This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

Quality Evaluation of Modern Code Reviews Through Intelligent Biometric Program Comprehension

Haytham Hijazi, Joao Duraes, Ricardo Couceiro, João Castelhano, Raul Barbosa, Júlio Medeiros, Miguel Castelo-Branco, Paulo de Carvalho, Henrique Madeira

Abstract— Code review is an essential practice in software engineering to spot code defects in the early stages of software development. Modern code reviews (e.g., acceptance or rejection of pull requests with Git) have become less formal than classic Fagan's inspections, lightweight, and more reliant on individuals (i.e., reviewers). However, reviewers may encounter mentally demanding challenges during the code review, such as code comprehension difficulties or distractions that might affect the code review quality. This work proposes a novel approach that evaluates the quality of code reviews in terms of bug-finding effectiveness and provides the reviewers with a clear message of whether the review should be repeated, indicating the code regions that may not have been well-reviewed. The proposed approach utilizes biometric information collected from the reviewer during the review process using non-intrusive biofeedback devices (e.g., smartwatches). Biometric measures such as Heart Rate Variability (HRV) and task-evoked pupillary response are captured as a surrogate of the cognitive state of the reviewer (e.g., mental workload) and inexpensive desktop eye-trackers compatible with the software development settings. This work uses Artificial Intelligence techniques to predict the cognitive load from the extracted biomarkers and classify each code region according to a set of features. The final evaluation considers various factors such as code complexity, time of the code review, the experience level of the reviewer, and other factors. Our experimental results show the approach could predict the review quality with 87.77%±4.65 accuracy and a Spearman correlation coefficient of 0.85 (p-value < 0.001) between the predicted and the actual review performance. This evaluation validates the cognitive load measurement using electroencephalography (EEG) signals as ground truth for the HRV and pupil signals.

Index Terms— Artificial Intelligence, Biometrics, Code inspections and walkthroughs, Human factors

---

1 INTRODUCTION

Software development is an intensive intellectual task. It consists of knowledge activities related to understanding the problem and designing an adequate solution. Therefore, human cognition plays a crucial role in software development. This high reliance on human cognition is a significant factor in software fragility and proneness to faults. For instance, analysis from a large set of industrial data reported [1] shows that 87% of the severe software defects in deployed code are caused by human cognitive failures, regardless of the software development process.

Software faults or defects (often called bugs) are one of the most persistent challenges of software reliability. Despite the modern tools available for developers and the intensive research on software reliability and quality, the general statistics for software developed (most of them related to software for critical applications) show high bug density figures, ranging from 1 to 5 bugs per 1000 lines of delivered code [2][3][4]. This problem is amplified by the constant pressure to minimize the time-to-market and due to the dramatic increase in code size witnessed by modern software. More lines of code (LoC) mean more bugs, as attested by the fact that the number of LoC is generally used as the most reliable metric to predict bug count in software products [5]. Knowing the high impact of software in our society, software bugs represent the most chronic and challenging problem, which might cause considerable negative consequences on the final product/service resulting from the software.

Among the large arsenal of techniques used to improve software quality, software inspections and, more specifically, code reviews are a well-established practice in software development. Classic code reviews [6][7] consist of the manual inspection of the source code by a group of reviewers to improve the overall quality of the software and detect software defects, among other quality aspects such as compliance to code standards.

Modern code reviews [8][9] are essentially asynchronous (no inspection meetings), based on proprietary tools, and have become more lightweight, more informal, and, consequently, much more dependent on individual skills rather than on the inspection group capacity. At Google [9], "even very large (code) changes on average require fewer than two reviewers."

If the code is reviewed by a single reviewer (or even by two), the quality of the review, and particularly the effectiveness in finding bugs, is highly dependent on the skills of the reviewer, but also on other very human aspects such as engagement level, distractions, fatigue, stress, and attention shifts. These are well-known sources of human cognitive errors [10][11] that may cause reviewers not to find bugs or point out non-existent bugs.

In addition to all these unavoidable reasons for individual reviewer failure in finding bugs, difficulties in understanding the code under review are one of the major issues faced by reviewers [8][12][13]. A very recent study [14] reports that when "reviewing code changes, about 41% of the respondents feel confusion at least half of the time, and only 10% do not feel confusion". If reviewers often have problems understanding the code, the effectiveness of bug findings will be affected.

- *H.Hijazi is with CISUC, University of Coimbra, Coimbra, Portugal, Email: haytham@dei.uc.pt*
- *J. Duraes is with CISUC, University of Coimbra and Polytechnic Institute of Coimbra, Coimbra, Portugal, email: jduraes@isec.pt*
- *R. Couceiro is with CISUC, University of Coimbra, Coimbra, Portugal, Email: rcouceir@dei.uc.pt*
- *J. Castelhano is with ICNAS, University of Coimbra, Coimbra, Portugal, Email: joaocastelhano@uc.pt*
- *R. Barbosa is with CISUC, University of Coimbra, Coimbra, Portugal, Email: rbarbosa@dei.uc.pt*
- *J. Medeiros is with CISUC, University of Coimbra, Coimbra, Portugal, Email: juliomedeiros@dei.uc.pt*
- *M. Branco, ICNAS/CIBIT, University of Coimbra, Coimbra, Portugal Email: mcbranco@fmed.uc.pt*
- *P. Carvalho is with CISUC, University of Coimbra, Coimbra, Portugal, Email: carvalho@dei.uc.pt*
- *H. Madeira is with CISUC, University of Coimbra, Coimbra, Portugal, Email: henrique@dei.uc.pt*

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

We propose the use of biometrics data collected from the reviewer (using non-intrusive sensors) and Artificial Intelligence (AI) techniques to estimate the engagement and how well the reviewer has covered and understood the different regions of the code under review, providing an evaluation of the quality of the review in terms of bug finding effectiveness.

It is well-known that the code review purposes are not limited to bug finding; it also aims at improving code quality, and removing code smells, improving compliance with coding standards, improving team cohesion, training young programmers/reviewers, among other goals. However, bug finding is often considered the top goal of code reviews. For example, in [8] the results of an extensive survey show that "almost all the managers included "finding defects" as one of the reasons for doing code reviews; for 44% of the managers, it is the top reason. Concerning surveyed developers/testers, "finding defects" is the first motivation for code review for 383 of the programmers (44%), second motivation for 204 (23%), and third for 96 (11%)."

A key aspect of modern code reviews is the use of specialized tools to facilitate code review. Established software companies have developed or adopted their tools, embodying the workflow process of their specific flavor of code reviews. Prominent examples of such tools are Critique (Google) [9], CodeFlow (Microsoft) [8], Phabricator (Facebook) [15]. Many other code review tools are currently available (see a survey of the "12 Best Code Review Tools for Developers - 2021 Edition" in [16][15]).

Notably, most of these code review tools also try to assess the quality of the code review through a process or product-oriented metrics such as comments quality, quality of the feedback provided, code (patch) quality, inspection rate, and LoC covered, among others [17]. Still, all of them fail in a crucial aspect: they do not evaluate the quality of the code review work provided by individual reviewers in terms of bug-finding effectiveness. Filling this gap is precisely the goal of the proposed approach.

This paper proposes a new approach to systematically evaluate the quality of modern code reviews in terms of bug-finding effectiveness at the code region level of granularity (a region is a small set of code lines, typically around ten LoC). This evaluation provides immediate feedback to code reviewers or developers on possible code regions/lines that were not well covered by the review. It also provides the reasons why such a review may have left bugs undetected. The reviewers could use this information to promptly improve the code review through a second pass over specific parts of the code under review, or project managers can ask for a second independent review.

The proposed approach uses well-established biometric techniques that use psychophysiological measures such as heart rate variability (HRV) and the pupillary response (pupillometry) to assess cognitive load while executing specific tasks and inherently infer the difficulty and mental effort associated with such tasks [18][19][20]. We also use the domain knowledge to extract useful features from the psychophysiological measures (feature engineering). Then, we adapted a multimodality augmented set of features, including other non-biometric features such as the code complexity, the reviewer experience, the number of revisits to a code region, and the code reading time. We use knowledge domain and data-driven algorithms (i.e., Kruskal-Wallis and Relief features selection) to select the best multimodal features. The best-chosen features are fed into k-nearest neighbors (KNN) and Logistic Regression classifiers (LR) — known for their explainability— to classify each code region as being either well-reviewed or not, with an explanation on why the code region was not well-reviewed.

We have evaluated the proposed approach through a controlled experiment including 21 code reviewers equipped with biometric sensors while performing a code review of four code snippets. The paper makes the **following contributions**:

- Proposes a new approach to evaluate the quality of code reviews individually (i.e., code reviewer), leveraging biometric measures and AI techniques to improve bug-finding effectiveness in the code review evaluation process.
- Evaluates the accuracy, precision, and recall of the proposed approach in a controlled experiment that uses code reviewers, actual code, realistic bugs, instrumentation to gather the biometric signals used to assess the reviewer's cognitive load.
- Validates the reviewer's cognitive load (assessed through non-intrusive devices) using Electroencephalography (EEG).
- Analyses the cognitive load of code reviewers at a fine-granularity (i.e., code region level), which is beyond the state-of-the-art in biometrics applied to task-level code analysis.

The paper is organized as follows: Section 2 presents relevant interdisciplinary background and covers the state-of-the-art of the pertinent topics. Section 3 details the proposed approach. Section 4 addresses the experimental setup to evaluate the proposal. Section 5 shows the analysis and classification of Biometrics. Section 6 discusses the results and the viability of the approach. Section 7 discusses the threats to the validity, and Section 8 summarizes the takeaway messages and outlines future work.

2 Background and Literature Review

Evaluating the code review process using physiological signals and pursuing a data-driven approach spans software engineering and cognitive & neuroscience, and AI. This section overviews the relevant topics and summarizes the state-of-the-art related to the proposed approach. For details on physiological features and their relationship with cognition, please see Supplement 1.

2.1 Code Reviews

The asynchronous style of code reviews emerged in the early 2000s, mainly in the context of OSS projects. In these approaches, reviewers can see the code and the code changes and can discuss specific lines of the code while the author of the code addresses the reviewer's comments. This flavor of light and asynchronous reviews has been brought by various tools that have emerged to help authors of patches to submit them for review before being integrated (i.e., merged) into the shared software.

As mentioned before, the modern style of code reviews has also been shaped by well-established companies that have developed their tools and review environments [8][9][15]. With such tools, the developer completes a change in the code and creates a review request, including a description of the change and specifying the candidate reviewer (or reviewers) that will receive such a request. Reviewers get notified via email to open the tool and review those changes. Additionally, reviewers can annotate these code lines containing the change with their comments.

Many commercial code review tools are currently available (see [16]). Most rely on the Git distributed version control system or even on cloud-based hosting services such as GitHub. These tools adopted the pull-based development model, and when developers need to make a change or add new code, they fork an existing Git repository and make those changes in their fork. The review tools embed the code review process in the pull request

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

workflow to facilitate the dialog between reviewers and developers and ensure that the code review is performed as expected. Ford et al. [21] show that deciding upon pull request (acceptance or rejection) depends on the social and technical aspects of the pull request initiator. In other words, to accept the review, you should look at the reviewer's (i.e., request initiator) profile and expertise. This result demonstrates to which extent the code review process is becoming more personal dependent. It means that helping reviewers with the proposed approach, which calls their attention to code that needs a second look, seems quite useful.

In practice, software companies adopt variants of code reviews. However, there are fundamental characteristics to achieve high-quality reviews. For instance, in classic reviews, Ackerman et al. [22] state that effective code reviews should be carried out by knowledgeable peers whose primary purpose is to find defects in the software product. Some key metrics mentioned [22] as essential for the effectiveness of the code review process include the average preparation effort per code unit; the average examination rate and effort per code unit; the average explanation rate per code unit; the average number of defects detected per code unit; the average number of defects per unit of code. These metrics point to reviewing best practices but fulfilling such best practices at the individual reviewer level is not easy to guarantee.

Best practices in modern code review [23] advise reviewers to ask themselves several questions before starting the review: do I understand the code? Does this code function as I expect? Does this code comply with regulatory requirements? It is also recommended not to review longer than 60 minutes and no more than 400 LoC at a time.

As we see from the examples above, code reviews rely on the individual capacities and skills of reviewers, knowing that in modern code reviews this dependency on the reviewers' skills is higher since only one peer typically reviews the code. The approach proposed in this paper addresses this dependency on the individual reviewer's skills, as well as on other human factors such as fatigue, stress, distractions, difficulties in code comprehension, to improve bug finding effectiveness through the evaluation of the code review and prompt feedback to the reviewer.

An important aspect of attesting to the relevance of the problem targeted by our proposal is to examine available studies on the effectiveness of code reviews concerning uncovered bugs. Shull et al. [87] surveyed the bug detection coverage of classic code reviews observed in eight studies and concluded that it is reasonable to expect code reviews to detect an average of 60% of the bugs. However, among the studies surveyed in [87], we can observe that the effectiveness of code reviews in bug detection can range from 19% up to 93%, depending on many factors.

Since modern code reviews are more informal than classic code reviews and do not rely on a group of inspectors and an inspection meeting, it is expected that the effectiveness of modern code reviews in finding bugs is lower than the average 60% reported by the analysis of the eight papers surveyed in [87]. An empirical study investigating modern code review quality for Mozilla found an overall of 54% of code reviews missed bugs in the approved commits. Interestingly, similar values were found consistently throughout the different modules analyzed in [24]. Concerning the type of defects detected by code reviews, an empirical study [25] classified the defects of nine industrial (C/C++) and 23 students (Java) code reviews and found that from 71% to 77% of the defects found are related to software evolvability aspects (code organization, solution approach, code formatting such as brackets usages and indentation, etc.) and do not affect code functionality (in any case, they are defects).

Code review effectiveness in bug detection of 60% (in average) for classic code reviews (but with a wide bug detection range from 19% up to 93%) **Error! Reference source not found.** and an average of 54% of bug detection for modern code reviews [24] show that there is a large room for improvement.

Several works suggest that personal (i.e., individual dependent) aspects play an important role in review quality, which concurs with our proposal of assessing individual reviewers' engagement and cognitive load using non-intrusive biometrics to help predict bug finding effectiveness. Baysal et al. [26] showed that non-technical factors such as "personal" dimensions affect code review time and the outcome of the review process. Kemerer et al. [27] indicated that defect detection and removal effectiveness depend on the individual review rate. Shull et al. [28] argued that inspections led by reading techniques are more effective at revealing defects. Hatton [29] showed that reviewers tended to show different defect detection capabilities and stated that the worst reviewer was ten times less effective than the best one. This result is consistent with the results of the eight studies surveyed (see section 3.6 of [87]). Kononenko et al. [17] also studied the characteristics of the good review, as perceived by developers, and found that personal metrics such as reviewer workload and experience play a relevant role in the code review quality. Notably, the authors of [17] also showed that human factors such as reviewer mood, personality, experience level, skills, productivity, and stress level are the most significant determining factors in the code review quality. Personal factors such as the reviewers' experience and workloads are also suggested as promising predictors of the code review quality in [24].

Al-Saiyd [30] took a further step in those human aspects and considered source code comprehension an essential part of the software maintenance process, including code reviews. The author [30] showed that code comprehension efforts highly rely on the reviewer's skills and experience and other factors, including the programming language and the code size/structure. In the same context, Huang et al. [31] argued that reviewers must spend a significant amount of time understanding the code during the code review process. Difficulties in understanding the code under review as a result of reviewers' confusion (i.e., reviewers do not understand something in the software) are pointed as a major factor in [14]. These works support our idea of associating review quality with the effectiveness of the code comprehension by the reviewer, with the significant difference that in our proposal, we infer code comprehension using biometrics features.

Although abundant studies have been conducted to assess developers' code comprehension (e.g., [32][33]), there is a lack of empirical studies that objectively assess the reviewer code comprehension and associate it with the review quality in terms of bug detection. Psychological and observational works on software developers and reviewers rely on indirect techniques to assess programmers' code comprehension, such as comparing task performance, surveys or articulating developers' thoughts in think-aloud protocols [30][31]. These techniques use self-reporting methods and require considerable efforts in transcription and data analysis, which might be inconsistent in the end.

2.2 Biometrics and Cognitive Neuroscience in SE

Recently, academia started to leverage cognitive neuroscience and AI in software development to understand the underlying cognitive mechanisms of human intelligence tasks in software

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

development. In a recent comprehensive survey [34], the authors proposed the term **NeuroSE** to "describe a research field in software engineering (SE) that makes use of neurophysiological methods and knowledge to understand software development better." This survey shows an exponential growth of the number of publications in NeuroSE since 2014, reaching 89 papers by mid-2020 (the period reported in [34]). Most of these studies report-controlled experiments often using heavy medical imaging equipment such as magnetic resonance imaging (fMRI) (see [34] for a comprehensive review) that obviously cannot be used in real software development settings. But these studies are essential to understand basic cognitive mechanisms related to human error and bug making/discovering in software code development and constitutes a relevant ground for the proposal of innovative software engineering approaches that move NeuroSE from controlled experiments into real-world software development setups, such as the approach proposed in this paper.

One of the first fMRI studies on programmers' mental effort in comprehending code is presented in [35]. In [36], the authors characterize the brain mechanisms involved in understanding natural language texts and source code, comparing the brain mechanisms involved in each case. A study identifying specific brain regions involved in code comprehension and syntax error identification, specifically the regions of language processing, working memory, and attention, is reported in [37].

The study of reviewers' brain activity during code inspections is addressed explicitly by our previous works in [38][39][40]. In particular, [39] suggests that the activity levels of the insula region of the brain are directly related to the quality of the bug detection, establishing a direct relationship of a brain signal with a code reviewing skill, and opening the possibility of using the activity of that brain structure as a predictor of accuracy of bug finding tasks. These findings have been confirmed and expanded in a second and more comprehensive fMRI study [40]

Wearable devices such as wrist-located sensors, bracelets, and smartwatches represent a fast-expanding industry of biometric devices that are primarily used in sports, fitness, and well-being applications but can also be used as base sensors to assess reviewer's cognitive load and engagement in reviewing tasks, as proposed in this paper.

Although the use of EEG, HRV, and pupillometry as sources of cognitive information from programmers and reviewers is relatively recent in the research literature, eye-tracking has been highly researched in the context of software development. More specifically, in the context of code review, Uwano et al. [41] used eye-tracking to characterize the performance of reviewers by analyzing their fixation data and eye movements while performing the review. The results indicated that reviewers who do not spend enough time scanning the code would likely spend more time in defect detection. Sharif et al. [42] replicated and expanded Uwano et al.'s work [41] to investigate how individuals find defects in source code by characterizing eye movements. The authors showed that the scanning time extracted from an eye-tracker is significantly correlated with defect detection time and visual effort on lines with defects.

Chandrika et al. [43] also used eye-tracking to understand the eye gaze behavior required for code review by both skilled and unskilled programmers. It was found that skilled subjects (i.e., better reviewers) tend to have eye-tracking traits such as better code coverage and attention span to error lines and comments. A recent work [44] conducted a study on 35 software engineers performing 40 code reviews while measuring their eye gaze. The authors could distinguish between time spent skimming the code vs. carefully reading it relying on eye-tracking. According to [45], careful reading is defined as "two standard deviations lower than the mean rate per person."

Although to the best of our knowledge, there are no previous works in the literature proposing the use of biometrics to evaluate code reviews in terms of bug-finding effectiveness, as proposed in our paper, several previous works established the feasibility of key aspects of our proposal. Vrzakova et al. [46] address the affect recognition of the code reviewers using physiological signals and mechanical signals (i.e., typing), showing that it is possible to unveil the reviewer state (valence and arousal). Several works have shown that it is possible to differentiate among several cognitive states such as stress and cognitive load [47][48]. Detecting cognitive distractions (a cognitive state strongly related to human errors) has been an essential goal for the automotive industry, and eye-tracking sensors have proven effective [49]. Works from Fritz, Muller, et al. show that it is possible to assess task difficulty in software development [50] and to optimize software testing through the prediction of code quality using programmers' cognitive load captured using HRV [51].

Our early works combine HRV, Pupillography, and eye-tracking to annotate code lines with the programmer's cognitive states collected while attempting to comprehend such code lines [52][53]. In [52], it was possible to conclude that the changes in cognitive load were mainly associated with the appearance of outliers above the mean value. In [53], we concluded that one of the features that is possible to extract (Low_freq/High_freq) showed high variations during small sections of code (similar in size to the code regions in the present paper) where the high cognitive load was expected. Once again, to capture this high variability (outliers), transforms need to be used.

Since we use data-driven approaches to classify the code review quality, it is relevant to examine previous works that used biometrics and data-driven approaches to assess programmers' cognitive load and code comprehension. In [51], a Random Forest classifier was used with HRV and EDA features to predict code quality. Floyd et al. [36] employed a Binary Gaussian Process classifier with fMRI and show that tasks involving programming languages and natural languages activate different brain areas, showing that it is possible to classify which task a participant is undertaking (code comprehension, code review and prose review) based on brain activity. Fucci et al. [32] replicated the study presented in [36] using lightweight EEG and EDA devices

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

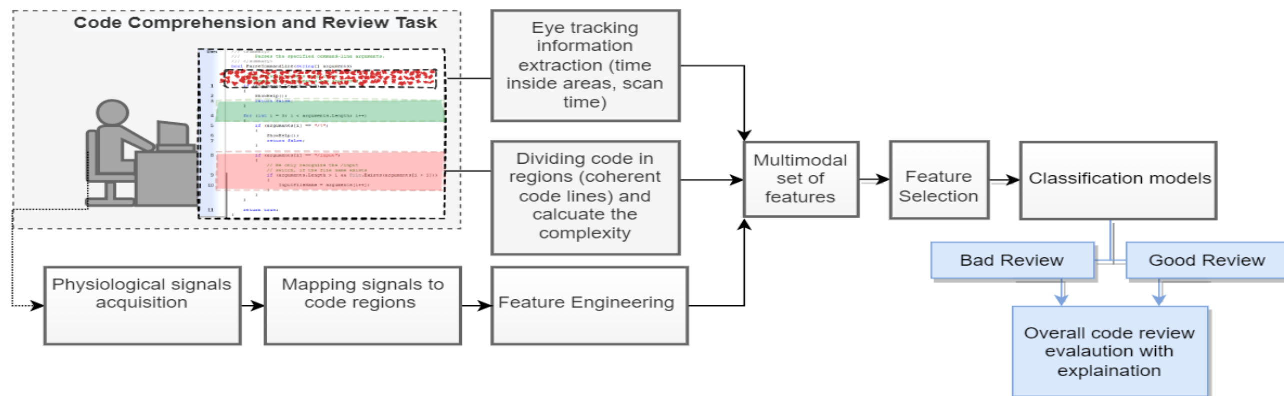

Fig. 1. Schematic Diagram of the Approach

and evaluated a comprehensive set of data-driven classifiers including Naive Bayes, K-Nearest Neighbors, Decision Tree, SVM, Neural Networks, Rule-based, Random Forest, and Boosting and showed that it is possible to distinguish between the different tasks (code versus natural language comprehension) with this much lighter setup. Lee et al. [54] used EEG and eye-tracking and a Support Vector Machine classifier to predict programmer expertise (novice/expert) and task difficulty (easy/difficult).

It is worth mentioning the exploratory nature of these previous works that used data-driven classifiers and the fact that most of them used high intrusive sensors such as EEG and fMRI that cannot be applied in real software development setups. The work presented in [51] is an interesting exception, as it predicts the quality of code using low intrusive sensors and random forest classifiers.

In the proposed approach, presented in the next section, we use both domain knowledge and data-driven feature selection methods are applied. We are particularly interested in using explainable classification models like the KNN and the LR. The KNN is interestingly explainable, whereas the LR is relatively explainable through the odd ratio changes relative to features.

3 The Proposed Approach

Figure 1 shows the key elements of the proposed approach. First, the code under review is divided into non-overlapping code regions to apply the code review quality assessment to relatively small code snippets that we call **code regions** instead of applying the technique to the entire code under review. If at least one code region is evaluated as not being well-reviewed that will be enough to classify the entire code review as not satisfactory and recommend a second look (i.e., a second review).

The definition of code regions in the code under review includes the following goals:

- **Resolution of the technique:** code regions define the basic grain of resolution of the technique to allow the precise identification of specific code lines that have not been well-reviewed for bug finding.
- **Non-overlapping and syntactically consistent blocks of code**: code regions do not split syntactic code constructs, allowing an automatic process (using formal grammar) to define regions in the code under review.
- **Human scale in terms of readability:** code regions represent code lines with a suitable size and complexity for a human reader. For example, the size of code regions must be limited to a given cap (in the evaluation of the technique presented in the paper, code regions have an average size of 8.9 lines of code and a maximum of 14 code lines).

This division of the code under review into regions is achieved using a parser to construct an abstract syntax tree during syntactic analysis and split it into sub-trees at the top-level constructs of the language. It is worth mentioning that the parser does not split syntactic code constructs. In other words, it does not break a block of statements inside a while/for loop and does not break apart the if portion from the else part of an if-else construct (i.e., the beginning and end of a given region are at the same nesting level).

Once the code is organized in regions, the complexity metrics of each region (Vg and LoC) are calculated using complexity metric tools to classify each region according to the complexity of the code snippet. The assessment of how well the reviewer understands each code region uses rule-based and explainable classifiers. For example, if the code lines the reviewer is looking at (obtained from the eye-tracker) are complex, and the cognitive load of the reviewer (data provided by biometrics) is low, this suggests that the reviewer is just skimming through the code snippet and is not making a real effort to understand such complex code. Or, as another example, if the complexity of the code region is low and the cognitive load of the reviewer while looking at those code lines is high, it suggests that the reviewer is having difficulties in comprehending such code or is distracted or mentally busy with something else. In both cases, the chances of overlooking bugs in the review of such code regions are high. It is worth mentioning that field studies show that bugs in deployed software may reside in low complexity code as much as in high complex code [14] [55], meaning that even simple code must be reviewed with the same care of complex code.

The eye-tracker is used to know when the reviewer looks at each code region and calculate region reading time and number of revisits. The reviewers' cognitive load is assessed from the physiologic signals collected by biometric sensors (i.e., ECG and Pupillography), which are processed for mapping the biomarkers with the corresponding code regions.

Feature engineering techniques are used to mine the raw data for meaningful features according to the knowledge in the domain. The features include the code complexity, the reviewer experience level, the code region reading time, the code region revisits, and the cognitive load indexed by the HRV and the Pupil features. The selected features are used to classify the code regions according to the code review quality, whether good or bad (see the middle-right part of Figure 1). The overall quality of review is then provided with the indication of the code regions that

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

should be better reviewed and an explanation achieved using explainable AI.

Our approach uses AI techniques to assess the quality and coverage of the code review by estimating how well the reviewer has comprehended different code snippets of the software under review. The AI role begins with the feature engineering of biomarkers extracted from the biosensors, classification, and decision optimization. The features are selected in a knowledge-driven and data-driven approach using Kruskal-Wallis (KW) based feature selection and Relief feature selection techniques [56][57]. KW is a computationally efficient non-parametric feature selection technique. It tests if two classes have an equal median and gives the value of P (low P values are selected for their discriminative power). On the other hand, the relief method is sensitive to feature interaction where it assigns weights to features based on finding k-nearest neighbors for the same class features and k-nearest misses for the other class. The classification is performed using an explainable conditional rule-based classifier and a simple explainable classification model based on the KNN and the Logistic Regression.

Concerning acquiring the heart signals for the HRV assessment, our proposal for developing concrete tools for real code reviewing scenarios is to use non-intrusive and wearable sensors that can be fully integrated with the software development environment. These sensors use wrist located Photoplethysmography (PPG) [58], typically done with bracelets/watches, which allow the assessment of reviewers' cognitive load using Pulse Rate Variability (PRV). PRV is like HRV, with the difference that it is based on the heart signals inferred through PPG instead of using a direct electric signal (from ECG sensors) such as in HRV.

Although we envisage the use of bracelets or watches with PPG for future concrete applications of the approach, in the evaluation of the approach presented in this paper, we used ECG sensors for the acquisition of the heart signals and to perform HRV assessment of the cognitive load. Since this paper is mainly focused on the validation of the concept (and not on the evaluation of a concrete tool implementing the proposed approach), we decided to use ECG sensors to ensure that we evaluate the idea using clean electric heart signals, removing the need of extra care in the experiments to ensure correct and consistent positioning of PPG watches in the wrists of the reviewers.

The feasibility of using PPG instead of ECG sensors was empirically shown by Pinheiro et al. [59]. They concluded that PRV could be a good alternative for HRV with significant correlations above 82% for time and frequency domain features. The results in [59] have also been attested by a recent and detailed study presented in [60]. The small delay (below 50 milliseconds [60]) introduced by PPG is not relevant for the proposed application.

The task-evoked pupillary response (pupillography) [20][61]is available in most eye-tracking systems, which are now more compatible with programming settings. The third type of wearable sensor that can also be used to measure electrodermal activity (EDA), also known as galvanic skin response, is known for its capacity of discriminating stress from cognitive load [47][48]. It is worth mentioning that in the experimental evaluation done in the current study, we have not used EDA (but, obviously, the use of EDA as an additional source to assess the reviewer's cognitive state is totally in line with our proposal).

The final key element of the proposed approach (see the box at the right-hand side of Figure 1) is the rule-based and data-driven classification model that receives the best-selected features from HRV, Pupillography, the complexity of the code region, time of reading the code region, and the number of code region revisits. The classifier then predicts the quality of the review of each code region to provide the indication of whether the reviewer should review the code of a given region again or not.

The design of the rule-based classification model considered established best practices of code reviews as described/proposed in [22][23] [26][27][28][29][41][44], as well as the expert opinion of the co-authors of the paper that have considerable experience in code reviews. Both best practices and expert opinion align with the current proposal's fundamental idea that relies on perceiving the reviewer's code comprehension as the crucial indicator of the effectiveness of a review. Table I summarizes the five criteria used to set up our rules, supported by a brief description of each one, and links each criterion with the code review

Table I CRITERIA USED TO EVALUATE THE CODE REVIEW QUALITY IN EACH CODE REGION

CRITERIA	DESCRIPTION	EFFECTS ON CODE REVIEW QUALITY
Reviewer's Cognitive Load	The main goal of assessing cognitive load (CL) is to infer to what extent the reviewer comprehended the code under review. CL is derived from the HRV (or PRV) and Pupillography features.	Extensive research has established a link between cognitive load assessment and code comprehension level [62]. Comprehension is a critical factor in an effective code review process. Bacchelli and Bird concluded that understanding is the main challenge when doing code reviews [8]. Dunsmore, Roper, and Wood supported the positive correlation between code comprehension and review effectiveness and quality [63] empirically.
The complexity of the Code Region	Based on the cyclomatic complexity metric (Vg) and lines of code (LoC)	The impact of code complexity on the difficulties perceived by reviewers in comprehending the code and detecting all bugs is not well established. However, Muller and Fritz in [51] used the code complexity metric and other features to predict code quality concerns. Nonetheless, our studies [52][64] show that code complexity does not always correlate with the subjective developers' perception of difficulty in comprehending code.
Reviewer's Expertise	The expertise here is in the context of programming skills. A written C exam was performed to distinguish between standards and experts.	Sauer et al. [65] identified individuals' expertise as a primary key to improving code review effectiveness. Other examples [66][67] showed a positive correlation between code review expertise and the number of defects found in the software, hence the code review quality.
No of Revisits to the Code Region	The number of revisits refers to the number of times the reviewer regresses to a specific code region. The number of revisits is extracted from the eye-tracker.	Many studies linked the quality of reading the code with revisits or regression (e.g., [33][68]) In [68] Busjahn et al. show that difficult texts induce more frequent regressions. Usually, complex code takes more revisits from expert reviewers, which is interpreted as good review practice. However, this is not always the case. Good code readers are characterized by few revisits and short fixations in simple code review tasks.
Time Spent Reviewing the Code Region (Reviewing Time)	The sum of times spent in all (re)visits of the reviewer in a code region, including reading, comprehending, and analyzing the code to find bugs (extracted from the eye-tracker).	Uwano et al. [41] show that the scan pattern reflects the cognitive action in code review. They show that the quality of the scan should significantly influence the individual efficiency of detecting bugs in the review. Authors show that the duration of scan time on specific code lines may indicate the strength of the reviewer and thus a high-quality review.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

quality established in the literature.

In this approach, we use two classification models: the Knowledge-Driven rule-based Classification Model (KCM) and the Data-driven rule-based Classifier Model (DCM). The KCM has predefined rules defined by experts, as shown below in Table II, while the DCM is based on the data-driven machine learning classifiers, namely the KNN and the Logistic Regression.

Table II shows the rules used in the evaluation of this approach. We understand that there is a wide range of possibilities for the definition of the set of rules and, very important, for the definition of the thresholds that determine the different categories of each criterion (e.g., reviewer's cognitive load high or low, code region simple or complex, reviewer expert or standard, etc.). Naturally, we know that many other alternatives do exist, particularly the ones that consider more categories for each criterion instead of just a simple Boolean approach. Furthermore, we can also consider additional criteria such as the reviewer's familiarity with the code, in addition to the five criteria shown in Table II. All these possibilities should be analyzed and evaluated. However, to keep the size of this paper within acceptable limits, we only propose the 32 rules using a Boolean approach for the different criteria.

TABLE II. CODE REVIEW EVALUATION RULES

Rule No.	Cognitive Load	Region Complexity	Reviewer Expertise	No. Revisits	Reading Time	Quality of Review
1	High	High	High	High	High	**Good**
2	High	High	High	High	Low	**Bad**
3	High	High	High	Low	High	**Good**
4	High	High	High	Low	Low	**Bad**
5	High	High	Low	High	High	**Good**
6	High	High	Low	High	Low	**Bad**
7	High	High	Low	Low	High	**Bad**
8	High	High	Low	Low	Low	**Bad**
9	High	Low	High	High	High	**Good**
10	High	Low	High	High	Low	**Good**
11	High	Low	High	Low	High	**Bad**
12	High	Low	High	Low	Low	**Good**
13	High	Low	Low	High	High	**Bad**
14	High	Low	Low	High	Low	**Bad**
15	High	Low	Low	Low	High	**Good**
16	High	Low	Low	Low	Low	**Bad**
17	Low	High	High	High	High	**Good**
18	Low	High	High	High	Low	**Bad**
19	Low	High	High	Low	High	**Good**
20	Low	High	High	Low	Low	**Bad**
21	Low	High	Low	High	High	**Bad**
22	Low	High	Low	High	Low	**Bad**
23	Low	High	Low	Low	High	**Bad**
24	Low	High	Low	Low	Low	**Bad**
25	Low	Low	High	High	High	**Good**
26	Low	Low	High	High	Low	**Good**
27	Low	Low	High	Low	High	**Good**
28	Low	Low	High	Low	Low	**Bad**
29	Low	Low	Low	High	High	**Good**
30	Low	Low	Low	High	Low	**Bad**
31	Low	Low	Low	Low	High	**Good**
32	Low	Low	Low	Low	Low	**Bad**

Most of the rules are relatively intuitive and easy to understand even by non-experts. For example, suppose the time spent reviewing a code region is below a given threshold (see Section 5.3 for the choice of threshold values used in the evaluation), then the review is most likely classified as a bad review. It is considered that the reviewer did not understand the code (it would be impossible in such a short amount of time), especially if the reviewer is not an expert. Or, as another example, if the code is complex, the reviewer's cognitive load is low, and the reviewer is not an expert, the quality of the review for that code region is classified as low, no matter the number of revisits or the reviewing time. The rationale of this conclusion is that a low cognitive load in non-expert reviewers indicates they are not trying to understand the complex code of such a region.

The reasoning supporting some of the rules is not so trivial, such as the number of revisits to the code region that follows some observations provided in [69]. For example, depending on the expertise of the reviewer, the complexity of the code, and the reviewer's cognitive load, a high number of revisits may indicate a careful review (e.g., when the code is complex, the cognitive load is high, and the reviewer is expert) or may show that the reviewer is insecure and does not understand the code well (e.g., when the reviewer is a non-expert, the cognitive load is high, and the code is complex).

The rules validation was performed by a panel of developers and code reviews experts in the following steps:

- List all the possible combinations of features contributing to the code review quality (cognitive load, experience, reviewing time, revisits, and code complexity).
- Revisit the combinations independently based on their code review experience and best practices in code review reports.
- The experts discussed each decision taking the average of their choices in the final form of the rules.
- Coverage testing: this test aims to explore each rule's frequency of occurrence in the dataset.
- The rule results were compared with the actual performance of the reviewers for further validation.

In any case, these rules represent expert opinion on the most plausible outcome (i.e., quality of the review) for the different combinations of the five criteria presented in Table I and used in Table II. In some cases, the impact of each criterion on the rules is quite debatable. Take, for example, the criterion "No. of revisits to the code region". If the number of revisits is high, it could mean hesitation of the reviewer or just the opposite, as the reviewer may be thoroughly confirming the content of the code region. As described in Table II (line 5), this individual criterion must be interpreted together with the expertise of the reviewer, the complexity of the code, and even with the cognitive load while analyzing the code region.

The second model used in this paper is the data-driven rule-based classification. In this model, we experimented with looking at the approach from a different angle. We provide the classifiers with the best-selected features using Relief and Kruskal-Wallis-based feature selection techniques, train the model, and predict the code review performance at the code-region level.

4 EXPERIMENTAL SETUP AND DESIGN

This section describes the controlled experiment designed to evaluate the proposed approach's accuracy. All the relevant data related (with the information related to individual participants fully anonymized) is available in this GitHub link[1] as supplementary material for this paper.

4.1 General Experimental Setup and Participants

The experiment was designed to monitor the cognitive load

[1] https://github.com/HaythamHijazi/Supplement

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

of reviewers using HRV and pupil signals while performing code reviews of different complexity. The Ethical Committee of the Faculty of Medicine of the University of Coimbra approved the study, following the Declaration of Helsinki and the standard procedures for studies involving human subjects. The subjects provided written informed consent, and all the data was anonymized. ECG was collected using BiosignalsPlux from Plux, and the pupil diameter was collected using an SMI eye tracker. Participants also wore an EEG cap (with 64 channels) for validation purposes to confirm that reviewers' mental effort was measured accurately. The left side of Figure 2 shows the EEG cap with 64 channels, and the right side shows the ECG sensors attached to the subject to evaluate the proposed approach.

The subjects consisted of 21 male programmers/reviewers, ranging from 19 to 40 years, with an average of 22 years. These programmers participated on a volunteer basis and were selected from a pool of 49 candidates through an interview-based screening process focused on assessing their C programming skills. Three volunteers were professionals from the software industry (real code reviewers), and they have worked on code reviews during their careers. The other volunteers were Ph.D. and MSc students (from computer science fields) who have sufficient programming and code review knowledge. Despite this mixture of professionals and students, the goal of the screening was to avoid selecting programming beginners. Additionally, we used a written C programming test **(see supplementary material[1])** to evaluate the programming skills of participants and filter out volunteers classified as "beginner" and would not realistically represent professionals in the industry (i.e., standards and experts on C programming). The 21 participants were classified into two skill levels: Standard: 16 participants and Expert 5 Participants. Participants who scored more than 7 out of 10 in the C test were considered Experts, while those who scored 4 to 7 out of 10 were considered Standard Reviewers. Participants that scored less than 4 (novices) were not considered.

Fig. 2. Experimental Setup EEG and ECG sensors

The Review tasks were presented to participants using Vizard software[2], assuring the same conditions to all participants. For each participant (reviewer), the experiment is composed of 4 consecutive runs, having a review task per run (one of the four programs is selected at random). Each run starts with a fixation cross shown in the middle of the screen for 30 seconds (i.e., a minimum stimulus in all runs) to get a baseline cognitive activity. Then three tasks are presented to the participant: natural language reading (a literary excerpt), a neutral and straightforward (bug-free) code reading, and one code review task. To avoid skewing the experiment results, within each run, the order of the tasks was randomized. The order of the four code reviewing tasks in each run was also randomized.

The natural language reading and neutral code tasks served to gauge the neutral cognitive load of each participant to calibrate and compare to the biometric readings when that participant was engaged in code reviews. Subjects were explicitly told about the goals of each type of task, and the neutral code tasks were identified as having no bugs. Subjects were asked just to read the code. The code review tasks were also explicitly marked as code that may or may not have bugs, and no hint was given about the bugs themselves.

Each review task was presented to the subject as a set of screens containing the code. The first screens showed a brief description of the code's goals and algorithm being reviewed, which is coherent with the actual code review tasks in the industry, where reviewers are given as much information as possible about the code. The subjects were free to move from one screen to another at any time. For visibility reasons, each screen contained at most 20 lines of code, and the lines were numbered. The subjects analyze the code, and if they suspect that a given line includes a bug, they will mark that line, which indicates a suspicion of a bug. Subjects could additionally activate a button "bug" further to confirm their suspicion about that line of code. The bug suspicion could be canceled at any time by marking the line once again and then activating a button "clear. "All controls available to the subject for line selection (e.g., buttons, etc.) were virtual, i.e., buttons drawn on the screen. The only physical device used in the experiments was a joystick used as a pointing device.

The review of the four programs (122 code lines) for each participant took about 1 hour, which is typically the maximum time recommended for industry code reviews. The participants and the four programs reviewed in the controlled experiment generated 84 code reviews (24 reviews were discarded due to missing one or more measurements from HRV and Pupil readings resulting from setup problems that affected the acquisition of data in some runs and were only detected afterward). Considering that each program under review included one or more code regions, the remaining 60 reviews generated a total of 149 code regions reviews, which is a reasonable number of samples to show the approach's feasibility.

One of the limitations in this stage is having all subjects as males. Unfortunately, in the call for voluntaries for this experiment, we could not recruit any female code reviewer, which is a consequence of the strong gender imbalance in the software development area, especially in tasks such as code programing and reviewing.

4.2 Code Review Tasks

The programs used in the code review tasks include both iterative and recursive programs. One simple and one medium/high complexity for each category to avoid skewing the experiment results based on code paradigm or size (shown in Table III). The code size was limited by the amount of time it was feasible to maintain the subject in the experiment before the accumulated tiredness started to influence the results. The programs were the following:

1. Bucket Sort ("bsort") implements a sorting algorithm. It is medium-size, iterative, and complex (Vg = 10). This is an

[2] https://www.worldviz.com/vizard-virtual-reality-software

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

example of data processing that one encounters when a library for data manipulation is unavailable.

2. Fibonacci ("fibo") is the implementation of the algorithm that generates the Fibonacci sequence. It is small, purely recursive, and very simple (Vg = 2). Recursive algorithms are harder to find in the industry. However, many industry-relevant languages rely on recursive algorithms, such as Haskell and Elixir.
3. Hondt ("hondt") is implementing the Hondt algorithm for seat allocation after an election. This task is iterative, small-sized, and medium complex (Vg = 5).
4. Matrix Determinant ("matdet") implements the recursive algorithm that computes the determinant of square matrices. It is medium-sized, mostly recursive (Vg = 10). The matrix determinant is an example of complex nested code controlled by many variables, including recursive logic.

The following table shows different features of the given programs:

TABLE III. CODE REVIEW TASKS

Task	Type	LoC	Complexity	No. of bugs
bsort	iterative	42	Vg = 10	4
fibo	recursive	9	Vg = 2	1
hondt	iterative	32	Vg = 5	4
matdet	recursive	39	Vg = 10	4

Realistic types of bugs that can be found in deployed software-based on findings of previous works (e.g., [70][71]) cover the most frequent ODC defect types [70][72], both Missing and Wrong cases. The bugs inserted do not cause syntactic errors and are not related to obscure aspects of language or libraries. Instead, they represent realistic software defects. The size of the programs limits the number of realistic bugs that can be inserted, which was one bug for Fibonacci and four bugs for each of the remaining programs.

4.3 Code Regions Defined in Each Program

As mentioned, the first step of the proposed approach is to (logically) divide the program under review into non-overlapping code regions. Every region is a coherent sequence of code lines. Code regions represent the reviewing unit evaluated by the proposed approach. We defined the regions of each program using a parser, and the complexity metrics were calculated using the Eclipse Matriculator plugin. Table IV shows the key features of the regions considered in the four programs under review.

TABLE IV. CODE REGIONS

Program	Region	Loc	Vg	No of bugs
bsort	Region 1	7	4	2
bsort	Region 2	8	2	0
bsort	Region 3	11	5	1
bsort	Region 4	11	3	1
fibo	Region 5	8	2	1
hondt	Region 6	2	1	1
hondt	Region 7	11	3	2
hondt	Region 8	12	3	1
matdet	Region 9	9	4	0
matdet	Region 10	14	4	3
matdet	Region 11	6	3	1
matdet	Region 12	8	2	0

Notice that not all the code lines indicated in Table III were included in the regions shown in Table IV. The following chunks of code were not considered: a) function prototypes and b) lines of variables declaration without initialization at the very beginning of functions. We are aware that some bugs might occur at either point. However, because of the code complexity metrics computation, we left these code lines unpartitioned, and we did not insert any bugs in these two exceptions.

5 BIOMETRIC SIGNALS ANALYSIS AND CODE REGIONS CLASSIFICATION

This section describes the preprocessing and the analysis performed on the biometric signals to capture the cognitive load in the biometric measures. We also show the two methods applied to classify each code region as either bad or well-reviewed. The classification is performed per code region after mapping the biometric measures to each code region using the timestamp index.

5.1 HRV Analysis and Features

The ECG signals were recorded at a sampling frequency of 10kHz (much higher than the required Nyquist frequency to avoid aliasing). After down-sampling the data to 1kHz, we used a standard Pan-Tompkins segmentation algorithm to extract the R-R intervals [73]. The spectral power ECG's R-R interval variability was assessed using 25 seconds, and a sliding window shifted with 1-sec increments.

Our goal is to detect whenever a difficulty occurs in understanding a code region. Since code regions under review consist of multiple LoC, the 25 seconds time window used to assess the R-R interval variability and extract the HRV features is adequate, even for code regions that have just a few LoC. The 25 second time window is also enough to accommodate the natural delays (e.g., the delay between stimuli and heart response) into consideration. In [74], it is stated that it takes about 5 seconds to increase HR after the onset of sympathetic stimuli and almost 20-30 seconds to reach its steady peak level. The power spectrum was computed using Burg's autoregressive power spectrum approach [75]. To capture the sympathetic and parasympathetic activations, several features from the time and frequency domain have been computed (e.g., absolute and relative area/peak of the spectrum low frequency, high frequency, and their ratio). From these features, several transforms (mean, std. deviation, median, min, max, and quantiles and peaks) have been calculated to access a discriminant HRV index for each code region under analysis. In the first batch of evaluation (i.e., rule-based classification), we used that correlate best with the complexity of the region being analyzed, namely, the 0.75 quantiles of the relative area of the spectra high-frequency interval (fsHRV).

As input to the rule-based classifier, since the selected feature presented a negative correlation, the log(1/fsHRV) was calculated within each code region and used to surrogate the cognitive load. This negative correlation was expected because the mental workload is associated with sympathetic activation (an increase of low frequencies) and parasympathetic withdrawal (decrease of high frequencies) [76].

For the sake of interpretability in the rule-based-classifier, we started with a minimal number of HRV features by selecting the most discriminant. However, in the comprehensive evaluation to unveil other HRV features, we used Kruskal-Wallis-based feature selection and Relief feature selection techniques [56][57]. The selected features represented the HRV time domain, such as the Approximate Entropy, which changes concomitantly with acute responses to cognitive load and stress, as shown in [77]. The HRV frequency domain peaks quantile (0.75 and 0.95). Some ultra-short features such as SDNN and RMSSD were used for fine-grained analysis (e.g., [78]).

5.2 Pupil signals Analysis and Feature

Eye pupil response is recognized as an indicator of cognitive

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

TABLE V. KCM FEATURES AND THRESHOLDS

Feature	Threshold
Cognitive Load	– LOW: <=1 .5 * baseline (mean/median of the text reading phase) – HIGH: > 1.5 * baseline (mean/median of the text reading phase)
Code complexity	– LOW - simple code: Vg <=4 – HIFG - complex code: Vg >= 5
Expertise	– LOW - standard reviewer: score of written test > 4 and < 7 out of 10 – HIGH - expert reviewer: score of a written C test >= 7 out of 10
No of revisits	– LOW: No of Revisits < 15: – HIGH: No of Revisits>15
Reviewing time	– LOW: Region complexity low and reviewer's expertise low and reading time (sec) < 4 * LoC of the region – HIGH: Region complexity low and reviewer's expertise low and reading time (sec) >= 4* LoC of the region – LOW: Region complexity low and reviewer's expertise high and reading time (sec) < 2 * LoC of the region – HIGH: Region complexity low and reviewer's expertise high and reading time (sec) >= 2 * LoC of the region – LOW: Region complexity high and reviewer's expertise low and reading time (sec) < 15 * LoC of the region – HIGH: Region complexity high and reviewer's expertise low and reading time (sec) >=15* LoC of the region – LOW: Region complexity high and reviewer's expertise high and reading time (sec) < 10 * LoC of the region – HIGH: Region complexity high and reviewer's expertise high and reading time (sec) >=10 *LoC of the region

Comments
Cognitive load is assessed through the LF/HF variability of the HRV and pupillography. The threshold was decided through analysis of the data collected from all the participants in the experiment using 1.5* baseline (mean/median of the text reading phase) as threshold criteria.
Code complexity is measured using cyclomatic complexity (Vg). The threshold was decided using common practices concerning the complexity of the code in unit testing and code reviewing.
Reviewer's expertise was previously assessed by a screening interview and a written C programming test that produced a score on a 0 to 10 scale. The threshold was decided by the experts (co-authors) that defined the written C programming test. Low expertise corresponds to the Standard reviewers, and high expertise correspond to the expert reviewers
Revisits are counted using the eye tracker and indicate the number of times the reviewer went back to the code region to review it again. The threshold was decided considering 1.5*mean number of revisits of all subjects in the experiment
Reviewing time is measured directly through the eye tracker. The thresholds were decided by experts considering several criteria that influence the reading time, such as the code complexity, the reviewer expertise, and the number of code lines.

and mental efforts. Researchers in [20][61], for example, established the evident association between pupil activity and attentional, cognitive efforts. Kahneman and Beatty described in [79] that when a person recalls something from memory or attempts to parse sentences, the pupil dilates slightly and returns to its normal size after the task is done. This reaction was called task-evoked pupillary response (TEPR) [20]. In our work, the frequency domain features of the pupil diameter change have been calculated using the same approach used in the HRV analysis, along with its transforms, as described in the previous section, and the extracted features transforms were used in the setup of the proposed classification models that will be illustrated in the next section.

5.3 Code Review Quality Classification

After performing the pre-processing and the analysis on the biometric signals, we map those signals to each code region of the programs using synchronous timestamps. We know from the eye-tracker at what time the reviewer was looking at a specific code region. This timestamp helps to map the biometric signals (i.e., measures) to each corresponding code region. We perform the code region classification from two perspectives: the knowledge-driven rule-based classification model (KCM), built according to the rules shown in Table II, and the data-driven classification model (DCM) using the KNN and Logistic Regression classifiers, which were chosen for explainability purposes. KNN is known as an instance-based, highly accurate classifier that does not need a training phase. Likewise, Logistic Regression is chosen because it is easy to implement, interpret, and suitable for the limited set of data we currently have.

As for the KCM, to classify the code review, the rules defined in Table II have been applied using thresholds that experts in the domain defined to each feature shown in Table V. After the thresholds had been identified, all the features indicated in Table V were labeled "High" or "Low" for each code region. We applied the rules in Table II to the data we collected in the controlled experiment.

The output results of the model are computed using the standard classification performance metrics and conditional ratio metrics. For example, if we consider the code region reviews that are classified as good while the reviewer has detected all bugs, the formula will be:

$$P(Good\ rev \mid all\ bugs\ detect) = \frac{P\ (Classf.\ good\ \cap all\ bugs\ detect.)}{P\ (Classf.\ good\ rev)} \quad (1)$$

The ideal case is to predict a good code review when the reviewer has detected all bugs in the code region (and to predict bad code review when not all bugs have been detected).

As we can see also from Table V, in KCM, the best biometric features that represent the cognitive load of reviewers are the LF/HF ratios of the HRV. In contrast, the DCM approach is built on the KNN and Logistic Regression. Since this approach is data-driven, the biometric features were upgraded to include 791 HRV and Pupil response features. Those biometric features were augmented with a) code region reviewing time, b) code complexity, c) the number of revisits of the code region. The augmented features vector is then fed into a feature selection module using the Kruskal-Wallis-based feature selection and Relief feature selection techniques mentioned earlier. The best ten selected features were: HRV frequency domain peaks mean, the code complexity, the HRV frequency domain peaks quantile 0.95 and the LF/HF ratio, the pupil diameter peaks median, the HRV time-domain SDNN, and RMSDD, the pupil diameter peaks mean, the pupil diameter quantile 0.85, the expertise, the time inside the area, and the revisits.

The classification classes were derived using the f-beta measure [80] of the subjects' performance. F-beta represents the harmony between the precision and the recall of the participant's

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

performance. Therefore, f-beta would characterize the essence of missing bus/identifying wrong bugs. The following formula gives the f-beta measure:

$$fbeta = \frac{(1 + beta^2) \times precision \times recall}{(beta^2 \times precision + recall)} \quad (2)$$

With beta = 2, we are giving more weight to the recall than the precision. This priority is because the false negatives (i.e., missing bugs) should be penalized in the code review evaluation. In real environments, companies can present different priories (i.e., different beta).

We hypothesize that missed bugs (even wrongly identified bugs) would be a matter of code comprehension challenges, stress, or distractors that would affect the code review quality and reviewer's focus. Because we are using a binary classification problem, a set of thresholds was tested to return a value of either 0 or 1 from the f-beta values. A threshold of 0.70 was determined after empirical experiments (testing thresholds against classifiers accuracy) and visualization of the data distribution visualization. We noticed that 0.7 is the closest value to dividing the data distribution between the good/bad performing participants. We observed that those who scored more than 0.7 on the f-beta measure had reasonably acceptable performance in terms of false negatives and false positives. In other words, the label of features is considered one or good review when the f-beta>= 0.70 and 0 or bad review otherwise.

6 Results and Discussion

Before evaluating the accuracy of the proposed approach with both KCM and DCM models, we analyzed each participant's performance in terms of bugs found, as shown in Figure 3 below. We noticed a wide amplitude of results, ranging from 0 precision and 0 recall to more than 0.9 precision and more than 0.75 in the recall. Participants 5, 6, and 11 performed the worst; participants 15, 16, 17, 18 (Experts) performed the best, having an above-average performance in terms of recall. The remaining ones show a well-dispersed across these extremes.

Experts were 49% better than the average of precision of all participants and 91% better than the participants' average of recall, confirming previous works that measured big differences in the quality of individual reviews [65][66][67]. Figure 3 shows us an interesting scenario to study as we can use the disparity of participants to relate to differences in the HRV and pupil signals collected during the experiments. In the end, the proposed approach would be particularly useful when the reviewers are not experts, which is the case for most of the reviewers used in our experiment. In fact, non-experienced reviewers tend to miss bugs in code reviews, which means that the review quality evaluation provided by our approach could be a useful tool telling the reviewers to repeat the review process in certain code regions.

From a task view, Figure 4 shows the average performance of participants (in terms of precision and recall) of finding the bugs is highly dependent on the program under review. We notice that the most straightforward program (i.e., "Fibo") has the best performance across the participants. In contrast, problematic programs such as the "Bsort" and the "Hondt" have relatively low performance, especially in the recall. Going deeper in the analysis, we observed that the complex programs such as the Matdet and the Bsort had the highest mental effort according to the NASA-TLX [81] that was distributed among participants (i.e., after each task participants scored their subjective assessment of mental effort, pressure with time, task fulfillment, and discomfort). In contrast, Fibo had the least mental effort score and the highest feeling of task fulfillment.

6.1 KCM Evaluation

To evaluate the knowledge-Driven Rule-Based Classification Model (KCM), we attempt to show that assessing the fine-grain code comprehension of the reviewer in each code region and applying a set of rules can give us an automated technique (i.e., the proposed approach) to evaluate the quality of code reviews and identifying code regions that should be reviewed again. After classifying the cognitive load in each code region of the programs, we applied the rules shown in Table II, defined by a panel of experts in the domain. We calculated the results by taking the average of all code regions in the programs. The results are displayed in Table VI.

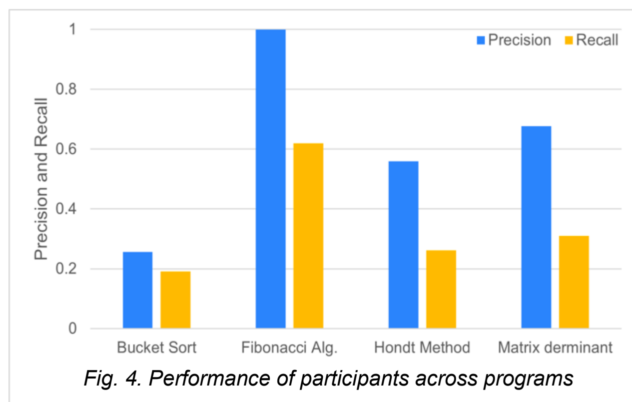

Fig. 4. Performance of participants across programs

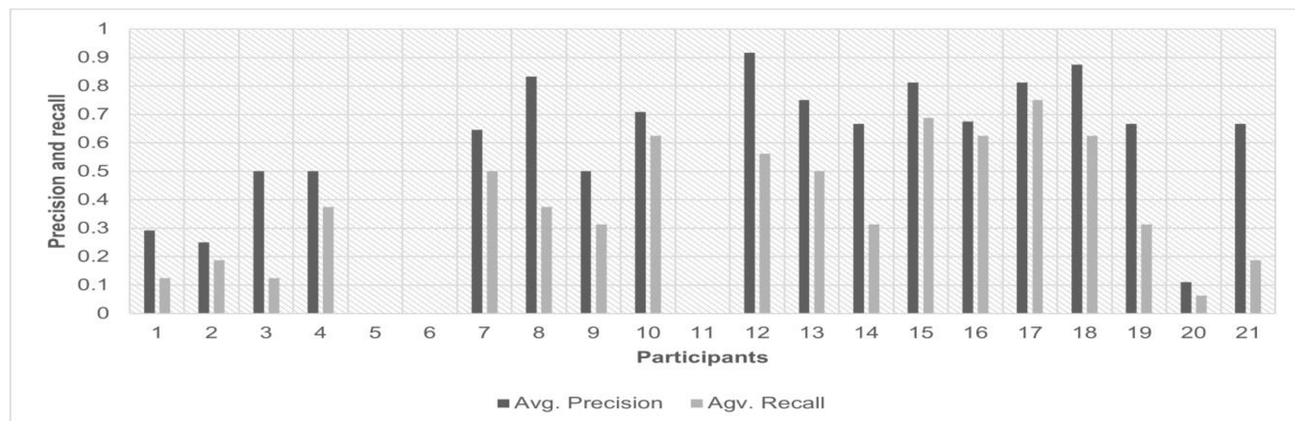

Fig. 3. Individual Performance of participants in Code Review tasks

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

Note that we know the number of bugs effectively detected by the reviewers in each code region. Thus, we could calculate necessary evaluation measures such as the True Positives (found the bug) and False Negatives (missed the bug). These measures enabled us to label each code region as either: "not all bugs were detected" or "all bugs were detected." These labels helped us evaluate the rules by comparing the prediction of the rules (i.e., good/bad review) with the label (i.e., ground truth). The ideal case would be achieving 100% in columns A and B in Table VI. Column A indicates that the rule predicts the review as bad whenever a reviewer leaves some bugs undetected. Column B shows that the rules predict the review as good whenever a reviewer detects all bugs in the code region.

TABLE VI. KCM EVALUATION

PROGRAM	A	B	C	D
ALL	84.62%±0.57	47.97%±0.88	15.38%±0.06	52.03%±0.08
BSORT	84.21%±1.03	37.93%±1.37	15.79%±0.10	62.07%±0.14
FIBO	100.00%±0.00	76.92%±2.2	0.00%±0.00	23.08%±0.22
HONDT	100.00%±0.00	19.44%±012	0.00%±0.00	80.56%±0.12
MATDET	66.67%±1.33	68.89%±1.33	33.33%±0.13	31.11%±0.13

A: Predicted bad/ not all bugs were detected
B: Predicted good/ all bugs were detected
C: Predicted bad/ all bugs were detected
D: Predicted good/ not all bugs were detected

As we observe in Table VI, the KCM approach predicts the review as bad **84.62%** of the time in all the programs, when the reviewer left some bugs undetected (i.e., **it is needed to review the code again**). Concerning each program, KCM indicates the review as bad 84.21% when the reviewer left some bugs undetected in the "Bsort." We can also observe that in "Fibo" and "Hondt," KCM indicates that 100% of the time, the approach marks the review as bad when not all the bugs were detected. Column C and Column D in Table VII should, in the ideal case, indicate low percentages. Column C represents that the KCM would predict the review as bad, but the reviewer has detected all the bugs. KCM indicated this case 15.38% of the time in all the programs.

In contrast, Column D shows that KCM would predict the review as good, but the reviewer missed some bugs. We can see that KCM was not as successful in this case as it was in Column A, especially in "Bsort" and "Hondt." In "Hondt," the first section has just two lines of code with one bug. Most of the reviewers missed that bug in the two lines of "Hondt." However, the KCM predicted the region as "Good" most of the time as reviewers in this simple region had good reading time with relatively low revisits. Still, in the end, they could not detect that tricky bug injected within the two lines.

We conclude that KCM could perform well in most of the programs (84.62% of accuracy in the most important result, which is classifying the review as bad when in fact, the reviewer did not find all the bugs), but not in all of them. The analysis shows that thresholds defined by experts may sometimes fail to capture the actual performance of the reviewer due to the dynamic nature of programs under review, expertise, code complexity, and other factors. Furthermore, some code regions (i.e., regions 2, 9, and 12) are bug-free. Thus, the label of these regions is always "all bugs were detected," either if the KCM predicted "good" or "bad" for the review quality, which complicates, even more, the task of predicting the quality of code region reviews for the results in column D.

It is worth noting that we are using a Boolean approach for each criterion (i.e., cognitive load high/low, code complexity high/low, etc.), which makes the definition of the thresholds very critical for the result. Thus, in our data-driven approach (i.e., DCM), we are examining the role of AI techniques to dynamically set the thresholds and unveil the hidden structure of the data. In the next section, we show the evaluation of the DCM.

6.2 DCM Evaluation

Here we evaluate the data-driven rule-based classification model (DCM) built on the K-nearest neighbors (KNN) and Logistic Regression (LR) classifier. We examined two different cross-validation techniques. First, we used the hold-out cross-validation, where the data was split into 33% testing and 67% training and validation. Second, we applied Leave One Subject Out Cross Validation to ensure the capacity of the model to generalize whenever an unseen subject is tested. The model received biometric and non-biometric features selected by Kruskal and Relief techniques. The biometric features from HRV and Pupil response included the following: a) HRV SDNN, b) HRV RMSDD, c) Very Low Frequency of HRV peaks and LF/HF ratio, d) Very Low Frequency of spectrum of normalized pupil peak, e) Median of pupil peaks.

The non-biometric features included the code complexity, the expertise level, the reading time, and the number of revisits. The multimodal features were labeled using the f-beta measure labels (Formula 2) in each code region. The predicted values of the classifiers were used to derive the exact measures of the KCM shown in Table VII (columns A, B, C, and D) to compare the two models. The following table shows the performance of the DCM approach.

TABLE VII. DCM EVALUATION

PRGRAM	A	B	C	D
K-NEAREST NEIGHBORS (KNN)				
ALL	84.85%±5.94	70.27%±7.57	15.15%±5.94	29.73%±7.57
BSORT	87.10%±5.55	66.67%±7.81	12.90%±5.55	33.33%±7.81
FIBO	100.00%±0.00	90.91%±4.76	0.00%±0.00	9.09%±4.76
HONDT	94.74%±3.7	85.71%±5.8	5.26%±3.70	14.29%±5.80
MATDET	61.54%±8.06	77.14%±6.96	38.46%±8.06	22.86%±6.96
LOGISTIC REGRESSION (LR)				
ALL	86.57%±5.65	68.00%±7.73	13.43%±5.65	33.00%±7.79
BSORT	92.31%±4.41	50.00%±2.28	7.69%±4.41	50.00%±7.28
FIBO	100.00%±0.00	90.91%±4.76	0.00%±0.00	9.09%±4.76
HONDT	95.24%±3.53	80.00%±6.63	4.76%±3.53	20.00%±6.63
MATDET	64.71%±7.92	83.87%±6.09	35.29%±7.92	16.13%±6.09

A: Predicted bad/ not all bugs were detected
B: Predicted good/ all bugs were detected
C: Predicted bad/ all bugs were detected
D: Predicted good/ not all bugs were detected

DCM evaluation in Table VII shows that the approach predicts the reviews as bad 84.85% and 86.57% when the reviewer left some bugs undetected in the whole set of programs by KNN and LR, respectively. Although this value is like the one obtained using KCM, the value in the remaining columns of Table VII presents much better performance. In column B, DCM predicts that the review is good 70.27% and 68.00% of the time when the reviewer detected all the bugs in the code regions as classified by KNN and LR. The results in column C are also quite good,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

showing that classifiers' decisions tend not to suggest reviewing again (i.e., classify as bad review) when all bugs have been detected, only in 15.15% and 13.43% of the cases classified by the KNN and the LR the code regions as bad reviews when the reviewer has discovered all the bugs.

It is worth mentioning that these results were obtained at the code review level, which is a fine grain evaluation of the review quality. Although the reviewers' subjective assessment performed through the NASA-TLX revealed that they feel accomplished about their task in "Fibo," the approach could efficiently predict the regions where they could not detect all the bugs as badly reviewed.

Concerning each program, DCM tells us that 87.10% and 92.30% of the review is predicted bad by the KNN and the LR, respectively, when the reviewer missed some bugs in the "Bsort." We can also observe that in "Hondt," the KNN and the LR classifiers predict 94.74% and 95.24%, respectively, of the time, the review as bad when the reviewer missed some bugs. Interestingly, in Matdet, we could notice how KCM outperformed the DCM in predicting bad reviews.

Nonetheless, we could relate the good performance of the DCM from the capacity of the KNN to perform well, given the limited dataset and number of features. The LR also showed the ability to tune the thresholds to decide the class as either 0 for bad review or 1 for the good review. In the case of DCM, the thresholds are dynamically updated and adapted to each reviewer's patterns. In Table VIII, we show the general performance of the KNN and the LR classifiers from the standard form of classification performance, including the accuracy, precision, recall, and the F1-score. The recall here embeds a critical performance metric which is the True Negative. True Negatives, as described earlier, represent the missed bugs in the code regions.

TABLE VIII. CLASSIFIERS PERFORMANCE

Class	Precision	Recall	F1-score
K-nearest neighbors (k=3)			
Bad review	80.00%±4.02	70.00%%±0.11	74.10%±0.21
Good review	60.01%%±1.24	72.00%%2.30	66.20%%±0.78
Accuracy	70.30%%±0.34		
Logistic Regression			
Bad review	81.00%±1.33	78.00%±0.11	77.10%±0.21
Good review	65.31%%±2.30	72.00%%0.80	71.60%%±0.89
Accuracy	73.70%%±1.60		

From the table above, the recall tells that 70.00% and 78.00% of the time, the classifiers correctly predicted the "bad review" from the actual bad reviews, with 70.30% and 73.70% KNN and LR accuracy, respectively. The chance level was calculated after shuffling the features with the labels 100 times randomly; the score of the random classification was significantly less than the accuracies mentioned in Table VIII with a p-value: 0.004. The p-value here is the classic p-value in hypothesis testing and represents the probability that the classification score would be obtained by chance.

The predicted values of code review performance were tested against the actual values of the reviewer's performance. Using the Spearman ranking correlation test, the rho value= 0.85 and with p-value= 0.001, which indicates a strong association between the predicted and actual values. Spearman was used due to the non-normality of the data distribution shown by Shapiro-Wilk's method.

6.3 Overall Code Review Evaluation and Explainability of The Results

This section describes how the overall outcome of the code review evaluation comes to the user.

In KCM, if the triggered rule (in Table II) indicates a bad review in one of the code regions, the approach localizes that code region through the eye-tracker and identifies it as a "red region" that should be reviewed again. The approach is conservative as it is enough to have one region to classify the whole review as a "bad" (and, for example, reject the proposed change to the code in a pull request).

It is easy in KCM to justify the rejection of the review because we know the rules defined by the experts. Imagine, for example, that a non-expert reviewer reviewed a complex code region, and this reviewer's cognitive load was low while reviewing that code. If this reviewer performed an insufficient number of revisits to that complex code region and did not read that code carefully (low reviewing time in the region), then the approach will trigger rule 24 in Table II. (rule with the combination mentioned before) and classify this code region as "bad" reviewed. Thus, the code review evaluation outcome would indicate that the reviewer should consider the entire review again while showing the problematic code regions.

In DCM, we do not have such predefined rules to justify the rejection or the acceptance of a code review. However, it is worth noting that in practical terms, when the global classification of the code review is classified as bad, the indication of the specific code regions that caused such classification is helpful information to tell the reviewer which code lines should be reviewed again. In practice, the approach will allow both KCM and DCM to operate depending on the software company priorities (e.g., accuracy vs. high interpretability).

Nevertheless, KNN and LR are considered *explainable* models. Although KNN has no parameters to learn, and thus, not interpretable on the modular level, it can explain the prediction at the local level. For example, we can observe the k neighbors classifier that was used to obtain the prediction of the code review. The locality of the model interpretation is relevant as we deal with the code region's level of granularity. The outcome of the approach can tell the code reviewer that, for instance, the code region was predicted as badly reviewed as it resides within a region of cases that have a high cognitive load and low complexity code region.

Regarding the LR interpretability, the LR uses the logistic function to transform the weighted sum into probability between 0 and 1. The way to interpret the LR is to witness the prediction changes when one of the features increases by 1 unit. The ratio of the two predictions (before and after increasing the feature value) can be an index of the feature importance in producing the prediction decision at the end.

6.4 Results validation using EEG

Our work hypothesizes that biomarkers extracted from HRV and Pupillography using biometric sensors could surrogate the cognitive load induced by the mental effort in the code review task. We hypothesize that using reviewers' cognitive load measured using HRV and Pupillography from non-intrusive wearable sensors (i.e., compatible with software development environments) is accurate enough to assure the feasibility of the proposed approach. Since the ANS signals do not flow directly from the brain (such as EEG signals) but rather from the peripheral expressions of the ANS, we believe it is essential to validate the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

accuracy of the cognitive load measured using HRV and Pupillography when compared to the measurements obtained using EEG. Although plenty of papers show that HRV and Pupillography can be used to measure cognitive load (e.g., [18][19][20]) we want to ensure further that they can be applied to intellectually complex and abstract tasks such as code reviews. Medeiros et al. [82] showed that EEG biomarkers could be used to fine-tune or validate the results obtained with ANS signals, which can be acquired using non-intrusive wearable devices.

This section presents an additional layer to the validation of the results using EEG biomarkers. EEG can be a robust measure to understand brain mechanisms behind mental tasks such as code comprehension and code reviews. To examine this validation empirically, a 64-channel EEG cap was used to record brain activity during the experiment that included the code review tasks described in Section 5.2. A standard preprocessing pipeline was performed to guarantee a reasonable signal-to-noise ratio of the neural signals for the remaining analysis. The preprocessing comprised different steps (see [82] for more details): (i) reduction of MR-induced EEG artifacts (gradient artifact and pulse artifact) due to the nature of the experimental protocol designed; (ii) standard pipeline where filtering, noisy channels interpolation, re-referencing, and blind source separation steps were performed to remove additional typical EEG artifacts (i.e., noise in the signal due to power lines and other equipment).

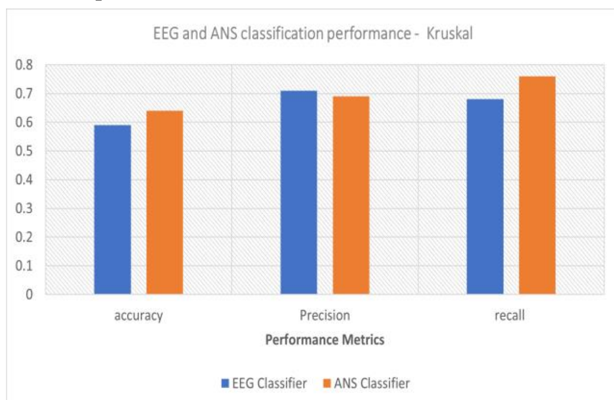

Fig. 5. EEG and ANS performance comparison in the same code regions

After the preprocessing of the EEG signals was performed, a feature engineering phase was completed. From the 60 preprocessed EEG signals, different standard features reported in cognitive state assessment studies were extracted for further analysis. In the context of cognitive state assessment, the most prominent and explored features are related to the spectral band power from the EEG bands (Delta, Theta, Alpha, Beta, and Gamma). In the feature extraction, linear univariate features (statistical features, Hjorth parameters, and spectral power features) and non-linear univariate features (Higuchi fractal dimension and Hurst exponent) were considered and extracted using a 1-second window with an 80% overlap. Then, second-order features (maximum, minimum, mean, standard deviation, and median) were computed to capture and enhance the software programmer's mental state. In the end, a total of 10,500 features were extracted (60 channels x 35 types of features x 5 second-order features).

If we use the EEG features to train the same classification model used by the ANS biomarkers (with the same labels) and we achieve similar performances (i.e., accuracy, precision, and recall), we can claim that EEG could verify the ANS biomarkers captured through non-intrusive biofeedback devices. To achieve this, we performed the following: a) a total of 10500 EEG features that correspond to each code region were fed into the Kruskal-Wallis-based feature selection, and Relief feature selection techniques, then the top 5 features from each technique were selected. The best features were as follows:

1) The maximum ratio of Theta Power / Beta Power from electrode C1.
2) The maximum ratio of Theta Power / Beta Power from electrode TP8.
3) The median of the relative power of Theta from electrode C6.
4) The maximum ratio of Theta Power / Beta Power from electrode P2
5) Median of the ratio Theta Power / Gamma Power from electrode PO3

As we can see from the selected features, the Theta band is dominating in all the features. This result complies with the studies that associated Theta band with the mental workload or the error making see [83].

We used the leave-one-subject-out cross-validation (LOSOCV) to assess the proposed model. Selecting LOSOCV could ensure the capacity of the model to generalize whenever unseen instances of data appear in the testing. These selected EEG features in isolation of the other features (i.e., code complexity, experience, scan time, revisits) were fed into the same classifiers to train the EEG features on the same label used in code regions classification. Likewise, the top selected ANS features were provided to the KNN and the LR using the same cross-validation technique (i.e., LOSOCV). We analyzed both classifiers (EEG and ANS classifier) to compare the performances.

The null hypothesis (H0) states that no significant difference exists between the two classifiers in terms of performance. In contrast, the alternative hypothesis (H1) states there is a significant difference between the two classifiers' performance. To test the hypothesis, we used the T-test (after testing the normality using Shapiro Wilk) at two significance levels of $\alpha = 0.05$ and 0.01. The results failed to reject H0 with a p-value of 0. 45. Thus, we conclude that both classifiers equivalently perform, and there would be no significant differences in their performance.

Figure 5 shows that both EEG classifiers and ANS classifiers (HRV and Pupillography) are performing similarly. Both classifiers correctly marked the "bad" review using either EEG or the ANS biomarkers with high recall in both classifiers (i.e., the EEG and the ANS). The same classifiers were tested against randomly shuffled features/targets, and the performance was significantly less than the original classification results with a P- value=0.03 As we can see from the performance, using ANS biomarkers extracted from available biofeedback devices can surrogate the cognitive load of the individuals. In other words, the ANS biomarkers could capture the cognitive load induced by the code comprehension and review as well as the EEG biomarkers could do. This result aligns with what Medeiros et al. [82] have achieved, showing that EEG can assess programmers' cognitive load accurately. Still, its intrusiveness compared to ANS signals is less prevalent in practice and natural software development environments.

7 Threats to Validity

Although the evaluation results of the approach are promising, there are some limitations in our experimental evaluation that we discuss in the next paragraphs as possible threats to our approach validity.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

Internal Validity: in our study, internal validity issues deal with the data acquisition environment. Data were acquired in a controlled experiment with the typical constraints of laboratory environments. Although we have assured all the participants that their performance as code reviewers were not under evaluation in any circumstances, it is impossible to avoid the "feeling of being observed", which could be amplified by the biometric sensors and the eye tracker. A real code review environment, in contrast, can be more dynamic with extra mind-wandering tasks (e.g., switching between tasks and interruptions). However, our focus was to evaluate the accuracy of the proposed approach, which, in our view, justifies the controlled code review environment used in our experiments.

Construct Validity: this study's construct validity touches on different aspects. First, the rules we used to assess the quality of code reviews were limited to 32 rules. The derivation of rules was based on different sources of information, such as our own experience in the field, opinions of experts from the industry, established best practices of modern code reviews, and related literature. We believe that more refined rules may allow a more accurate evaluation of the quality of the code reviews.

Second, the number of categories used for each criterion in the rules and the thresholds that differentiate each category impact the quality of the code review evaluation. In our experiments, we used a Boolean approach for each criterion of the rules (e.g., only two levels: low or high). Although we think this will impact the results, most likely, the use of more refined rules, more levels for the criteria, and refined thresholds would lead to even better results for the accuracy of the proposed approach.

Third, the source code used for the code review tasks might not perfectly represent real-world software. Although having different complexity and including both iterative and recursive paradigms, the four programs used are relatively small compared to the existing software. Naturally, a controlled experiment cannot use extensive code samples as the duration of the task asked of the participants would be prohibitively long.

Fourth, one of the vital construct validity issues deals with heart measurement accuracy. ECG cannot be used to acquire heart measurements in a real software environment. Various studies (e.g., [59]) show that it is feasible to use PPG based on bracelets and smartwatches as an alternative to ECG to extract HRV/PRV features.

Fifth, the use of KNN could be efficient in the low dimensionality of data and could be explainable in this limited setting. The LR could also provide some explainability without getting insights into the interaction of features. Nevertheless, in the next round of data acquisition, regression algorithms such as the Multivariate adaptive regression spline (MARS) might be an alternative due to its simplicity, ability to model non-linear data, and explainability with higher dimensionality.

External Validity: first, in the modeling phase, as we used the features mentioned in this study, the threat of obtaining a training-serving skew might present when deploying a future code review tool in a realistic software development environment. New features could be added in the serving time (i.e., deployment), or a change in the data type of features would occur, and many other skews (e.g., feature distribution skew) could happen; however, in ML end-to-end open-source platforms like TensorFlow [84], there are various solutions to this threat, such as TensorFlow Data Validation (TDF) which monitors the ML model during the deployment for any data skew anomaly.

Second, the limited number of subjects (i.e., 21) is considered a challenge in this study. Although most software engineering studies that use neurophysiological methods use a similar number of participants (see the comprehensive survey published in 2021 [34]), the difficulty in using many participants in these kinds of studies is a real problem. Moreover, male preponderance in the experiment represents one of the external validity issues

8 CONCLUSION AND FUTURE WORK

Modern code reviews are highly dependent on individual reviewers' performance to identify bugs and other quality problems in the software under review. Natural human limitations such as reviewers' distraction, fatigue, or difficulties in fully comprehending the code under review have a clear negative impact on the quality of code reviews and may leave bugs undetected.

We propose an innovative approach that monitors the reviewer's performance at code line reviewing level (considering small code snippets called code regions) and evaluates the overall quality of the code reviews by providing three relevant outcomes: **a)** an overall evaluation with a clear indication of whether the review should be repeated or not, **b)** pointers to code regions that may not have been well-reviewed, and **c)** an explanation of why the review of the pointed code regions was considered not satisfactory.

The proposed approach uses biometric information to assess the cognitive state of the reviewer (particularly, cognitive load) during the code review process. This can be accomplished through non-intrusive devices (e.g., smartwatches and bracelets) that capture biomarkers such as Heart Rate Variability (HRV) and Pupillography to assess the reviewer's cognitive load, and an inexpensive desktop eye tracker to associate the measured reviewer's cognitive load to the reviewing of specific code regions.

The evaluation of code review quality of each code region of the software under review is achieved by combining the information on the reviewer's cognitive load with other code reviews quality factors such as the code complexity of that region, the number of revisits to that region, the reviewing time of the region, and the experience level of the reviewer. The review of each code region is classified as Good or Bad in terms of bug-finding effectiveness. The proposed approach assumes the conservative approach of classifying the whole code review as Bad (i.e., needs to be repeated) if one or more code regions under review are classified as bad.

The proposed approach was implemented as part of a controlled experiment to evaluate the accuracy of the approach. The goal was to validate the accuracy of the classifications of code region reviews as Good or Bad, as this is the central element to consider the proposed approach useful. We believe the approach is valid if the code regions classified as badly reviewed contain undetected bugs, justifying the need for a second review. The implementation of the proposed approach relies heavily on Artificial Intelligence to assess the reviewer's cognitive load and, more specifically, to classify the code region reviews. Two types of classifiers were considered in the approach evaluation: a Knowledge-Driven rule-based Classification Model (KCM) and the Data-driven rule-based Classifier Model (DCM).

The controlled experiment included 21 code reviewers that reviewed four programs of different types and complexity. Programs were seeded with real bugs to have a ground truth to evaluate the actual performance of the reviewers in finding the bugs. Results show that the data-driven classifiers (DCM) provided, in

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

general, better results than the knowledge-driven classifier. On average, for all the code regions of all the four programs, our results show that **85.65% and 87.77%** of the code region reviews classified by KNN and LR, respectively, as Bad correspond to code regions where the reviewers left one or more bugs undetected (i.e., it is necessary to repeat the review) and **78.14% and 74.56%** of the code region reviews classified by KNN and LR, respectively, as Good correspond to code regions where the reviewers detected all the bugs.

Since the proposed approach's accuracy depends on the reviewers' cognitive load assessed by HRV and Pupillography, we also compared the cognitive load assessed through HRV and Pupillography with the cognitive reference load assessed using an assessment EEG cap with 64 channels. Results show no significant difference, which means that the low-intrusive HRV and Pupillography are accurate enough to be used in practice to assess reviewers' cognitive load.

Future research directions may allow to expand and improve further these positive results. Firstly, we are currently developing a real-world implementation of the proposed approach as an extension of the GitHub code reviewing tool (see a first description of the tool architecture and screenshots in [85]). The goal is to use such a tool in real code review environments to allow us to evaluate other aspects of the proposed approach, namely the reviewer's reactions and utilization of the tool.

A second research direction consists of including additional sources of information to characterize reviewers' cognitive state in a better way (e.g., differentiate cognitive load, stress fatigue, and other states). For example, EDA (electrodermal activity) is a promising additional source. It is non-intrusive and is referred to in the literature as an excellent biometric source to assess stress.

A third research direction is to refine the levels and thresholds considered for the different criteria used in the evaluation rules of code reviews. For example, we can consider several levels for the reviewer's cognitive load, the complexity of code regions, the reviewers' expertise, etc., instead of the two levels alternative (i.e., low or high) used in the present evaluation. Having several levels of reviewer's expertise, code region complexity, reviewers' cognitive load, etc., would make the definition of the set of rules much more sophisticated. Fuzzy logic reasoning could be used where there are no crisp values but membership functions that enable a more realistic human type of reasoning

A fourth research line is related to the evaluation of the complexity of the code regions, as metrics such as Vg and LoC do not capture well code complexity from a human perspective. The promising idea is to use cognitive weights established for the different code constructs [86] to compose a more effective measure of code region complexity.

Although HRV and Pupil response were good predictors for the reviewer cognitive load using the KNN and the LR in the DCM approach, we believe that future work should focus on the extraction and fusion of other features that are known in the literature to reflect the changes in cognitive load, such as the frequency domain features of the pupil diameter variability as well as features extracted from the EDA sensor and context information. By fusing these features, we aim to achieve a more robust surrogate of the reviewer's cognitive state and, therefore, more precise application of the proposed rules. Furthermore, we believe that future work should consider other cognitive states of the reviewers, such as stress, distraction, and fatigue, and not only the cognitive load imposed by code comprehension difficulties.

An interesting aspect to include in future work is to integrate the defect classification (i.e., functional and evolvability) with the evaluation metrics, as proposed in [25].

Finally, we are confident that the interdisciplinary nature of the proposed approach and the encouraging results obtained in the evaluation presented in this paper have a good potential to open new research avenues in the assessment of code comprehension and improvement of software reliability.

Acknowledgment

The authors thank the volunteers who took part in the controlled experiment. The authors acknowledge that the BASE project partially funded this work under Grant POCI - 01-0145 - FEDER- 031581, in part by the Centro de Informática e Sistemas da Universidade de Coimbra (CISUC), and in part by Coimbra Institute for Biomedical Imaging and Translational Research (CIBIT), Institute of Nuclear Sciences Applied to Health (ICNAS), the University of Coimbra under Grant PTDC/PSI-GER/30852/2017 | CONNECT-BCI.

REFERENCES

[1]. Huang, F., Liu, B., Wang, S., & Li, Q. (2015). The impact of software process consistency on residual defects. Journal of Software: Evolution and Process, 27(9), 625-646.

[2]. S. M. A. Shah, M. Morisio, and M. Torchiano, "The impact of process maturity on defect density," in Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, 2012, pp. 315–318.

[3]. N. Honda and S. Yamada, "Empirical analysis for high quality sw development," Ameri. Jour. Op. Research, 2012.

[4]. H. Zhang, "An investigation of the relationships between lines of code and defects," in 2009 IEEE International Conference on Software Maintenance. IEEE, 2009, pp. 274–283.

[5]. I. Sandu, A. Salceanu, and O. Bejenaru, "New approach of the customer defects per lines of code metric in automotive SW development applications," in Journal of Physics: Conference Series, vol. 1065, no. 5. IOP Publishing, 2018.

[6]. M. E. Fagan, "Design and code inspections to reduce errors in program development", IBM System J. vol. 15, No. 3, pages 183–211, 1976.

[7]. J. Barnard and A. Price, "Managing code inspection information," in IEEE Software, vol. 11, no. 2, pp. 59-69, March 1994, doi: 10.1109/52.268958.

[8]. Alberto Bacchelli and Christian Bird, "Expectations, Outcomes, and Challenges of Modern Code Review", International Conference on Software Engineering. ICSE, 2013.

[9]. Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018, May). Modern code review: a case study at Google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 2018.

[10]. James Reason, "Human Error," Cambridge University Press, https://doi.org/10.1017/CBO9781139062367, 1990.

[11]. A. Fuqun Huang, B. Bin Liu, and C. Bing Huang, "A Taxonomy System to Identify Human Error Causes for Software Defects", 18th ISSAT Int. Conference on Reliability and Quality in Design, 2012.

[12]. Cohen J, Teleki S, Brown E (2006) Best kept secrets of peer code review. Smart Bear Inc, Somerville D'Mello S, Graesser A (2014) Confusion and its dynamics during device comprehension with breakdown scenarios. Acta Psychol 151:106–116.

[13]. Tao Y, Dang Y, Xie T, Zhang D, Kim S (2012) How do software engineers understand code changes?: an exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. FSE '12. ACM, New York, pp 51:1–51:11.

[14]. Ebert, F., Castor, F., Novielli, N. et al., "An exploratory study on confusion in code reviews", Empirical Software Engineering, 26, 12 2021.

[15]. Joe Brockmeier, "A Look at Phabricator: Facebook's Web-Based Open Source Code Collaboration Tool", ReadWrite blog, Sep. 28, 2011 (https://readwrite.com/2011/09/28/a-look-at-phabricator-facebook/), accessed on Dec. 6, 2020

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

[16]. Shaumik Daityari, “12 Best Code Review Tools for Developers (2021 Edition)”, March 19, 2021, (https://kinsta.com/blog/code-review-tools/), accessed on March 30, 2021.

[17]. O. Kononenko, O. Baysal, and M. Godfrey, “Code review quality: how developers see it”, in Proceedings of the 38th International Conference on Software Engineering, 2016.

[18]. J. Veltman and A. Gaillard, “Physiological workload reactions to increasing levels of task difficulty,” Ergonomics, vol. 41, no. 5, 1998.

[19]. G. F. Walter and S. W. Porges, “Heart rate and respiratory responses as a function of task difficulty: The use of discriminant analysis in the selection of psychologically sensitive physiological responses,” Psychophysiology, vol. 13, no. 6, 1976.

[20]. B. Pfleging, D. K. Fekety, A. Schmidt, et al., “A model relating pupil diameter to mental workload and lighting conditions,” in Proc. of the 2016 CHI conference on human factors in computing systems, 2016.

[21]. D. Ford, M. Behroozi, A. Serebrenik, and C. Parnin, “Beyond the Code Itself: How Programmers Really Look at Pull Requests”, 41st ACM/IEEE International Conference on Software Engineering [ICSE SEIS], May 2019, DOI: 10.1109/ICSE-SEIS.2019.00014

[22]. Ackerman, A. F., Buchwald, L. S., & Lewski, F. H. (1989), “Software inspections: an effective verification process”, IEEE software, 6(3), 31-36, 1989.

[23]. Richard Bellairs, “Best Practices for Code Review”, Dec. 4, 2019, https://www.perforce.com/blog/qac/9-best-practices-for-code-review, accessed: 2022-03-01.

[24]. O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," *2015 IEEE International Conference on Software* Maintenance and Evolution (ICSME), pp. 111-120, 2015.

[25]. M. V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?," in *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430-448, May-June 2009.

[26]. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, “The influence of non-technical factors on code review”, In Proc. of the Working Conference on Reverse Engineering, 2013.

[27]. C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. IEEE Trans. Softw. Eng., 35(4):534–550, July 2009.

[28]. F. Shull, I. Rus, and V. Basili. “Improving software inspections by using reading techniques”, In International Conference on Software Engineering, pages 726–727, 2001.

[29]. L. Hatton, “Testing the value of checklists in code inspections”, IEEE Software, 25(4):82–88, 2008

[30]. A. N. Al-Saiyd, “Source code comprehension analysis in software maintenance”, in 2017 2nd International Conference on Computer and Communication Systems (ICCCS), 2017.

[31]. Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng, “Code Review Knowledge Perception: Fusing Multi-Features for Salient-Class Location”, IEEE Trans. on Software Engineering, 2020.

[32]. D. Fucci, D. Girardi, N. Novielli, L. Quaranta, F. Lanubile, “A replication study on code comprehension and expertise using lightweight biometric sensors”, Proceedings of the 27th International Conference on Program Comprehension, Pages 311–322, May 2019.

[33]. Jessup, S., Willis, S. M., Alarcon, G., Lee, M., “Using Eye-Tracking Data to Compare Differences in Code Comprehension and Code Perceptions between Expert and Novice Programmers”, In Proc. of the 54th Hawaii International Conference on System Sciences (p. 114).

[34]. B. Weber, T. Fischer, R. Riedl, “Brain and autonomic nervous system activity measurement in software engineering: A systematic literature review”, Journal of Systems and Software, March 2021.

[35]. T. Nakagawa, Y. Kamei, H. Uwano, et al., “Quantifying programmers’ mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment,” in Companion Proc. of the 36th Int. Conf. on Software Engineering. ACM, 2014.

[36]. Floyd, B., Santander, T., & Weimer, W. (2017, May). Decoding the representation of code in the brain: An fMRI study of code review and expertise. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 175-186). IEEE.

[37]. N. Peitek, J. Siegmund, S. Apel, C. Kastner, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann, “A look into programmers’ heads,” IEEE Transactions on Software Engineering, vol. 46, 2018.

[38]. J. Duraes, H. Madeira, J. Castelhano, et al., “Wap: Understanding the brain at software debugging,” in 2016 IEEE 27th International Symp. on Software Reliability Engineering (ISSRE). IEEE, 2016.

[39]. J. Castelhano, I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, and M. Castelo Branco, “The role of the insula in intuitive expert bug detection in computer code: an fMRI study,” Brain imaging and behavior, vol. 13, no. 3, pp. 623–637, 2019.

[40]. J. Castelhano, I. C. Duarte, J. Duraes, H. Madeira, M.Castelo-Branco, "Reading and calculation neural systems and their weighted adaptive use for programming skills", Neural Plasticity, 2021.

[41]. Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. I., “Analyzing individual performance of source code review using reviewers' eye movement”, In Proceedings of the 2006 symposium on Eye tracking research & applications, 2006.

[42]. Sharif, B., Falcone, M., & Maletic, J. I., “An eye-tracking study on the role of scan time in finding source code defects”, In Proceedings of the Symposium on Eye Tracking Research and Applications, 2012.

[43]. Chandrika, K. R., Amudha, J., & Sudarsan, S. D., “Recognizing eye tracking traits for source code review”, In 2017 22nd IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA), 2017

[44]. Begel, A., & Vrzakova, H., “Eye movements in code review”, In Proc. of the Workshop on Eye Movements in Programming (pp. 1-5), 2018.

[45]. Jason Cohen, “11 proven practices for more effective, efficient peer code review”, IBM Developer, January 2011.

[46]. Vrzakova, H., Begel, A., Mehtätalo, L., & Bednarik, R. (2020), “Affect Recognition in Code Review: An In-situ Biometric Study of Reviewer’s Affect”, Journal of Systems and Software, 159, 110434.

[47]. C. Setz, B. Arnrich, J. Schumm, et al., “Discriminating stress from cognitive load using a wearable EDA device,” IEEE Tran. on information technology in biomedicine, vol. 14, no. 2, pp. 410–417, 2009.

[48]. K. Kyriakou, B. Resch, G. Sagl, et al., “Detecting moments of stress from measurements of wearable physiological sensors,” Sensors, vol. 19, no. 17, p. 3805, 2019.

[49]. Anh Son Le, Tatsuya Suzuki, and Hirofumi Aoki, “Evaluating driver cognitive distraction by eye tracking: From simulator to driving”, Transp. Research Interdisciplinary Perspectives journal, Vol. 4, 2020.

[50]. T. Fritz, A. Begel, S. C. Muller, et al., “Using psycho-physiological measures to assess task difficulty in software development,” in Proceedings of the 36th International Conference on Software Engineering, ICSE, ACM, 2014, pp. 402–413.

[51]. S. C. Muller and T. Fritz, “Using (bio) metrics to predict code quality online,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 452–463.

[52]. R. Couceiro, G. Duarte, J. Duraes, et al., “Pupillography as indicator of programmers’ mental effort and cognitive overload,” in IEEE Int. Conf. on Dependable Systems and Networks, IEEE, 2019.

[53]. R. Couceiro, R. Barbosa, G. Duraes Duarte, et al., “Spotting problematic code lines using nonintrusive programmers’ biofeedback,” International Symp. on Software Reliability Engineering, ISSRE, 2019.

[54]. Lee, S., Hooshyar, D., Ji, H. et al. Mining biometric data to predict programmer expertise and task difficulty. Cluster Comput 21, 1097–1107 (2018). https://doi.org/10.1007/s10586-017-0746-2

[55]. (16) R. Minelli, A. Mocci, and M. Lanza, “I know what you did last summer: an investigation of how developers spend their time,” in Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. IEEE Press, 2015, pp. 25–35.

[56]. Ali Khan, S., Hussain, A., Basit, A., & Akram, S. (2014). Kruskal-Wallis-based computationally efficient feature selection for face recognition. The Scientific World Journal, 2014.

[57]. Urbanowicz, R. J., Meeker, M., La Cava, W., Olson, R. S., & Moore, J. H. (2018). Relief-based feature selection: Introduction and review. Journal of biomedical informatics, 85, 189-203.

[58]. Xiao Zhang and Yongqiang Lyu and Xin Hu and Ziyue Hu and Yuanchun Shi and Hao Yin, “Evaluating Photoplethysmogram as a Real-Time Cognitive Load Assessment during Game Playing”, International Journal of Human–Computer Interaction}, Vol. 34, No 8, 2018.

[59]. Pinheiro, N., Couceiro, R., Henriques, J., Muehlsteff, J., Quintal, I., Goncalves, L., & Carvalho, P. “Can PPG be used for HRV analysis?”, 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC) (pp. 2945-2949), 2016.

[60]. Mejía-Mejía, Elisa et al. “Heart Rate Variability (HRV) and Pulse Rate Variability (PRV) for the Assessment of Autonomic Responses.” Frontiers in physiology vol. 11 779. 23 Jul. 202.

[61]. V. Peysakhovich, M. Causse, S. Scannella, and F. Dehais, "Frequency analysis of a task-evoked pupillary response: Luminance-independent measure of mental effort," International Journal of Psychophysiology, vol. 97, pp. 30-37, 2015.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3158543, IEEE Transactions on Software Engineering

[62]. Andrzejewska, M., & Skawińska, A, "Examining Students' Intrinsic Cognitive Load During Program Comprehension–An Eye Tracking Approach", Int. Conference on Artificial Intelligence in Education (pp. 25-30). Springer, Cham, 2020.

[63]. Dunsmore, A., Roper, M., & Wood, M. (2000), "The role of comprehension in software inspection", Journal of Systems and Software, 52(2-3), 121-129.

[64]. R. Couceiro, G. Duarte, J. Duraes, J. Castelhano, C. Duarte, C. Teixeira, et. al, "Biofeedback augmented software engineering: monitoring of programmers' mental effort", Int. Conference on Software Engineering, New Ideas and Emerging Results, ICSE 2019.

[65]. Sauer, C., Jeffery, D. R., Land, L., & Yetton, P., "The effectiveness of software development technical reviews: A behaviorally motivated program of research", IEEE Transactions on Software Engineering, 26(1), 1-14, 2000.

[66]. Rigby, P. C., & Storey, M. A., "Understanding broadcast based peer review on open source software projects", 33rd International Conference on Software Engineering (ICSE) (pp. 541-550). IEEE, 201.

[67]. Biffl, S., & Halling, M. (2002, June). Investigating the influence of inspector capability factors with four inspection techniques on inspection performance. In Proceedings Eighth IEEE Symposium on Software Metrics (pp. 107-117). IEEE

[68]. Busjahn, T., Schulte, C., & Busjahn, A. (2011, November). Analysis of code reading to gain more insight in program comprehension. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research (pp. 1-9).

[69]. R. Chandrika, J. Amudha and S. D. Sudarsan, "Recognizing eye tracking traits for source code review," 22nd IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA), pp. 1-8, 2017.

[70]. J. Duraes and H. Madeira "Emulation of SW Faults: A Field Data Study and a Practical Approach", IEEE Transactions on SW Engineering, vol. 32, no. 11, pp. 849-867, November 2006.

[71]. D. Cotroneo, L. Simone, P. Liguori, R. Natella, N. Bidokhti, "How Bad Can a Bug Get? An Empirical Analysis ofSoftware Failures in the OpenStack Cloud Computing Platform", Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019.

[72]. R. Chillarege, I.S. Bhandari, J.Chaar, and M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements, IEEE Transactions on Software Engineering 18(11):943 - 956, Dec. 199.

[73]. Sacha J., "Interaction between heart rate and heart rate variability", Annals Noninvasive Electrocardiology. 2014, 19:207–216.

[74]. McCraty, R., & Shaffer, F. Heart rate variability: new perspectives on physiological mechanisms, assessment of self-regulatory capacity, and health risk. Global adv. in health and medicine, 4(1), 2015

[75]. Burg, J.P. "Maximum Entropy Spectral Analysis", Proc. of the 37th Meeting of the Society of Exploration Geophysicists, 1967.

[76]. Delliaux, S., Delaforge, A., Deharo, J. C., & Chaumet, G. (2019). Mental workload alters heart rate variability, lowering non-linear dynamics. Frontiers in physiology, 10, 565.

[77]. Blons, E., Arsac, L. M., Gilfriche, P., McLeod, H., Lespinet-Najib, V., Grivel, E., & Deschodt-Arsac, V. (2019). Alterations in heart-brain interactions under mild stress during a cognitive task are reflected in entropy of heart rate dynamics. Scientific reports, 9(1), 1-10

[78]. Pereira, T., Almeida, P., Cunha, J., & Aguiar, A. (2017). Heart rate variability metrics for fine-grained stress level assessment. Computer methods and programs in biomedicine, 148, 71-80.

[79]. Daniel Kahneman, Jackson Beatty, "Pupil Diameter and Load on Memory", Science 23: Vol. 154, Issue 3756, pp. 1583-1585, 1966.

[80]. Y. Sasaki, "The truth of the F-measure", Teach Tutor Mater, vol. 1, no. 5, 2007.

[81]. Hart, S. G. (2006, October). NASA-task load index (NASA-TLX); 20 years later. In Proceedings of the human factors and ergonomics society annual meeting (Vol. 50, No. 9, pp. 904-908). Sage CA: Los Angeles, CA: Sage publications.

[82]. Medeiros, J., Couceiro, R., Duarte, G., Duraes, J., Castelhano, J., Duarte, C., ... & Teixeira, C. (2021). Can EEG Be Adopted as a Neuroscience Reference for Assessing Software Programmers' Cognitive Load?. Sensors, 21(7), 2338, 2021.

[83]. Crk, I., & Kluthe, T., "Assessing the contribution of the individual alpha frequency (IAF) in an EEG-based study of program comprehension", 38th Annual Int. Conf. of the IEEE Engineering in Medicine and Biology Society (EMBC) (pp. 4601-4604). IEEE, 2016.

[84]. Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., ... & Saurous, R. A. (2017). Tensorflow distributions. arXiv preprint arXiv:1711.10604.

[85]. H. Hijazi, J. Cruz, J. Castelhano, R. Couceiro, M. Castelo-Branco, P. d. Carvalho and H. Madeira, iReview: An Intelligent Code Review Evaluation Tool using Biofeedback, in the 32nd International Symposium on Software Reliability Engineering (ISSRE 2021), 2021

[86]. Jain, Leena and Satinderjit Singh. "Designing The Code Snippets for Experiments on Code Comprehension of Different Software Constructs." Int. Journal of Computer Sciences and Engineering (2019).

[87]. F. Shull, V. Basili, B. Boehm, A. W. Brown, et al, "What We Have Learned About Fighting Defects", METRICS '02: Proceedings of the 8th International Symposium on Software Metrics, June 2002

Haytham Hijazi is a Ph.D. Research Fellow with the Center for Informatics and Systems, University of Coimbra (CISUC), Portugal. His research interests include Artificial Intelligence, Biofeedback, and Software Engineering. His current thesis work is focused on intelligent biofeedback systems for augmenting content comprehension.

Joao Duraes has been with the Software and Systems Engineering research group of the Centre for Informatics and Systems of the University of Coimbra since 1994. His research is mainly focused on software reliability, and his contributions include novel techniques and models concerning software fault injection, dependability benchmarking, and security evaluation.

Ricardo Couceiro has been a Research Member of the Center for Informatics and Systems, University of Coimbra (CISUC) Since 2006. His research interests include biomedical signal processing, pattern recognition, and modeling applied to the analysis of cardiovascular systems.

João Castelhano Joao Castelhano is a Biomedical Engineer with a PhD in Health sciences. He is currently a researcher at CIBIT/ICNAS - University of Coimbra and is an expert in multimodal medical imaging data acquisition and analysis.

Raul Barbosa is an Assistant Professor at University of Coimbra, where he is a member of the Software and Systems Engineering group. His main research interests focus on the design of dependable distributed systems, resilience in cloud systems, software intensive systems, and assessment by complementing experimental approaches with formal methods.

Júlio Medeiros is currently a Ph.D. student at the University of Coimbra. His research interests include Signal Processing, Biofeedback, and Machine Learning applied to understanding and mitigating human error in software development tasks.

Miguel Castelo-Branco is the Director of the Coimbra Institute for Biomedical Imaging and Translational Research (CIBIT). He has been the Director of IBILI, a leading Vision Research Institute in Portugal. He is the Scientific Coordinator of the National Functional Brain Imaging Scientific initiative. He was also the Director of ICNAS, Medical Imaging Infrastructure, University of Coimbra. His achievements are well reflected in publications in top General Journals, such as Nature and PNAS.

Paulo de Carvalho is currently an Associate Professor with Habilitation with the University of Coimbra. His main research interests include intelligent algorithms for personal health solutions and clinical decision support systems. His publications include over 250 papers in refereed international journals and conferences. He has been involved in several organizations and program committees of international conferences in the health informatics domain and he serves as editorial board member in several scientific journals. He is the Director of the Working Group on Health Informatics of the IFMBE.

Henrique Madeira is Full Professor at the University of Coimbra, where he has been involved in research on dependable computing since 1989. He has coordinated or participated in dozens of research projects funded by the Portuguese Government and by the European Union. His current funded projects are in the areas of software quality and software reliability, verification and validation, human factors in software engineering, and safety and security evaluation Artificial Intelligence in safety-critical functions.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/