on the UI thread. Hence, they should probably use extra threads (background or worker threads) to perform these long-running operations. However, the Android rule[40] is to update UI elements only by the main thread (Android OS does not let other worker threads update UI elements). That said, developers do not seem to follow these thread related principles and violate the rules, which can lead to crashes. For example, in commit 7f03292 in the Poet-Assistant app, the crash was fixed by using the post method (View.post(Runnable)) to avoid updating UI from a worker thread. Android offers several ways to access the UI thread from other threads: View.post(Runnable), Activity.runOnUiThread(Runnable) and so on[41]. Concurrency utilities (java.util.concurrent) or AsyncTask class are also used to handle long running tasks. For example, in the AmazefileManager app, commit 1b37a24 is a fix to a crash due to copying and pasting on a remote server. To fix this crash, some SMB file operations were moved to run in the background thread using callable. The Callable interface, java.util.concurrent.Callable, represents an asynchronous task which can be executed by a separate thread similar to Runnable[42]. However, the related code segment keeps changing in later commits (e.g., commit 4a07c24) involving conversations on app hanging[43], which indicates that developers still make mistakes while applying these techniques, which could introduce crashes to the code again.
- **Deprecated APIs.** The rapid evolution of Android API and Android fragmentation (range of hardware devices supporting Android) is unavoidable since Android continuously provides new features to keep up with users' emerging needs. Both fragmentation and Android API evolution lead to API incompatibility issues in Android apps

[38] https://github.com/AntennaPod/AntennaPod/issues/264

[39] https://github.com/TeamAmaze/AmazeFileManager/pull/2287/commits/91194efbcf500a3b14160fed06c4880511993bfa

[40] https://developer.android.com/guide/components/processes-and-threads

[41] https://developer.android.com/guide/components/processes-and-threads

[42] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html

[43] https://github.com/TeamAmaze/AmazeFileManager/pull/1977

(Scalabrino et al. 2019). During API evolution, some APIs become deprecated, and some new APIs are introduced. As a consequence, app developers also need to quickly react to deprecated APIs by adopting their code to avoid compatibility issues in Android apps. Commits that involve dealing with such issues are included in this category (32 commits). App developers can make their apps support a range of Android versions by declaring the API levels that their apps support, using two attributes in the Android manifest file: android:minSdkVersion and android:targetSdkVersion (Scalabrino et al. 2019; Xia et al. 2020). The minSdkVersion defines the minimum API level required for the app to run, while targetSdkVersion defines the API Level that the application targets. If this is not set, the default value will be equal to the value that was given to minSdkVersion. Setting targetSdkVersion is crucial to enable forward and backward compatibility, even though this is sometimes ignored by app developers. For example, at the time of commit e48ac79 in the Tucan app the targetSdkVersion was not defined in the AndroidManifext.xml which is fixed in the current version.

Also, according to the latest Android version, classes such as ProgressDialog, AsyncTask, PreferenceManager, Preference and so on are deprecated and often involved in the commits. Android recommends using alternative classes instead of these deprecated classes to utilize new functionalities without failures. In addition, developers can add conditional checks to verify the current Android version on which the app is running, and based on the results of these checks, instruct the code to which code statements to be executed or which specific APIs should be invoked. If an app invokes an API that is not defined by the API level, it will trigger a "NoSuchMethodError" and result in app crashes (Xia et al. 2020). For example, in the Twidere app, issue #974[44] is about potential crashes occurring due to the use of a deprecated API, which was fixed by adding a condition to specific APIs (CookieSyncManager for lower API and CookieManager for higher API)

- **Other**
  - General programming errors: The changes in the above categories mostly involve Android components and classes. There can be other changes that do not involve Android components, and solely Java programming mistakes such as incorrect access modifier, number format changes (e.g., duration is captured in 'int' when 'double' required), incorrect casting, computational errors, variable initialization and so on, which may also introduce crashes in the code.
  - Unknown: There are some other commits that we are unable to identify what might have caused the commits to be crash-inducing, since these commits include only the changes related to: formatting code, renaming variables, removing unused parameters or changes only in error message. The lack of information, such as poor issue titles or limited developers' conversations, makes it difficult to debug the source code in such commits. One possible reason could be the limitations of RA-SZZ. RA-SZZ is improved to handle only certain types of equivalent changes (changes that do not modify system behaviour) (Neto et al. 2019). Hence, RA-SZZ may still flag some other equivalent changes (e.g., swapping condition of an if-statement when breaking a conditional expression into multiple if-statements, adding/removing 'this' keyword) as bug inducing (Neto et al. 2019, 2018). Furthermore, RA-SZZ may not always find the true locations of a bug that occurred due to external changes

[44] https://github.com/TwidereProject/Twidere-Android/issues/974

(Costa et al. 2016). For example, issue #5852[45] in the Ankidroid app is a crash that occurred due to low memory space on the user's device, which was not introduced by the source code.

4.4 RQ4: How can we further improve JIT crash predictions of mobile apps?

We are also interested in further investigating how we can improve the performance of our JIT prediction models. The metrics investigated in our models, i.e., JIT metrics by Kamei et al. (Kamei et al. 2012) and static analysis warnings by PMD are originally developed using non-mobile software. Despite the fact that these metrics were originally developed with the focus of general defects in non-mobile software, our results for RQ1 and RQ2 showed that these metrics are useful in predicting crashes (not only general defects) in mobile apps (not only non-mobile software) as well. We are concerned about the low precision values of our prediction model, and a potential reason for these low values may be the lack of metrics that capture the context of mobile apps. The addition of more metrics that describe an Android/mobile context may be beneficial to improve the performance of Android apps JIT models. Therefore, we further analysed the types of changes identified in RQ3 to learn whether they can be used as metrics for JIT modelling, so that we can use such metrics to enhance model performance.

4.4.1 RQ4-approach

Figure 4 shows the analysis process that was used for answering RQ4.

As shown in Fig. 4(1), first the extracted change types of commits analysed for answering RQ3 were transformed to numerical values, with the purpose of using the change types as metrics for crash prediction. As such, all types of changes are transformed to the binary form (0 or 1, for true or false). Table 7 shows how we transformed these different aspects of changes to numerical form using some commits from our dataset. For example, if a change in a commit focuses on 'Android core component', the corresponding record is marked as '1', else '0'. Note that we focus only on common types of changes that we identified from our manual analysis in RQ3. For example, the lines added and deleted in commit 902b3a[46] (see Table 7) show that the changes are related to Java imports and Android UI elements. Therefore, "Other" and "UI component & utilities" metrics were assigned as '1' and the remaining metrics are assigned '0'. The changed lines in commit 7b312c[47] does not involve Android specific changes other than a removal of a normal Java class. Hence, only "Other" for that commit was assigned to '1' and the remaining metrics are '0'. Commit c9c15c involves two types of changes: "Core Components" and "Third Party" related changes. This way, the new metrics values for the crash-inducing commits sample were assigned.

Since our purpose is only to understand the predictive ability of different types of changes happen in the commits and not prediction at this point, we performed some preliminary modelling using a one-class classification technique, i.e., isolation forest, since we only have data for one class (data labelled only for crash-inducing commits) as dependent variable at this stage. Isolation forest is known as an anomaly detection technique which can also be applied in a JIT context to predict defects (He et al. 2017). Here, defective

[45] https://github.com/ankidroid/Anki-Android/issues/5852
[46] https://github.com/Tyde/TuCanMobile/commit/902b3a0596c92e1a9156366596669152745eb1da
[47] https://github.com/Tyde/TuCanMobile/commit/7b312c14d785685608639e17e91a465583a64e22

commits are considered as anomalies since they are rare and irregular compared to normal clean commits. We applied isolation forest to the crash-inducing commits sample, by splitting it into training and test set (20%). Isolation forest was able to predict all the cases in the test set correctly. This indicates that manually extracted characteristics can be used as features and have the ability to enhance JIT prediction models' performance.

Therefore, our next focus was to combine these new Android apps/mobile specific metrics with existing metrics (14 metrics in Table 3) and analyse prediction performance to check whether the new metrics can enhance the prediction performance. For this purpose, we add back data from non-crash-inducing commits (refer to Fig. 4(2)), using the same sampling approach used while selecting the sample for crash-inducing commits in RQ3. This is because we need data from both classes to apply random forest. Therefore, we manually extracted the same set of mobile specific change characteristics, converted to numeric values (as done for crash-inducing commits in Table 7) to obtain metrics for non-crash-inducing commits making the total sample size of 642 commits (263 of crash-inducing and 379 non-crash-inducing commits). The proportion of change types within non-crash-inducing commits was reported following the same form of analysis conducted for crash-inducing commits in RQ3 (refer to Table 8). Additionally, a Fisher's exact test was performed to verify whether the number of change types are indeed statistically different between crash-inducing changes and non-crash-inducing changes. Following the same pipeline that we used to build JIT models to answer RQ2, we re-trained the random forest with the combined features and evaluated its performance. Finally, an ablation analysis was conducted to understand the contribution of each metric type to the performance outcomes.

We also checked odds ratio values of these new metrics and variable importance scores. For this, we re-run logistic regression and random forest on the sample of 642 commits and analyse odds ratio values based on logistic regression and variable importance scores based on AUC score returned by the random forest model (varimpAUC() function from 'Party' package[48] in R was used for this).

4.4.2 RQ4-results

Table 8 reports the change type distribution in non-crash-inducing commits. This table follows the structure of Table 6, reporting the results specific to non-crash-inducing commits. In comparison to the change type distribution reported in Table 6, Table 8 reveals lower percentages for all types, except for the type labelled 'other'. For instance, in the case of core component-related changes, 26% of such changes were observed in crash-inducing commits (as per Table 6), while only 2% were found in non-crash-inducing commits for the K-9 project (as per Table 8). This result shows that Android-specific change types tend to be more prevalent in crash-inducing commits. On the other hand, 'other' types (which are not Android-specific) occur with higher frequencies in non-crash-inducing commits. For example, 78% of non-crash-inducing commits (as shown in Table 8) involve 'other' types, such as general Java programming-related changes for K-9. In contrast, only 16% of crash-inducing commits involve 'other' types of changes. This analysis shows a clear difference in change type distribution between crash-inducing and non-crash-inducing commits, with Android-specific changes dominating in crash-inducing commits. The results of a Fisher's exact test reported a p-value much lower than the commonly accepted p-value (<0.05)

[48] https://cran.r-project.org/web/packages/party/party.pdf

indicating a statistically significant difference in the number of change types between crash-inducing and non-crash-inducing commits.

Table 9 shows the performance scores of the model after adding new metrics. Comparing the results of tenfold cross validation for the random forest model in Table 5 and 9, we observe that we can improve the precision score from 0.12 to 0.85, recall score from 0.66 to 0.84, F1-score from 0.18 to 0.84, AUC score from 0.83 to 0.94 and MCC score from 0.23 to 0.73 when new metrics are combined with the previous metrics set. This shows that combining metrics from traditional JIT defect prediction with the metrics related to Android context can substantially enhance model performance when predicting crashes at the commit level for mobile apps.

Furthermore, Table 10 reports the results of ablation analyses, showing the impact of each metric group. For this purpose, we analysed the prediction performance by removing individual metric groups under two validation settings (tenfold cross validation and time-wise validation). According to both validation settings, we observe that the highest performance across all measures is achieved when all types of metrics are combined. In particular, the MCC score is much higher (MCC is 0.73 for tenfold cross validation) when all three types of metrics are combined. Therefore, adding the new mobile specific metrics are beneficial for producing more effective JIT crash prediction models. Considering the tenfold cross validation, the new metrics solely contribute to enhance performance by 0.36 (0.73–0.37) in terms of MCC score. The performance based on time-wise validation is slightly lower compared to the results obtained from tenfold cross validation. This can be expected since tenfold cross validation tends to achieve more optimized training with a random selection of folds, allowing for model building and validation based on diverse sets of training and testing data (Falessi et al. 2020). This is a less conservative approach compared to time-wise validation, which involves training models on past data and testing them on more recent data, where data imbalance issues are likely to be more significant. It is interesting to see that there is only a small difference between #5 and #7 (with and without PMD) in terms of both AUC (0.92) and MCC (0.62, 0.66) scores with respect to time-wise validation. This may indicate that the predictive power of PMD metrics may shift over the lifetime of projects and the model may need to be retrained overtime by including different metrics related to the programming language being use.

Figure 5 shows the values for variable importance scores returned by random forest. Inside parenthesis, we also include odds ratio values for the new metrics (as we have analysed the same for existing metrics in Section 4.1) which provide some understanding about the nature of relationship with likelihood of a commit to be crash inducing. All the new metrics are risk increasing factors except the metric "Other" which is risk decreasing (odds ratio < 1). Hence, the greater the number of these risk increasing changes in a commit, the more the likelihood of a commit to be crash-inducing. Among these seven types of changes, changes focusing on Threads (6.09) and Core_component (5.17) have the strongest relationship with crashes. Hence, developers need to be the most careful while handling changes related to "Threads" and "Android core components". The values for variable importance scores returned by the random forest model provide information from a modelling perspective: the metric "Other" has the highest variable importance value, which means that "Other" contributes the most to the prediction outcome achieved by random forest model. The metric "Other" represents mainly the general programming changes that do not fall into the seven types of mobile specific changes in our classification of changes (refer to Section 4.3). Commits involving only the "Other" type of changes are more likely to not introduce crashes. Therefore, the "Other" metric contains valuable information that

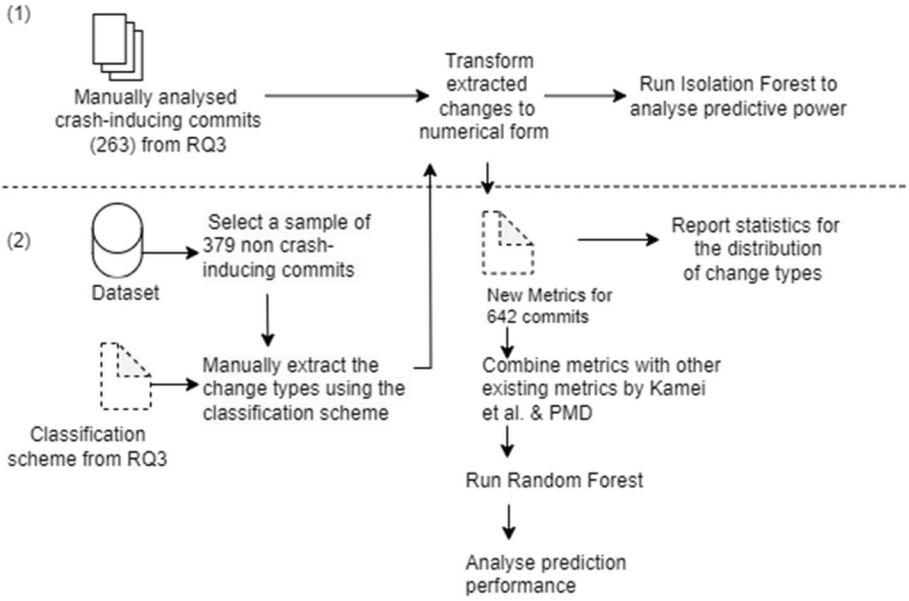

Fig. 4 Analysis process for answering RQ4

might be highly relevant for differentiating commits that are likely to crash or not. Furthermore, we observe that some new metrics show higher variable importance scores than some PMD and Kamei, et al. (Kamei et al. 2012) metrics. For example, the first three highest importance scores are for metrics 'Other', 'Core_components' and 'Threads' which are some of the newly identified metrics. This means that these new metrics are highly important to increase the performance of JIT crash prediction.

5 Discussion and Implication

In this section, we further discuss our findings and outline implications in revisiting the research questions from Section 5.1 to Section 5.4. We next outline recommendations for research and practice in Section 5.5.

5.1 Importance of existing metrics (RQ1)

In answering our first research question (RQ1) regarding the important metrics for just-in-time crash prediction for mobile apps, we found that both standard just-in-time metrics by Kamei et al.(Kamei et al. 2012) and PMD warnings are important. This is consistent with the results of the study by Alexander et al. (Trautsch et al. 2020) who studied static analysis warning for JIT defect prediction for non-mobile software.

Here we see that number of lines added (LA) and Entropy show the strongest relationship with the crash-inducing commits. LA which indicates code size seems to be among the top most important metrics for both traditional and mobile software (Kamei et al.

Table 7 Transforming types of changes from manual analysis to numerical values

App	Commit hash	1.Core comp	2.UI comp. & utilities	3. External resources	4 Lists & Arrays	5 Third party	6 Threads	7 Deprecated API	8. Other
TuCanMobile	902b3a	0	1	0	0	0	0	0	1
TuCanMobile	7b312c	0	0	0	0	0	0	0	1
AmazeFileManager	edbc93	1	1	1	1	1	0	0	0
AmazeFileManager	c9c15c	1	0	0	0	1	0	0	0

Table 8 Distribution of different characteristics of non-crash-inducing commits

App	# commits	1.Core comp	2.UI comp. & utilities	3. External resources	4 Lists & Arrays	5 Third party	6 Threads	7 Deprecated API	8. Other
K-9 Mail	64	1(2%)	3(5%)	7(11%)	5(8%)	11(17%)	1(2%)	1(2%)	50(78%)
ConnectBot	15	-	1(7%)	1 (7%)	-	-	-	-	14(93%)
My Expenses	64	3(5%)	7(11%)	8 (13%)	2(3%)	-	-	2(3%)	64(100%)
Anki-Android	72	1(1%)	4(6%)	2 (3%)	1(1%)	2(3%)	1(1%)	1(1%)	69(96%)
Twidere	23	-	1(4%)	1 (4%)	-	-	-	2(9%)	22(96%)
AntennaPod	49	4(8%)	5(10%)	9 (18%)	3(6%)	4(8%)	-	-	34(69%)
AmazeFileManager	30	1(3%)	2(7%)	2 (7%)	1(3%)	1(3%)	-	-	28(93%)
FastHub	8	1(13%)	2(25%)	1(13%)	-	3(38%)	-	-	5(63%)
TuCanMobile	4	-	1(25%)	-	-	-	-	-	4(100%)
Transdroid	8	-	-	-	-	-	-	-	8(100%)
AnyMemo	19	-	-	-	2(11%)	-	-	-	18(95%)
KISS	19	1(5%)	-	1(5%)	1(5%)	-	-	-	16(84%)
Poet Assistant	4	-	2(50%)	-	-	-	-	-	3(75%)
TOTAL	379	12(3%)	28(7%)	32(8%)	15(4%)	21(6%)	2(1%)	6(2%)	335(88%)

Table 9 Performance of Random Forest model after adding new metrics from manual analysis

Metrics	Score	Precision	Recall	F1	AUC	MCC
Existing metrics (metrics by Kamei et al. (Kamei et al. 2012) and PMD) +New metrics (Android specific features)	Train-score (tenfold cross validated)	0.88	0.85	0.87	0.95	0.74
	Test-score	0.85	0.84	0.84	0.94	0.73

Table 10 Performance for different metrics groups

	JIT	PMD	New	tenfold cross validation					Time-wise validation				
				Precision	Recall	F1	AUC	MCC	Precision	Recall	F1	AUC	MCC
#1	✓			0.60	0.66	0.60	0.81	0.35	0.64	0.67	0.61	0.77	0.37
#2		✓		0.49	0.57	0.49	0.66	0.20	0.45	0.57	0.47	0.62	0.14
#3			✓	0.70	0.83	0.74	0.92	0.59	0.69	0.81	0.72	0.91	0.61
#4		✓	✓	0.67	0.76	0.70	0.93	0.55	0.66	0.77	0.68	0.92	0.58
#5	✓		✓	0.74	0.82	0.75	0.93	0.59	0.76	0.77	0.74	0.92	0.62
#6	✓	✓		0.61	0.68	0.62	0.84	0.37	0.61	0.65	0.59	0.79	0.35
#7	✓	✓	✓	0.85	0.84	0.84	0.94	0.73	0.79	0.83	0.79	0.92	0.66

2012). This is because the greater the number of lines added, the more likely a defect can be introduced. The other code size related metric LT (#lines of code in a file before the change) shows a less strong relationship. This may be due to the nature of continuously changing source code of mobile apps, as Catolino (Catolino 2017) argue. Due to shorter release cycles, code is changed continuously, therefore the usefulness of metrics computed based on previous commits may or may not be useful. This explains why metrics such as LT is less significant for mobile apps crash prediction. Moreover, changes with high Entropy are more likely to be crash introducing, since developers need to track and recall large number of scattered changes across each file. Since Android apps have different levels of abstraction, such as high-level configuration files (manifest file, layout files, String. xml, etc.) in addition to the changes being made during app code implementation (Sadeghi et al. 2016), changes are scattered across several files. Hence, 'entropy' is an important metric in the mobile context, similar to traditional software.

Furthermore, Android apps use Java programming language. Hence, we found three PMD metrics, which are more important than some of the standard JIT defect prediction metrics. They are, ControlStatementBraces, UseProperClassLoader, and DoNotUseThreads, which are also risk increasing factors. After LA and Entropy, the third most important metric is ControlStamentBraces, i.e., omission of curly braces from if-statements, for/do/while-loops and switch statement. This finding is not surprising given that code changes in Java applications could involve a large number of "if conditional statements", which tends to introduce bugs (Yang et al. 2014). Not using braces as recommended may increase code complexity and affect readability and understandability of the code (Gopstein et al. 2018; Chirila et al. 2011). It is possible that developers make such mistakes due to misunderstanding the code, which might ultimately lead to crashes. Similarly, threads (concurrent) play a dominant role in Java programming to achieve both functional and non-functional properties (Callan et al. 2022; Yu et al. 2018). Hence, UseProperClassLoader, and DoNotUseThreads appear to be important metrics, which are related to threads constrains. Improper handling of threads may cause several issues such as data race and resource leak (Su et al. 2020), which can manifest in some types of exceptions.

The next important metric is NDEV, i.e., number of developers that touched the modified files in a commit. Although this is a risk decreasing factor for non-mobile software (Kamei et al. 2012), it is a risk increasing factor for mobile apps. This might be due to the nature of mobile app development. Even though the number of developers in the

development team increased, it is a common problem that developers face difficulties with quickly getting familiar with updated Android versions with the fast evolution of Android platform (Su et al. 2020). More stack overflow posts related to Android specific issues indicates that developers are not as familiar as they are with non-mobile programming such as Java (Su et al. 2020). Furthermore, the feature FIX, i.e., whether a change is a fix to a previous bug is not as important in non-mobile software as it is in mobile apps. FIX was among the top three most important metrics for predicting crashes in the study by Kamei et al. (Kamei et al. 2012), but it is the least important among the selected metrics in our study. This is also the metric with the least contribution (Info. Gain = 0.02, see Table 3) to the predictions, thus, usefulness of the FIX metric can be considered limited. This result is consistent with Catolino et al. (Catolino et al. 2019), where FIX was the least contributing metric in their JIT bug prediction for mobile apps. This can be due to different practices of developers. For example, we noted in our manual analysis that developers use 'fix' keyword in their messages while not only referring to fixing a previous bug but also to refer to the addition/enhancement of features when they close such issues. This might impact computation of the metric, since Commit Guru calculate this metric based on the keywords used in developers' commits logs, which makes this metric less effective.

5.2 Best modelling approach (RQ2)

The results for our second research question (RQ2) regarding the best modelling approach for JIT crash prediction show that random forest is the best machine learning algorithm. This can be expected since random forest is a well-known technique for producing robust and highly accurate results, and is especially resilient to noisy data (Kamei et al. 2016). Also, previous research that explored random forest (e.g., (Pascarella et al. 2019; Trautsch et al. 2020)) in JIT defect prediction showed that random forest is the best performing approach. However, different techniques could perform differently depending on the context, such as, the nature of the dataset (e.g., level of imbalance and skewness of data) or granularity level of predictions (e.g., file level, method-level, or commit-level). For instance, in the study by Xia et al. (Xia et al. 2016), naive bayes achieved the best performance, outperforming random forest for predicting crashes in release-level. Random forest is also less affected by the imbalanced datasets, where it has been shown to outperform other algorithms in detecting defective modules, even though random forest is not necessarily tuned to directly address imbalanced data (Kim et al. 2011). For instance, classifiers such as Multilayer perceptron can be biased towards the dominant class (Kim et al. 2011).

Considering the different modelling approaches we compared (in terms of performance), we noted that crash predictions with traditional JIT defect prediction metrics (metrics by Kamei et al. (Kamei et al. 2012) and static analysis warnings by PMD) tend to yield low precision values. This is a common problem in software defect prediction, mostly due to highly imbalance datasets (Kamei et al. 2012). Models with low precision and high recall values are extremely useful in practice, since developers are willing to put effort on reducing crashes by checking false positives. However, the higher the precision values, the less time consuming it is for developers, since less false positives are evident. Given that precision and recall are based on a threshold value (i.e., 0.5 in this study) and considering the trade-off between the two measures, practitioners can adjust the threshold value to achieve higher recall at the cost of lower

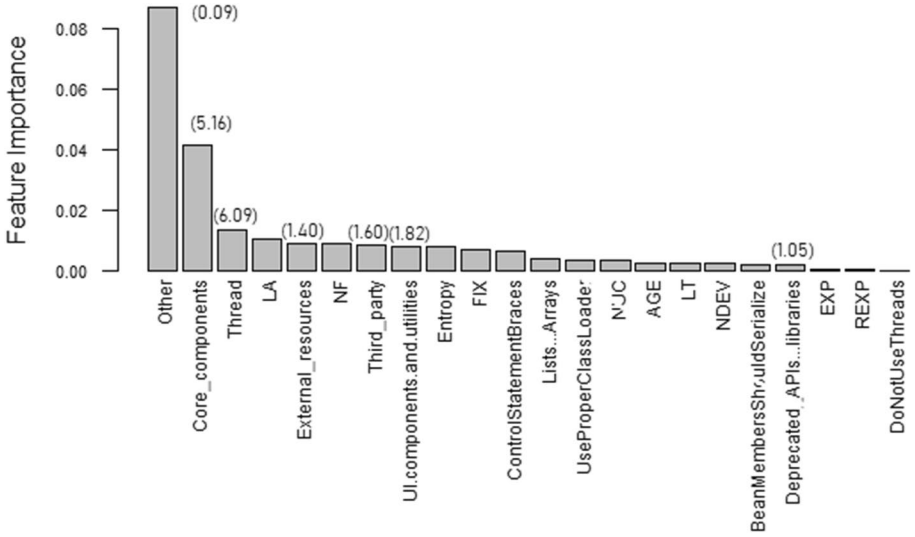

Fig. 5 Variable importance scores returned by random forest model

precision, depending on specific requirements of how much recall or precision matter to the problem in hand (Xia et al. 2016).

5.3 Types of changes (RQ3)

Our manual analysis found that the most common types of changes occur in crash-inducing commits are changes involving modifications related to Android core components and Android UI related components. Changes related to handling external resources, lists and arrays, third-party dependencies, and deprecated APIs are other common types of changes. A previous research by Su et al. (Su et al. 2020) also provides a classification focusing on mobile apps crashes. Their classification is different to ours since they analysed exceptions traces (we analysed commits) of mobile apps and provides a classification of root causes of exceptions where ours is a classification of changes. Their taxonomy includes 12 categories of error types that lead to crashes (for example, concurrency error, UI update error etc.). Their results show that the fixing rate of Android specific errors is lower compared to other Java related errors. On the other hand, our results show that Android component related changes appear as the most frequent type of change in the commits that introduced crashes in our studied mobile apps. Altogether, these findings indicate that developers try to fix Android specific issues by often changing the code. Su et al. (Su et al. 2020) who studied StackOverflow posts on the topic of Android, reported that developers need more discussions when it comes to Android specific issues. This could be due to Android developers not being familiar with the Android domain, while not being able to quickly adapt with the fast evolution of the Android API and features (Kamei et al. 2012).

5.4 Importance of Android specific metrics (RQ4)

Results of RQ4 shows some Android specific metrics are more important than existing JIT metrics for mobile apps. Considering these Android domain specific metrics to form new independent variables has the potential to enhance the performance of prediction models. Developers also can pay more attention while dealing with these types of changes in their commits to limit future crashes. Combining these metrics or adding features that represent mobile specific context with existing JIT defect prediction metrics can improve the performance of prediction significantly, since these metrics carry useful information about crashes specifically related to mobile apps that are not considered in the metrics proposed by Kamei et al. (Kamei et al. 2012) and static analysis warnings by PMD. We note that these types of metrics representing Android mobile context are highly important to achieve a better performance by reducing false positives (which can save more developers' effort) and solve the problem of low precision in JIT approaches for mobile apps.

With regards to previous researchers who explore JIT defect prediction for mobile apps (Catolino et al. 2019; Cheng et al. 2022; Zhao et al. 2021a, 2021b), none of them considered Android specific metrics in their models. Catolino (Catolino 2017) argue that more research is needed in order to define new metrics that are useful to build effective JIT defect prediction models in a mobile context. Answering this research question, we support Catolino et al.'s argument by providing evidence that performance of JIT prediction models can be improved by adding Android domain specific features.

5.5 Recommendations for research and practice

Importance of Existing Metrics Considering metrics by Kamei et al. (Kamei et al. 2012) the number of lines added and entropy have a relatively strong relationship with app crashes compared to other metrics. Hence, if developers are working with long code or changes are widely scattered across several files, such changes should require more attention (e.g., extra code review sessions). For example, a peer review of the code by a senior developer should be considered in these cases. With regards to PMD warnings, we observed that some warnings are related to threads and loops/control statements that have a positive correlation with Android app crashes (Section 5.1). However, developers sometimes ignore fixing static analysis warnings due to the large volume of warnings generated by the tools (Imtiaz et al. 2019). One solution to this is that developers can prioritize warnings (Do et al. 2020), so that developers know which warnings need immediate addressing. Hence, app developers can prioritize some warnings such as ControlStatementBraces and DoNotUseThreads (see Section 5.1), considering their severity, and it would be better for developers not to postpone fixing them. In terms of research, researchers should not ignore static analysis warnings in research related to Just-In-time defect/crash predictions as such metrics can be useful in addition to traditional JIT metrics to optimize the prediction performance, as demonstrated in this work.

Importance of prediction techniques According to a comparison of seven different ML algorithms (see Section 4.2), random forest is the best approach for Just-in-Time predictions of app crashes, i.e., which predicts whether a commit is crash-inducing at the commit time in open-source Android apps. Nevertheless, different techniques could perform differently under different contexts (see Section 5.2). Therefore, we recommend that researchers

should experiment with various classification approaches depending on the different settings (e.g., mobile, non-mobile, open-source, or closed source etc.).

Importance of domain specific metrics In addition to experiment with different ML algorithms (or different data balancing techniques and feature engineering techniques) on existing metrics, researchers should consider defining new metrics that are domain specific. While commonly used metrics from other domains can be adapted to the mobile domain and may be effective to some extent, improvements are needed, especially in terms of low precision values. Hence, it is important to integrate the domain knowledge (in our case, Android specific metrics) with standard metrics to enhance JIT crash predictions for Android apps. Beyond Android, researchers may also be able to adapt more general models to any domain, such as iOS, by contextualizing them with the specific domain metrics.

The new Android-specific metrics used in this study are based on the types of changes (e.g., core components, threads) that are enacted in the commits. Some commits involve multiple types of changes, while others involve only a single type of change. One could argue that predicting commits within the same type of change might be challenging. For example, if a commit is thread-related, whether the model is able to identify when the particular thread-related change is crash inducing or not would be difficult. Thus, we further investigated this issue by manually checking the predicted values against the actual values of the commits within the same type of change. Our analysis found that commits within the same change type are not common in the real world; the majority of commits involve multiple types of changes rather than being only related to one Android component. Out of 642 commits in the data sample used in RQ4, only 130 commits were within the same change type. While this is a small sample, among these 130 commits, 63% (with a recall of 0.78) of commits were predicted correctly based on time-wise validation. Of note is that the time-wise approach was used in order to study successive changes. The recall value of 0.78 indicates an acceptable level of performance (Pascarella et al. 2019; Zheng et al. 2022) in predicting crash-inducing commits within the same change type. We also noted that crashes in third party-related changes are the most challenging to predict. There can be various third-party dependencies that developers have to deal with when accessing such services (refer to Sect. 4.3.2). More commit data involving third-party-related changes is required to capture the variety of third-party dependencies during training to overcome such prediction challenges.

Furthermore, the model can be used to predict crashes of commits, including suspicious commits. It should also be noted that, if developers know beforehand that some change types are crash-inducing they might change their behaviour, subsequently affecting the models' behaviour and performance as well. For instance, if developers started using our models and learned that code components involving threads are most likely to be crash-inducing, they may pay extra attention while committing thread-related changes, in order to reduce the likelihood of defects. The models' behaviour may subsequently change, as the incidences of crashes in thread-related changes would reduce. This situation demands the constant retraining of models to maintain relevance. Keeping models up-to-date when models are in production is still an open problem however (Paleyes et al. 2022). There is never a final version of a machine learning model, which requires regular retraining to adapt to the evolving environment and data (Wan et al. 2019). Monitoring evolving input data and understanding how to trigger warnings when models deviate from normal behaviour are important future research areas (Paleyes et al. 2022). As previous literature (Paleyes et al. 2022; Wan et al. 2019) suggested, implementing feedback loops allows for the

adjustment of inputs to the model, which can influence the model behaviour while it is live and running, which is still a challenging aspect requiring further research.

In addition to the Android-specific metrics, the 'other' class, which involves different types of general programming-related changes, seems to contribute significantly to identifying non-crashing commits, possibly explaining the highest importance score for the random forest. This type can be further divided into change types such as incorrect access modifiers, data types and incorrect casting. Further research aimed at understanding the relationship among these specific types of changes in influencing potential crashes would provide additional insights into identifying risky/non-risky changes.

Furthermore, considering the costs associated with manually extracting these new features, we believe that the extraction process can be automated. A tool can be developed to calculate the number of occurrences of various types of changes (e.g., Android core components, threads, etc.) in a commit. For example, a relevant similar attempt for such an approach can be found in a previous study by Lu et al. (Lu et al. 2017), who employed similar metrics, wherein the number of occurrences of activities and services were calculated. In such cases, if a commit involves excessive use (perhaps based on a threshold value) of such change types, the related metric can be assigned as 'True' otherwise 'False', or the number of occurrences itself could be the metric value. Likewise, future research may consider automating the extraction of such mobile context related metrics.

Limited tool support It is important to note that we have focused only on Java apps including some apps such as Twidere and Ankidroid that use both Kotlin and Java. We excluded apps written 100% in Kotlin because PMD only provides supports for Java apps, meaning that we were unable to find suitable static analysis tools for calculating the required metrics for Kotlin apps. In detail, our analysis of metrics selection started with a large collection of metrics, including some object-oriented/source code metrics calculated by PMD as they are related to reliability. We were unable to calculate these metrics for Kotlin (Kotlin is a combination of both functional and object-oriented programming) due to the unavailability of supporting tools to collect the required metrics. We further confirmed this lack of tool support from a study by Andrä et al. (Andrä et al. 2020), who evaluated available static analysis tools for Kotlin, concluding that there were no suitable tools. Regardless, during the metrics selection process, we noted that object-oriented metrics did not strike as important for JIT crash prediction for Java apps although no evidence to understand their importance in Kotlin apps is available. Furthermore, a tool integrating support for both Kotlin and Java would be useful, since some Android mobile apps use both languages (e.g., app My Expenses is mainly written in Java 69.0% and Kotlin 30.5%, with minor portions in JavaScript and Python). For example, Mateus and Martinez (Góis Mateus and Martinez 2019) and Hecht et al. (Hecht et al. 2015a) selected the tool Paprika (Hecht et al. 2015b) for their studies as Paprika is able to detect both object-oriented and Android specific code smells (which is not our focus in this study) from binaries (apk) of mobile apps. The only tool we found supporting calculation of source code metrics for both Java and Kotlin is CodeMR[49]. Unfortunately, the free version of the tool is limited to only a specific number of classes. Also, CodeMR does not provide APIs to easily access the tool. Therefore, tools like CodeMR, which can calculate several code quality metrics related to code complexity, cohesion, coupling and size, will be useful for research in Android apps

[49] https://www.codemr.co.uk/

related to quality/reliability. Researchers who are interested in app reliability issues can investigate these matters in future research.

Better documentation and training There are several concepts and rules related to Android system and APIs exception handling that are not fully grasped by developers (Tan et al. 2018; Fan et al. 2018b). During our study, we found that this issue is still prominent, since we observed that developers often make mistakes when it comes to Android specific programming, causing crashes. Developers often make changes related to Android core components and components life-cycle, which is frequent in the commits that introduced crashes (refer to Section 4.3.2). Also, we observed that some developers could not fix these issues in one attempt, but needed intense discussions among team members before fixing the problem. However, we acknowledge that some Android concepts are not explained enough in the official documentation. For example, when a developer refers to the Kotlin version[50] of the document, the Activity life cycle is explained without any lifecycle diagram although it is explained using a diagram on Java version[51]. Some diagrams also can be a bit confusing; For example the diagram for handling Lifecycles with Lifecycle-Aware Components[52] presents the activity states as "STARTED" and "CREATED" instead of "PAUSED" and "STOPPED" after calling onPause() and onStop() functions. Developers have raised both these issues in issue tracker (see issue ID 223914384[53], 221,561,985[54]) since these can make specially the junior developers' learning complicated. Also, explanation of some concepts miss code samples, and it would be better if downloadable code samples are provided in both Java and Kotlin. For example, a developer requested a Java version of a code sample for using ViewPager2[55] (see issue 249,936,436[56]) since the given downloadable code sample is in Kotlin. Hence, adding more visual aids and code samples will make documentation easier to understand. Also, due to the rapidly evolving Android system, developers are challenged to remain intimate with the Android API as new features are constantly emerging. A comprehensive documentation that is kept it up-to-date is very important to make app developers fully understand the Android system while reducing programming mistakes.

6 Threats to Validity

As proposed by Perry et al. (Perry et al. 2000), there are three types of threats that should be considered in empirical research studies, namely construct validity, internal validity and external validity. Below we briefly discuss these validity threats identified in our study.

[50] https://developer.android.com/reference/kotlin/android/app/Activity

[51] https://developer.android.com/reference/android/app/Activity

[52] https://developer.android.com/topic/libraries/architecture/lifecycle#lc

[53] https://issuetracker.google.com/issues/223914384?fbclid=IwAR0w7uMJVB0CdGtFRfiKYosmgoAjkoFoLikDuF8fRoyOS_MyEIwNOC47eYg

[54] https://issuetracker.google.com/issues/221561985?fbclid=IwAR0Xzq-b_yd_NwNcDWOjsbP8yQc6tHwUG232R2qO3aruzF9GfFPZqU2Y1j4

[55] https://developer.android.com/jetpack/androidx/releases/viewpager2

[56] https://issuetracker.google.com/issues/249936436?fbclid=IwAR3GhFENykVsVxbEoB7vp9QNQd2JUkgRo4kBNmBZMAj8v2Q_bGDXS3-_nFw

Threats to construct validity This threat refers to the relationship between theory and observation. This is mainly concerned with measurements of the independent variables and dependent variable (Wright et al. 2010). To mitigate this threat, we rely on well-established metrics, tools and techniques such as RA-SZZ, Commit Guru, PMD and ML algorithms. Of course, these approaches are not without limitations. For example, if a defect is not recorded in version control systems, such as GitHub, a change will not be flagged as defect-inducing. Also, output from RA-SZZ (i.e., the commits that are labelled as crash-inducing) are still affected by some types of refactoring changes, and changes may incorrectly be flagged as crash-inducing (Neto et al. 2018). Furthermore, the metrics related to developers' experience (EXP, REXP) computed by Commit Guru are based on the number of modifications applied by a developer (Rosen et al. 2015). It might be possible that a committer is not the actual developer; when new developers are in the team, they may modify the source code but are not allowed to perform a push to the repository (Pascarella et al. 2019). In that case, the actual developer is not the same person as the committer. This may be a reason for the experience related metrics in our model showing a very low contribution to the model (Info. Gain = 0.04, 0.09) compared to other metrics. Catolino et al. (Catolino et al. 2019) received Info Gain values close to ours for the same metrics and argue that computation of these metrics is not effective in a mobile context and new metrics related to developer experience are needed. Nevertheless, the techniques such as RA-SZZ, Commit Guru are the most effective approaches available in the literature, even though the tools bear some limitations. In terms of suitable ML algorithm, we selected them based on the algorithms that have been widely used in software defect prediction.

Threats to internal validity Threats in this category are concerned with errors in the code and experiments (Xia et al. 2016). Since we use our own implementation in some phases of our work (e.g., extract crash issues, develop ML models), each phase was double checked by another author to mitigate this threat. Also, we followed guidelines and the examples provided in scikit learn documentation to implement ML models. As noted in Section 3.2.3, the reliability of links between crash issues and fixes are dependent on the reliability of our implementation and the techniques we used (e.g., keywords-based search). We manually checked the links and filtered out two invalid links to ensure reliability of our work (see Section 3.2.3). However, it is possible that developers do not always link fixes with issue reports or issue IDs with commits logs. In that case, we may miss some links. This is a known issue of detecting links between issues and commit logs (Wu et al. 2011).

Threats to external validity This refers to the generalizability of our results. To mitigate this threat, we consider 14 open-source mobile apps with different characteristics, scope, size and of different domains (e.g., finance and game). At the same time, we considered different ML techniques for predictions. We built and trained models using tenfold cross validation and report the average performance. During cross validation, the test split of the data was unseen to training and 20% of the dataset (i.e., our test set) was held out separately during training. Hence, the built models may be generalized to an unseen dataset. As future works, we need to further extend our dataset to commercial apps, apps developed based on other platforms such iOS and other programming languages such as Kotlin to improve the generalizability of our results.

7 Summary and Future Work

We conducted a comprehensive study to investigate the prediction of crashes in mobile apps. In particular, our focus was to predict whether a new commit is likely to introduce a crash in a mobile app. For this purpose, we adapted JIT defect prediction techniques which were originally developed for traditional software systems. JIT defect prediction techniques are an alternative to traditional defect prediction and able to predict the presence of defects in a commit, i.e., as soon as a change is committed on a repository. While JIT defect prediction has been studied in a few previous works, limited effort was committed to analysing crashes specifically, and especially for mobile apps.

In this study, we started with a quantitative analysis to understand which metrics from Kamei et al. (Kamei et al. 2012), source code metrics and static analysis warnings by PMD are important in a mobile context for JIT prediction. We also explored what ML technique is the best approach to predict JIT crashes. Accordingly, first we applied feature selection techniques in answering RQ1, such as information gain and sequential feature forward techniques with logistic regression to filter only the relevant metrics for crash inducing commits of mobile apps. Then, we compared the performance of seven state-of-the-art classification techniques (i.e., logistic regression, decision tree, naive bayes, random forest, multilayer perceptron, adaboost and xgboost) while tuning hyperparameters in answering RQ2 to achieve the best performing model. The study includes more than 30,000 commits from 14 mobile apps. Next, a qualitative analysis was performed by manually analysing commit logs and source code of a selected sample of 642 commits (263 of crash-inducing and 379 non-crash-inducing commits) to understand what caused crashes and if any common characteristics for crash-inducing commits are evident. A classification of changes in crash-inducing commits was developed based on this analysis. Moreover, our study provides following findings:

- In terms of existing metrics in the literature, both Kamei et al. (Kamei et al. 2012) metrics and PMD warnings are important to identify crash-inducing commits effectively. Some PMD warning showed a higher importance than some metrics proposed by Kamei et al. (Kamei et al. 2012). The most important metrics are the number of lines of code added and entropy (i.e., distribution of modified code across each file). The next three important metrics are related to violations of Java rules by PMD: ControlStatementBraces which is about use of braces on 'if…else' statements, and two others (UseProperClassLoader, and DoNotUseThreads) are related to the use of threads.
- The best performance with existing metrics is achieved by the random forest algorithm, i.e., with AUC 83%, Precision 12%, Recall 66%, F-measure 18% and MCC 23%, (Recall is extremely useful for developers according to related research (Kamei et al. 2012)), outperforming other classifiers including some ensemble techniques. Random forest performed 4–5% better than the baseline model. This shows that the

choice of the ML classification techniques impact on the performance of JIT crash prediction models.

- Additional metrics representing the mobile/Android domain are required to improve crash prediction performance. A preliminary analysis shows that by adding mobile specific metrics with traditional metrics (i.e., those of Kamei et al. (Kamei et al. 2012) and PMD warnings), we improve prediction performance substantially. These mobile specific metrics are based on the types of Android specific changes involved in commits which were extracted manually.

Our study also shows that prediction models and most of the metrics used for JIT defect prediction used for non-mobile software can be adopted to mobile context. At the same time, some metrics need to be modified due to the nature of the development practices in mobile context. Our study opens new challenges for future research. We plan to extend our work to additional mobile apps including Kotlin apps and commercial apps. Based on our findings, we also believe that new metrics representing mobile context are very important to increase prediction performance. Our future work will investigate whether only these metrics are enough on its own to predict crashes. If such metrics are enough on their own (without traditional JIT prediction metrics) for JIT prediction, we may not need all 22 metrics to be collected. If so, cost of data collection can be reduced. However, still there is no automated support to extract these new metrics, hence future works is required to develop supportive tools. Additionally, we will delve deeper into understanding how general programming practices (non-Android-specific changes such as general Java programming related changes) which are captured by the "other" category, impact commits' crashing or non-crashing potential. This more granular level analysis may provide better understanding of (non) risky changes. Quality of commits messages also plays an important role in JIT defect prediction techniques. Hence, researchers can assist with further exploring new techniques or improving existing techniques which can be used by developers to enhance the quality of commit messages. For example, use consistent methods to link fixes or issue IDs in commit logs. In terms of machine learning techniques, we can investigate several directions to improve modelling outcomes including more advanced techniques, such as deep and ensemble learning techniques, as the choice of classifier has an impact on the prediction performance. However, deep learning techniques require large datasets, and such data is not available for JIT crash prediction in a mobile context. Furthermore, given that software developers expect explainable model outcomes (Santos et al. 2020), and our primary goal is also understanding, currently we use only machine learning due to limited explainable power of deep learning techniques. Our future works will explore deep learning techniques in this context. Also, we can further improve in terms of hyperparameter tuning. As suggested by Shu et al. (Shu et al. 2019), hyperparameter optimization on data pre-processing (i.e., to adjust training data by controlling outlier removal or data balancing problems) may be more beneficial than optimizing the classifier. This can be explored in future works, since we have applied parameter tuning only for the classifiers in this study. For example, to learn how to balance the dataset, parameter tuning can be applied to SMOTE and the best setting can be used for modelling. Finally, we hope to evaluate our model from different perspectives. We will extend our work for effort-aware evaluation (e.g., the effort required to inspect the modified code for a change) and for evaluation in industrial contexts.

Appendix A: Selected PMD Warnings for our study

1. AssignmentToNonFinalStatic (ATNFS)
2. AvoidAssertAsIdentifier (AAAI)
3. AvoidBranchingStatementAsLastInLoop (ABSALIL)
4. AvoidCallingFinalize (ACF)
5. AvoidCatchingNPE (ACNPE)
6. AvoidCatchingThrowable (ACT)
7. AvoidDecimalLiteralsInBigDecimalConstructor (ADLBDC)
8. AvoidDuplicateLiterals (ADL)
9. AvoidEnumAsIdentifier (AEAI)
10. AvoidInstanceofChecksInCatchClause (AICCC)
11. AvoidLosingExceptionInformation (ALEI)
12. AvoidMultipleUnaryOperators (AMUO)
13. AvoidUsingOctalValues (AUOV)
14. BadComparison (BC)
15. BeanMembersShouldSerialize (BMSS)
16. BrokenNullCheck (BNC)
17. CallSuperFirst (CSF)
18. CallSuperLast (CSL)
19. CheckSkipResult (CSR)
20. ClassCastExceptionWithToArray (CCEWTA)
21. CloneMethodMustBePublic (CMMBP)
22. CloneMethodMustImplementCloneable (CMMIC)
23. CloneMethodReturnTypeMustMatchClassName (CMRTMMCN)
24. CloneThrowsCloneNotSupportedException (CTCNSE)
25. CloseResource (CR)
26. CompareObjectsWithEquals (COWE)
27. ConstructorCallsOverridableMethod (CCOM)
28. DoNotCallSystemExit (DNCSE)
29. DoNotHardCodeSDCard (DNHCSC)
30. DoNotThrowExceptionInFinally (DNTEIF)
31. DontUseFloatTypeForLoopIndices (DUFTFLI)
32. EmptyCatchBlock (ECB)
33. EmptyFinalizer (EF)
34. EmptyFinallyBlock (EFB)
35. EmptyIfStmt (EIS)
36. EmptyInitializer (EI)
37. EmptyStatementBlock (ESB)
38. EmptyStatementNotInLoop (ESNIL)
39. EmptySwitchStatements (ESS)
40. EmptySynchronizedBlock (ESB)
41. EmptyTryBlock (ETB)
42. EmptyWhileStmt (EWS)
43. EqualsNull (EIN)
44. FinalizeDoesNotCallSuperFinalize (FDNCSF)
45. FinalizeOnlyCallsSuperFinalize (FOCSF)
46. FinalizeOverloaded (FOL)
47. FinalizeShouldBeProtected (FSBP)
48. IdempotentOperations (IO)
49. UnusedNullCheckInEquals (UNCIE)
50. UseCorrectExceptionLogging (UCEL)
51. UseEqualsToCompareStrings (UETCS)
52. ImportFromSamePackage (IFSP)
53. InstantiationToGetClass (ITGC)
54. JumbledIncrementer (JI)
55. JUnitSpelling (JUS)
56. JUnitStaticSuite (JUSS)
57. LoggerIsNotStaticFinal (LINSF)
58. MisplacedNullCheck (MNC)
59. MissingBreakInSwitch (MBIS)
60. MissingSerialVersionUID (MSVU)
61. MissingStaticMethodInNonInstantiatableClass (MSMINIC)
62. MoreThanOneLogger (MTOL)
63. NonCaseLabelInSwitchStatement (NCLISS)
64. NonStaticInitializer (NSI)
65. OverrideBothEqualsAndHashcode (OBEAH)
66. ProperCloneImplementation (PCI)
67. ProperLogger (PL)
68. ReturnEmptyArrayRatherThanNull (REARTN)
69. ReturnFromFinallyBlock (RFFB)
70. SimpleDateFormatNeedsLocale (SDFNL)
71. SingleMethodSingleton (SMS)
72. SingletonClassReturningNewInstance (SCRNI)
73. StaticEJBFieldShouldBeFinal (SEFSBF)
74. StringBufferInstantiationWithChar (SBIWC)
75. TestClassWithoutTestCases (TCWTC)
76. UnconditionalIfStatement (UIS)
77. UnnecessaryBooleanAssertion (UBA)
78. UnnecessaryCaseChange (UCC)
79. UnnecessaryConversionTemporary (UCT)
80. UselessOperationOnImmutable (UOOI)
81. UseLocaleWithCaseConversions (ULWCC)
82. UseProperClassLoader (UPCL)
83. UseTryWithResources (UTWR)
84. AvoidCatchingGenericException (ACGE)
85. AvoidSynchronizedAtMethodLevel (ASAML)
86. AvoidThreadGroup (ATG)
87. DoNotUseThreads (DNUT)
88. DontCallThreadRun (DCTR)
89. DoubleCheckedLocking (DCL)
90. NonThreadSafeSingleton (NTSS)
91. UnsynchronizedStaticFormatter (USF)
92. UseNotifyAllInsteadOfNotify (UNAON)
93. LooseCoupling (LC)
94. CouplingBetweenObjects (CBO)
95. ExcessiveImports (EI)
96. AvoidDeeplyNestedIfStmts (ADNIS)
97. ControlStatementBraces (CSB)
98. CognitiveComplexity (CC)
99. ExcessiveParameterList (EPL)

Appendix B: Information Gain values (> 0.01)

1. BeanMembersShouldSerialize 0.50
2. ld 0.49
3. la 0.49
4. nuc 0.45
5. nf 0.44
6. ndev 0.42
7. ControlStatementBraces 0.41
8. nd 0.40
9. CognitiveComplexity 0.40
10. AvoidDuplicateLiterals 0.32
11. LooseCoupling 0.28
12. AvoidCatchingGenericException 0.27
13. lt 0.21
14. sexp 0.20
15. entrophy 0.18
16. EmptyCatchBlock 0.18
17. CloseResource 0.17
18. AvoidDeeplyNestedIfStmts 0.15
19. UseConcurrentHashMap 0.13
20. AvoidSynchronizedAtMethodLevel 0.13
21. EmptyIfStmt 0.12
22. DoNotUseThreads 0.11
23. UseLocaleWithCaseConversions 0.10
24. ConstructorCallsOverridableMethod 0.10
25. ns 0.09
26. exp 0.09
27. UseProperClassLoader 0.08
28. age 0.07
29. UseTryWithResources 0.06
30. ExcessiveParameterList 0.06
31. CompareObjectsWithEquals 0.05
32. AvoidCatchingNPE 0.05
33. CallSuperLast 0.04
34. CallSuperFirst 0.04
35. NonThreadSafeSingleton 0.04
36. rexp 0.04
37. ReturnEmptyArrayRatherThanNull 0.04
38. MissingSerialVersionUID 0.03
39. AvoidCatchingThrowable 0.03
40. SimpleDateFormatNeedsLocale 0.03
41. SingletonClassReturningNewInstance 0.03
42. CouplingBetweenObjects 0.02
43. EmptyStatementNotInLoop 0.02
44. DoNotHardCodeSDCard 0.02
45. AssignmentToNonFinalStatic 0.02
46. UseEqualsToCompareStrings 0.02
47. OverrideBothEqualsAndHashcode 0.02
48. fix_False 0.02
49. AvoidInstanceofChecksInCatchClause 0.02

Appendix C: List of hyper-parameters optimized in different classifiers

Model	Tuning parameters and values
Logistic regression	penalty: ['l1', 'l2', 'elasticnet', 'none'], tol: [np.random.uniform(0.1, 1.0)], C: [20, 100, 200, 500, 700, 1000, 1500, 1800, 2000], max_iter: [20, 10, 50, 100, 500,700, 1000, 1500, 1800, 2000]
Naïve Bayes	var_smoothing: np.logspace(0,-9, num = 100)
Multilayer perceptron	max_iter: [200, 250, 500, 1000], activation: ['identity', 'logistic', 'tanh', 'relu'], learning_rate: ['constant', 'invscaling', 'adaptive']
Random forest	criterion: ['entropy', 'gini'], max_depth: [int(x) for x in np.linspace(start = 1, stop = 110, num = 15)], n_estimators: [250, 400, 500, 1000, 1500, 2000], max_leaf_nodes: (Zhao et al. 2021a; Williams and Spacco 2008; Shihab 2012; Thitichaimongkhol and Senivongse 2016; Meldrum et al. 2020), max_features: ['auto', 'sqrt', 'log2'], min_samples_leaf: (Inc 2012; Dickerson 2016; McIlroy et al. 2016; Xia et al. 2016), min_samples_split: [0.8, 2,4,6], bootstrap: [True, False]
AdaBoost	n_estimators": [10, 50, 100, 500, 1000, 5000]
XGB	learning_rate: [0.35, 0.5, 0.8], n_estimators": [1000, 1500,2000,4000], max_depth: [20, 50, 100,200], min_child_weight: [0.5, 0.2, 1,3,5,7], gamma: [0.5, 0.6, 0.8], colsample_bytree: [0.6, 0.7,0.8]

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

Data Availability The dataset and scripts we used in this study are available in the online repository: https://tinyurl.com/2p3ntuet

Declarations

Conflicts of Interest The authors declare that they have no conflict of interest.

References

Aljamaan H, Alazba A (2020) Software defect prediction using tree-based ensembles, in Proceedings of the 16th ACM international conference on predictive models and data analytics in software engineering, pp. 1–10

Allison P (2013) in What's the Best R-Squared for Logistic Regression? vol. 2022, ed. Statistical Horizons

An L, Khomh F (2015) An empirical study of crash-inducing commits in mozilla firefox, in Proceedings of the 11th international conference on predictive models and data analytics in software engineering, pp. 1–10

Andrä L-M, Taufner B, Schefer-Wenzl S, Miladinovic I (2020) Maintainability Metrics for Android Applications in Kotlin: An Evaluation of Tools, in Proceedings of the 2020 European Symposium on Software Engineering, pp. 1–5

Arnaoudova V, Eshkevari LM, Di Penta M, Oliveto R, Antoniol G, Guéhéneuc Y-G (2014) Repent: Analyzing the nature of identifier renamings. IEEE Trans Software Eng 40(5):502–532

Asaduzzaman M, Bullock MC, Roy CK, Schneider KA (2012) Bug introducing changes: A case study with android, in 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), IEEE, pp. 116–119

Barnett JG, Gathuru CK, Soldano LS, McIntosh S (2016) The relationship between commit message detail and defect proneness in java projects on github, in 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), IEEE, pp. 496–499

Bessey A et al (2010) A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM 53(2):66–75

Black TR (1999) Doing quantitative research in the social sciences: An integrated approach to research design, measurement and statistics. sage

Boehm BW (1987) Industrial software metrics top 10 list. IEEE Softw 4(5):84–85

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Callan J, Krauss O, Petke J, Sarro F (2022) How do Android developers improve non-functional properties of software? Empir Softw Eng 27(5):113

Carka J, Esposito M, Falessi D (2022) On effort-aware metrics for defect prediction. Empir Softw Eng 27(6):152

Cataldo M, Mockus A, Roberts JA, Herbsleb JD (2009) Software dependencies, work dependencies, and their impact on failures. IEEE Trans Software Eng 35(6):864–878

Catolino G (2017) Just-in-time bug prediction in mobile applications: the domain matters!, in 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, pp. 201–202.

Catolino G, Di Nucci D, Ferrucci F (2019) Cross-project just-in-time bug prediction for mobile apps: an empirical assessment, in IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems, pp. 99–110

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357

Chen T, Guestrin C (2016) Xgboost: A scalable tree boosting system, in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785–794

Cheng T, Zhao K, Sun S, Mateen M, Wen J (2022) Effort-aware cross-project just-in-time defect prediction framework for mobile apps. Front Comp Sci 16(6):1–15

Chirila C-B, Juratoni D, Tudor D, Creţu V (2011) Towards a software quality assessment model based on open-source statical code analyzers, in 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), IEEE, pp. 341–346

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study, in Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp. 332–343

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Measur 20(1):37–46

Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2016) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans Software Eng 43(7):641–657

"Data Leakage And Its Effect On The Performance of An ML Model." https://www.analyticsvidhya.com/blog/2021/07/data-leakage-and-its-effect-on-the-performance-of-an-ml-model/ Accessed Nov 2022

DeMaris A (2002) Explained variance in logistic regression: A Monte Carlo study of proposed measures. Soc Methods Res 31(1):27–74

Di Nucci D, Palomba F, De Rosa G, Bavota G, Oliveto R, De Lucia A (2017) A developer centered bug prediction model. IEEE Trans Software Eng 44(1):5–24

Dickerson J (2016) Mobile apps: what consumers really need and want. A global study of consumers expectations and experiences of mobile applications. [Online]. Available: https://silo.tips/download/mobile-apps-what-consumers-really-need-and-want-a-global-study-of-consumers-expe#. Accessed May 2022
Dietterich TG (2000) Ensemble methods in machine learning. International workshop on multiple classifier systems. Springer, pp 1–15
Ding J, Fu L (2018) A Hybrid Feature Selection Algorithm Based on Information Gain and Sequential Forward Floating Search①. J Intell Comput 9(3):93
Do LNQ, Wright J, Ali K (2020) Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. IEEE Transactions on Software Engineering
Falessi D, Huang J, Narayana L, Thai JF, Turhan B (2020) On the need of preserving order of data when validating within-project defect classifiers. Empir Softw Eng 25:4805–4830
Fan Y, Xia X, Da Costa DA, Lo D, Hassan AE, Li S (2019) The impact of mislabeled changes by szz on just-in-time defect prediction. IEEE Trans Software Eng 47(8):1559–1586
Fan L et al. (2018a) Efficiently manifesting asynchronous programming errors in android apps, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 486–497
Fan L et al. (2018b) Large-scale analysis of framework-specific exceptions in android apps, in IEEE/ACM 40th International Conference on Software Engineering, pp. 408–419
Fenton NE, Neil M (1999) A critique of software defect prediction models. IEEE Trans Software Eng 25(5):675–689
Feurer M, Hutter F (2019) Hyperparameter optimization, in Automated machine learning: Springer, Cham, pp. 3–33
Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N (2014) An empirical study of just-in-time defect prediction using cross-project models, in Proceedings of the 11th working conference on mining software repositories, pp. 172–181
Gardner MW, Dorling S (1998) Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmos Environ 32(14–15):2627–2636
Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models, in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1: IEEE, pp. 789–800
Góis Mateus B, Martinez M (2019) An empirical study on quality of Android applications written in Kotlin language. Empir Softw Eng 24(6):3356–3393
Gopstein D, Zhou HH, Frankl P, Cappos J (2018) Prevalence of Confusing Code in Software Projects, in Proceedings of the 15th International Conference on Mining Software Repositories-MSR 18: 281–291
Gyimóthy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Software Eng 31(10):897–910
Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Software Eng 38(6):1276–1304
"How to Avoid Data Leakage When Performing Data Preparation." (n.d.) Machine Learning Mastery. https://machinelearningmastery.com/data-preparation-without-data-leakage/. Accessed May 2022
He Y, Zhu X, Wang G, Sun H, Wang Y (2017) Predicting bugs in software code changes using isolation forest, in 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, pp. 296–305
Hecht G, Benomar O, Rouvoy R, Moha N, Duchien L (2015a) Tracking the software quality of android applications along their evolution (t), in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 236–247
Hecht G, Rouvoy R, Moha N, Duchien L (2015b) Detecting antipatterns in android apps, in 2015 2nd ACM international conference on mobile software engineering and systems, IEEE, pp. 148–149
Hemmert GA, Schons LM, Wieseke J, Schimmelpfennig H (2018) Log-likelihood-based pseudo-R 2 in logistic regression: deriving sample-sensitive benchmarks. Sociol Methods Res 47(3):507–531
Hoang T, Dam HK, Kamei Y, Lo D, Ubayashi N (2019) DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction, in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp. 34–45
Huang J, Borges N, Bugiel S, Backes M (2019) Up-to-crash: Evaluating third-party library updatability on android, in 2019 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, pp. 15–30
Immaculate SD, Begam MF, Floramary M (2019) Software bug prediction using supervised machine learning algorithms, in 2019 International conference on data science and communication (IconDSC), IEEE, pp. 1–7

Imtiaz N, Murphy B, Williams L (2019) How do developers act on static analysis alerts? an empirical study of coverity usage, in 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp. 323–333

ISO/IEC (2011) ISO/IEC 25010: 2011-Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models, vol 25010. BSI, London

Jović A, Brkić K, Bogunović N (2015) A review of feature selection methods with applications, in 2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO), Ieee, pp. 1200–1205

Kamei Y et al (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Software Eng 39(6):757–773

Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empir Softw Eng 21(5):2072–2106

Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto K-I (2007) The effects of over and under sampling on fault-prone module detection, in First international symposium on empirical software engineering and measurement (ESEM 2007), IEEE, pp. 196–204

Kaur A, Kaur K, Kaur H (2016) Application of machine learning on process metrics for defect prediction in mobile application, in Information Systems Design and Intelligent Applications: Springer, pp. 81–98

Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans Software Eng 34(2):181–196

Kim S, Zimmermann T, Pan K, James Jr E (2006) Automatic identification of bug-introducing changes, in 21st IEEE/ACM international conference on automated software engineering (ASE'06), IEEE, pp. 81–90

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction, in Proceedings of the 33rd International Conference on Software Engineering, pp. 481–490

Koziarski M (2021) CSMOUTE: Combined synthetic oversampling and undersampling technique for imbalanced data classification, in 2021 International Joint Conference on Neural Networks (IJCNN), IEEE, pp. 1–8

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Laradji IH, Alshayeb M, Ghouti L (2015) Software defect prediction using ensemble learning on selected features. Inf Softw Technol 58:388–402

Li N, Shepperd M, Guo Y (2020a) A systematic review of unsupervised learning techniques for software defect prediction. Inf Softw Technol 122:106287

Li W, Zhang W, Jia X, Huang Z (2020b) Effort-aware semi-supervised just-in-time defect prediction. Inf Softw Technol 126:106364

Li R, Zhou L, Zhang S, Liu H, Huang X, Sun Z (2019) Software defect prediction based on ensemble learning, in Proceedings of the 2019 2nd International conference on data science and information technology, pp. 1–6

Lomio F, Iannone E, De Lucia A, Palomba F, Lenarduzzi V (2022) Just-in-time software vulnerability detection: Are we there yet? J Syst Softw 188:111283

Lu X, Chen Z, Liu X, Li H, Xie T, Mei Q (2017) Prado: Predicting app adoption by learning the correlation between developer-controllable properties and user behaviors. Proc ACM Interact Mob Wearable Ubiquit Technol 1(3):1–30

McIlroy S, Ali N, Hassan AE (2016) Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. Empir Softw Eng 21(3):1346–1370

McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction, in Proceedings of the 40th International Conference on Software Engineering, pp. 560–560

Meldrum S, Licorish SA, Owen CA, Savarimuthu BTR (2020) Understanding stack overflow code quality: A recommendation of caution. Sci Comput Program 199:102516

Menzies T et al (2012) Local versus global lessons for defect prediction and effort estimation. IEEE Trans Software Eng 39(6):822–834

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Tech J 5(2):169–180

Munson JC, Khoshgoftaar TM (1992) The detection of fault-prone programs. IEEE Trans Software Eng 18(5):423

Nayebi M, Adams B, Ruhe G (2016) Release Practices for Mobile Apps--What do Users and Developers Think?, in 2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner) 1: IEEE, pp. 552–562

Neto EC, Da Costa DA, Kulesza U (2018) The impact of refactoring changes on the szz algorithm: An empirical study, in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp. 380–390

Neto EC, da Costa DA, Kulesza U (2019) Revisiting and improving szz implementations, in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, pp. 1–12

Okutan A, Yıldız OT (2014) Software defect prediction using Bayesian networks. Empir Softw Eng 19(1):154–181

Paleyes A, Urma R-G, Lawrence ND (2022) Challenges in deploying machine learning: a survey of case studies. ACM Comput Surv 55(6):1–29

Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. J Syst Softw 150:22–36

Pascarella L, Geiger F-X, Palomba F, Di Nucci D, Malavolta I, Bacchelli A (2018) Self-reported activities of android developers, in Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, pp. 144–155

Perry DE, Porter AA, Votta LG (2000) Empirical studies of software engineering: a roadmap, in Proceedings of the conference on The future of Software engineering, pp. 345–355

Phong MV, Nguyen TT, Pham HV, Nguyen TT (2015) Mining user opinions in mobile app reviews: A keyword-based approach (t), in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 749–759

Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106

Rahman F, Posnett D, Hindle A, Barr E, Devanbu P (2011) BugCache for inspections: hit or miss?, in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 322–331

R-Documentation. "Pseudo R^2 Statistics." https://search.r-project.org/CRAN/refmans/DescTools/html/PseudoR2.html Accessed Nov 2022

Rish I (2001) An empirical study of the naive Bayes classifier. IJCAI workshop on empirical methods in artificial intelligence, Seattle, pp 41–46

Rodriguez D, Herraiz I, Harrison R, Dolado J, Riquelme JC (2014) Preliminary comparison of techniques for dealing with imbalance in software defect prediction, in Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, pp. 1–10

Rodríguez-Pérez G, Robles G, González-Barahona JM (2018) Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Inf Softw Technol 99:164–176

Rokach L (2010) Ensemble-based classifiers. Artif Intell Rev 33(1):1–39

Rosen C, Grawi B, Shihab E (2015) Commit guru: analytics and risk prediction of software commits, in Proceedings of the 2015 10th joint meeting on foundations of software engineering, pp. 966–969

S. Inc. "Mobile operating systems' market share worldwide from January 2012 to November 2022." https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ Accessed Nov 2022

Sadeghi A, Bagheri H, Garcia J, Malek S (2016) A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Trans Software Eng 43(6):492–530

Santos G, Figueiredo E, Veloso A, Viggiato M, Ziviani N (2020) Predicting software defects with explainable machine learning, in Proceedings of the XIX Brazilian Symposium on Software Quality, pp. 1–10

Scalabrino S, Bavota G, Linares-Vásquez M, Lanza M, Oliveto R (2019) Data-driven solutions to detect api compatibility issues in android: an empirical study, in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp. 288–298

scikit-learn. "sklearn.model_selection.StratifiedKFold." https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html#sklearn.model_selection.StratifiedKFold Accessed Nov 2022

Shihab E (2012) An exploration of challenges limiting pragmatic software defect prediction. Queen's University (Canada)

Shin J, Aleithan R, Nam J, Wang J, Wang S (2021) Explainable Software Defect Prediction: Are We There Yet?. arXiv preprint arXiv:2111.10901

Shu R, Xia T, Williams L, Menzies T (2019) Better security bug report classification via hyperparameter optimization, arXiv preprint arXiv:1905.06872

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM Sigsoft Software Eng Notes 30(4):1–5

Sommerville I (2011) Software engineering 9th Edition, ISBN-10, vol. 137035152, p. 18

Song Q, Guo Y, Shepperd M (2018) A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans Software Eng 45(12):1253–1269
Su T et al. (2020) Why my app crashes understanding and benchmarking framework-specific exceptions of android apps, IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2020.3013438
Tan SH, Dong Z, Gao X, Roychoudhury A (2018) Repairing crashes in android apps. in IEEE/ACM 40th International Conference on Software Engineering, pp. 187–198
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Software Eng 43(1):1–18
Thitichaimongkhol K, Senivongse T (2016) Enhancing usability heuristics for android applications on mobile devices. Proc World Congress Eng Comput Sci 1:19–21
Trautsch A, Herbold S, Grabowski J (2020) Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction, in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 127–138
Vang J (2019) Data science topics. one-off coder. https://datascience.oneoffcoder.com/psuedo-r-squared-logistic-regression.html
Wan Z, Xia X, Lo D, Murphy GC (2019) How does machine learning change software development practices? IEEE Trans Softw Eng 47(9):1857–1871
Wang Q (2014) A hybrid sampling SVM approach to imbalanced data classification, in Abstract and Applied Analysis, vol. 2014: Hindawi
Wen M, Wu R, Cheung S-C (2016) Locus: Locating bugs from software changes, in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 262–273
Williams C, Spacco J (2008) Szz revisited: verifying when changes induce fixes, in Proceedings of the 2008 workshop on Defects in large software systems, pp. 32–36
Wimalasooriya C, Licorish SA, da Costa DA, MacDonell SG (2022) A systematic mapping study addressing the reliability of mobile applications: The need to move beyond testing reliability. J Syst Softw 186:111166
Wright HK, Kim M, Perry DE (2010) Validity concerns in software engineering research, in Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 411–414
Wu R, Wen M, Cheung S-C, Zhang H (2018) Changelocator: locate crash-inducing changes based on crash reports. Empir Softw Eng 23(5):2866–2900
Wu J, Chen X-Y, Zhang H, Xiong L-D, Lei H, Deng S-H (2019) Hyperparameter optimization for machine learning models based on Bayesian optimization. J Electron Sci Technol 17(1):26–40
Wu R, Zhang H, Kim S, Cheung S-C (2011) Relink: recovering links between bugs and changes, in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 15–25
Xia X, Shihab E, Kamei Y, Lo D, Wang X (2016) Predicting crashing releases of mobile applications, in 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10. https://doi.org/10.1145/2961111.2962606
Xia H et al. (2020) How android developers handle evolution-induced api compatibility issues: A large-scale study, in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, pp. 886–898
Xu Z et al (2021) Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. IEEE Trans Reliab 71(1):204–220
Xu Z, Liu J, Yang Z, An G, Jia X (2016) The Impact of Feature Selection on Defect Prediction Performance: An Empirical Comparison, in 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 309–320. https://doi.org/10.1109/ISSRE.2016.13
Yang H, Wang C, Shi Q, Feng Y, Chen Z (2014) Bug Inducing Analysis to Prevent Fault Prone Bug Fixes, in SEKE, pp. 620–625
Yang X, Lo D, Xia X, Zhang Y, Sun J (2015) Deep learning for just-in-time defect prediction, in 2015 IEEE International Conference on Software Quality, Reliability and Security, IEEE, pp. 17–26
Yu T, Wen W, Han X, Hayes JH (2018) Conpredictor: Concurrency defect prediction in real-world applications. IEEE Trans Software Eng 45(6):558–575
El Zarif O, Da Costa DA, Hassan S, Zou Y (2020) On the Relationship between User Churn and Software Issues, in 17th International Conference on Mining Software Repositoriespp. 339–349. https://doi.org/10.1145/3379597.3387456
Zein S, Salleh N, Grundy J (2016) A systematic mapping study of mobile application testing techniques. J Syst Softw 117:334–356
Zhao G, da Costa DA, Zou Y (2019) Improving the pull requests review process using learning-to-rank algorithms. Empir Softw Eng 24:2140–2170

Zhao K, Xu Z, Zhang T, Tang Y, Yan M (2021a) Simplified deep forest model based just-in-time defect prediction for android mobile apps. IEEE Trans Reliab 70(2):848–859

Zhao K (2022) Pre-Process Data with Pipeline to Prevent Data Leakage during Cross-Validation. Medium https://towardsdatascience.com/pre-process-data-with-pipeline-to-prevent-data-leakage-during-cross-validation-e3442cca7fdc Accessed 29/07/2022

Zhao K, Xu Z, Yan M, Tang Y, Fan M, Catolino G (2021b) Just-in-time defect prediction for Android apps via imbalanced deep learning model, in Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 1447–1454

Zheng W, Shen T, Chen X, Deng P (2022) Interpretability application of the Just-in-Time software defect prediction model. J Syst Softw 188:111245

Zhihao P, Fenglong Y, Xucheng L (2019) Comparison of the different sampling techniques for imbalanced classification problems in machine learning," in 2019 11th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), IEEE, pp. 431–434

Zhu A (2021) Select Features for Machine Learning Model with Mutual Information, ed: Medium