Noname manuscript No.
(will be inserted by the editor)

Tracking Bad Updates in Mobile Apps: A Search-based Approach

Islem Saidani · Ali Ouni · Md Ahasanuzzaman · Safwat Hassan · Mohamed Wiem Mkaouer · Ahmed E. Hassan

Received: date / Accepted: date

Abstract The rapid growth of the mobile applications development industry raises several new challenges to developers as they need to respond quickly to the users' needs in a world of continuous changes. Indeed, mobile apps undergo frequent updates to introduce new features, fix reported issues or adapt to new technological or environment changes. Hence, introducing changes in this context is risky and can harmfully impact the application rating and competitiveness. Thus, ensuring that the applications updates are deployed in a controlled way is of crucial importance. To better support mobile applications evolution and cut-off the costs of users dissatisfaction, we propose in this paper, AppTracker, a novel approach to automatically track bad release updates in Android applications (*i.e.*, releases with higher percentage of negative reviews relative to the prior releases). We formulate the problem as a three-class classification problem to label the apps updates as *bad*, *neutral* or *good*. To solve this problem, we evolve bad release detection rules using Multi-Objective Genetic Programming (MOGP) based on the adaptation of the Non-dominated Sorting Genetic Algorithm (NSGA-II). In particular, the search process aims to provide the optimal trade-off between two conflicting objectives to deal with the considered classes. We evaluate our approach and investigate the performance of both within-project and cross-project validation scenarios on a benchmark of 50,700 updates from 1,717 free Android apps from Google Play Store. The statistical tests revealed that our approach achieves a clear advantage over machine learning approaches (*e.g.*, random forest, decision tree, etc.) with

I. Saidani, A. Ouni
ETS Montreal, University of Quebec, Montreal, QC, Canada
E-mail: islem.saidani.1@ens.etsmtl.ca, ali.ouni@etsmtl.ca

M. Ahasanuzzaman, A. E. Hassan
Queen's University, Kingston, ON, Canada
E-mail: md.ahasanuzzaman@queensu.ca, ahmed@cs.queensu.ca

S. Hassan
Thompson Rivers University, Kamloops, BC, Canada
E-mail: shassan@tru.ca

M. W. Mkaouer
Rochester Institute of Technology, NY, USA
E-mail: mwmvse@rit.edu

significant improvements of 18% and 6% in terms of F1-score within-project and cross-project validations, respectively. Furthermore, the features analysis reveals that (1) the previous updates ratings and (2) the APK size are the most important features for both within and cross-project scenarios.

Keywords Mobile Apps · Software Releases · Bad Updates · User Rating · Search-Based Software Engineering · Android · Google Play Store

1 Introduction

Over the last years, software development and releasing activities have shifted from a traditional process, in which software projects are released following a clearly defined road-map, towards a modern process in which continuous releases become available on a weekly/daily basis. Nowadays, such an agile strategy is massively adopted by mobile applications (apps). Indeed, with more than two billion users relying on smart-phones and tablets [1], mobile apps development undergoes continuous changes to add new features, fix reported issues, or adapt to new technological and environmental changes. Hence, many mobile applications often release daily updates of their applications to quickly deliver up-to-date applications to end users [2].

In this context, software change management represents a fast-paced task of extreme complexity [3] while mobile release engineering in a non-trivial and risky task that requires comprehensive information and knowledge [4]. In fact, the tension between release speed and quality is a major concern for mobile apps developers as bad changes adversely affect users experience and may drive them away over time in a very competitive mobile apps market [5, 6]. Indeed, one of the unique and important features that mobile app platforms, such as Google Play Store, provide is the users reviews and rating. User reviews represent a powerful asset to reflect users' (dis)satisfaction and can provide a complementary view on the app's success and quality as large amount of reviews contain bug reports [7, 8, 9]. For instance, unexpected or poor app changes may cause even loyal users to explore alternative apps as pointed out by Martens and Maalej [10]. Recently, Hassan et al. [11] showed that various app changes such as feature removal and user interface (UI) issues have the influence to increase the number of negative user reviews, while bad updates having crashes and functional issues tend to be fixed in subsequent updates. Therefore, the analysis of user reviews about a specific update is of pivotal importance as pointed out by Hassan et al. [11]. Hence, providing developers with relevant tools to track and prevent bad updates before pushing them into the marketplace is crucial to maintain and improve the rating of their apps.

To address this issue, we introduce a novel approach, namely AppTracker, to automate the tracking of mobile app *bad updates* (*i.e.*, updates with higher percentage of negative user reviews relative to the prior updates of the app [11]). The problem is formulated as a three-class classification problem to classify releases into "good", "bad" or "neutral". In particular, we adopt the One-Versus-All (OVA) method [12] which consists of decomposing our multi-class classification problem into multiple binary problems. Then, we evolve various binary classifiers to generate classification rules using Multi-Objective Genetic Programming (MOGP) as a base learner. In the context of OVA method, for each class i, we train a base

MOGP learner using all instances of this class as positive data points, while the remaining classes are considered as negative data points. Our MOGP formulation is based on an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) [13]. MOGP techniques have been widely adopted in search-based software engineering (SBSE) [14, 15] to solve various classification-related software engineering problems [16, 17, 18, 19, 20, 21], due to their efficiency in exploring large search spaces and searching optimal solutions. More specifically, AppTracker approach aims at learning patterns from examples of bad app releases that have been experienced by end users. These patterns are expressed in the form of tree-based solution representation that is expressed as logical combinations of metrics and their corresponding threshold values. These solutions are refined through a multi-objective evolutionary search process to converge towards the optimal detection rules that should cover as much as possible the accurately detected (1) bad, (2) good and (3) neutral releases from the base of real-world app release examples.

To evaluate AppTracker, we performed an empirical study on a large benchmark of 50,700 releases extracted from 1,717 popular apps in the Google Play Store [1]. Based on two different validation scenarios, within-project and cross-project settings, our obtained results confirm that AppTracker statistically advances the baseline techniques. Moreover, we leverage our generated rules by analyzing the obtained Pareto fronts (*i.e.*, the non-dominated solutions) achieved by the MOGP algorithm. In particular, we measure the features' importance using the Permutation Feature Importance (PFI) technique [22, 23], and then rank them using Scott-Knott (SK) algorithm [24, 25] in order to prioritize the refactoring efforts during the app's maintenance. The results of this analysis reveal that the previous updates ratings and the APK size are the most important features for both within and cross-project scenarios.

1.1 Contributions

The paper makes the following main contributions:

1. A novel approach, AppTracker, formulating the detection of mobile apps releases as multi-class classification problem based on MOGP. We adopted MOGP as a base learner to support multi-class classification by decomposing it into multiple binary problems using the one-versus-all method. To the best of our knowledge, this is the first search-based technique for the detection of bad releases in mobile applications.
2. An empirical evaluation on a benchmark of 50,700 releases from 1,717 Android apps, shows that AppTracker outperforms various baseline machine learning techniques by achieving median F1 scores of 46% and 47% in within-project and for cross-project validations, respectively, across the three classes.
3. A qualitative analysis to discover which features are the most prominent using the optimal rules based on the Pareto fronts analysis. The results reveal that the previous updates ratings and the APK size are the most important features for both within and cross-project scenarios.

[1] `https://play.google.com/`

4. A longitudinal labeled data from the Google Play Store from 1,717 free-to-download Android apps having over 50,700 release updates for a period of over thee years [26].

1.2 Replication Package

We provide our replication package containing all the materials to reproduce and extend our study [26].

1.3 Paper Organization

In Section 2, we motivate the problem of tracking bad mobile releases with a real-world example. Then, we explain our approach in Section 3. Section 4 describes the experimental setup of our empirical study while Section 5 presents the results of this study. In Section 6, we elaborate on the implications of our results. Section 7 discusses the threats to validity. In Section 8, we survey the related work. Section 9 finally concludes and discusses future research directions.

2 Motivating Example

To show the importance of early identification of bad release updates in mobile apps, we describe in this section a motivating example from a real-world Android app. Let us consider `Dubsmash`[2], a popular video sharing Android app (in the Video Players category). Dubsmash used to maintain a stable rating history of 4.2/5 and most of its updates were either "neutral" or "good" suggesting that the app have had a negligible negative user rating for its updates. However, looking at its release history, we observe that the negativity ratio (*i.e.* as the ratio of the percentage of negative reviews before update U_i to the percentage of negative reviews of update U_i) highly increased immediately after the release of U43 (16 October 2018) as shown in Figure 1. For instance, users were unhappy and complaining about the recent updates leaving comments such as "*This apps was very fun but got progressively worse but got used to it, now it's practically unusable ...*" (cf. Figure 2). While the app developers started to deploy more frequent updates with shorter delays (with less than two weeks on average) to address the users concerns, users continued expressing their complaints. Within few months, the negative ratings increased from 10% to reach 25% on 14 March 2019.

A closer examination of the `Dubsmash` app change history has shown that during this period, many features were deleted from the app. Thus, the app installation size (*i.e.*, APK file) has decreased by 71% (from 30 MB to 8.7 MB) and consequently the number of activities has dropped from 53 to 31 (-154%) and so is the number of intents that decreased by 78%. In addition, the minimum SDK version (*i.e.*, the required Android version to run the app) has been upgraded from 4.1 to 4.4, which led to losing users who are using older SDK versions in their devices. These changes have led to many user complaints, as shown in Figure 2.

[2] `https://play.google.com/store/apps/details?id=com.mobilemotion.dubsmash`

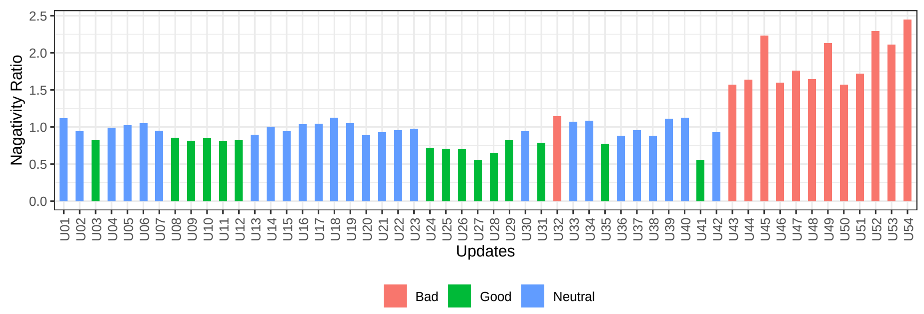
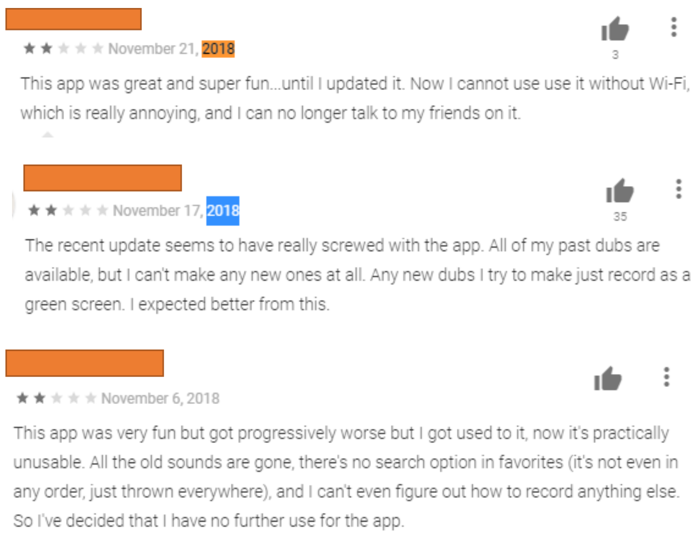


Fig. 1: A snapshot of updates fluctuations in Dubsmash app between 2016-05-17 and 2019-03-14.



[redacted]

★★☆☆☆ November 21, 2018 3

This app was great and super fun...until I updated it. Now I cannot use use it without Wi-Fi, which is really annoying, and I can no longer talk to my friends on it.

[redacted]

★★☆☆☆ November 17, 2018 35

The recent update seems to have really screwed with the app. All of my past dubs are available, but I can't make any new ones at all. Any new dubs I try to make just record as a green screen. I expected better from this.

[redacted]

★★☆☆☆ November 6, 2018

This app was very fun but got progressively worse but I got used to it, now it's practically unusable. All the old sounds are gone, there's no search option in favorites (it's not even in any order, just thrown everywhere), and I can't even figure out how to record anything else. So I've decided that I have no further use for the app.

Fig. 2: Examples of users' reviews on the `Dubsmash` app from Google Play Store.

This example indicates the usefulness and the need for an automated tool to track bad updates in order to avoid negatively affecting users experience and ensuring the success of the app and this by learning from previous types of updates (good, bad, or neutral). However, this task is not trivial in practice. In fact, the main difficulty lies in the complex search space as the number of possible combinations of update features (*e.g.*, changes in the user interface, features removal or addition, SDK update, release size, changes in the app permissions, changes in the libraries, etc.) and their associated values is very large. Hence, tracking bad updates can be formulated as a search-based optimization problem to explore this large search space, in order to find the optimal detection rules for each class. Additionally, a practical tool should provide the developers with human explainable detection rules to help them gain insights into bad changes, especially when these changes are not trivial, as shown in this example. For instance, one possible detection rule for `Dubsmash` app, as illustrated in Figure 3, indicates that to avoid a bad update, the decrease in the APK size (*i.e.*, Chang_perc_APK_size) and in the number of intents (*i.e.* Chang_perc_Nintent) should not exceed 70% and 77%, respectively. Additionally, the number of activities (*i.e.*, Nact) should be more than 52. On the other side, the minimum SDK version (*i.e.*, Min_SDK) should not exceed 4.4. These conditions could be leveraged as refactoring recommendations that guide the developers in the maintenance process in order to maintain the app's rating.

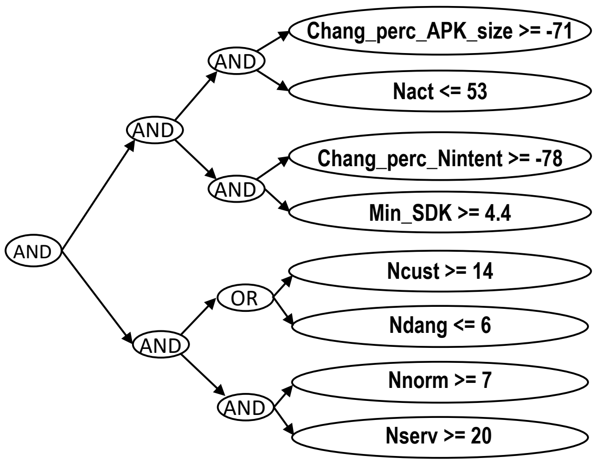


Fig. 3: An illustrative example of bad releases detection rule for the `Dubsmash` app.

In the next section, we describe AppTracker and show how we formulated the bad updates tracking problem as a multi-objective combinatorial optimization problem to address the above mentioned problems.

3 The AppTracker Approach

In this section, we describe our AppTracker approach to automatically track bad mobile apps releases using multi-objective genetic programming (MOGP).

3.1 Approach Overview

Figure 4 illustrates an overview of AppTracker, a two-phase framework (1) training and (2) detection. In the training phase, our main goal is to decompose the multi-class problem into multiple binary problems to build a set of binary detection rules from real-world examples of various Android apps releases. In the detection phase, we use these generated rules to detect the appropriate label (neutral, good or bad) for new unlabeled data (*i.e.*, a new release).

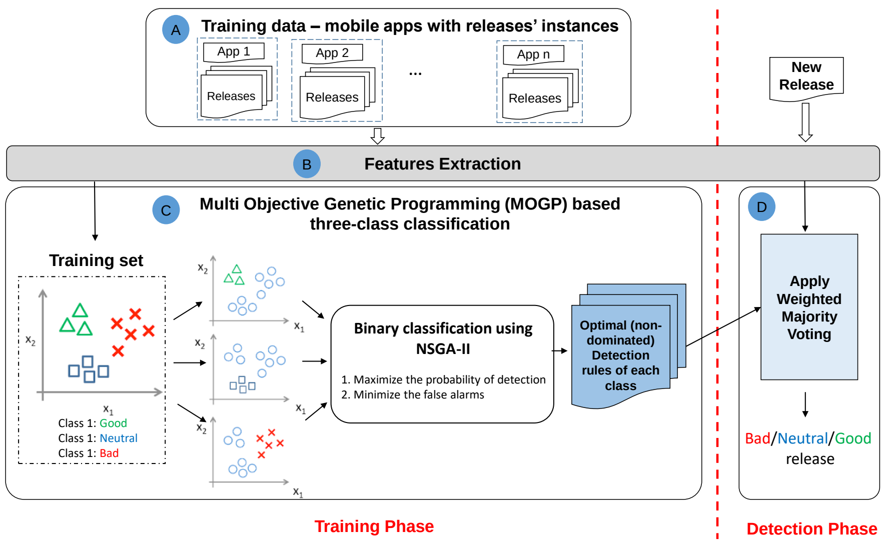

A
Training data – mobile apps with releases' instances
App 1
Releases
App 2
Releases
...
App n
Releases
New Release
B
Features Extraction
C
Multi Objective Genetic Programming (MOGP) based three-class classification
Training set
Class 1: Good
Class 1: Neutral
Class 1: Bad
Binary classification using NSGA-II
1. Maximize the probability of detection
2. Minimize the false alarms
Optimal (non-dominated) Detection rules of each class
D
Apply Weighted Majority Voting
Bad/Neutral/Good release
Training Phase
Detection Phase

Fig. 4: Approach overview.

As shown in Figure 4, our framework takes as input a set of mobile app releases with known labels, *i.e.*, "bad", "good" or "neutral" (Step A). Then, the Step B consists of extracting a set of features characterizing the considered releases in order to feed the search-based algorithm using multi-objective genetic programming (MOGP). As output, a set of non-dominated rules (*i.e.*, a solution that has a score for each objective such that no other solution within the set has a better score across all objectives) will be built (Step C). Thereafter, in the detection phase, the framework assigns the proper class of a new release knowing its characteristics (Step D) using an ensemble majority voting based on each rule's score. In the next subsections, we detail each step.

3.2 Step A: Training data preparation: collecting apps data

Our data collection follows a three-steps process. First, we collected app updates data (*e.g.*, the APK files of the releases) of popular free Android apps in the Google Play Store. Then, we extracted app manifest information. Finally, we collected data about advertisement (Ads) libraries that are used in each app.

3.2.1 Collecting updates of the Google Play Store apps

To collect Google Play Store apps, we proceeded as follows:

A. Selecting Top Free-to-Download Apps. In this study, we focused on free-to-download apps of the Google Play Store [27]. In particular, we selected a set of mobile apps with respect to the following criteria:

– *App popularity:* We considered popular Android apps in Google Play Store as we expect that these apps are developed and maintained by developers who care about their apps rating, and have a large user-base.
– *App diversity:* We considered the top popular Android apps across all categories in the Google Play Store to ensure that there is no bias towards specific app categories in our observations.

Our selection of the top free-to-download apps is based on App Annie's report on popular apps [28] in the Google Play Store since 2016. Then, we selected the top-hundred apps in each app category so that our study does not impact by the variances across the different app categories. Next, we filtered out those apps that was repeated across the categories, and that were already removed from the Google Play Store during our study period. In total, we selected 1,717 apps having over 50,700 releases during our study period. Table 2 provides some statistics about the studied apps.

B. Crawling App Data over three years. We used a Google Play crawler [29] to gather longitudinal data during the period 20 April 2016 to 20 September 2019 from the Google Play Store. Thereafter, for each studied app, we collected the following data:

– *General data:* The app title, description, current number of downloads, and rating.
– *Updates data:* The release notes of each update.
– *User reviews data:* The review title, review contents, rating, and review time.

At the end of this step, we collected a total of 50,700 updates that were released during our study period. Table 2 summarizes the statistics about the collected updates for each category.

3.2.2 Extracting app manifest information

To collect metadata of an app, its components, and its requirements, we need to extract the app manifest file (*i.e.*, `AndroidManifest.xml`) from the APK file of the app. We reverse engineered the APKs of each app update using the Apktool[3] and extracted `AndroidManifest.xml` files from the collected APKs. Then, we parsed the `AndroidManifest.xml` and collected app metadata (*e.g.*, permissions, activities, services, and target SDK versions, etc.).

3.2.3 Collecting data about integrated ad libraries

To collect integrated advertisement (Ad) libraries, we followed Ahasanuzzaman et al.'s technique [30, 31]. In particular, we extract the fully qualified class names of each class, and manually searched them on the web to identify ad library packages. Thereafter, we collected the list of integrated ad libraries in each update of the studied apps.

[3] `https://github.com/iBotPeaches/Apktool`

3.3 Characterizing the studied updates

We follow a similar approach of Hassan et al. [11] to characterize the updates (*e.g.*, good or bad updates) of an app based on the app user ratings. First, we calculate the *Ratio of Negative Ratings* $RNR(U_i)$ of an update U_i of an app as the ratio of one or two star ratings of the update U_i to the total number of ratings of all updates. Then, we calculate the Median Ratio of Negative Ratings ($MRNR(U_i)$) of an update U_i which is the median of the Ratio of Negative Rating of all the previous updates of U_i. Finally, to characterize an update U_i, we measure the *Negativity Ratio (NR)* of U_i based on the $RNR(U_i$ and $MRNR(U_i)$ as follows:

$$\textit{Negativity Ratio}(U_i) = \frac{RNR(U_i)}{MRNR(U_i)} \tag{1}$$

For instance, if an update with 10 user ratings (four ratings with two stars and six ratings with four stars), then the *RNR* score of this update is 0.4 ($RNR = \frac{4}{10} = 0.4$). If the *MRNR* of this update is 0.1, then its negativity ratio (*NR*) is 4 ($NR = \frac{0.4}{0.1} = 4$).

We characterize an update of an app into three classes using the negativity ratio. Table 1 shows the rules for characterization of an update[11]. These classes are the target labels of an update in our dataset.

Table 1: Characterization of the updates of an app.

Update Class	**Rule**
Good Update	Negativity ratio <1.0 - SD*
Bad Update	Negativity ratio >1.0 + SD*
Neutral Update	Otherwise

SD = Standard Deviation of the negativity ratio

Table 2 shows the number of good, bad and neutral updates across the studied categories.

3.4 Step B: Features Extraction

In our approach, we extracted a total of 41 metrics divided into ten dimensions that characterize the update's rating and thus its likelihood of being label as "bad", "good" or "neutral". In Table 3, we list our metrics suite and explain the rational behind each of them. In particular, they identify the following categories:

– **Size of the App:** consists of metrics related to the APK size of an app at the time of release. Larger size apps typically provide more features but at the same time, it requires more space and bandwidth to download the update which could impact the app's rating. We also collect the number of activities and services in each app release. An activity provides a screen for users to interact, whereas a service is used to perform operations in the background.

Table 2: Summary of the collected data.

Row Labels	**# of apps**	**# of good updates**	**# of neutral updates**	**# of bad updates**	**# of updates**
ART_AND_DESIGN	2	14	7	1	22
AUTO_AND_VEHICLES	10	107	70	132	309
BEAUTY	3	32	12	6	50
BOOKS_AND_REFERENCE	71	690	476	698	1,864
BUSINESS	80	789	677	762	2,228
COMICS	43	207	165	174	546
COMMUNICATION	74	1,116	1,111	1,066	3,293
DATING	7	71	124	142	337
EDUCATION	71	786	600	665	2,051
ENTERTAINMENT	50	418	468	427	1,313
EVENTS	2	21	34	40	95
FINANCE	38	337	210	484	1031
FOOD_AND_DRINK	7	155	184	166	505
GAME	85	626	613	405	1644
HEALTH_AND_FITNESS	72	861	504	853	2,218
HOUSE_AND_HOME	10	161	122	236	519
LIBRARIES_AND_DEMO	15	58	34	43	135
LIFESTYLE	43	471	383	380	1,234
MAPS_AND_NAVIGATION	69	474	471	761	1,706
MEDIA_AND_VIDEO	5	4	12	8	24
MEDICAL	50	468	180	307	955
MUSIC_AND_AUDIO	66	683	692	568	1,943
NEWS_AND_MAGAZINES	76	837	483	880	2,200
PARENTING	7	77	28	56	161
PERSONALIZATION	87	766	663	586	2,015
PHOTOGRAPHY	92	1,200	928	743	2,871
PRODUCTIVITY	78	747	956	884	2,587
SHOPPING	53	593	575	884	2,052
SOCIAL	85	1,155	1,222	1,213	3,590
SPORTS	68	551	308	643	1,502
TOOLS	96	1,317	1,424	1,184	3,925
TRAVEL_AND_LOCAL	76	1,086	712	863	2,661
VIDEO_PLAYERS	56	493	602	437	1,532
WEATHER	70	671	365	547	1,583
Total	**1,717**	**18,042**	**15,415**	**17,244**	**50,701**

We also consider the app intents which define the app's "intent" to perform an action.

- **Ad libraries:** This dimension captures any changes in the number of displayed advertisements (ads). It has been shown that frequency and size of displayed Ad increases the number of negative reviews [30, 32].
- **SDK version:** This dimension includes metrics related to the minimum and the target Software Development Kit (SDK). Higher minimum and target SDK versions might suggest that app included many new features but at the same time can lead to losing users who are using older SDK versions in their devices.
- **Permissions:** In this dimension, we collect information about the user permissions. Higher number of permissions increases privacy risks, thus it might impact the release's rating.
- **Marketing effort:** This dimension includes metrics related to the release description (*i.e.*, note) that is displayed to all users to present the new features

or the resolved issues. Many changes in the release notes would signify many feature updates and improvements in the app
- **Link to last releases (s):** This dimension is related to the app's releasing stability overtime. Previous release ratings can help in predicting future release ratings.
- **Release time:** This dimension is dedicated to measuring the release frequency. More frequent updates may still have the bug unsolved or frequent updates may increase more issues in the apps as developers try to give an update in quick succession. However, frequent updates may solve the issue and that may satisfy the users. Hassan et al. [33] analyzed the emergency updates and found that the ratio of negative reviews is small for the emergency.

3.5 Step C: MOGP-based three-class classification

To address the three-class classification problem, we divide it into three binary classification problems using the *One-Versus-All* (OVA) method. For each binary classification instance, one class is labeled as a "positive class" (=1) and all the other classes as "negative classes" (=0), then we train the corresponding classification model. The main merit of this strategy is its interpretability since it allows gaining valuable knowledge about a given class by checking its corresponding model. Additionally, this strategy is commonly used and usually set as a default choice for Machine Learning (ML) models to handle multi-class classification problem [34, 12].

3.5.1 Overview of NSGA-II

In this paper, we use NSGA-II as an intelligent search-based algorithm, that has been widely adopted to solve many software engineering problems [21, 35, 16, 36, 37, 15, 38, 39], to generate binary detection rules of each release class.

NSGA-II starts by randomly creating an initial population of individuals encoded using a specific representation. Then, a child population is generated from the population of parents using genetic operators (crossover and mutation). The whole population (that contains children and parents) is sorted according to their dominance level [13] and only the best N solutions are chosen (N is the population size, which is a parameter to be set). Then, a new population is created using selection, crossover and mutation. This process will be repeated until reaching the last iteration according to a stop criteria.

3.5.2 Adaptation of NSGA-II for binary classification

To adopt a search algorithm to a given problem, a set of elements need to be defined. In fact, it is insufficient to merely apply a search technique *out of the box*, as problem-specific adaptations need to be defined to ensure the best performance such as (*i*) solution representation, (*ii*) solution evolution, and (*iii*) solution evaluation.

Table 3: AppTracker metrics.

Dimension	Feature	Explanation	Rationale
Size of App release	APK_size	The size of the app's APK (in MB) at the time of release.	Larger install size of an app requires more space and bandwidth to download the update which can impact on the app's rating.
	APK_size_changed	Indicates whether APK_size has changed from previous update.	
	chang_perc_APK_size	The percentage of change in APK_size from the previous update. The positive value shows the increase and negative value shows the decrease in the value.	
	Nact	# of activities that is defined for the app screens.	An activity provides the window in which the app draws its User Interface(UI). Many app screens can be difficult to handle and may cause issues in the app.
	Nact_changed	A boolean value shows any change in Nact from the previous update.	
	chang_perc_Nact	The percentage of change in Nact from the previous update.	
	Nserv	# of services in the app.	A Service is an app component that can perform long-running operations in the background. Running many services can negatively impact the app battery's lifespan.
	Nserv_changed	A boolean value shows any change in Nserv from the previous update.	
	chang_perc_Nserv	The percentage of change in Nserv from the previous update.	
	Nintent	# of intents in the app.	The intent is a messaging object used to launch a specific app component or to request an action from another app component .
	Nintent_changed	A boolean value shows any change in Nintent from the previous update.	
	chang_perc_Nintent	The percentage of change in Nintent from the previous update.	
Ads libraries	Nlib	# of integrated libraries for displaying ads.	Apps users may get frustrated by the obtrusiveness of many Ad in the middle of running the app.
	lib_changed	A boolean value shows any change in Nlib from the previous update.	
	chang_perc_lib	The percentage of change in Nlib from the previous update.	
SDK version	Min_SDK	The minimum SDK version in which the app can run in a device.	Higher is the min_SDK_version would include many new features but lose users who are using older SDK versions in their device.
	Min_SDK_changed	A boolean value shows any change in Min_SDK from the previous update.	
	chang_perc_min_SDK	The percentage of change in Min_SDK from the previous update.	
	Targ_SDK	The SDK version of a user's device which can run the app.	The higher the Targ_SDK, the more the app utilizes new features of the updated SDK.
	targ_SDK_changed	A boolean value shows any change Targ_SDK from the previous update.	
	chang_perc_targ_SDK	The percentage of change in Targ_SDK from the previous update.	
Permissions	Nperm	# of permissions required to run the application properly.	Users need to accept those permissions to run the app. If # of permissions becomes very high, app users can not trust the app.
	Nperm_changed	A boolean value shows any change in Nperm from the previous app update.	
	chang_perc_Nperm	The percentage of change in Nperm from the previous update.	
	Dang_perm	# of permissions that are defined as dangerous by Google.	Dangerous permissions require resources that involve the user's private information, or can potentially affect the user's stored data or the operation of other apps.
	Ndang_changed	A boolean value shows any change in Dang_perm from the previous update.	
	chang_perc_Ndang	The percentage of change in Dang_perm from the previous update.	
	Norm_perm	# of permissions that are defined as normal by Google.	Normal permissions typically have negligible risk to the user's privacy or the operation of other apps. For example, permission to set the time zone is a normal permission.
	Nnorm_changed	A boolean value shows any change in # of normal permissions from the previous update.	
	chang_perc_Nnorm	The percentage of change in Norm_perm from the previous app update.	
	Cust_perm	# of permission that are defined by developers for the domain of the app.	Custom app permissions allow an app to share its resources and capabilities with other apps.
	Ncust_changed	A boolean value shows any change in Cust_perm from the previous update.	
	chang_perc_Ncust	The percentage of change in Cust_perm from the previous app update.	
Marketing Effort	Note_length	The total number of words in the release note.	Many changes in the release notes would indicate new features/improvements in the app.
	chang_perc_note	Percentage of the modified words in the release note.	
Link to last release(s)	hist_perc_neg_rating	The median percentage of negative rating of all previous updates.	All previous negative ratings can impact the current rating.
	last_perc_neg_rating	The median percentage of negative rating of the previous update.	
	hist_rating	The median aggregated rating (a value between 0 to 5) of all previous updates.	
	last_rating	The median aggregated rating of the previous update.	
Release Time	delay_last_release	The release time of this update (days).	Faster release time can introduce more bugs.
	release_time	The median release time of all previous deployed updates (days).	

Solution representation: In MOGP, a candidate solution, *i.e.*, a detection rule, is represented as an IF–THEN rules with the following structure [16, 40, 19, 20]:

IF *"Combination of metrics with their thresholds"* **THEN** *"RESULT"*.

The antecedent of the IF statement describes the conditions, *i.e.*, pairs of metrics and their threshold values connected with mathematical operators (*e.g.*, $=, >, \geq, <, \leq$), under which a release is considered as good, bad, or neutral. These pairs are combined using logic operators (OR, AND in our formulation). Figure 5 provides an example of a solution. This rule, represented by a binary tree, detects a bad release if it fulfills the situation where (1) the minimum change in the required version of SDK (*Min-SDK_chang*) equal to 1% or (2) the Ad library size (*Nlib*) is greater or equals to 5 or (3) the number of dangerous permissions (related to security) (*dang_perm*) is greater or equals to 2.

IF *Min-SDK_chang =1* OR *Nlib* ≥ 5 OR *dang_perm* ≥ 2
THEN BAD release.

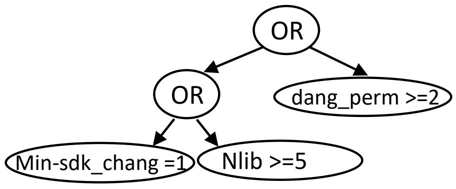


Fig. 5: A simplified example of a solution representation.

To generate the initial population, we start by randomly selecting a set of metrics and their threshold values and then assign them different nodes of a given individual, *i.e.*, trees. To control for complexity, each solution size, *i.e.* the tree's length, should vary between lower and upper-bound limits based on the total number of considered metrics to use within the detection rule. More precisely, for each solution, we assign:

– For each leaf node one metric and its corresponding threshold. The latter is generated randomly between lower and upper bounds according to the values ranging of the related metric.
– Each internal node (function) is randomly selected between AND and OR operators.

Genetic operators: We formulated our genetic operators as follows:

Mutation: In MOGP, the mutation can be applied to (i) a terminal or (ii) a function node. First, the mutation operator randomly selects a node in the tree to be mutated. Then, if the selected node is a terminal, it will be then replaced by another terminal (*i.e.*, other metric or other threshold value, or both), and if it is a function node (*i.e.*, AND, OR operators), it will be replaced by a new random function. Then, the node and its sub-tree will be replaced by the new randomly generated sub-tree. Figure 6 depicts an example of a mutation process, in which we replace the terminal containing *Min-SDK_chang* feature, by another terminal composed of the condition *targ_sdk* $= 5$. Thus, we obtain the new following rule:

IF *Nlib* ≥ 5 OR *targ_sdk* $= 5$ OR *dang_perm* ≥ 2
THEN BAD release.

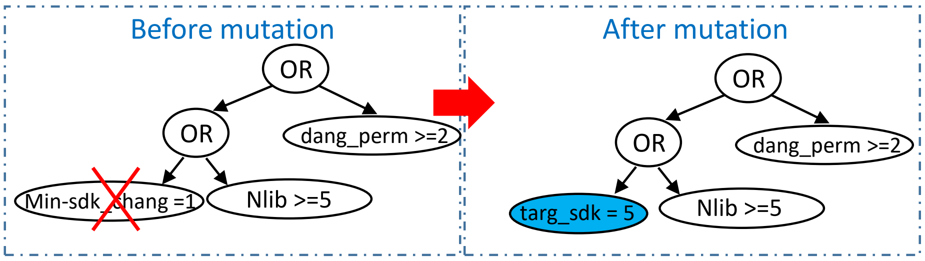


Fig. 6: An example of mutation operator.

Crossover: For MOGP, we use the standard single-point crossover operator where two parents are selected and a sub-tree is extracted from each one. Figure 7 depicts an example of the crossover process. In fact, rules P1 and P2 are combined to generate two new rules. For instance, the new rule C2 will be:

IF *min_sdk* ≥ 5 AND *Nlib* ≥ 3 **THEN** BAD release.

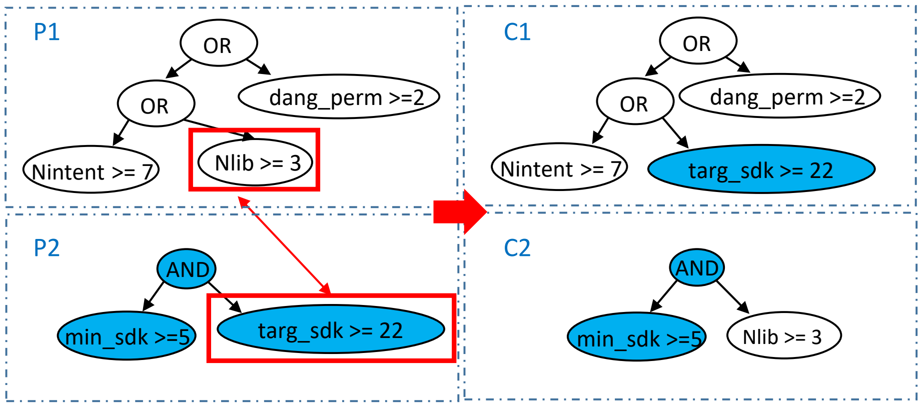


Fig. 7: An example of crossover operator.

Solution Evaluation: Appropriate fitness function, also called objective function, should be defined to evaluate how good is a candidate solution. For the binary classification problem, we seek to optimize the two following objective functions:

1. Maximize the coverage of expected positive class instances over the actual list of positive class instances known as the True Positive Rate (TPR), or also the probability of detection (PD).

$$TPR(S) = \frac{\{Detected\ Positive\ class\ instances\} \cap \{Expected\ Positive\ class\ instances\}}{\{Detected\ Positive\ class\ instances\}}$$

2. Minimize the coverage of actual non-skipped commits that are incorrectly classified as skipped also known as False Positive Rate (FPR), or the probability of false alarm (FP).

$$FPR(S) = \frac{\{Detected\ Positive\ class\ instances\} \cap \{Expected\ Negative\ class\ instances\}}{\{Detected\ Positive\ class\ instances\}}$$

Additionally, since NSGA-II returns a set of optimal (*i.e.* non-dominated) solutions in the *Pareto front* without ranking, we extract a single best solution which is the nearest to the ideal solution known as *True Pareto* in which TPR value equals to 1 and FPR equals to 0. Formally, the distance is computed in terms of Euclidean Distance [37, 40, 16] as follows:

$$BestSol = \min_{i=1}^{n} \sqrt{(1 - TPR[i])^2 + FPR[i]^2} \qquad (2)$$

where n represents the cardinality the Pareto front generated by NSGA-II.

3.6 Step D: Detection Phase

After the optimal binary rules are built in the training phase, they will be then used to detect the corresponding label for a new app release. This step takes as input the set of features extracted from a given release using the feature extraction module. As output, it returns the label, *i.e.*, good, bad, or neutral based on the majority voting principle.

3.6.1 Majority voting

Each detection rule returns (*i*) either +1 (to indicate that the input belongs to its class) or -1 (to indicate that the input does not belong to its class), and (*ii*) a confidence level measured by its fitness function value (the average between both objective function scores). Thus, we obtain for each class a two-dimension vector containing the weighted sum (*i.e.*, multiplied by the confidence level measures) of positive and negative votes as its entries. However, two situations should be taken into consideration. First, in the case of conflict, *i.e.*, two or more rules return +1, the final label is assigned to the class having the highest confidence. Second, when no rule recognizes the input as its class (all the rules return -1), we assign the label to the class associated with the most negative confidence level.

4 Empirical Study Design

In this section, we describe the design of our empirical study to evaluate our AppTracker approach. Figure 8 provides an overview of our experimental design. First, we evaluate the predictive performance of our AppTracker approach based on NSGA-II against mono-objective search and state-of-the art machine learning algorithms to address the two first research questions. We run non-deterministic algorithms used in this empirical study 31 times to deal with their stochastic nature as suggested by Arcuri and Briand [41]. Afterward, we conduct an experiment to qualitatively investigate the most important metrics for our approach. In the following, we describe each step in detail.

To facilitate the replication and extension of our study, we provide the experimental material in our online replication package [26].

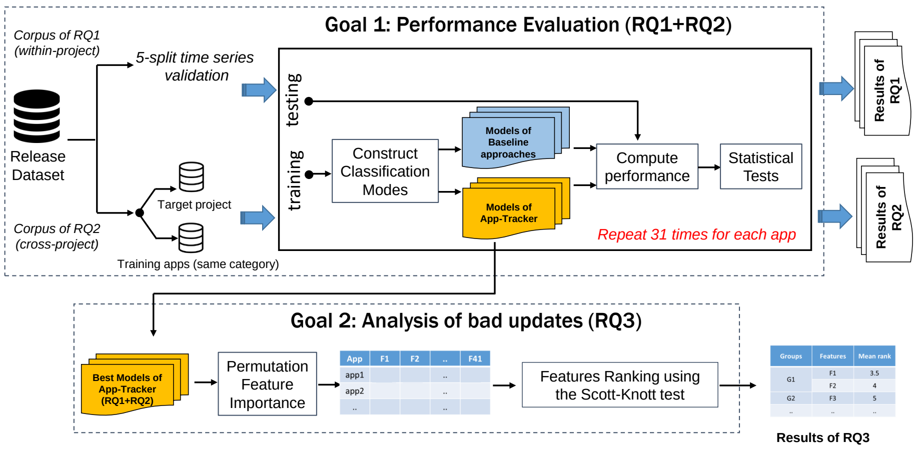

Goal 1: Performance Evaluation (RQ1+RQ2)
Corpus of RQ1 (within-project)
5-split time series validation
Release Dataset
Target project
Corpus of RQ2 (cross-project)
Training apps (same category)
testing
training
Construct Classification Modes
Models of Baseline approaches
Models of App-Tracker
Compute performance
Statistical Tests
Repeat 31 times for each app
Results of RQ1
Results of RQ2
Goal 2: Analysis of bad updates (RQ3)
Best Models of App-Tracker (RQ1+RQ2)
Permutation Feature Importance
App
F1
F2
..
F41
app1
app2
Features Ranking using the Scott-Knott test
Groups
Features
Mean rank
G1
G2
F1
F2
F3
3.5
4
5
Results of RQ3

Fig. 8: An overview of our experimental design.

4.1 Research Questions

We designed our experiments to answer three research questions (RQs):

- **RQ1 (Within-project evaluation).** *How does our* AppTracker *approach perform compared to baseline techniques in within-project scenario?*
- **RQ2 (Cross-project evaluation).** *How effective is our* AppTracker *approach when applied in cross-project scenario?*
- **RQ3 (Features importance analysis).** *What are the most important features for our tool?*

4.2 Predictive performance (RQ1-2)

The first *objective* of our experimental study is to assess the efficiency of our AppTracker approach in solving the three-class classification of apps releases problem considering two different scenarios: within-project (RQ1) and cross-project validation (RQ2).

4.2.1 Evaluation Scenarios and Apps Filtering

In RQ1, we conduct a *time-aware* validation in which the chronological order is considered, similar to previous studies [42, 43, 44, 45, 46]. Specifically, we consider time series validation [4] which is a variation of k-fold where train/test sets are observed at fixed time intervals. In the k^{th} split, the time series validation returns first k folds as the train set and the $(k+1)^{th}$ fold as the test set. In this study, k is set to 5, the default value. Since this scenario is only useful for apps with sufficient historical data, we consider only apps that had at least 100 release versions. Additionally, we only select apps with at least one representation of each class in both training and testing sets. This filtering left 19 apps with 2,518 versions. An overview of the studied apps is reported in Table 4.

Table 4: Statistics about RQ1 corpus.

# of apps	19
# of updates	2,518
bad updates ratio	0.31
neutral updates ratio	0.36
good updates ratio	0.33

Then, in RQ2, we investigate the extent to which our approach can be generalized through a cross-project prediction. In fact, mobile apps might not always have sufficient historical labeled data to build a classifier [47] (especially with small or new apps in the market), which may prevent the mobile app team from using within-project prediction tools, such as AppTracker. Hence, cross-project validation is a common state-of-the-art technique to solve the lack of training data in software engineering [48]. To evaluate our approach on the cross-project scenario, we train each app based on the other collected apps from the same category. Then, we test our AppTracker approach on the target app data. Training on apps from the same category is useful for developers as this would help them track the bad updates of their competitors and attempt to avoid them (*e.g.* privacy violations).

Similar to RQ1, we only study apps with at least one instance of each update class in training/testing sets which left 1,313 apps with a total of 48,395 updates as shown in Table 5.

Note that for both RQ1 and RQ2, all the studied approaches are evaluated on unseen data (*i.e.* the testing data is not used at the training phase).

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

Table 5: Statistics about RQ2 corpus.

# of apps	1,313
# of updates	48,395
bad updates ratio	0.35
neutral updates ratio	0.30
good updates ratio	0.35

4.2.2 Baseline approaches

As a basis for comparisons with our MOGP method, we have employed representative families of classification, a GP-based approach and common Machine Learning (ML) families that are widely used in solving several software engineering problems. In each algorithm family, we consider two approaches, *discretized-based* classifiers where the instances are classified into one of the three classes and *regression-based* classifiers that build a regression model first based on the negativity ratio, then perform classification according to Table 1. The considered baselines are presented in Table 6.

Furthermore, as ML models are sensitive to the scale of the inputs, the data are normalized in the range $[0, 1]$ by using feature scaling. In addition, to mitigate the issue related to the imbalanced nature of the dataset, we rely on Synthetic Minority Oversampling Technique (SMOTE) method [57], to resample the training data. Note that with XGB there is no need for resampling as it is internally handled by the algorithm similarly to our approach. Also, it is worth to mention that we only resample the training data in order to assess these algorithms in a real-world situation.

ML and XGB models are implemented using Scikit-learn [58] and XGB [59] Python libraries, respectively. As for the search-based algorithms, we used MOEA Framework[5], an open-source framework for developing and experimenting with search-based algorithms [60].

4.2.3 Evaluation Metrics.

To compare the predictive performance of AppTracker with other techniques, we employ for binary classification, **F1-score**, the commonly used metric in predictive models comparison [61] which is defined as the harmonic mean of the precision and recall of prediction. The **Precision** measures to the ability of a classifier not to label as positive a sample that is negative, while **Recall** measure the ability of a classifier to find all the positive samples. We also use Area Under the ROC Curve (**AUC**) which indicates how much a prediction model/rule is capable of distinguishing between postive and negative classes. In our study, we consider the following binary measures:

– True Positive (TP): the number of positive class instances that are correctly classified;
– True Negative (TN): the number of negative instances that are correctly classified as CI negative;

[5] `http://moeaframework.org/`

Table 6: Selected baselines from each family.

Family	Classifier	Regression-based classifier
Decision-based	**- Mono-objective Genetic Programming (mono-GP):** This technique has been widely used in various multi-class classification problems [49, 50, 51, 52, 53, 54]. Hence, it is important to assess our multi-objective formulation against mono-objective GP (called GP-multi). If considering separate conflicting objectives fails to outperform aggregating them into a single objective function, then the proposed formulation is inadequate. In this evaluation, we adopt GP to our problem similarly by relying on one-vs-all method.	None
	- Decision tree: Similarly to GP-based techniques, this model generates decision trees consisting of nodes and branches. In this tree we can go from conditions about an instance (represented in the branches) to conclusions about the instance's target value (leaf nodes). The algorithm can naturally handle multi-class classification problems.	**- Regression Tree (RT):** In regression, the predicted outcome is the negativity ratio then converted to their corresponding class value.
Statistical	**- Logistic Regression (Log-Reg):** uses a sigmoid function as a learning model, then it optimizes a cost function that measures the likelihood of the data given the classifier's class probability estimates; then for the multi-class problem, one-vs-all solution can be is used.	**- Linear Regression (LinReg):** a linear approach for modelling the relationship between a scalar response and one or more explanatory variable.
	- Bayes Network:The model is constructed based on Bayes' Theorem and can be naturally extended to the multi-class classification.	None
Support-Vector Machines (SVM)	**- Support Vector Classification (SVC):** A classifier that partitions the data in such a way that it maximizes the margin of separation between the decision boundary and the class instances. SVC can be generalized to multi-class case using one-vs-all classification strategy.	**- SVM regression (SVMR):** uses the same principle as the SVC but it predicts discrete values (*i.e.* negativity ration) then convert them to their corresponding class values.
Ensemble Learning (EL)	**- Random Forest (RF):** A bagging-based EL approach used to train other DT by dividing the data into N subsets of the same size, and then each subset is used to create a DT classifier (called estimator) and uses averaging to improve the predictive accuracy. Finally, the whole classification model is built by aggregating these estimators.	**- RF regressor (RFR):** For regression tasks, the mean or average prediction of the individual trees is returned. Then, the discrete values (*i.e.* negativity ratio) are converted to their correspon--ding classes (categorical numbers).
	- eXtreme Gradient Boosting (XGB): a boosting-based EL approach that improves the performance of separate DT classifiers by combining them into a composite whole. The classifiers are learned sequentially, aiming to reduce the errors of the previously modeled ones. This algorithm supports both binary and multi-class classifications and has specifically shown success in treating multi-class imbalanced problem [55, 56].	**- XGBRegressor (XGBR):** uses the same principle as the XGB but it predicts discrete values (*i.e.* negativity ration) then convert them to their corresponding class values.
Nearest Neighbor	**- K-Nearest Neighbor (KNN):** A simple well-known ML classifier that is based on the distances between the patterns in the feature space. Specifically, a pattern is classified according to the majority class of its K-nearest neighbors. The algorithm can be naturally extended to multi-class classification.	**- KNN regressor (KNNR):** uses the same principle as the KNN but it predicts discrete values (*i.e.* negativity ration) then convert them to their corresponding class values.

– False Positive (FP): the number of negative instances classified as positive;
– False Negative (FN): the number of positive instances that identified as negative.
– n, m and p represents the number of instances of bad, good and neutral release classes, respectively.

For multi-class classification, we consider **Matthews Correlation Coefficient (MCC)** [62] computed as a correlation coefficient between the observed and predicted classifications. Additionally, we calculate the *Standard* (also called macro) averages of the binary metrics as done by previous studies [63, 64, 65] and the *Weighted* (*i.e.*, weighted by the number of instances per class) averages in order to account for class imbalance [66, 67, 68]. All the used measures are defined in Table 7.

4.2.4 Dealing with stochastic approaches.

Due to the stochastic nature of genetic algorithms, decision tree (DT) and random forest (RF) algorithms, we compare their performance by performing 31 independent runs for each experimentation then we choose the rule/model with the median value as suggested in Arcuri and Briand [41] work.

4.2.5 Statistical Tests methods Used.

Before selecting the statistical tests, we should first assess the data normality. To this end, we employ Shapiro-Wilk's W test [69] to assess whether the data distribution is normal (*i.e.*, $\rho{-}value \geq 0.05$). Using this test, We found that $\rho{-}value < 0.05$ for all the used metrics suggesting that a non-parametric test should be used.

In order to provide support for the conclusions derived from the obtained results, we use Wilcoxon signed rank test [70] with a 95% confidence level while using Bonferroni correction [71]. Vargha-Delaney A (VDA) [72] is also used to measure the effect size. This non-parametric method is widely recommended in SBSE context [73] and indicates the probability that one technique will outperform another technique in a given performance measure. When comparing the performance of two techniques, a Vargha-Delaney A measure equals to 0.5 indicates that the two techniques are of comparable performance (*i.e.*, do not differ), while a measure above or below 0.5 indicates that one of the techniques outperforms the other [74]. The Vargha-Delaney statistic also classifies the magnitude of the obtained effect size value into four different levels: (i) *negligible* (ii), *small*, (iii) *medium*, and (iv) *large* [75].

4.2.6 Parameters' Tuning and Setting

One of the most important aspects of research on prediction approaches is *parameters' tuning* which has a critical impact on the algorithm's performance [76]. This is also compulsory when using ML techniques [77]. There is no optimal parameters setting to solve all problems, therefore, we used a trial-and-error method to select the hyper-parameters [21] to handle parameters' tuning for search-based algorithms which is a common practice in SBSE [21]. These parameters are fixed as follows: population size = 100; maximum # of generations = 500; crossover probability =0.7; and mutation probability = 0.1.

As for ML techniques, we employed *Grid Search* (GS)[78], an exhaustive search-based tuning method widely used in practice. In order to facilitate the replication of our results, we provide the selected main parameters and their respective search spaces for ML techniques as shown in Table 8. Please note that parameters' tuning is only applied to the training set and hence we cannot guarantee an optimal result on the testing set; as the parameters' tuning may lead to over-fitting [25].

Table 7: Performance measures.

Classification	Measure	Formula
Binary	Recall	$\frac{TP}{TP+FN}$
	Precision	$\frac{TP}{TP+FP}$
	F1-score	$2 * \frac{Precision*Recall}{Precision+Recall}$
	AUC	$\frac{1+\frac{TP}{TP+FN}-\frac{FP}{FP+TN}}{2}$
Three-class	Standard-F1	$\frac{F1_{bad}+F1_{good}+F1_{neutral}}{3}$
	Standard-AUC	$\frac{AUC_{bad}+AUC_{good}+AUC_{neutral}}{3}$
	Standard-Precision	$\frac{Precision_{bad}+Precision_{good}+Precision_{neutral}}{3}$
	Standard-Recall	$\frac{Recall_{bad}+Recall_{good}+Recall_{neutral}}{3}$
	Weighted-F1	$\frac{F1_{bad}*n+F1_{good}*m+F1_{neutral}*p}{n+m+p}$
	Weighted-AUC	$\frac{AUC_{bad}*n+AUC_{good}*m+AUC_{neutral}*p}{n+m+p}$
	Weighted-Precision	$\frac{Precision_{bad}*n+Precision_{good}*m+Precision_{neutral}*p}{n+m+p}$
	Weighted-Recall	$\frac{Recall_{bad}*n+Recall_{good}*m+Recall_{neutral}*p}{n+m+p}$
	MCC	$\frac{[(tp_{bad}+tp_{neutral}+tp_{good})*(n+m+p)]-[(fp_{bad}+tp_{bad})*n+(fp_{good}+tp_{good})*m+(fp_{neutral}+tp_{neutral})*p]}{\sqrt{[(n+m+p)^2-[(fp_{bad}+tp_{bad})^2+(fp_{good}+tp_{good})^2+(fp_{neutral}+tp_{neutral})^2]]*[(n+m+p)^2-[(n^2+m^2+p^2)]]}}$

Table 8: Configuration space for the hyper-parameters of ML models.

Model	Hyper-parameters	Search Space
SVC, SVM-Reg	C	[1,10]
	kernel	range ['linear', 'rbf']
	max of iterations	range [200,50000]
DT, RT	Criterion	['gini', 'entropy']
	max depth	range [10,100], None
	min samples split	range [2,10], None
	min samples leaf	range [1,5], None
	max features	['sqrt', 'log2', None]
RF, RF-Reg	Number of estimators	range [50,600]
	max depth	range [10,100], None
	Criterion	['gini', 'entropy']
	min samples split	range [2,10], None
	min samples leaf	range [1,5], None
	max features	['sqrt', 'log2', None]
XGB, XGB-Reg	max depth	range [10,100], None
	eta	[0.01,0.7]
	min child weight	[1,5,10]
	num of estimators	[50,600]
	learning rate	[0.01,0.3]
NB	alpha	[0,1]
	binarize	[0.0,0.5,1.0,None]
	fit prior	True or False
LR	max of iterations	range [200,50000]
	penalty	['l1','l2','none']
	solver	['newtoncg', 'lbfgs', 'sag','saga','liblinear']
Lin-Reg	fit intercept	True or False
	normalize	True or False
	copy_X	True or False
	positive	True or False
KNN, KNN-Reg	number of neighbors	range [2,5]
	algorithm	['auto', 'ball_tree', 'kd_tree', 'brute']
	leaf size	range [1,50]
	weights	['uniform','distance']

4.3 Features' importance analysis (RQ3)

The second goal of our empirical study is to analyze the most important features. This analysis provides actionable insights for (1) practitioners who might want to identify the factors that can help them maintaining/improving the rating of their apps, and (2) researchers who are interested in understanding which/how features can be influential in mobile app releasing activities.

To address RQ3, we use *Permutation Feature Importance* (PFI) technique, introduced by Breiman [22] and Fisher et al. [23], to discover which features are the most useful for prediction. The importance of a certain feature is computed as the degree of change in the prediction performance in terms of **Gini** measure (defined as *2 * AUC - 1*). Since the dataset may contain multicollinear features, the permutation importance can perform poorly. Hence, to handle multicollinearity issues, we perform hierarchical clustering on the Spearman rank-order correlations [79], and keep only one single feature from each cluster. Once the (PFI) is computed, we rank the features using Scott-Knott algorithm [24, 25] into statistically homogeneous groups so that the obtained rankings within the same group are not significantly different (*i.e.*, $\rho-value \geq 0.05$). Scott-Knott algorithm has been widely applied to different software engineering domains such as identifying the most influential variables [80, 81, 82, 83]. It should be noted that we use the non-parametric version of of the Scott-Knott algorithm that does not require the assumptions of normal distribution.

5 Empirical Study Results

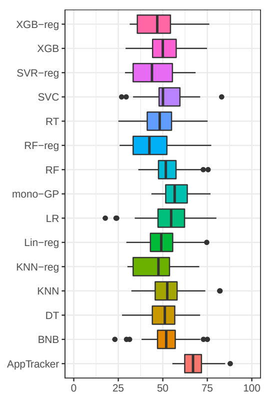
5.1 Results of RQ1 (Within-project validation)

Table 9 reports the median F1, AUC and MCC scores achieved by AppTracker compared to the baseline approaches; while Table 10 shows the statistical tests comparison using the Wilcoxon signed rank test and Vargha-Delaney A estimate and effect size. In addition, we show the different distributions of the studied scores in Figure 9.

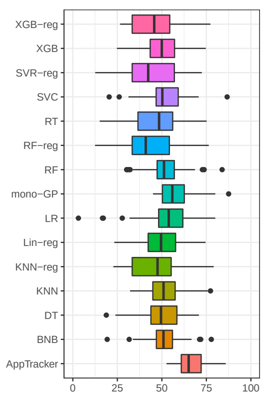
As shown in Table 9, our approach achieved satisfactory results for standard and weighted measures and can reach 81% and 90% in terms of standard and weighted F1 measures, respectively (cf. Figure 10). More specifically we obtained in median 61% for Weighted-F1, 66% for Weighted-Precision, 62% for Weighted-Recall and %67 in terms of Weighted-AUC. With regards to standard scores, we obtained 52% in terms of Standard-F1, 58% for Standard-Precision, 62% for Standard-Recall and 68% for Standard-AUC. The results are well above 1/3 (33.33 %) which is the random chance of guessing that an update belongs to one of three classes labels (*i.e.*, in a three-class classification problem). To get more insights, we investigated the performance of each binary classification. As Figure 10 demonstrates, the binary classification of bad updates performs better compared to others by reaching in median 57% in terms of $F1_{bad}$, 60% for $Precision_{bad}$, 56% for $Recall_{bad}$ and 68% for AUC_{bad} respectively. However, the statistical difference tests reveal that the scores are comparable for the three classes and this is applied to all the studied metrics (*i.e.* F1, Precision, recall and AUC).

In comparison with the mono-objective formulation, we clearly see that our MOGP technique outperforms mono-GP with a substantial improvement for all the studied metrics. For example, we achieved an improvement of 15% and 16% for the Standard and Weighted F1 measures, respectively. Moreover, the statistical test results (Table 10) reveal that over 2,945 runs (5 validation folds x 19 app x 31 repetitions), the difference in scores is significant with **large** VDA effect sizes. These findings confirm that multi-objective formulation is adequate for this problem comparing to aggregating the objectives into a single fitness function. Hence, our problem formulation passes the "sanity check" in this RQ.

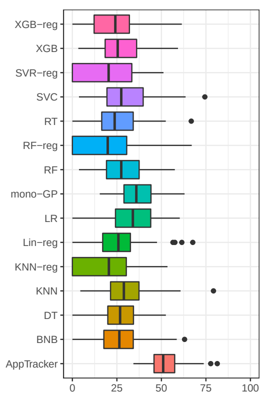
Compared to ML techniques, we find that our AppTracker approach is advantageous over the studied techniques. For instance, AppTracker provides an improvement of at least 24% in terms of MCC over the best ML algorithm (LR). Additionally, the statistical analysis underlines the significant differences with **large**



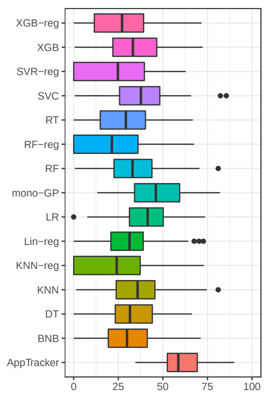
(a) Standard AUC



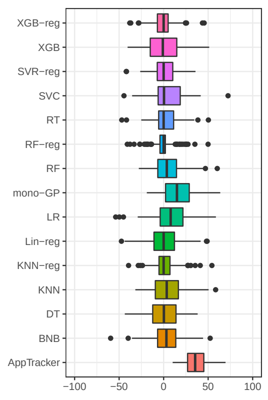
(b) Weighted AUC



(c) Standard F1



(d) Weighted F1



(e) MCC

Fig. 9: Boxplots comparing scores of AppTracker against baseline approaches within-project validation.

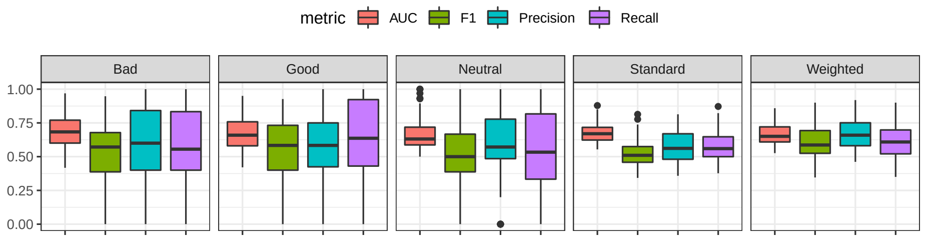


Fig. 10: Results of AppTracker within-project validation.

VDA effect sizes (cf. Table 10). Overall, the results reveal that AppTracker can reach the best balance between the three class accuracies. It is worth noting that all ML techniques are trained using re-sampled training sets unlike in NSGA-II which uses the original data without sampling. These results confirm that the multi-objective formulation is efficient in addressing with the data imbalance problem [84, 16].

Finally, it is worth to note, the regression-based classifiers perform less than other ML techniques (The discretized classifiers used in the study) as well as AppTracker. We also performed statistical test between AppTracker and the other ML approaches. We observe that AppTracker statistically outperforms other ML approaches (with a large effect size in the majority cases). Tables 10 and 12 present the statistical test results for within project and cross-project scenarios, respectively. These results indicate that the discretized classification is more adequate for the three-class classification of mobile releases.

Summary for RQ1. AppTracker *can achieve higher predictive performance than the studied techniques with a statistically significant difference within-validation; with an improvement of 19% in terms of MCC, 10% in terms of AUC (for standard and weighted scores) and 15% in terms of F1 (for standard and weighted scores) approximately.*

Table 9: Performance of AppTracker vs. the state-of-the-art within-project validation (Median scores among the studied apps in percentage).

Algorithm	**MCC**	**AUC**		**F1**	
		Standard	*Weighted*	*Standard*	*Weighted*
AppTracker	**35**	**68**	**67**	**52**	**61**
mono-GP	16	57	57	37	45
LR	11	56	55	34	39
KNN	6	53	53	29	36
SVC	5	53	52	29	33
RF	4	52	52	28	32
BNB	2	51	51	27	30
XGB	2	51	51	29	35
RT	2	48	48	24	27
KNN-reg	1	46	47	19	24
DT	0.4	50	50	24	32
RT	0.2	49	48	25	29
SVR-reg	0	47	47	19	24
Lin-reg	0	49	49	25	30
RF-reg	0	43	44	15	23
XGB-reg	-1	46	45	22	27

5.2 Results of RQ2 (Cross-project validation)

In this RQ, we compare AppTracker with the examined baseline approaches under cross-project validation, using our evaluation metrics, the standard and weighted average scores of F1-score, AUC and MCC, to measure the performance of our approach. Table 11 presents the effectiveness of cross-project modeling compared to the baseline techniques while Table 12 reports the statistical tests results. In addition, we show the different distributions of the studied scores in Figure 12.

Table 10: Statistical tests results of AppTracker compared to ML techniques (within-project).

Metric	**F1**						**AUC**						**MCC**		
	Weighted			**Standard**			**Weighted**			**Standard**					
	p-value	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*
vs. LR	$< 10^{-16}$	0.85	L	$< 10^{-16}$	0.88	L	$< 10^{-16}$	0.83	L	$< 10^{-16}$	0.84	L	$< 10^{-16}$	0.88	L
vs. SVC	$< 10^{-16}$	0.87	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.88	L	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.92	L
vs. KNN	$< 10^{-16}$	0.89	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.89	L	$< 10^{-16}$	0.89	L	$< 10^{-16}$	0.92	L
vs. XGB	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.93	L	$< 10^{-16}$	0.89	L	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.91	L
vs. DT	$< 10^{-16}$	0.93	L	$< 10^{-16}$	0.97	L	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.94	L
vs. RF	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.95	L	$< 10^{-16}$	0.90	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.92	L
vs. BNB	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.94	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.93	L
vs. mono	$< 10^{-16}$	0.72	M	$< 10^{-16}$	0.86	L	$< 10^{-16}$	0.80	L	$< 10^{-16}$	0.81	L	$< 10^{-16}$	0.80	L
vs. SVM-Reg	$< 10^{-16}$	1	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L
vs. XGB	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.89	L	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.93	L	$< 10^{-16}$	0.91	L
vs. XGB-Reg	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L
vs. KNN-Reg	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L
vs. Lin-Reg	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	0.9	L
vs.RF-Reg	$< 10^{-16}$	1	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L
vs. RT	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	0.9	L	$< 10^{-16}$	1	L	$< 10^{-16}$	1	L

L: Large, M: Medium, S: Small, N: Negligible

First, the average values of standard and weighted F1-scores obtained by our AppTracker are acceptable by achieving median scores of 47% and 56% respectively and can reach 90% (cf. Figure 11). Regarding the binary classifications, Figure 11 shows that the scores obtained for the "good" class are generally better which is in line with the statistical tests results. Thus, we believe that further research is needed to improve the prediction of "neutral" and "bad" updates classes.

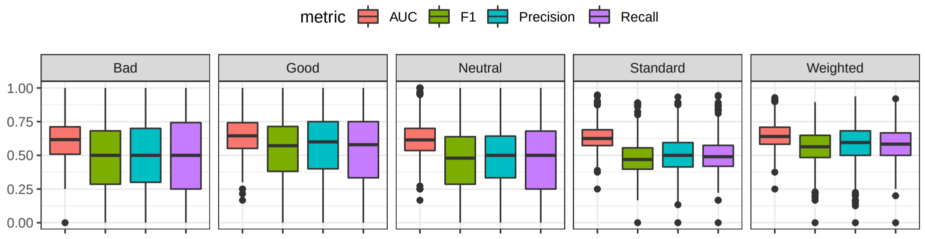
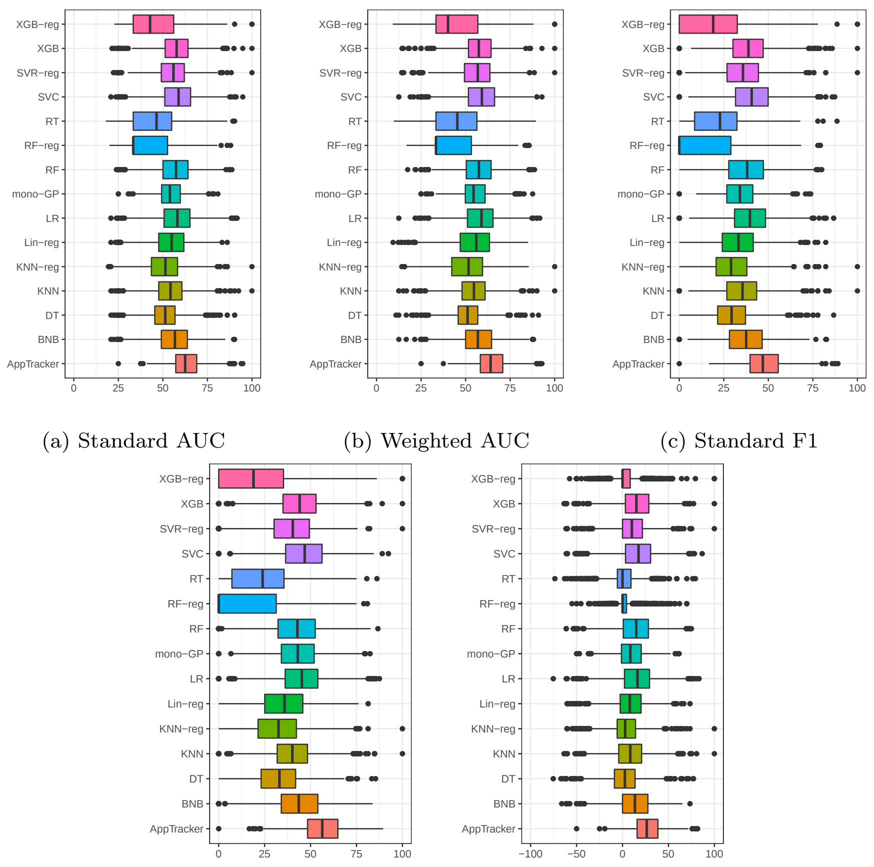


Fig. 11: Results of AppTracker for cross-project validation.

Compared to the baseline approaches, we clearly see that, similar to RQ1, AppTracker remains the best approach. For instance, AppTracker achieves 9% of improvement in terms of MCC over SVC, the best ML technique, and 17% compared to mono-GP. Moreover, the statistical analysis confirms that all results are significantly different with small to large effect sizes as reported in Table 12.

Compared to the within-project validation (RQ1), the results of our approach have decreased, with 9% in terms of MCC and 3-5% in terms of AUC and F1 scores but with negligible (for F1-standard) to small effect sizes. But overall, we believe that AppTracker still is a promising solution that allows mitigating the lack of data, especially for new mobile apps having no enough release history, and outperforms the state-of-the-art approaches.



(a) Standard AUC (b) Weighted AUC (c) Standard F1

(d) Weighted F1 (e) MCC

Fig. 12: Boxplots comparing scores of AppTracker against baseline approaches for cross-project validation.

Summary for RQ2. *Under cross-project scenario, our* AppTracker *still outperforms state-of-the-art techniques with a significant improvement by achieving an average of 26% in terms of MCC, 64% for Weighted-AUCand and 56% in terms of Weighted-F1. Compared to within-project scenario, our results has slightly decreased but overall, our approach still a promising solution that can be practically used when little/no data is available for new apps.*

5.3 Results of RQ3 (Feature importance analysis)

While in the previous RQs, we investigated the predictive performance of AppTracker, in this stage we are interested in understanding how important is each feature for the generated rules, as this would be helpful to prioritize the refactoring efforts during the maintenance process. To this end, we apply the Permutation

Table 11: Performance of AppTracker vs. the state-of-the-art for cross-project validation (Median scores among the studied apps in percentage).

Algorithm	**MCC**	**AUC**		**F1**	
		Standard	*Weighted*	*Standard*	*Weighted*
AppTracker	**26**	**63**	**64**	**47**	**56**
SVC	17	59	59	41	47
LR	16	58	59	40	45
RF	15	57	57	38	43
XGB	15	58	57	39	44
BNB	14	57	57	38	44
SVM-reg	10	56	57	36	40
mono-GP	9	54	54	34	43
Lin-reg	8	55	56	33	36
KNN	8	54	55	35	40
KNN-reg	3	51	52	29	33
DT	3	51	51	29	33
XGB-reg	0	43	40	19	19
RF-reg	0	33	33	0	0
RT	0	46	45	23	24

Table 12: Statistical tests results of AppTracker compared to ML techniques for the cross-project scenario.

Metric	**F1**						**AUC**						**MCC**		
	Weighted			**Standard**			**Weighted**			**Standard**					
	p-value	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*	*p-value*	*A*	*Effect*
vs. BNB	$< 10^{-16}$	0.75	L	$< 10^{-16}$	0.70	M	$< 10^{-16}$	0.69	M	$< 10^{-16}$	0.71	M	$< 10^{-16}$	0.69	M
vs. DT	$< 10^{-16}$	0.90	L	$< 10^{-16}$	0.85	L	$< 10^{-16}$	0.83	L	$< 10^{-16}$	0.86	L	$< 10^{-16}$	0.84	L
vs. KNN	$< 10^{-16}$	0.83	L	$< 10^{-16}$	0.77	L	$< 10^{-16}$	0.75	L	$< 10^{-16}$	0.76	L	$< 10^{-16}$	0.77	L
vs. LR	$< 10^{-16}$	0.74	M	$< 10^{-16}$	0.66	S	$< 10^{-16}$	0.64	S	$< 10^{-16}$	0.67	M	$< 10^{-16}$	0.66	S
vs. mono-GP	$< 10^{-16}$	0.77	L	$< 10^{-16}$	0.76	L	$< 10^{-16}$	0.77	L	$< 10^{-16}$	0.80	L	$< 10^{-16}$	0.78	L
vs. RF	$< 10^{-16}$	0.77	L	$< 10^{-16}$	0.69	M	$< 10^{-16}$	0.67	M	$< 10^{-16}$	0.70	M	$< 10^{-16}$	0.68	M
vs. SVC	$< 10^{-16}$	0.71	M	$< 10^{-16}$	0.65	S	$< 10^{-16}$	0.63	S	$< 10^{-16}$	0.65	S	$< 10^{-16}$	0.65	S
vs. within	**0.01**	0.42	S	**0.03**	0.43	N	10^{-7}	0.34	S	10^{-5}	0.37	S	10^{-7}	0.34	S
vs. XGB	$< 10^{-16}$	0.76	L	$< 10^{-16}$	0.69	M	$< 10^{-16}$	0.66	S	$< 10^{-16}$	0.70	M	$< 10^{-16}$	0.67	M
vs. KNN-reg	$< 10^{-16}$	0.90	L	$< 10^{-16}$	0.82	L	$< 10^{-16}$	0.81	L	$< 10^{-16}$	0.85	L	$< 10^{-16}$	0.84	L
vs. Lin-reg	$< 10^{-16}$	0.87	L	$< 10^{-16}$	0.73	M	$< 10^{-16}$	0.73	M	$< 10^{-16}$	0.80	L	$< 10^{-16}$	0.78	L
vs. RF-reg	$< 10^{-16}$	0.95	L	$< 10^{-16}$	0.88	L	$< 10^{-16}$	0.88	L	$< 10^{-16}$	0.92	L	$< 10^{-16}$	0.87	L
vs. RT	$< 10^{-16}$	0.94	L	$< 10^{-16}$	0.87	L	$< 10^{-16}$	0.86	L	$< 10^{-16}$	0.91	L	$< 10^{-16}$	0.88	L
vs. SVR-reg	$< 10^{-16}$	0.82	L	$< 10^{-16}$	0.71	M	$< 10^{-16}$	0.71	M	$< 10^{-16}$	0.75	L	$< 10^{-16}$	0.75	L
vs. XGB-reg	$< 10^{-16}$	0.94	L	$< 10^{-16}$	0.85	L	$< 10^{-16}$	0.85	L	$< 10^{-16}$	0.90	L	$< 10^{-16}$	0.87	L

L: Large, M: Medium, S: Small, N: Negligible

Feature Importance (PFI) technique then, we cluster the results using the Scott-Knott test. In the following, we report the results of feature importance analysis within-project and under cross-project validations. For the sake of readability, we report only the top-5 metrics (in terms of their importance scores). For more details, please refer to our replication package [26].

5.3.1 Within-project results

Table 13 shows the top-5 metrics ranked and grouped by their importance scores, as determined by the Scott-Knott test.

Link to the last update. The results show that the median percentage of negative rating of the previous update (*last_perc_neg_rating*) is the most important feature for our approach, with an average score of 9%. A closer examination reveals that this feature achieves the highest scores in 6 out of 19 apps. For example, in `com.lionmobi.powerclean` 85% of bad updates have *last_perc_neg_rating* $\geq$ 3.9%. In this app, removing this feature would result in a decrease of 13% in the prediction accuracy of AppTracker. A similar observation can be applied to `com.google.android.youtube` app in which we also observed that eliminating *last_perc_neg_rating* would a decrease of 20% in the prediction accuracy. This can be explained by the fact that the app may have some unstable moments in which users continue expressing their complaints related to an issue from the previous update. Hence, our findings comply with prior work showing that developers may need to perform changes through multiple updates until they recover from a bad update [11].

Release Size. The installation size of an app (*APK_size*) is the second most important feature across the studied apps with an average score of 7.3% and being the most important feature for one app, namely `air.com.playtika.slotomania` in which the feature obtained 17% of importance score. Furthermore, the percentage of change in the installation size (*chang_perc_APK_size*) is the top-3 feature but with no statistical difference compared to *APK_size* according to Scott-Knott tests results. Additionally, *chang_perc_APK_size* is the top-1 for two out of 19 apps, which indicates that the change in the size of an app at the time of the release could affect the current rating. For instance, we found in `com.emoji.coolkeyboard` app, that 56% of bad updates have *last_perc_neg_rating* $\geq$ 2%; which indicates that larger volume of code implies higher probability to contain a bug [82] and thus may lead to the user's dissatisfaction.

Release time. *release_time* and *delay_last_release* (G3) have also helped in discriminating the updates. While *release_time* has on average an importance score of 6%, *delay_last_release* obtained 5% and appears on top of the most important features for one app namely `com.emoji.ikeyboard`. In this app, eliminating *delay_last_release* feature in this app can lead to a decrease of 10% in the prediction accuracy of AppTracker. Additionally, a manual investigation has revealed that all the bad updates in this app have *delay_last_release* $\leq$ 31 days which suggests that faster release time can introduce more bugs and thus lead to negative ratings. We also advocate that developers may need to employ proper testing tools to assure the quality of their quickly deployed releases.

Table 13: The ranking of the top-5 features within-project scenario, divided into distinct groups that have a statistically significant difference in the mean.

Groups	**Feature**	**Average Importance Score (in %)**
G1	last_perc_neg_rating	9.22
G2	chang_perc_Apk_size	7.32
	Apk_size	7.05
G3	release_time	6.33
	delay_last_release	5.16

5.3.2 Cross-project results

The PFI analysis results under cross-project scenario are displayed in Table 14.

The APK size. This dimension appears again in the top-3 list of most important features with two factors (*APK_size* and *chang_perc_APK_size*) and the scores of these features are comparable as revealed by Scott-Knott ESD test (*i.e.* clustered in the same group G1). While *APK_size* appears on the top-1 list of 278 apps, *chang_perc_APK_size* is the top-1 feature in 183 out of 1,313 apps. Hence, developers can consider optimizing their code complexity as a mean to fix/avoid update issues.

Library. The number of integrated libraries (Nlib) is the top-3 most important feature with an average score of 6.1% and being the top-1 in 99 apps. By examining our generated rules of bad updates, we have found that *Nlib* is usually associated with $\geq$. This result is in line with Ahasanuzzaman et al. [30] and Gui et al. [32] studies' results as the authors showed that the frequency and size of displayed Ad increases the number of negative reviews.

SDK. The SDK dimension seems to be helpful to differentiate the updates under cross-project scenario. In fact, the minimum SDK (*min_SDK*) is the top-4 most important feature with an average score of 5.7% and being the top-1 for 47 apps. This finding is in line with previous study by Tian et al. [82] in which authors found that, high-rated apps have a higher minimum and target SDK as users are benefiting from the latest features provided by SDK.

Link to previous updates. The results in the table clearly indicate that The median aggregated rating of all previous updates (*hist_rating*) is among the most important features for the all studied apps with an average score of 5.1%. We also found this feature to be dominant in 83 out of 1,313 apps, which strengthens our previous findings claiming that if the previous update's rating highly affects the label of the current update. Being in line with our motivating example in Section 2, this finding indicates that it is indeed hard to keep the users' confidence if a bad release occurs. That is, getting back the users' satisfaction may need time.

Table 14: The ranking of the top-5 features for cross-project scenario, divided into distinct groups that have a statistically significant difference in the mean.

Groups	Feature	Average Importance Score (in %)
G1	Apk_size	7.61
	chang_perc_Apk_size	7.09
G2	Nlib	6.11
G3	min_sdk	5.72
G4	hist_rating	5.16

Summary for RQ3. *Within project validation, the Permutation Feature Importance (PFI) analysis reveals that (1) the link between the current and last update(s) ratings, (2) the APK size and its related percentage of change and (3) the release time dimension are the most prominent features for* AppTracker*. When it comes to the cross-project rules, These findings are further strengthened for (1) and (2). In addition, we found that for cross-project scenario, the number of Ad libraries and the minimum required SDK version helped a lot to characterizing the updates.*

6 Discussion and Implications

In this section, we discuss the implications of our results in practice.

Supporting mobile apps developers track bad updates. The usefulness of our AppTracker approach has been shown through its achieved performance in both within and cross-project validations. Nevertheless, we believe that the key innovation of our approach is its ability to provide the user with a comprehensible justification for the classification especially when the changes made in the release are non-trivial. Moreover, it is worth noting that, thanks to the flexibility of MOGP techniques, it can be possible to reduce the complexity of the generated detection rules (*e.g.*, tree size and/or depth) in order to generate more comprehensible justification by considering this objective in the fitness function (or as a constraint in the solution encoding), but at the cost of scarifying the accuracy as these objectives are in conflict [16].

Android developers need to pay attention to the quality of their app next release. Our results indicate that the history of the previous negative rating (*i.e.*, the *hist_perc_neg_rating* and *lastt_perc_neg_rating* features) is among the top important features. Hence, if an app loses reputation through repeated bad releases, it will be hard to get back its reputation in the future. Often, time constraints push mobile apps developers to release faster, however, they should consider a trade-off between time and quality. That is, given that the mobile apps market is evolving quickly with many competitors, developers should pay special care to their updates and should maintain their reputation over time.

The smaller the release, the smaller the risk of releasing. Our results, for the most important features, indicate that the change in the release size (*APK_size*) is among the most influencing features. While users typically tend to see new features, improvements, and bug fixes, released regularly, as a sign of evolution, there is a dilemma with this. Moving features around and changing behavior can be confusing and harm app's user experience, so it's important to manage how new changes are released. Little and often is a good way to go as small releases are less risky. For instance, suppose a developer releases ten features at once, the risk of having a bug is high. In worst scenario, each of the ten released feature, can have a bug. If this happens, the developer would be in a bad situation, trying to fix ten serious bugs and get an update out as soon as possible. To minimize your risk, releasing smaller and more frequent is likely a successful strategy.

Learn best practices for the next release in mobile apps development. Teaching the next generation of engineers best practices for the release management process and its impact on the users is of crucial importance. Educators can use our study results and our dataset [26] to teach and motivate students to follow best release practices while avoiding bad updates that may cause user dissatisfaction or regression in their apps. In particular, our real world dataset of 50,700 updates from 1,717 Android apps, represents a valuable resource that could enable the introduction of mobile apps release to students using a "*learn by example*" methodology, illustrating best releasing practices that should be followed and bad practices that should be avoided.

Other formulations for the problem. Within the evolutionary process, our technique evolves detection rules, mimicking the creation of decision trees, to solve the three-class classification problem. While in this paper we showed that this tree-based approach can achieve satisfactory results, there is a room of improvement. For instance, it is interesting to explore solving the three-class classification problem without decomposing it to multiple binary classifications.

7 Threats to Validity

In this section, we review the main threats to the validity of our findings:

Threats to internal validity are concerned with the factors that could have affected the validity of our results. The main concern could be related to the stochastic nature of search-based algorithms, and some ML techniques (*e.g.* DT). To address this issue, we repeated each experimentation 31 times and considered the median scores values used to evaluate the predictive performance. Threats to internal validity could also be related to possible errors in our experiments. To conduct our experiments, we used real-world dataset collected from Google Play Store, the largest market place for mobile applications and mined user reviews on real time in a period of over three years using a dedicated tool. Another possible threat to internal validity could be related to bias in the replication of the benchmark approaches. We employed widely used tools and implementation of the search algorithms, MOEA Framework [60], and the Scikit-learn [58] and XGB [59] Python libraries for the machine learning algorithms. respectively. Thus, we believe that there is a negligible bias towards internal threats to validity.

Threats to construct validity are mainly related to the rigor of the study design. First, we relied on three standard performance metrics namely F1-score widely employed in predictive models comparison [61]. Second, although we used different families learning algorithms, there exist other techniques. As a future work, we plan to extend our empirical study with other baseline techniques. Another threat to construct validity could be related to parameters' tuning as setting different parameters can lead to different results for search-based as well as ML techniques. We mitigated this issue by applying several trial and error iterations to tune search-based algorithms and relied on Grid Search [78] method to find the optimal settings of ML techniques. Thus, future replication of this work should explore other ranges/parameters and their impacts on the predictive performance. An additional threat to internal validity is related to training and test sets selection. As an attempt to mitigate this issue, we considered in RQ1 the time series validation which is a realistic scenario as it considers the chronological order of apps' releases. In RQ2, we selected a typical scenario in which we train AppTracker on data from the same category (*i.e.* similar characteristics). Future work is planned to validate our approach considering a time-aware selection in the cross-project setting.

Conclusion threats to validity. Conclusion threats to validity concern the relationship between the treatment and the outcome. To provide support for the conclusions derived from the obtained results, we use Wilcoxon signed rank test [70] with a 95% confidence level while using Bonferroni correction [71]. Vargha-Delaney A (VDA) [72] is also used to measure the effect size. This non-parametric method is widely recommended in SBSE context [73]. The employed statistical analysis provides strong evidence for validating our assumptions and our experimental study. Hence, we believe that there is negligible threat to the validity of our conclusions.

Threats to external validity are concerned with the generalizability of results since the experiments were based on free-to-download android apps. Hence, future replications of this study are necessary to confirm our findings in other contexts such as paid Android applications and iOS mobile applications.

8 Related Work

In this section, we review the related works that can be divided into 2 classes (1) analysis of user feedback, and (2) release engineering in mobile apps.

8.1 Studies on user review analysis in mobile apps

Several research works have analyzed user reviews in mobile apps to extract knowledge about different mobile app development aspects. Pagano and Maalej [85] investigated user reviews and found that users tend to provide their review feedback shortly after a new app release, while negative feedback (*e.g.*, shortcomings) is generally destructive. Later, Maalej and Nabil [7] used various techniques to collect different features from user reviews, then used different ML algorithms to label reviews into four categories: (i) feature request, (ii) bug report, (iii) user experience, and (iv) unspecified. Similarly, Panichella et al. [8, 9] proposed an approach named as App Reviews Development Oriented Classifier (ARdoc) which classifies user reviews into five categories: (i) feature request, (ii) bug report, (iii) providing information, (iv) requesting information and (v) others. El Zarif et al. [86] studied users' feedback and found that users express their intentions to switch to competitors when facing issues in the used software systems. Hence, Assi et al. [87] proposed FeatCompare that extracts features from user reviews of competitor apps. The obtained results show that FeatCompare outperforms the existing state-of-the-art approaches with 14.7% on average. They also found that 70% of the surveyed app developers agree on the potential benefits of using FeatCompare to extract features of competitor apps. Hu et al. [88] studied the consistency of star ratings and reviews of popular free hybrid Android and iOS apps. They found that some hybrid apps do not obtain coherent user ratings across platforms. Sarro et al. [89] showed the possibility of predicting user ratings for an app based on the features it offers in Android and BlackBerry with high accuracy.

To investigate the user rating influence, Harman et al. [90] studied over 32k BlackBerry applications and found a high correlation between the average user rating and the number of downloads of an app. Later, Martin et al. [91, 92] found that paid and free app releases tend to have a positive impact on the success of an app and that free apps with significant releases are more likely to have positive effects on the user ratings. Moreover, they found that app releases related to bug fixes and new features are more likely to increase user ratings. Moreover, Noei et al. [27] found that some specific mobile device characteristics (such as CPU) have a high relation with the user-perceived quality. Recently, Hassan et al. [33] investigated emergency updates in Android apps and revealed that these emergency updates are unlikely to be followed by other emergency updates so that they tend to have a long longevity. The study also revealed that emergency updates are often preceded by updates having more negative user reviews than the emergency ones themselves. Moreover, Khalid et al. [93] investigated iOS mobile apps user complaints from 20 iOS app reviews and identified 12 types of complaints. Most of these complaints were related to functional errors, as well as privacy and ethics-related issues. Chen et al. [94] studied User Interface (UI) issues mentioned in the reviews of 31,579 apps in the Google Play Store and found that UI-related reviews have lower ratings than the other reviews. Moreover, Chen et al. identified seventeen issue types (e.g., layout and navigation) related to the UI of mobile apps. Gui et al. [32] studied various aspects of advertisement (ads) libraries and found that most Ad complaints leading to negative reviews were related to user interface concerns such as the frequency, timing and location of the displayed ads.

Several studies exploited user reviews and complaints to help in various maintenance and evolution activities. For instance, Ciurumelea et al. [95] studied the textual description of user reviews then leveraged machine learning and information retrieval techniques to plan for the next release. Their approach aim at categorizing reviews and recommending the relevant source code files that should to be modified to address the issue described in the user review. Palomba et al. [96] introduced an approach namely ChangeAdvisor that automatically analyzes user reviews from which it recommends source code artifacts to be changed using natural language processing and clustering algorithms.

8.2 Studies on releases engineering in mobile apps

Several research works focused on studying release practices in mobiles apps. Nayebi et al. [4] performed a survey to study release strategies adopted for mobile apps and their impact on users. Their study shows that experienced developers are mostly aware that their release strategy affects user review and expressed interest in accommodating users' feedback in their release strategy. From user perspective, the study revealed that while users value apps with frequent updates, they also point out that frequent updates could negatively affect users' opinion about an app. Later, Domínguez-Álvarez and Gorla [97] studied mobile apps releasing practices in both iOS and Android and found that developers make new releases of their apps more frequently in Android than on iOS. They also found that there is no synchronization in releasing apps on both platforms.

Calciati et al. [98, 99] studied the evolution of Android apps to investigate how apps behavior changes across different releases of the same app. Most of the observed changes are related to an increased leak of sensitive data, an increase of added of permission, and an increase of API calls related to dangerous permissions in posterior releases over time. Nayebi et al. [100] built different analogical reasoning models to predict Android apps release marketability based on changes in release and code attributes. The obtained results indicate that Android app releases follow certain patterns over time that allow predicting the success of future releases success.

Xia et al. [47] are the first to propose a machine learning technique to predict crashing mobile releases. Using a number of change factors such as complexity, time, and diffusion, they trained a Naive Bayes classifier to predict crashing releases for 10 open source apps. Their results revealed that the technique can improve the prediction of random guessing by 50% and 28% in terms of F1 and AUC, respectively. Later, Su et al. [101] studied crashing releases in open source and commercial Android apps based on thrown exceptions and found that Android framework-related exceptions (*e.g.*, app Management, Database and Widget) and library exceptions are the main root causes. Recently, Yang et al. [102] analyzed the release notes of 69,851 releases for 2,232 apps in the Google Play Store and identified six patterns of release notes (*e.g.*, apps with short and rarely updated release notes). The obtained results show that apps with long release notes have higher ratings than other apps. They also found that apps with shifting in their release notes patterns have encountered an increase in the average rating of these apps. Recently, Hamdi et al. [103, 104] conducted a longitudinal study on refactoring activities in Android apps. They found that while developers often refactor their apps source code, bad coding and design practices are unlikely to be removed though refactoring.

While there are several studies on user reviews and release practices and issues in mobile apps, there are no specific approaches to predict bad releases. In our approach, our goal is to track all bad releases, including crashing ones by leveraging the user feedback as reviews typically include the experienced crashes/issues by end users.

9 Conclusion and Future Work

This paper proposed a novel search-based approach for bad mobile apps tracking, AppTracker, in which we adapted NSGA-II to generate optimal detection rules for each class (*i.e.* bad, good and neutral). The rules have tree-like representations in order to find the best trade-off between two conflicting objective functions to (1) maximize the true positive rate, and (2) minimize the false positive rate of the binary classification. An empirical study is conducted on a benchmark of 50,700 updates of 1,717 free Android apps having over 50,700 release updates. Considering two validation scenarios namely cross-validation and cross-project validation, the statistical analysis of the obtained results reveals that AppTracker is advantageous over mono-objective Genetic Programming (mono-GP) and seven Machine Learning (ML) techniques, which confirms that our formulation is better to solve the problem. Regarding the bad updates analysis, we found that (1) the previous updates ratings and (2) the APK size are the most important features for both within and cross-project scenarios.

Our future research agenda includes performing a larger empirical study with apps from other stores with free and paid applications. We also plan to consider other metrics, *e.g.*, code-level quality. Furthermore, we plan to extend our AppTracker approach in the form of a bot to integrated into the development pipeline of Android apps to notify developers of their updates' risk before releasing a new version to end-users. Furthermore, while the prediction of the corresponding class (good, bad, or neutral) is helpful for Android developers to follow better release practices and improve users experience, predicting the specific negativity ratio would provide a more fine-grained analysis. Hence, as future work it is interesting to build regression-based models to estimate the negativity ratio. Moreover, we plan to implement a bot based on AppTracker and conduct a user study with our industrial partner to better evaluate our approach in an industrial setting.

Acknowledgements This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2018-05960.

References

1. Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In *International Conference on Mobile Software Engineering and Systems*, pages 99–110, 2019.
2. Moses Openja, Bram Adams, and Foutse Khomh. Analysis of modern release engineering topics:–a large-scale study using stackoverflow–. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 104–114, 2020.
3. Sebastian Klepper, Stephan Krusche, Sebastian Peters, Bernd Bruegge, and Lukas Alperowitz. Introducing continuous delivery of mobile apps in a corporate environment: A case study. In *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, pages 5–11. IEEE, 2015.
4. Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices for mobile apps – what do users and developers think? In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 552–562, 2016.
5. Fabio Palomba, Mario Linares-Vasquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *IEEE international conference on software maintenance and evolution (ICSME)*, pages 291–300, 2015.
6. Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. Release planning of mobile apps based on user reviews. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2016.
7. Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference (RE)*, pages 116–125. IEEE, 2015.
8. Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 281–290. IEEE, 2015.
9. Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. Ardoc: App reviews development

oriented classifier. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 1023–1027, 2016.
10. Daniel Martens and Walid Maalej. Release early, release often, and watch your users' emotions: Lessons from emotional patterns. *IEEE Software*, 36 (5):32–37, 2019.
11. Safwat Hassan, Cor-Paul Bezemer, and Ahmed E Hassan. Studying bad updates of top free-to-download apps in the google play store. *IEEE Transactions on Software Engineering*, 2018.
12. Anderson Rocha and Siome Klein Goldenstein. Multiclass from binary: Expanding one-versus-all, one-versus-one and ecoc-based approaches. *IEEE Transactions on Neural Networks and Learning Systems*, 25(2):289–302, 2013.
13. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. In *A fast and elitist multiobjective genetic algorithm: NSGA-II*, volume 6, pages 182–197, 2002.
14. Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.
15. Ali Ouni. Search based software engineering: challenges, opportunities and recent applications. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1114–1146, 2020.
16. Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.
17. Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering*, 40(9): 841–861, 2014.
18. Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *15th ACM International Conference on Global Software Engineering*, pages 43–54, 2020.
19. Marouane Kessentini and Ali Ouni. Detecting android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 122–132, 2017.
20. Ali Ouni, Marouane Kessentini, Katsuro Inoue, and Mel O Cinnéide. Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4):603–617, 2015.
21. Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.
22. Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
23. Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *J. Mach. Learn. Res.*, 20(177):1–81, 2019.
24. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. (1), 2017.
25. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization for defect

prediction models. 2018.
26. Dataset for bad releases detection. Available at : https://github.com/stilab-ets/AppTracker, 2021.
27. Ehsan Noei, Mark D Syer, Ying Zou, Ahmed E Hassan, and Iman Keivanloo. A study of the relation of mobile device attributes with the user-perceived quality of android apps. *Empirical Software Engineering*, 22(6):3088–3116, 2017.
28. AppAnnie. App annie. available online:. https://www.appannie.com/en/,, 2020. Accessed: 2020-04-01.
29. Akdeniz. Google play crawler. available online:. https://github.com/Akdeniz/google-play-crawler,, 2013. Accessed: 2021-03-1.
30. Md Ahasanuzzaman, Safwat Hassan, Cor-Paul Bezemer, and Ahmed E Hassan. A longitudinal study of popular ad libraries in the google play store. *Empirical Software Engineering*, 25(1):824–858, 2020.
31. Md Ahasanuzzaman, Safwat Hassan, and Ahmed E Hassan. Studying ad library integration strategies of top free-to-download apps. *IEEE Transactions on Software Engineering*, 2020.
32. Jiaping Gui, Meiyappan Nagappan, and William GJ Halfond. What aspects of mobile ads do users care about? an empirical study of mobile in-app ad reviews. *arXiv preprint arXiv:1702.07681*, 2017.
33. Safwat Hassan, Weiyi Shang, and Ahmed E Hassan. An empirical study of emergency updates for top android mobile apps. *Empirical Software Engineering*, 22(1):505–546, 2017.
34. Scikit learn. Scikit-learn multiclass-classification. https://scikit-learn.org/stable/modules/multiclass.html#multiclass-classification,, 2006. Accessed: 2021-01-10.
35. Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2010.
36. Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17, 2015.
37. Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):23, 2016.
38. Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.
39. Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. Search-based refactoring: Towards semantics preservation. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 347–356, 2012.
40. Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, 2013.
41. Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd International*

Conference on Software Engineering (ICSE), pages 1–10, 2011.

42. Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, 2020.
43. Fangcheng Qiu, Meng Yan, Xin Xia, Xinyu Wang, Yuanrui Fan, Ahmed E Hassan, and David Lo. Jito: a tool for just-in-time defect identification and localization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1586–1590, 2020.
44. Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. Effort-aware just-in-time defect identification in practice: a case study at alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1308–1319, 2020.
45. Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2017.
46. Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 157–168, 2016.
47. Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.
48. Jing Xia, Yanhui Li, and Chuanqi Wang. An empirical study on the cross-project predictability of continuous integration outcomes. In *14th Web Information Systems and Applications Conference (WISA)*, pages 234–239, 2017.
49. J Krishna Kishore, Lalit M. Patnaik, V Mani, and VK Agrawal. Application of genetic programming for multicategory pattern classification. *IEEE transactions on evolutionary computation*, 4(3):242–258, 2000.
50. Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, volume 2, pages 1070–1077. IEEE, 2001.
51. William Smart and Mengjie Zhang. Using genetic programming for multiclass classification by simultaneously solving component binary classification problems. In *European Conference on Genetic Programming*, pages 227–239. Springer, 2005.
52. Zheng Chen and Siwei Lu. A genetic programming approach for classification of textures based on wavelet analysis. In *2007 IEEE International Symposium on Intelligent Signal Processing*, pages 1–6. IEEE, 2007.
53. Pedro G Espejo, Sebastián Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(2): 121–144, 2009.

54. Saad M Darwish, Adel A EL-Zoghabi, and Doaa B Ebaid. A novel system for document classification using genetic programming. *Journal of Advances in Information Technology Vol*, 6(4), 2015.
55. Jafar Tanha, Yousef Abdi, Negin Samadi, Nazila Razzaghi, and Mohammad Asadpour. Boosting methods for multi-class imbalanced data classification: an experimental review. *Journal of Big Data*, 7(1):1–47, 2020.
56. Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pages 1–4, 2015.
57. Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
58. Scikit learn. Scikit-learn classification and regression models. `https://scikit-learn.org/stable/supervised_learning`,, 2006. Accessed: 2021-01-10.
59. XGBoost. Xgboost python package. `https://xgboost.readthedocs.io/en/latest/python/index.html`,, 2006. Accessed: 2021-01-10.
60. David Hadka. Moea framework. `http://moeaframework.org/`,. Accessed: 2020-12-01.
61. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.
62. Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):1–13, 2020.
63. Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4): 427–437, 2009.
64. Paula Branco, Luís Torgo, and Rita P Ribeiro. Relevance-based evaluation metrics for multi-class imbalanced domains. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 698–710. Springer, 2017.
65. Mohammad Hossin and MN Sulaiman. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):1, 2015.
66. Benjamin P Evans, Bing Xue, and Mengjie Zhang. What's inside the black-box? a genetic programming method for interpreting complex machine learning models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1012–1020, 2019.
67. Julian Eberius, Katrin Braunschweig, Markus Hentsch, Maik Thiele, Ahmad Ahmadov, and Wolfgang Lehner. Building the dresden web table corpus: A classification approach. In *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*, pages 41–50. IEEE, 2015.
68. Mohammad Mehedi Hassan, Sana Ullah, M Shamim Hossain, and Abdulhameed Alelaiwi. An end-to-end deep learning model for human activity recognition from highly sparse body sensor data in internet of medical things environment. *The Journal of Supercomputing*, pages 1–14, 2020.
69. Patrick Royston. Approximating the shapiro-wilk w-test for non-normality. *Statistics and computing*, 2(3):117–119, 1992.
70. Frank Wilcoxon, SK Katti, and Roberta A Wilcox. Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank

test. *Selected tables in mathematical statistics*, 1:171–259, 1970.
71. Richard A Armstrong. When to use the b onferroni correction. *Ophthalmic and Physiological Optics*, 34(5):502–508, 2014.
72. András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
73. Shiva Nejati and Gregory Gay. *11th International Symposium Search-Based Software Engineering*, volume 11664. 2019.
74. Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.
75. Simone Scalabrino, Giovanni Grano, Dario Di Nucci, Rocco Oliveto, and Andrea De Lucia. Search-based testing of procedural programs: Iterative single-target or multi-target approach? In *International Symposium on Search Based Software Engineering*, pages 64–79, 2016.
76. Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47. Springer, 2011.
77. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2018.
78. Scikit-learn.org. Parameter estimation using grid search with scikit-learn. available online:. `https://scikit-learn.org/stable/modules/grid_search.html`,, 2006. Accessed: 2020-12-01.
79. Jerrold H Zar. Spearman rank correlation. *Encyclopedia of biostatistics*, 7, 2005.
80. Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. Examining the stability of logging statements. *Empirical Software Engineering*, 23(1):290–333, 2018.
81. Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865, 2017.
82. Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–310, 2015.
83. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823. IEEE, 2015.
84. Urvesh Bhowan, Mengjie Zhang, and Mark Johnston. Genetic programming for classification with unbalanced data. In *European Conference on Genetic Programming*, pages 1–13, 2010.
85. Dennis Pagano and Walid Maalej. User feedback in the appstore: An empirical study. In *21st IEEE international requirements engineering conference (RE)*, pages 125–134, 2013.

86. Omar El Zarif, Daniel Alencar da Costa, Safwat Hassan, and Ying Zou. On the relationship between user churn and software issues. In *17th International Conference on Mining Software Repositories (MSR)*, pages 339–349. ACM, 2020.
87. Maram Assi, Safwat Hassan, Yuan Tian, and Ying Zou. Featcompare: Feature comparison for competing mobile apps leveraging user reviews. *Empirical Software Engineering*, 26(5):94, 2021.
88. Hanyang Hu, Shaowei Wang, Cor-Paul Bezemer, and Ahmed E Hassan. Studying the consistency of star ratings and reviews of popular free hybrid android and ios apps. *Empirical Software Engineering*, 24(1):7–32, 2019.
89. Federica Sarro, Mark Harman, Yue Jia, and Yuanyuan Zhang. Customer rating reactions can be predicted purely using app features. In *IEEE 26th International Requirements Engineering Conference (RE)*, pages 76–87, 2018.
90. Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *IEEE working conference on mining software repositories (MSR)*, pages 108–111, 2012.
91. William Martin, Federica Sarro, and Mark Harman. Causal impact analysis for app releases in google play. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 435–446, 2016.
92. William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2016.
93. Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE software*, 32(3):70–77, 2014.
94. Qiuyuan Chen, Chunyang Chen, Safwat Hassan, Zhengchang Xing, Xin Xia, and Ahmed E. Hassan. How should I improve the UI of my app?: A study of user reviews of popular apps in the google play. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):37:1–37:38, 2021.
95. Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C Gall. Analyzing reviews and code of mobile apps for better release planning. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 91–102, 2017.
96. Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 106–117, 2017.
97. Daniel Domínguez-Álvarez and Alessandra Gorla. Release practices for ios and android apps. In *ACM SIGSOFT International Workshop on App Market Analytics*, pages 15–18, 2019.
98. Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. What did really change with the new release of the app? In *15th International Conference on Mining Software Repositories (MSR)*, pages 142–152, 2018.
99. Paolo Calciati and Alessandra Gorla. How do apps evolve in their permission requests? a preliminary study. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 37–41, 2017.

100. Maleknaz Nayebi, Homayoon Farahi, and Guenther Ruhe. Which version should be released to app store? In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 324–333, 2017.
101. Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering*, 2020.
102. Aidan Z.H. Yang, Safwat Hassan, Ying Zou, and Ahmed E. Hassan. An empirical study on release notes patterns of popular apps in the google play store. *Empirical Software Engineering*, pages 1–41, 2021.
103. Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.
104. Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel O Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39, 2021.