Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study

Anderson Uchôa*, Caio Barbosa*, Daniel Coutinho*, Willian Oizumi*, Wesley K. G. Assunção*,
Silvia Regina Vergilio†, Juliana Alves Pereira*, Anderson Oliveira*, Alessandro Garcia*
*Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil
†Computer Science Department, Federal University of Paraná (UFPR), Brazil

***Abstract*—Companies have adopted modern code review as a key technique for continuously monitoring and improving the quality of software changes. One of the main motivations for this is the early detection of design impactful changes, to prevent that design-degrading ones prevail after each code review. Even though design degradation symptoms often lead to changes' rejections, practices of modern code review alone are actually not sufficient to avoid or mitigate design decay. Software design degrades whenever one or more symptoms of poor structural decisions, usually represented by smells, end up being introduced by a change. Design degradation may be related to both technical and social aspects in collaborative code reviews. Unfortunately, there is no study that investigates if code review stakeholders, e.g, reviewers, could benefit from approaches to distinguish and predict design impactful changes with technical and/or social aspects. By analyzing 57,498 reviewed code changes from seven open-source systems, we report an investigation on prediction of design impactful changes in modern code review. We evaluated the use of six ML algorithms to predict design impactful changes. We also extracted and assessed 41 different features based on both social and technical aspects. Our results show that *Random Forest* and *Gradient Boosting* are the best algorithms. We also observed that the use of technical features results in more precise predictions. However, the use of social features alone, which are available even before the code review starts (e.g., for team managers or change assigners), also leads to highly-accurate prediction. Therefore social and/or technical prediction models can be used to support further design inspection of suspicious changes early in a code review process. Finally, we provide an enriched dataset that allows researchers to investigate the context behind design impactful changes during the code review process.**

***Index Terms*—design changes; code review; machine learning**

I. Introduction

Modern code review is a practice that has been widely adopted by major companies [1], [2]. It is typically a lightweight, informal, asynchronous, and tool-assisted practice aimed at monitoring, detecting and removing issues that were introduced during development tasks [2]. Supported by platforms such as Gerrit and GitHub, the code review process is initiated by one developer referred to as the *owner*, which modifies the original codebase and submits a new code change to be reviewed by other developers – the so-called *reviewers*.

A key concern of all stakeholders involved in a code review, including code owners, reviewers, and team managers, is to become aware of ongoing changes impacting the design [3]–[5]. The underlying motivation is to monitor and inspect those design impactful changes so that stakeholders can anticipate, find, and remove signs of design degradation before the end of a code review. Otherwise, those design harmful changes can become prevalent after the code review [6], [7]. If design impactful changes are not discriminated and brought to attention early, it will increase the likelihood of those changes finding their way into the software system for several reasons. These include reviewers deviating their effort to other quality checks in later review stages or even starting to focus only on the main purpose of the issue being resolved. Moreover, design-related changes become harder to revert towards the end of the review as many other inter-related modifications were already realized as the review progresses. In fact, these reasons explain why many design impactful changes get unnoticed as nearly 40% of pull request rejections are related to design issues [8].

Early identification of impactful changes that degrade the software design is important during code review [3], [9], [10]. Code reviewers are expected to inspect ongoing changes and provide prompt feedback to code owners in the form of comments. In turn, the code owner should fix and forward the new version of the code for inspection. Such a procedure is repeated in multiple iterations, which are called *revisions*. This sequence of revisions ends up with either the acceptance or rejection of the change into the codebase [11], [12].

Despite its importance, recent studies found these modern code review practices are far from being sufficient to prevent design-degrading changes [3], [13]. Design degradation occurs whenever a change introduces poor structural decisions, i.e., design smells [14]–[17]. Tools for detecting design smells tend to be inefficient when a change is still at its early stages. For instance, the full addition of a new feature can be complex and its realization needs many revisions to be accomplished. Moreover, other types of change can be complex as well. In fact, several changes in software projects are fully realized only after many revisions in a single review.

If these harmful changes are not reversed early, i.e., before a code review is ended, rework will be necessary after the changes of the last merged revision. Further changes with time-consuming refactorings will have to be applied later. Given the costs of design refactorings, they are unlikely to be applied and smells will be compounded over time, thereby accelerating the design degradation [3], [4], [18].

Due the importance of the early identification of design relevant changes during the review process, stakeholders must use all available information during the code reviewing process. In code review platforms, stakeholders have either technical

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
To be published in the Proceedings 18th International Conference on Mining Software Repositories (MSR), 2021 in Madrid, Spain.

or social information at their disposal to be used as additional information, both before or after each change. Social information are often available as soon as code review starts. Social information includes: number of prior code changes submitted by the code owner, and centrality of the code owner on the collaboration graph [19]–[22]. Technical information are available after changes and revisions done during the review. Examples of technical information include: number of times a file has been changed and types of change [19], [23]–[25].

The advantage of using technical and social metrics to characterize and predict failures have widely been studied [26]–[29]. However, their use to discriminate and predict design impactful changes is rarely studied [3], [13]. In fact, such metrics can act as indicators of design impactfulness of ongoing changes along the code review process. Hence, to the best of our knowledge, there is no study on which types of metrics can be used as effective features in Machine Learning (ML) algorithms to accurately predict design impactful changes.

This paper presents results of a large-scale empirical study that investigates whether and how technical and social metrics can be used to predict design impactful changes. To this end, we analyzed more than 50k code reviews of seven real-world systems from two large open source communities. We mined and examined if a comprehensive suite of technical and social metrics can discriminate design (un)impactful changes. Then, we explored the use of these metrics, as features for six interpretable ML algorithms, which tend to offer an effective prediction for different tasks and contexts, e.g., [29], [30]. Finally, we evaluated the predictive power of the selected features and algorithms to assist developers to automatically determine whether a code change is impactful.

Our key findings and contributions are: (1) both social and technical metrics are able to distinguish design (un)impactful changes; (2) the use of technical features results in more accurate predictions, when compared to the social ones; (3) features related to the code change, commit message, and file history dimensions are effective for differentiating (un)impactful changes; (4) *Random Forest* and *Gradient Boosting* have shown to be the most accurate in predicting design impactful changes; and (5) an enriched dataset and replication package that allows researchers to investigate the context and motivations behind design impactful changes during code reviews.

II. Motivating Example

Next we show the importance of considering design impact and using social and technical aspects during code review. To this end, we rely on two scenarios of the jgit system in which code review is conducted on the Gerrit platform.

Scenario A. Let us consider the review 3345 [31], composed of seven revisions, in which two developers performed a major change to "*Replace TinyProtobuf with Google Protocol Buffers*". After the last revision, 12,215 insertions and 2,404 deletions were performed in 58 files. Additionally, 104 design smells were introduced, leading to structural degradation related to the lack of abstraction (46), encapsulation (17), and modularity (42). Interestingly, design impact was not mentioned during revisions of the reviews of this major change. In other words, the replacement of a third-party component was conducted without an explicit concern with possible side effects that the change could introduce into the system.

Scenario B. Now consider the review 825 [32], which aims to "*Implement a Dircache checkout (needed for merge)*". This review had four reviewers and 18 revisions. After the first revision (R_1), we observed a inclusion of three smells: two Unutilized Abstraction and one Insufficient Modularization. Such smells were perceived by a developer, according to the following comment: "*One problem I faced here: we do have an abstraction to access the WorkTree when walking (reading) on it.*" Additionally, during the revisions (R_3 to R_{10}), we observed removals and reintroductions of the smells Unutilized Abstraction and Insufficient Modularization, characterizing a high fluctuation of design degradation during revisions. In other words, despite the developers identify degradation symptoms, they still are not able to see all the ramifications and impacts of their changes along with revisions.

Both scenarios illustrate the need for mechanisms to aid developers on identifying and preventing design impactful changes. Such mechanisms can rely on the large, diverse, rich information from social and technical aspects of the system and stakeholders in the code review. In **Scenario A**, when developers are unaware of the design impact of their change, a mechanism could have supported reviewers by automatically analyzing the discussions that took place on previous revisions and the components involved in the changes to predict the impact of changes in the current revision on the system design. In **Scenario B**, a mechanism could have analyzed the previous behavior of the code owner, and reviewers could better understand which changes are harming the source code.

In this context, one could argue: why not only using existing tools, such as Designite [33], to identify design degradation? Despite useful, such tools are limited. Besides the reasons already mentioned in Section I, they rely only on static analysis in which detection strategies do not adapt per revision, not exploring the history of changes and multiple sources of information. They ignore the variation of technical and social aspects inherent to the code review process [3], [13], [20].

In summary, the motivations for our work are: (i) a real-world need for mechanisms to aid stakeholders in identifying impactful design changes during code review, (ii) availability of large and rich sources of information that can be used to make stakeholders aware of their changes' impact, and (iii) limitation of existing tools on using historical and dynamic information to support stakeholders during code review.

III. Study Settings

A. Research Questions

Our study is guided by four research questions (RQs).

RQ$_1$: *Are design impactful changes significantly different from unimpactful ones in terms of social and technical metrics?* – Social and technical aspects may be avoiding or amplifying design degradation. To capture such aspects, we used a set of metrics detailed in Section III-E. **RQ**$_1$ aims at investigating which metrics are able to distinguish between design impactful changes and unimpactful ones.

RQ$_2$**:** *What is the performance of ML algorithms to predict design impactful and unimpactful changes?* – Once we show empirical evidence that distinguishes impactful and unimpactful changes, **RQ**$_2$ aims at investigating the use of supervised ML techniques to assist developers in automatically make their decisions. In practice, some prediction algorithms perform better than others, depending on the task. Thus, we compare the performance of six interpretable ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting. We chose these algorithms since they provide an intuitive and easy to explain model [34], [35].

RQ$_3$**:** *How effective are the social and technical features as a proxy to the design impactfulness changes?* – **RQ**$_3$ aims at evaluating and comparing the performance of both kinds of features. To this end, we applied the ML algorithm using three feature sets: a set using only social features, a set using only technical ones, and a set using technical and social features together. By answering **RQ**$_3$, we will be able to identify which kind of features are the best predictor, as well as the effectiveness of combining social and technical features. Furthermore, we also evaluated the effectiveness of a feature selection step for the three sets.

RQ$_4$**:** *What features are the best indicators of impactful design changes?* – **RQ**$_4$ aims at understating which features are considered the most relevant by the models. Such knowledge is essential because, in practice, a model should be as simple as and require as little data as possible. By answering **RQ**$_4$, we will be able to provide insights to practitioners and researchers as to what factors best indicate design impactful changes.

B. Code Review Data

To answer our RQ$_s$, we need not only information of the source code to distinguish design impactful changes, but also to analyze every code revision submitted along the code review process and investigate technical and social information related to those revisions. Thus, instead of mining code review data ourselves, we used the data provided by the Code Review Open Platform (CROP) [36], an open-source dataset that links code review data to software changes. All systems in CROP employ Gerrit as their code review tool. Hence, by using CROP, we have access to a rich dataset of code changes. To this end, given a certain system, CROP provides a complete copy of the entire codebase for each revision and its respective parent, which represents the system's codebase at the time of review. In other words, unlike the Git repository of a system, which contains only accepted revisions (i.e., the changes in the final revisions in a review), the CROP stores all revisions.

In our study, we adopt all Java systems included in the CROP dataset: four systems from the Eclipse community and three systems from the Couchbase community, as presented in Table I. For sake of completeness, we remove the reviews whose status is "Open" since they may not have been assigned to reviewers, and the set of reviewers may still change.

TABLE I
SOFTWARE SYSTEMS INVESTIGATED IN THIS STUDY

Community	System	# of Reviews	# of Revisions	Time span
Eclipse	**jgit**	5,304	13,578	10/09 to 11/17
	egit	5,220	12,814	9/09 to 11/17
	platform.ui	4,527	13,418	20/13 to 11/17
	linuxtools	4,074	11,418	6/12 to 11/17
Couchbase	**java-client**	909	2,622	11/11 to 11/17
	jvm-core	828	2,269	4/14 to 11/17
	spymemcached	536	1,379	5/10 to 7/17

C. Detection of Degradation Symptoms within Code Reviews

We investigated two categories of degradation symptoms, which are fine-grained (FG) and coarse-grained (CG) smells [37]. Although we do not focused on architectural smells, we empirically observed they follow similar trends in a complementary analysis. *FG smells* are indicators of structural degradation in the scope of methods and code blocks [37]. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. *CG smells* are symptoms that may indicate structural degradation related to object-oriented principles, e.g., abstraction, encapsulation, and modularity [37], [38]. An example of CG smell is Insufficient Modularization [37]. This symptom occurs in classes that are large and complex due to the accumulation of responsibilities. Such categories encapsulate a set of symptoms that are more perceived and used by developers in practice to identify and refactor source code locations degraded [15], [39]–[41].

For automatically detecting symptoms of such categories, we used a state-of-the-practice tool called DesigniteJava [33], which detected a total of 27 degradation symptoms types: 17 CG smells, and 10 FG smells. Hence, for each system, we identified these symptoms by considering each submitted revision that has undergone the code review process. Thus, we used CROP to access the versions of the system before and after the revision took place. Next, we detected the degradation symptoms in each version before and after revision. By following this methodology, we are guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in the revision. Table II lists the 27 symptoms types investigated in our study. We provide all descriptions, detection strategies, and thresholds for each type of symptom in the replication package [42].

TABLE II
DEGRADATION SYMPTOMS INVESTIGATED IN THIS STUDY

Coarse-grained Smells
Imperative Abstraction, Multifaceted Abstraction, Unutilized Abstraction, Unnecessary Abstraction, Deficient Encapsulation, Unexploited Encapsulation, Broken Modularization, Insufficient Modularization, Hub Like Modularization, Cyclic Dependent Modularization, Rebellious Hierarchy, Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Missing Hierarchy, Broken Hierarchy.
Fine-grained Smells
Abstract Function Call From Constructor, Complex Conditional, Complex Method, Empty Catch Block, Long Identifier, Long Method, Long Parameter List, Long Statement, Magic Number, Missing Default.

D. Identification of Design Impactful Change Instances

We identified design impactful change instances in two steps: (i) identification of smelliness files by considering each revision of a code change, where each revision was compared to its parent, i.e., the codebase's version before any revision; and (ii) the computation of design degradation indicators.

To illustrate the first step, let R_s = {r$_1$, r$_2$, ..., r$_n$} be the set of submitted revisions for a given review s. For each revision in R_s (i.e., $\forall r_i,\ r_i \in R_s$), we use the CROP to retrieve the system's versions before and after r_i. Next, we check the presence of degradation symptoms (both CG and FG smells) in the files of r_i. The two file sets (before and after) might not be exactly the same, due to files created and deleted during the review process. Since the before version of each revision is its parent, we guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in r_i, avoiding the collateral effects of the rebase [43]. The output of this step is, for each revision in R_s, the version of the files impacted before and after the revision.

In the second step, we rely on an existing grounded theory [15] that explains that developers tend to consider multiple degradation characteristics in terms of *density* and *diversity* of symptoms. In addition, the use of *density* and *diversity* of symptoms for detecting design degradation is supported by other studies [18], [40]. Such studies show that degraded code elements tend to be affected by higher *diversity* and *density* of symptoms when compared to other code elements. However, before selecting *diversity* and *density* of symptoms as metrics, we compared the two lists of smells (by density and diversity) before and after revisions. As a result, we observed a small average variation (<1 type of smell/revision). Thus, computing the degradation in(de)crease with density imposes only a minor threat. Therefore, we take into account only the *density*, as a metric to measure the level of design degradation.

Therefore, for each selected system, we computed this characteristic in the context of each symptom category (CG and FG smells), for all the collected revisions. *Density* was computed for each version before and after revision, as the sum of the number of symptom instances in the set of smelliness source code files. The computation of *density* before and after revisions, allowed us to generate two different indicators of design degradation for each revision, where each indicator represents the differences in *density* of FG and CG smells.

In summary, a positive difference in the density of symptoms indicates an *increase in the degradation* as a result of the revision, therefore, harming the design. Similarly, a negative difference indicates a *reduction of the degradation* as a result of the revision. Finally, no variation indicates that there has been no structural design change. We consider that a design change is impactful when an *increase* or *reduction* in design degradation was observed as a result of submitted changes. Conversely, unimpactful changes are submitted changes that do not affect design degradation. Table III shows the number of revisions identified as design (un)impactful changes for each system and symptom category.

TABLE III
THE NUMBER OF REVISIONS IDENTIFIED AS DESIGN IMPACTFUL CHANGES

Project	Coarse-grained Smells			Fine-grained Smells		
	Impactful	Unimpactful	All	Impactful	Unimpactful	All
java-client	751 (29%)	1,871 (71%)	2,622	1,340 (51%)	1,282 (49%)	2,622
jvm-core	630 (28%)	1,639 (72%)	2,269	1,054 (46%)	1,215 (54%)	2,269
spymemcached	423 (31%)	956 (69%)	1,379	586 (42%)	793 (58%)	1,379
platform.ui	2,431 (18%)	10,987 (82%)	13,418	4,460 (33%)	8,958 (67%)	13,418
egit	2,973 (23%)	9,841 (77%)	12,814	5,192 (41%)	7,622 (59%)	12,814
jgit	3,995 (29%)	9,583 (71%)	13,578	6,082 (45%)	7,496 (55%)	13,578
linuxtools	3,321 (29%)	8,097 (71%)	11,418	4,837 (42%)	6,581 (58%)	11,418
Total	**14,524 (25%)**	**42,974 (75%)**	**57,498**	**23,551 (41%)**	**33,947 (59%)**	**57,498**

E. Features for Design Impactful Change Prediction

We extracted a set of features able to capture both technical and social aspects of the changes involved in each revision of a code review. Each feature corresponds to a metric. We detail each feature and its description per dimension on Table IV.

TABLE IV
TECHNICAL AND SOCIAL FEATURES ADOPTED IN OUR STUDY

Technical Features		
Dimension	**Name**	**Description**
Size	**NLA**	Number of inserted lines in this code change
	NLD	Number of deleted lines in this code change
	CHURN	Number of lines added to and removed in this code change
	NFA	Number of added files in this code change
	NFD	Number of deleted files in this code change
Diffusion	**NCF**	Number of changed files in this code change
	NMD	Number of modified directories in this code change
	ME	Distribution of modified code across files in this code change
	NLANG	Number of programming languages used in this code change
	NFT	Number of file types in this code change
Complexity	**NSA**	Number of added code segments in this code change
	NSD	Number of deleted code segments in this code change
	NSU	Number of updated code segments in this code change
File history	**FM**	Number of times files in this code change were modified before
	FD	Number of developers who changed files in this code change
Textual	**ML**	Number of words in description of this code change
	BUG	Whether description of this code change contains word "bug"[1]
	FEAT	Whether description of this code change contains word "feature"[1]
	IMPR	Whether description of this code change contains word "improve"[1]
	DOC	Whether description of this code change contains word "document"[1]
	REFC	Whether description of this code change contains word "refactor"[2]
Social Features		
Dimension	**Name**	**Description**
Developer's Experience	**NC**	Number of prior code changes submitted by the owner of this code change
	NRC	**NC** in recent 120 days
	NDC	**NC** that contain at least one directory affected by this code change
	NR	Number of prior code changes the owner of this code change is assigned to inspect
	MR	Merged rate of prior code changes submitted by the owner of this code change
	RMR	**MR** in recent 120 days and normalized over the recent change number
	DMR	**MR** that contain at least one directory affected by this code change
Discussion Activity	**NIC**	Number of inline comments made by reviewers.
	NWIC	Sum of the all words of each inline comment.[3]
	PWIC	**NWIC** weighted by the number of inline comments.[3]
	NGC	Number of general comments made by reviewers.
	NWGC	Sum of the all words of each general comment.[3]
	PWGC	**NWGC** weighted by the number of general comments.[3]
	DL	Number of general comments and inline comments written by reviewers
Collaboration networks	**SD**	The degree centrality for a node v is the fraction of nodes it is connected to.[4]
	SCLOS	The inverse of the sum of all distances to all other nodes.[4]
	SB	The sum of the fraction of all-pairs shortest paths that pass through v.[4]
	SE	The centrality for a node based on the centrality of its neighbors.[4]
	SCLUST	The geometric average of the subgraph edge weights.[4]
	SKC	Maximal subgraph that contains nodes of degree k or more.[4]

[1]And more keywords based on previous work [44], [45]
[2]And more keywords based on previous work [46]
[3]We discarded comments made by non-human participants and applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers
[4]The initial node is the committer of the revision and all other nodes are the reviewers of each revision

From the **technical perspective**, we extracted 21 features related to source code, modification history of the files, and the textual description of the change. Moreover, these features were categorized into five dimensions: (i) **Size** consists of features related to source in their smallest granularity. Prior studies have found that large patches may need more effort to review [47]; (ii) **Diffusion** comprehends the features about changes distributed on two or more files (e.g., number of changed files). Prior studies also found that revisions, where their changes scatter across a large number of files or directories, may need more effort to review [19], [25]. Thus, we expected that the diffusion of a change could influence the likelihood of the change being impactful; (iii) **Complexity** comprehends the features on the complexity of a change. A code change with more code segments modified is likely more complex and requires more effort and time to be reviewed [48]; (iv) **File history** is composed of features related to the history of the files. The number of prior changes to a file can be a good indicator to detect degraded files. Moreover, files that are previously touched by more developers are more likely to introduce degradation symptoms [23]–[25]; and (v) **Textual** consists of features that capture textual characteristics of the commit message. Previous studies [49], [50] found that the description length of a patch is related to its likelihood of receiving poor comments. Additionally, the commit message may contain more information about a code change that may help reviewers comprehend the change more easily.

From the **social perspective**, we extracted 20 features that characterize the developer's experience, collaboration network, and participation in discussions. Moreover, these features were grouped into three dimensions: (i) **Developer's experience** comprise the features related to the previous experience of the code change owner. Previous studies [19]–[21] found that developer experience is essential information for predicting design issues. Such studies claim that if a developer often submits changes in recent times prior to the change, they will be more familiar with the recent developments of the system, and thus the code change may be fewer design issues; (ii) **Discussion activity** comprise the features of communication between developers and reviewers. In fact, classes having degraded symptoms can create more discussion among the reviewers [3], [13]. As well as, discussions with a high number of comments around code changes would find possible design symptoms, improving or maintaining the quality; and (iii) **Collaboration networks** consists of features of social networks. Previous studies [20], [22], [48] found that collaboration factors (i.e., level of participation within the system) could influence code review outcomes. For this reason, we constructed a network based on the collaboration of owners and reviewers to use the features proposed by [22].

F. Development of the Impactfulness Prediction Models

We experimented with six different (binary classification) supervised ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting.

Training and testing the models. We trained and tested the models as follows. Firstly, we collected the design (un)impactful changes instances for a given system. We merged them into a single dataset, where design impactful changes instances are marked with a *true* value, and design unimpactful changes instances are marked with a *false* value. Secondly, before training the models, we dealt with imbalanced data, a common issue with software engineering data [51]. In our case, the number of design impactful changes instances varies; i.e., the design impactful changes instances might be greater than or smaller than the number of unimpactful ones. To that end, we relied on the undersampling algorithm, which randomly selects instances of the oversampled class. Thirdly, we scaled all the features to a [0, 1] range to speed up the learning process of the algorithms [52]. Fourthly, we used the grid search to tune the hyperparameters of each model using five folds. Grid search is an exhaustive search that examines all of the combinations of a specified set of candidate settings to find the best combination [53].

Finally, to train the model, we employed a 10-fold cross-validation strategy using the hyperparameters established by the search [54]. This strategy randomly partitions the dataset into 10 folds of equal size, in which each fold has the same proportion of the various criticality classes. A single fold is then used as a test set, while the remaining ones are employed for training the model, i.e., they are independent of each other.

Performance evaluation. We evaluated the performance of each generated model, by analyzing confusion matrices, obtained from the testing strategy described above, and reporting the values of well-known measures [55]. *Precision* is the percentage of detected code changes that are actually impactful ($Pr = \frac{TP}{TP+FP}$). *Recall* is the percentage of correctly predicted impactful design change relative to all of the changes that are actually unimpactful ($Re = \frac{TP}{TP+FN}$). The *F1-score* ($F1$) is the harmonic mean of precision and recall. Additionally, to mitigate the limitation of choosing a fixed threshold when calculating precision and recall, we compute the *Area Under the ROC Curve (AUC)* values. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a design change is predicted as impactful or unimpactful.

Replication package. All data described previously, features used for training and testing the ML algorithms, hyperparameters analyzed, generated ML models, as well as the confusion matrix and statistical analysis are available in [42].

IV. Results and Discussions

A. Design impactful changes vs. unimpactful ones

To answer $\mathbf{RQ}_1$, we used the *Wilcoxon Rank Sum Test* [56] and the *Cliff's Delta (d)* measure [57] to verify which metrics are able to discriminate between impactful and unimpactful design changes. To this end, we explore each metric described in Section III-E. The *Cliff's Delta (d)* measure [57] shows how strong is the difference between design impactful changes and unimpactful ones in terms of the analyzed metrics. Since we are performing multiple comparisons, we need to adjust the *p-values* to consider the increased chance of rejecting the null hypothesis simply due to random chance. To do so, we apply the widely used *Bonferroni correction* [58], which controls the familywise error rate. For this method, we consider that each system is a family, which means that we perform the correction in the *p-values* of the features at the system level.

Table V shows the results obtained for coarse- (CG) and fine-grained (FG) smells, where the 1st column contains the type of metric while the 2nd column shows the evaluated metric. The 3rd to 16th columns show the Wilcoxon Test and d results for each system, each group of two columns represents the results for CG and FG smells, respectively. Statistical significant differences (*p-value* < 0.05) are highlighted as gray cells. To interpret the *Cliff's Delta (d)* effect size, we employ a well-known classification [59], that defines four categories of magnitude, which are represented in Table V: *negligible* (without symbol), *small* (*), *medium* (**), and *large* (***). The positive d magnitudes are represented by the $(+)$ symbol and the negative ones are represented by the $(-)$ symbol.

We emphasize that $\mathbf{RQ}_1$ does not consider the FEAT, IMPR, IMPR, DOC, and REFC metrics since the *Wilcoxon Test* and the *Cliff's Delta* measure are not suitable for boolean metrics.

Correlation between inline comments and impactful changes. Among all metrics in the dimension of discussion, we highlight those related to inline comments, *inline comments*

TABLE V
STATISTICAL SIGNIFICANCE OF THE WILCOXON RANK SUM TEST AND THE CLIFF'S DELTA (D) FOR COARSE AND FINE-GRAINED SMELLS

Dimension	Metric	spymemcached		java-client		jvm-core		platform.ui		jgit		egit		linuxtools		all	
		CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG
Developer's Experience	NC	(−)	(−)	(−)	(−)	(−)	(−)*	(−)	(−)	(−)	(−)	(−)	(−)*	(−)	(−)	(−)	(−)
	NRC	(+)	(−)	(−)	(−)	(−)*	(−)	(−)*	(−)*	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	NDC	(+)*	(+)	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
	NR	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(−)	(+)	(+)
	MR	(−)	(−)	(+)	(+)	(+)	(−)	(+)	(+)	(−)	(+)	(−)	(−)	(−)	(−)	(−)	(−)
	RMR	(−)*	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(−)	(−)	(−)	(−)	(+)
	DMR	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Discussion Activity	NIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NWIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	PWIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NGC	(−)	(−)	(+)	(−)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NWGC	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	PWGC	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	DL	(+)	(+)	(+)	(+)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Collaboration Networks	SD	(−)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SCLOS	(−)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SB	(−)	(−)	(−)	(−)	(+)	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SE	(−)*	(−)	(−)	(−)	(−)	(+)	(−)	(−)*	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SCLUST	(+)*	(+)	(+)	(−)	(−)	(−)	(+)	(+)	(−)	(−)	(−)	(−)	(+)	(+)	(+)	(+)
	SKC	(+)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(−)
Size	NLA	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NLD	(+)*	(+)	(+)*	(+)**	(+)**	(+)***	(+)*	(+)*	(+)	(+)	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*
	CHURN	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NFA	(+)***	(+)*	(+)***	(+)**	(+)***	(+)***	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*
	NFD	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Diffusion	NFC	(+)**	(+)*	(+)**	(+)**	(+)***	(+)***	(+)*	(+)*	(+)*	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**
	ND	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)**	(+)***	(+)**	(+)***	(+)**	(+)***	(+)**
	ME	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)**	(+)***	(+)**	(+)***	(+)**	(+)***	(+)**
	NLANG	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
	NFT	(+)	(+)	(+)	(−)	(+)	(+)	(+)**	(+)*	(+)*	(+)	(+)***	(+)*	(+)**	(+)*	(+)**	(+)*
Complexity	NSA	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NSD	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)
	NSU	(+)*	(+)*	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
File History	FD	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)*	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**
	FM	(+)**	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**	(+)*	(+)*	(+)**	(+)*	(+)**	(+)**	(+)**	(+)*
Textual	ML	(−)	(+)	(+)*	(+)**	(+)	(+)*	(+)*	(+)*	(+)*	(+)**	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*

(NIC), *words in inline comments (NWIC)*, and *percentage of words in inline Comments (PWIC)*. For CG smells, they were statistically significant for most systems, except for the java-client (all three) and jvm-core (PWIC) systems. Such metrics also showed positive magnitude in all cases. This means that impactful changes, represented by both CG and FG smells, are often associated with a higher volume of inline comments. Nevertheless, the magnitude of these metrics was negligible in all cases. Thus, we conclude that discussion activity metrics in isolation are not enough for differentiating impactful changes.

Developer's experience dimension. The *directory changes (NDC)* metric presented consistent results in most systems both for CG e FG smells. *NDC* was the only metric in the developer's experience dimension that presented small and medium (and positives) magnitudes across all projects. This result means that *NDC* is the best metric in its dimension for differentiating impactful changes. **Collaboration networks dimension.** Results were similar on both CG e FG smells. Magnitudes were negative in 76% and 80% of cases also for both smells. Such a result indicates that prior collaboration between author and reviewers contributes to the production of changes that do not impact the design. However, in many cases, the results were not statistically significant.

Finding 1: The usefulness of the metrics from the discussion activity, developer's experience, and collaboration networks dimensions to differentiate design impactful changes is limited. However, the *number of directory changes* presents promising results.

Code metrics as strong indicators of impactful changes. The most relevant metrics for distinguishing between impactful and unimpactful changes were the ones related to code. Their results were not statistically significant in only three cases for FG smells. Moreover, with the exception of the *files deleted (NFD)* metric in the java-client system, all code metrics presented positive magnitudes. Among all code metrics, we highlight the *lines added (NLA)*, *changed lines (CHURN)*, and *segments added (NSA)* metrics, which presented statistically significant results with large magnitude in all cases, for both types of smells. If we restrict our analysis to CG smells, the *files added (NFA)*, *segments deleted (NSD)*, and *modify entropy (ME)* metrics also stand out. The highlighted metrics are closely related to the size (*NLA*, *CHURN*, *NFA*) and complexity (*NSA*, *NSD*, *ME*) of the changes. Thus, such metrics can be used by reviewers to decide when a change requires more attention to the design impact.

Textual dimension. The *message length (ML)* metric showed statistically significant results with small or medium positive magnitudes in most cases. We conjecture that this is because changes with detailed descriptions tend to be more complex, leading to a higher impact on design. **File history dimension.** The *file developers (FD)* and *file modifications (FM)* metrics, showed statistically significant results with positive magnitudes. In this case, the *FD* metric showed medium or large magnitudes in most cases. This result suggests that the more people interacting with a file set, the greater the chance that the next changes in such files will impact the design.

Finding 2: Code, textual, and file history dimensions contain metrics that are relevant to differentiate impactful changes. The most relevant ones can be combined to predict changes that require more attention to design.

B. ML performance for predicting design impactful changes

We address **RQ**$_2$, by reporting and comparing different models after the 10 stratified cross-fold executions. We apply stratified sampling in all the cross-fold executions to make sure both training and test datasets contain the same amount

TABLE VI
PERFORMANCE OF USING DIFFERENT LEARNING ALGORITHMS TO PREDICT DESIGN IMPACTFUL CHANGES

Coarse-grained Smells																								
System	**SVM (linear)**				**Decision Tree**				**Random Forest**				**Naive Bayes (gaussian)**				**Gradient Boosting**				**Logistic Regression**			
	Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
spymemcached	0.77	0.55	0.64	0.69	0.92	0.93	0.92	0.92	0.96	0.93	0.94	0.95	0.84	0.28	0.42	0.61	0.94	0.94	0.94	0.94	0.80	0.62	0.69	0.73
java-client	0.80	0.73	0.76	0.77	0.96	0.95	0.96	0.96	0.97	0.97	0.97	0.97	0.88	0.41	0.56	0.68	0.97	0.95	0.96	0.96	0.80	0.74	0.77	0.78
jvm-core	0.85	0.76	0.80	0.81	0.95	0.96	0.95	0.95	0.97	0.96	0.96	0.96	0.68	0.92	0.78	0.73	0.96	0.96	0.96	0.96	0.85	0.80	0.82	0.83
platform.ui	0.78	0.51	0.62	0.68	0.90	0.93	0.92	0.92	0.94	0.95	0.94	0.94	0.59	0.88	0.67	0.61	0.93	0.95	0.94	0.94	0.71	0.65	0.68	0.69
egit	0.76	0.62	0.68	0.71	0.89	0.91	0.90	0.90	0.93	0.93	0.93	0.93	0.75	0.56	0.64	0.69	0.93	0.94	0.93	0.93	0.75	0.69	0.72	0.73
jgit	0.72	0.61	0.66	0.69	0.90	0.91	0.91	0.91	0.95	0.93	0.94	0.94	0.67	0.58	0.59	0.63	0.94	0.94	0.94	0.94	0.74	0.70	0.72	0.73
linuxtools	0.75	0.62	0.68	0.71	0.92	0.94	0.93	0.93	0.96	0.96	0.96	0.96	0.82	0.18	0.29	0.57	0.95	0.96	0.96	0.96	0.73	0.71	0.72	0.72
All	0.70	0.61	0.66	0.68	0.86	0.87	0.86	0.86	0.93	0.92	0.93	0.93	0.65	0.64	0.64	0.64	0.93	0.92	0.93	0.93	0.70	0.70	0.70	0.70
Average	**0.77**	**0.63**	**0.69**	**0.72**	**0.91**	**0.93**	**0.92**	**0.92**	**0.95**	**0.94**	**0.95**	**0.95**	**0.74**	**0.56**	**0.57**	**0.65**	**0.94**	**0.95**	**0.95**	**0.95**	**0.76**	**0.70**	**0.73**	**0.74**
Median	**0.77**	**0.62**	**0.67**	**0.70**	**0.91**	**0.93**	**0.92**	**0.92**	**0.96**	**0.94**	**0.94**	**0.95**	**0.72**	**0.57**	**0.62**	**0.64**	**0.94**	**0.95**	**0.94**	**0.94**	**0.75**	**0.70**	**0.72**	**0.73**

Fine-grained Smells																								
System	**SVM (linear)**				**Decision Tree**				**Random Forest**				**Naive Bayes (gaussian)**				**Gradient Boosting**				**Logistic Regression**			
	Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
spymemcached	0.89	0.80	0.84	0.85	0.95	0.96	0.95	0.95	0.97	0.97	0.97	0.97	0.92	0.38	0.53	0.67	0.97	0.97	0.97	0.97	0.90	0.82	0.86	0.86
java-client	0.84	0.74	0.79	0.80	0.90	0.95	0.92	0.92	0.94	0.97	0.95	0.95	0.92	0.50	0.64	0.73	0.93	0.96	0.95	0.95	0.84	0.76	0.80	0.81
jvm-core	0.89	0.74	0.81	0.82	0.93	0.96	0.95	0.94	0.96	0.97	0.97	0.97	0.80	0.76	0.77	0.78	0.95	0.97	0.96	0.96	0.88	0.77	0.82	0.83
platform.ui	0.80	0.55	0.65	0.71	0.92	0.96	0.94	0.94	0.96	0.97	0.97	0.97	0.75	0.44	0.50	0.62	0.96	0.97	0.97	0.97	0.79	0.66	0.72	0.74
egit	0.85	0.72	0.78	0.80	0.91	0.95	0.93	0.93	0.95	0.96	0.96	0.96	0.84	0.62	0.71	0.75	0.95	0.97	0.96	0.96	0.84	0.76	0.80	0.81
jgit	0.90	0.62	0.73	0.77	0.91	0.93	0.92	0.92	0.96	0.95	0.95	0.95	0.91	0.36	0.52	0.66	0.96	0.95	0.96	0.96	0.86	0.71	0.78	0.80
linuxtools	0.82	0.65	0.72	0.75	0.92	0.95	0.93	0.93	0.95	0.97	0.96	0.96	0.90	0.23	0.36	0.60	0.95	0.97	0.96	0.96	0.81	0.72	0.76	0.77
All	0.74	0.67	0.70	0.72	0.88	0.90	0.89	0.89	0.95	0.94	0.95	0.95	0.68	0.67	0.67	0.67	0.95	0.94	0.94	0.94	0.75	0.70	0.72	0.73
Average	**0.84**	**0.69**	**0.75**	**0.78**	**0.92**	**0.95**	**0.93**	**0.93**	**0.96**	**0.96**	**0.96**	**0.96**	**0.84**	**0.50**	**0.59**	**0.69**	**0.95**	**0.96**	**0.96**	**0.96**	**0.83**	**0.74**	**0.78**	**0.79**
Median	**0.85**	**0.70**	**0.76**	**0.79**	**0.92**	**0.95**	**0.93**	**0.93**	**0.96**	**0.97**	**0.96**	**0.96**	**0.87**	**0.47**	**0.59**	**0.67**	**0.95**	**0.97**	**0.96**	**0.96**	**0.84**	**0.74**	**0.79**	**0.81**

of design (un)impactful changes instances. Table VI shows the precision (Pr), recall (Re), F1-score (F1), and AUC values of each ML algorithm for each target system and smell levels. The row "all" represents the generated model, when training and testing in the entire dataset and using all features.

The performance of ML algorithms for predicting design impactful changes. Table VI shows that for coarse-grained smells, the average of precision values across models ranges between 0.74 and 0.95, while the recall values range between 0.56 and 0.95. Similarly, the F1-score values range between 0.57 and 0.95, while the AUC values range between 0.65 and 0.95. A similar observation applies when we consider fine-grained smells, in which the average of precision, recall, F1-score, and AUC values ranges between 0.83 and 0.96, 0.50 and 0.96, 0.59, and 0.96, and 0.69, and 0.96, respectively.

To assess statistical differences between the models, we applied Friedman non-parametric test [60] with Nemenyi's post hoc multiple pairwise comparison (p-value $\leq$0.05). We observed that both *Random Forest* and *Gradient Boosting* outperformed the other ML models at coarse-grained (CG) and fine-grained (FG) levels, with significant difference according to the statistical test. Furthermore, both *Random Forest* and *Gradient Boosting* present a similar performance, i.e., without a statistical difference, considering the CG level, with a similar average of 0.95 for both F1 and AUC. However, with a slight difference of 1% for the average of precision and recall values. A similar observation applies when at FG level. Moreover, to ascertain if the level of accuracy is adequate, we compared our model performance results with other approaches both for ML-based smell detection [61], [62] and code review analysis [63]. On average, our two best models achieved similar or better performance results than those previous works.

Finding 3: *Random Forest* and *Gradient Boosting* are the most accurate in predicting design impactful changes within code reviews. Both algorithms achieve an average of F1-score of 0.95 and 0.96, for predicting design impactful changes at a CG and FG level, respectively.

C. The role of social and technical features as predictors

We address **RQ**$_3$, by investigating the performance of different feature sets as a proxy to predict design impactful changes at CG and FG smell levels, namely (i) social features only, (ii) technical features only, and (iii) social + technical features together. Additionally, and given the great number of features, we investigate the difference of the impactfulness prediction between to use or not a step for feature ranking and selection. Instead of simply removing the highly-correlated features by following a filtering method, we decided to apply a wrapper method to feature selection. We applied feature ranking with recursive feature elimination and cross-validated selection of the best number of features for our data. For feature ranking and selection we used the RFECV function available in the scikit-learn's feature selection package [64]. We run RFECV using 5-fold cross-validation and SVC linear as the estimator. After the cross-validation process, and RFECV recommendations, three new sets, namely feature selection sets, were generated, a set with 19, 9, and 40 features respectively for the social, technical, and social + technical dimensions. Similarly to **RQ**$_2$, we check the statistical difference of the results using Friedman test [60] and Nemenyi's post hoc multiple pairwise comparison, with a confidence level of 95%.

The effectiveness of social and technical features as predictors to design impactful changes. To evaluate the effectiveness of social and technical features, we rely on the best ML algorithms, i.e., *Random Forest* and *Gradient Boosting*, based on the **RQ**$_2$ results. Table VII shows the mean values of precision (Pr), recall (Re), F1-score (F1), and AUC for both algorithms, grouped by dimension and feature set.

TABLE VII
PERFORMANCE OF SOCIAL AND TECHNICAL FEATURES AS PROXY TO PREDICT DESIGN IMPACTFUL CHANGES

Random Forest									
Dimension	**Feature set**	**Coarse-grained Smells**				**Fine-grained Smells**			
		Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
Social	all features	0.80	0.81	0.81	0.80	0.79	0.79	0.79	0.79
	feature selection	0.80	0.81	0.81	0.80	0.79	0.79	0.79	0.79
Technical	all features	**0.96**	**0.96**	**0.96**	**0.96**	**0.94**	**0.94**	**0.94**	**0.94**
	feature selection	0.90	0.93	0.91	0.91	**0.94**	**0.94**	**0.94**	**0.94**
Social + technical	all features	0.95	0.94	0.95	0.95	0.93	0.92	0.93	0.93
	feature selection	0.87	0.91	0.89	0.88	0.93	0.92	0.93	0.93

Gradient Boosting									
Dimension	**Feature set**	**Coarse-grained Smells**				**Fine-grained Smells**			
		Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
Social	all features	0.82	0.83	0.82	0.82	0.80	0.81	0.80	0.80
	feature selection	0.81	0.83	0.82	0.82	0.80	0.81	0.80	0.80
Technical	all features	**0.95**	**0.96**	**0.96**	**0.96**	**0.94**	**0.94**	**0.94**	**0.94**
	feature selection	0.90	0.92	0.91	0.91	**0.94**	**0.94**	**0.94**	**0.94**
Social + technical	all features	**0.95**	0.94	0.94	0.94	0.93	0.92	0.93	0.93
	feature selection	0.88	0.91	0.90	0.89	0.93	0.92	0.93	0.93

We observed that social features for impactful changes, at both levels of granularity, reached mean values of Pr, Re, F1, and AUC around 0.8. A similar performance is reached when feature selection was used. Technical features reach mean values of Pr, Re, F1, and AUC around 0.96 and 0.94 for impactful changes at, respectively, coarse- and fine-grained levels. But we also observed that the use of features selection at the coarse-grained level leads to a performance reduction, decreasing values of Pr, F1, and AUC. The Friedman test did not point a significant difference when using feature selection compared to all features, as well as between the two levels of granularity, in the same set of features. In summary, we can conclude that both sets of features are good predictors. They reach values of Pr, Re, F1, and AUC ≥ 0.79 at both levels.

Finding 4: Both social and technical features are effective as a proxy to detect impactful changes. In this way, code review stakeholders may choose the set of features to be used according to their interests and roles.

Social features vs. technical features vs. social + technical features. The set of technical features are better predictors than the set of social features in terms of Pr, Re, F1, and AUC for coarse-grained and fine-grained smells, and both algorithms, with or without feature selection. We observe a significant statistical difference between both sets. Nevertheless, there is no statistical difference, in terms of performance, between technical and social + technical sets. The combination of both kinds of features leads to results that are statistically equivalent to the best results, obtained by the technical feature set. This happens even for the set of social and technical features together when the double number of features is used.

Finding 5: The use of technical features leads to the best results. Moreover, such kind of features can be used in combination with social features without reducing the performance of the ML algorithms.

D. The best features for predicting design impactful changes

We address **RQ**$_4$, by reporting how often each feature appears among the top-1 and top-5 most important features of all the generated models without feature selection. To better understand the importance of social and technical features for predicting design impactful changes per symptom category, we generate different sets of rankings. To this end, we vary the configuration of each model according to the different feature sets, i.e., social features only, technical features only, and both together, and reported the five most frequent features per ranking and feature sets. We used scikit-learn's implementations to extract the feature importance of the SVM (linear), Decision Tree, Random Forest, Gradient Boosting, and Logistic Regression models [64]. We highlight that some models, e.g., SVM, might return the importance of a feature as a negative number, indicating that the feature is important for the prediction of the design unimpactful changes, in our case. Thus, we consider such a feature also important to the models, and thus, we build the ranking using the absolute value of feature importance returned by the models.

The best features using social and technical features in isolation for predicting design impactful changes. Table VIII lists the ranking of the best features across ML algorithms and systems grouped by smell category, i.e., *coarse-grained (CG)* and *fine-grained (FG) smells*, and feature set, i.e., *technical feature only*, and *social feature only*. For each ranking, we show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears by feature set.

TABLE VIII
THE RANKING OF THE MOST IMPORTANT FEATURES ACROSS ML ALGORITHMS USING SOCIAL AND TECHNICAL FEATURES IN ISOLATION

Coarse-grained Smells						
Ranking	Technical Features Only			Social Features Only		
	Dim.	Feature Name	Freq.	Dim.	Feature Name	Freq.
1	Size	# Lines Added	13	Dev Exp	# Directory Changes	23
	Size	# Files Added	11	Disc Act	# Inline Comments	5
	Size	# Changed Lines	6	Dev Exp	# Changes	3
	Complexity	# Segments Added	2	Colab Net.	Social Closeness	2
	Diffusion	# Changed Files	2	Dev Exp	# Recent Changes	1
5	Size	# Lines Added	29	Dev Exp	# Directory Changes	31
	Size	# Files Added	28	Dev Exp	# Changes	25
	Size	# Changed Lines	23	Dev Exp	Merged Ratio	22
	Complexity	# Segments Added	20	Dev Exp	# Recent Changes	20
	Diffusion	Modify Entropy	13	Dev Exp	Recent Merged Ratio	11

Fine-grained Smells						
Ranking	Technical Features Only			Social Features Only		
	Dim.	Feature Name	Freq.	Dim.	Feature Name	Freq.
1	Size	# Lines Added	16	Dev Exp	# Directory Changes	28
	Size	# Changed Lines	10	Disc Act	# Inline Comments	4
	Complexity	# Segments Added	6	Disc Act	# Words in General Comments	1
	File History	# File Modifications	2	Disc Act	% Words in General Comments	1
	Diffusion	# Changed Files	1	Disc Act	Discussion Length	1
5	Size	# Lines Added	30	Dev Exp	# Directory Changes	33
	Size	# Changed Lines	23	Dev Exp	# Changes	22
	File History	# File Modifications	23	Dev Exp	# Recent Changes	22
	Textual	Message Length	17	Dev Exp	Merged Ratio	20
	Complexity	# Segments Added	16	Disc Act	# Inline Comments	11

By considering FG smells with technical features only, we observed that features quantifying the size of the code changes such as, *lines added (NLA)*, and *changed lines (CHURN)* frequently appear in the top-1 ranking, followed by features that quantify complexity (*segments added (NSA)*), history information about files modified (*file modifications (FM)*), and diffusion of a change (*changed files (NCF)*). A similar observation applies in the top-5 ranking, but with the presence of one textual feature, *message length (ML)*, that appears 17 times. Interestingly, when we compare both types of smells, we observed that *NLA*, *CHURN*, *NSA*, *NCF* also appear as important for both symptom categories in the top-1 and top-5 rankings, except for the features *files added (NFA)*, and *modify entropy (ME)* that only appear for CG smells.

On the other hand, when we consider social features only: for FG smells, we observed that the feature *directory changes (NDC)*, that quantifies the developer's experience in terms of the number of prior code changes submitted by the owner that contains at least one directory affected by the current submitted code change, is the most important social feature in the top-1 ranking appearing 28 times across models. Next, less frequently, features that quantify *inline comments (NIC)* made by reviewers on the code change submitted by the owner, and features that quantify discussion activities among the owner and reviewers, such as, *# words in general comments (NGC)*, *% words in general comments (PWGC)*, and *discussion length (DL)* also appear as important features.

Interestingly, by looking at the top-5 ranking, we observed that 4 out of 7 (57%) features that quantify different aspects of the developer's experience are considered as important across models, followed by the *NIC* that appears 11 times. By comparing both categories, we observed that *NDC* keeps its importance in the top-1 and top-5 rankings. Finally, two social features appear as important for CG smells, *social closeness (SCLOS)*, and *recent merged ratio (RMR)*. These observations also reinforce that senior developers should be allocated as reviewers to keep the quality of code review high [3], [7] and promote the knowledge transfer along revisions [2], [65].

Finding 6: In isolation, social features that quantify the developer's experience and discussion activities are indeed considered important across models in both smells categories. Also, as expected, technical features that quantify size, complexity, and diffusion of the code changes are considered important across all models.

The best features using social and technical features together for predicting design impactful changes. Table IX also lists the ranking of the best features across ML algorithms grouped by CG and FG smells. However, in this table, we consider social and technical features as a single feature set. We also show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears.

TABLE IX
THE RANKING OF THE MOST IMPORTANT FEATURES ACROSS ML ALGORITHMS USING SOCIAL AND TECHNICAL FEATURES TOGETHER

Coarse-grained Smells						
Ranking	**Dim.**	**Technical features**	**Freq.**	**Dim.**	**Social features**	**Freq.**
1	Size	# Lines Added	12	-	-	-
	# Files Added	# Files Added	12	-	-	-
	Size	# Changed Lines	7	-	-	-
	Diffusion	# Changed Files	1	-	-	-
	Diffusion	Modified Entropy	1	-	-	-
5	Size	# Lines Added	29	Dev Exp	# Changes	2
	Size	# Files Added	29	Dev Exp	# Recent Change	2
	Size	# Changed Lines	21	Dev Exp	Recent Merged Ratio	2
	Complexity	# Segments Added	19	Dev Exp	# Directory Changes	1
	Diffusion	Modified Entropy	13	-	-	-
Fine-grained Smells						
Ranking	**Dim.**	**Technical features**	**Freq.**	**Dim.**	**Social features**	**Freq.**
1	Size	# Lines Added	12	-	-	
	Size	# Changed Lines	12	-	-	
	Complexity	# Segments Added	7	-	-	
	File History	# File Modifications	2	-	-	
	Diffusion	# Changed Files	1	-	-	
5	Size	# Lines Added	30	Dev Exp	# Changes	2
	Size	# Changed Lines	26	Dev Exp	# Recent Change	2
	Complexity	# Segments Added	18	Dev Exp	Recent Merged Ratio	2
	Diffusion	# Languages	17	Disc Act	% Words in General Comments	1
	File History	# File Modifications	13	Dev Exp	# Directory Changes	1

For FG smells with social and technical features aggregated, we observe that the same technical features listed in Table VIII appears in the top-1 and top-5 rankings, i.e., when we consider the technical features in isolation, the same features also appear as the most important features for FG smells, except for the *message length (ML)*, that only appears when the technical features are in isolation, and the *number of languages (NLANG)* feature, which only appears in the rankings with the combined features. Both of those features appear 17 times each in their respective top-5 rankings. We also observed that when the social and technical features are considered together, social features only appear in the top-5 ranking.

This result indicates that technical features, when combined with social ones, tend to be the most important features across models, especially in the top-1 ranking. However, social features that appear in the top-5 ranking are majority features that quantify the developer's experience. This result, reinforces our previous finding, on the importance of the developer's experience to predicting fine-grained design impactful changes.

We also observed similar behavior for CG smells, in which the same set of technical features when we considered the technical features in isolation, also appear as the most important features across models. The exception to this is the feature *modified entropy (ME)*, which appears 1 time in the top-1 ranking in Table IX. Finally, similar to FG smells the social features only appear in the top-5 ranking, with the prevalence of features that quantify the developer's experience.

Finding 7: Technical features tend to be considered the most important features across models when compared with social features. However, social features that quantify the developer's experience are also considered important across models in both symptom categories.

Some features never appear in any of the rankings. Considering the full rankings of features importance, we observed that for both FG and CG smells, when technical and social features are considered in isolation, the technical features that capture textual characteristics of the commit message such as, *has bug (BUG)*, *has feature (FEAT)*, *has improvement (IMPR)*, *has document (DOC)*, and *has refactor (REFC)* do not appear even when we consider the top-5 ranking. A similar observation applies when we consider all social and technical features together, except for the feature *FEAT*. On the social features, for both smell types the features *directory merged ratio (DMR)* and *social betweenness (SB)* never appear in the top-1 and top-5 feature importance ranking. Additionally, the feature *percentage words in general comments (PWGC)* also does not appear in the rankings for CG smells.

On the other hand, when we combine all social and technical features together, eight social features do not appear among the rankings for both symptoms category. These features, four are from the collaboration network dimension, *social closeness (SCLOS)*, *social clustering (SCLUST)*, *social eigenvector*, and *social betweenness (SB)*. Another group of three features that do not appear are from the discussion activity dimension: *inline comment (NIC)*, *general comments (NGC)*, and *discussion length (DL)*. Finally, *reviews (NR)* from the developer's experience dimension also does not appear. Specifically for FG smells, we observed that *merged ratio (MR)*, *recent merged ratio (RMR)*, *directory merged ratio (DMR)*, *social k coreness (SKC)*, *# words in inline comments (NWIC)*, *# words in general comments (NWGC)*, and *% words in general comments (PWGC)* also never appear in the rankings.

Finding 8: Features from the textual dimension consistently did not appear in any of the rankings when technical features are used in isolation. Conversely, when both types of features are used in conjunction, features related to collaboration networks and discussion activity tend to not appear for both symptom categories.

V. Threats to Validity

Construct and Internal Validity. The precision and recall of degradation symptoms detection may have influenced our results. We mitigate this threat by selecting a detection tool, successfully used in recent studies on design degradation [3], [13], [33], [66], and previous work [67] indicated a precision of 96% and a recall of 99%. Moreover, there is evidence that developers tend to refactor code elements with a high density and diversity of the selected symptoms [18]. Although, there are more instances of design unimpactful changes in our dataset we mitigate this imbalanced dataset by removing instances from the over-represented class through random under-sampling strategy. Moreover, the selection of the ML algorithms and their parameter settings may affect the accuracy and influence interpretability. We mitigate this threat by selecting the most widely used interpretable ML algorithms and, for a fair comparison, we searched for their best parameters through an extensive hyperparameter search via grid search, and 10-fold cross-validation strategy. Finally, we selected and computed a wide number of social and technical features that helped us measure different social and technical dimensions of the changes involved in each code revision, e.g., developer experience, and file history. The rationales for using metrics are supported by prior studies, e.g., [22], [47], [48], [68]. We wrote scripts to automate compute these metrics, and all implementations were validated by three paper authors.

Conclusion and External Validity. About the descriptive analysis, four paper authors contributed to the analysis of design (un)impactful changes. For the statistical analysis, we rely on the *Wilcoxon Rank Sum Test*, *Bonferroni correction*, and *Cliff's Delta (d)* measure to verify which metrics are able to discriminate between (un)impactful design changes. We also computed largely used performance measures [55], precision, recall, F1-score, and AUC score. Furthermore, we rely on the Friedman non-parametric test, and Nemenyi-tests to avoid subjective opinion regarding the best accurate model and the role of social and technical features as predictors. About the feature importance, our ML pipeline does not currently have a way to get the feature importance of Naive Bayes, but we have no reason to believe the lack of this model affected the conclusion of RQ_4. We are aware that there is an inherent threat to transferring our findings to other systems, i.e., our results may be subject to the degradation characteristics of these specific systems. However, we focused on seven systems to be able to make robust and reliable statements about if learning approaches can be used in such settings.

VI. Related Work

Previous studies applied ML to help in the code review activity [63]. In fact, ML has been used intensively for smell characterization and detection [61], [62], [69], [70]. Recent reviews on this subject show that distinct algorithms and smells have been addressed. They differ in the set of metrics, regarding product and process [71], [72]. There are also works that investigate how smells affect program comprehension and bug proneness, and how code becomes smelly. Sae-Lim et al. [73], [74] proposes an ML approach that uses information from the current release to predict modules that will decay in the next release, i.e., modules that will become smelly. The prediction model is built based on source code metrics, but it can be improved by using developers' context that is expressed by a set of modules to be modified by the developers. We have not found ML-based studies concerned with how changes performed by developers in the code review context can impact on the degradation symptoms like smells.

Works that study the impact of code review on the overall software quality do not usually apply ML. Some works investigate the relationship between code review and bug introduction [75], [76], and between code review and design quality [6], [7]. Morales et al. [6] studied the impact of code review coverage (proportion of changes code reviewed) and reviewer involvement on smells. They observe that high coverage and review participation can reduce the occurrence of smells. Pascarella et al. [7] investigated the influence of code reviews in the code smell severity. They conclude that higher values of code review quality dimensions (activity and participation) are related to a decrease in code smell severity.

In a previous work [3], we observed that the majority of code reviews are unimpactful on the evolution of design degradation and that certain code review practices can both combat or even accelerate design degradation. Our findings showed that design discussions may not be enough for avoiding degradation and that certain practices, such as long discussions and a high rate of reviewers' disagreement, might increase the design degradation risk. We also observed a wide design degradation fluctuation during the revisions of each review, meaning that developers were both introducing and removing degradation symptoms along with a single code review. This fluctuation often resulted in the amplification of design degradation at the end of the review. This fact shows the importance of reviewers to know a *priori* the impact of their revisions. Our study is the first initiative to contribute in this direction. We propose an ML approach that uses social and technical features. This allows reviewers to distinguish design (un)impactful changes and prioritize revisions to avoid design degradation.

VII. Final Remarks

In summary, our main findings pointed out that: (i) both social and technical features are able to distinguish between design impactful changes and unimpactful ones; (ii) Random Forest and Gradient Boosting are the most accurate algorithms; and (iii) both social and technical features are effective as a proxy to predict impactful changes. Our findings also provide insights for new studies and be the basis for tool builders creating a new generation of tools to aid developers in automatically predicting design impactful changes during code reviews. We also show that: (i) existing detection tools should be more interactive, in a stepwise manner, to anticipate, find, and remove signs of degradation before finishing a review; (ii) In addition to only technical features, the combination with social features is promising for predicting design (un)impactful changes; and (iii) qualitative studies should be performed to explain other aspects governing the decision-making process discriminating and predicting design impactful changes.

Acknowledgment. This work was partially funded by CNPq (434969/2018-4, 312149/2016-6, 140919/2017-1, 141285/2019-2, 131020/2019-6, 104254/2019-0), CAPES/Procad (175956), CAPES/Proex, and FAPERJ (200773/2019, 010002285/2019, PDR-10 202073/2020).

References

[1] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Comput.*, vol. 17, no. 4, pp. 8–17, 2013.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th ICSE*, 2013, pp. 712–721.

[3] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *36th ICSME*. IEEE, 2020, pp. 511–522.

[4] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "The impact of code review on architectural changes," *IEEE Trans. Softw. Eng. (TSE)*, pp. 1–19, 2019.

[5] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *17th MSR*, 2020, pp. 125–136.

[6] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the Qt, VTK, and ITK projects," in *22nd SANER*. IEEE, 2015, pp. 171–180.

[7] L. Pascarella, D. Spadini, F. Palomba, and A. Bacchelli, "On the effect of code review on code smells," in *27th SANER*, 2020.

[8] M. C. O. Silva, M. T. Valente, and R. Terra, "Does technical debt lead to the rejection of pull requests?" in *12th SBSI*. ACM, 2016, pp. 248–254.

[9] E. Fernandes, A. Uchôa, A. C. Bibiano, and A. Garcia, "On the alternatives for composing batch refactoring," in *3rd IWoR*. IEEE, 2019, pp. 9–12.

[10] R. de Mello, A. Uchôa, R. Oliveira, W. Oizumi, J. Souza, K. Mendes, D. Oliveira, B. Fonseca, and A. Garcia, "Do research and practice of code smell identification walk together? a social representations analysis," in *13th ESEM*. IEEE, 2019, pp. 1–6.

[11] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *11th MSR*, 2014, pp. 202–211.

[12] T. Hirao, A. Ihara, Y. Ueda, P. Phannachitta, and K.-i. Matsumoto, "The impact of a low level of agreement among reviewers in a code review process," in *12th OSS*, 2016, pp. 97–110.

[13] C. Barbosa, A. Uchôa, F. Falcao, D. Coutinho, H. Brito, G. Amaral, A. Garcia, B. Fonseca, M. Ribeiro, V. Soares, and L. Sousa, "Revealing the social aspects of design decay: A retrospective study of pull requests," in *34th SBES*, 2020, pp. 364–373.

[14] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw. (JSS)*, vol. 101, pp. 193–220, 2015.

[15] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira *et al.*, "Identifying design problems in the source code: A grounded theory," in *40th ICSE*, 2018, pp. 921–931.

[16] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[17] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study," in *34th SBES*, 2020, pp. 52–61.

[18] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *30th ISSRE*, 2019, pp. 346–357.

[19] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[20] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *20th WCRE*. IEEE, 2013, pp. 122–131.

[21] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? case study on the linux kernel," in *10th MSR*. IEEE, 2013, pp. 101–110.

[22] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *35th ICSE*. IEEE, 2013, pp. 1032–1041.

[23] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng. (TSE)*, vol. 26, no. 7, pp. 653–661, 2000.

[24] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *6th PROMISE*, 2010, pp. 1–9.

[25] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng. (TSE)*, vol. 39, no. 6, pp. 757–773, 2012.

[26] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *7th MSR*. IEEE, 2010, pp. 31–41.

[27] H. S. Yazdi, M. Mirbolouki, P. Pietsch, T. Kehrer, and U. Kelter, "Analysis and prediction of design model evolution using time series," in *26th CAiSE*. Springer, 2014, pp. 1–15.

[28] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, "Are change metrics good predictors for an evolving software product line?" in *7th PROMISE*, 2011, pp. 1–10.

[29] J. Alves Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, "Sampling effect on performance prediction of configurable systems: A case study," in *11th ICPE*, 2020, pp. 277–288.

[30] J. A. Pereira, H. Martin, M. Acher, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning software configuration spaces: A systematic literature review," *arXiv preprint arXiv:1906.03018*, 2019.

[31] Eclipse, https://git.eclipse.org/r/#/c/3345/, 2020, accessed in: August 2020.

[32] ——, https://git.eclipse.org/r/#/c/825/, 2020, accessed in: August 2020.

[33] T. Sharma, P. Mishra, and R. Tiwari, "Designite: a software design quality assessment tool," in *1st BRIDGE*. ACM, 2016, pp. 1–4.

[34] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, "Machine learning: a review of classification and combining techniques," *Artificial Intelligence Review*, vol. 26, no. 3, pp. 159–190, 2006.

[35] T. Menzies and M. Shepperd, ""bad smells" in software analytics papers," *Inf. Softw. Technol. (IST)*, vol. 112, pp. 35–47, 2019.

[36] M. Paixao, J. Krinke, D. Han, and M. Harman, "Crop: Linking code reviews to source code changes," in *15th MSR*, 2018, pp. 46–49.

[37] T. Sharma and D. Spinellis, "A survey on software smells," *J. Syst. Softw. (JSS)*, vol. 138, pp. 158–173, 2018.

[38] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[39] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 ICSME*. IEEE, 2014, pp. 101–110.

[40] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *38th ICSE*, 2016.

[41] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *31st ICSME*. IEEE, 2015, pp. 121–130.

[42] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W. K. G. Assunção, S. Regina Vergilio, J. Alves Pereira, A. Oliveira, and A. Garcia. (2021) Replication package for the paper: "predicting design impactful changes in modern code review: A large-scale empirical study". Accessed: 2021-02-26. [Online]. Available: http://doi.org/10.5281/zenodo.4563214

[43] M. Paixao and P. H. Maia, "Rebasing in code review considered harmful: A large-scale empirical investigation," in *19th SCAM*. IEEE, 2019, pp. 45–55.

[44] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? a taxonomical study of large commits," in *5th MSR*, 2008, pp. 99–108.

[45] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *10th FSE*, 2015, pp. 966–969.

[46] V. Soares, A. Oliveira, P. Farah, A. Bibiano, D. Coutinho, A. Garcia, S. Vergilio, M. Schots, D. Oliveira, and A. Uchôa, "On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns," in *34th SBES*, 2020, pp. 788–797.

[47] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *5th MSR*, 2008, pp. 67–76.

[48] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Emp. Softw. Eng. (ESE)*, vol. 23, no. 6, pp. 3346–3393, 2018.

[49] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," *Emp. Softw. Eng. (ESE)*, vol. 22, no. 2, pp. 768–817, 2017.

[50] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th ICSE*. IEEE, 2013, pp. 392–401.

[51] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng. (TSE)*, vol. 45, no. 12, pp. 1253–1269, 2018.

[52] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *32nd ICML*, 2015.

[53] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[54] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2. Montreal, Canada, 1995, pp. 1137–1145.

[55] T. Fawcett, "An introduction to roc analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, 2006.

[56] E. Whitley and J. Ball, "Statistics review 6: Nonparametric methods," *Critical care*, vol. 6, no. 6, p. 509, 2002.

[57] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[58] J. H. McDonald, *Handbook of biological statistics*. sparky house publishing Baltimore, MD, 2009, vol. 2.

[59] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.

[60] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, vol. 32, no. 200, pp. 675–701, 1937.

[61] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Emp. Softw. Eng. (ESE)*, vol. 21, pp. 1143 – 1191, 2016.

[62] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Trans. Softw. Eng. (TSE)*, vol. 41, no. 5, pp. 462–489, 2015.

[63] H. Lal and G. Pahwa, "Code review analysis of software system using machine learning techniques," in *11th ISCO*, 2017, pp. 8–13.

[64] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[65] M. Caulo, B. Lin, G. Bavota, G. Scanniello, and M. Lanza, "Knowledge transfer in modern code review," in *28th ICPC*, 2020, pp. 230–240.

[66] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, p. 12, 2018.

[67] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Emp. Softw. Eng. (ESE)*, 2020.

[68] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *36th ICSE*, 2014, pp. 345–355.

[69] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43 – 58, 2017.

[70] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *25th SANER*. IEEE, 2018, pp. 612–621.

[71] F. Luiz Caram, B. R. de Oliveira Rodrigues, A. Campanelli, and F. Silva Parreiras, "Machine learning techniques for code smells detection: A systematic mapping study," *Int. J. Softw. Eng. Knowl. Eng*, vol. 29, pp. 285–316, 02 2019.

[72] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115 – 138, 2019.

[73] N. Sae-Lim, S. Hayashi, and M. Saeki, "Toward proactive refactoring: An exploratory study on decaying modules," in *3rd IWoR*. IEEE, 2019, pp. 39–46.

[74] ——, "Can automated impact analysis techniques help predict decaying modules?" in *35th ICSME*, 2019, pp. 541–545.

[75] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Emp. Softw. Eng. (ESE)*, vol. 21, no. 5, pp. 2146–2189, Oct. 2016.

[76] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *31st ICSME*. IEEE, Sep. 2015, pp. 81–90.