Predicting Bugs by Monitoring Developers During Task Execution

Gennaro Laudato*, Simone Scalabrino*, Nicole Novielli†, Filippo Lanubile†, Rocco Oliveto*
*STAKE Lab @ University of Molise, Italy
†University of Bari, Italy

***Abstract*—Knowing which parts of the source code will be defective can allow practitioners to better allocate testing resources. For this reason, many approaches have been proposed to achieve this goal. Most state-of-the-art predictive models rely on product and process metrics, *i.e.,* they predict the defectiveness of a component by considering *what* developers did. However, there is still limited evidence of the benefits that can be achieved in this context by monitoring *how* developers complete a development task. In this paper, we present an empirical study in which we aim at understanding whether measuring human aspects on developers while they write code can help predict the introduction of defects. First, we introduce a new developer-based model which relies on behavioral, psycho-physical, and control factors that can be measured during the execution of development tasks. Then, we run a controlled experiment involving 20 software developers to understand if our developer-based model is able to predict the introduction of bugs. Our results show that a developer-based model is able to achieve a similar accuracy compared to a state-of-the-art code-based model, *i.e.,* a model that uses only features measured from the source code. We also observed that by combining the models it is possible to obtain the best results (∼84% accuracy).**

***Index Terms*—bug prediction, human aspects of software engineering, biometric sensors, empirical software engineering**

I. Introduction

Quality assurance is of paramount importance in software development. While activities such as code reviewing or testing allow developers to increase the quality of software products, bugs are still inevitable. Predicting which components will likely be affected by bugs in the near future would allow practitioners to dedicate attention to such components, thus (i) reducing the costs of failure consequences and defect fixing, as well as (ii) improving the overall software quality. To this aim, many automated techniques have been proposed to *predict* the presence of defects in the source code [1]–[7].

The currently available defect prediction techniques are mainly based on machine learning [8]–[10]. A binary classifier is trained on past data to automatically predict if new software components will contain bugs. These approaches leverage features classically derived from *product metrics*, which measure characteristics of the source code (e.g., LOC) [11], and *process metrics*, which capture aspects related to the development activities (e.g., number of changes, or number of developers who applied changes) [6]. While such factors are important, the aforementioned approaches fail to capture the human aspects relevant when developers introduce bugs.

Therefore, some studies have tried to shift the focus on the developer's side [2], [5], [7] by proposing novel metrics able to capture the human factors behind writing and maintaining code. However, such metrics are mostly computed by mining software repositories, and they only focus on *what* the developers did, rather than on *how* they did it. The latter aspect, neglected in previous work, is of utmost importance: For example, previous work showed that developers who experience fatigue may be more likely to introduce bugs [12].

In this paper, we present an empirical study in which we aim at understanding to what extent human aspects that can be monitored and measured *while programming* can help predict the introduction of defects in the source code. To achieve this goal, we first build a conceptual framework including *behavioral*, *psychophysical*, and *control* factors, which we conjecture could help to determine the outcome of a development task in terms of presence of errors (*i.e.,* bugs). We operationalize these factors through a set of metrics that can be concretely assessed during software development using behavioral measurements (e.g., the counts of keystrokes and mouse clicks), biometrics (e.g., the developers' heart-rate or attention level) or by acquiring contextual information (e.g., the developers' programming experience).

We ran a controlled experiment involving 20 software developers who were monitored while performing coding tasks of different difficulty levels and, specifically, two *implementation* tasks and two *bug-fixing* tasks. We used the collected data to compare the predictive power of two machine learning models: one based on our framework (developer-based) and one based on state-of-the-art product metrics (code-based).

Our results show that a developer-based model, alone, does not outperform the code-based baseline, achieving an accuracy of 76% (vs 79% obtained by the baseline). However, we observe that a model that combines developer- and code-based features achieves the best results (84% accuracy). Some behavioral and psycho-physical features, such as the ones based on keystrokes and heart-rate, resulted to be important predictors. Therefore, we envision that the definition of training sessions aimed at improving the underlying cognitive processes related to such factors might help developers increase the quality of their code. Also, it would be possible and beneficial, in the future, to train developers to become self-aware of the conditions in which it is better for them to take breaks or completely stop writing code to reduce the risk of introducing errors.

To summarize, we make the following contributions:

- We define the first bug prediction model based on a comprehensive set of developer-based factors acquired while a task is being performed;
- We present the results of a controlled experiment based on a rigorous protocol to assess how a model relying on developer-based features compares with one depending on code-based features;
- We release the replication package of this study including the study protocol, the scripts, and the dataset to foster future studies in this field.

The remainder of this paper is organized as follows: Section II reports the related work on bug prediction and biometrics in software engineering; Section III describes our theoretical framework and its operationaization; in Section IV-A we describe the study design and protocol, present the empirical results, and discuss their implications. Finally, we report the threats to validity and discuss the ethical issues in Section VI-C, and conclude the paper in Section VII.

II. Related Work

A. Bug Prediction

The scientific community has devoted much effort to investigating the problem of automatic bug prediction. D'Ambros *et al.* [13] conducted a large comparison of bug prediction methods that rely on the usage of products and process metrics. The results showed that there is not a technique that globally performs better than the others. Along this line, Palomba *et al.* [8] designed a bug prediction model for smelly classes based on machine learning. Specifically, they proposed combining predictors based on product and process metrics with a measure of code smell intensity. The findings revealed that this combination of information increased the accuracy of the bug prediction model, with a +13 % gain in F1-score, compared to the state-of-the-art.

Pandey *et al.* [10] performed a large machine-learning study aimed at designing and evaluating a bug-prediction model capable of dealing with the class imbalance problem. Indeed, the class imbalance is a major problem for research studies in this field. Pandy and colleagues proposed a classification framework composed of a sequence of deep-learning and two ensemble learning layers. Specifically, these latters were the layers in charge of dealing with the class imbalance problem in software bug prediction. This mixed model allowed to achieve better results than most state-of-the-art approaches.

Recently, Qu *et al.* [14] presented a study where they investigated the relationship between the number of software developers and the probability that a file contains buggy code. To this aim, they included nine open-source systems and observed that when multiple developers worked on the same source code file there is a higher chance that the file contains buggy code. Considering this trend, they designed an unsupervised method in the context of effort-aware bug prediction. The results showed prediction performance comparable or better to other supervised baseline approaches).

Ferenc *et al.* [15] released a novel dataset, namely the BugHunter Dataset, which was built according to a different perspective: the data indeed was acquired by considering the buggy and the fixed states of the same source code components in terms of time frames and therefore not in terms of the typical release versions. The authors designed a bug prediction experimentation on this dataset with bug characterization metrics at three source code levels: file, class, and method. This study revealed that method-level metrics show the highest performances in the classification. Indeed, the best model achieved an F-measure value over 0,74.

To the best of our knowledge, this is the first work that aims at predicting the presence of bugs in source code by including consideration of biometric features.

B. Biometrics in Software Engineering

In recent years, the software engineering research community started to use physiological signals to investigate the relationship between developers' cognitive and affective states and several aspects of software development, such as code comprehension [16], productivity [17], and software quality [18]. Such studies include consideration of several biometrics, including hearth-related metrics, the measurement of electrodermal activity (EDA), electroencephalography (EEG), electromyography (EMG), or eye-tracking.

Code comprehension and task difficulty Fritz and colleagues [19] combined features based on eye-tracking, EDA and EEG, to predict the difficulty of a task as perceived by developers during and after a programming task. Their study demonstrates that it is possible to predict whether a new developer will experience difficulties in a code comprehension task with a precision of 70% and a recall 62%. Parnin [20] studied the complexity of programming tasks by relying on the analysis of sub-vocal signals. The study shows how EMG correlates with cognitive patterns involved in dealing with easy and hard programming tasks. Fucci et al. [21] employ lightweight biometric sensors for EEG, EDA, and heart-related measurements to distinguish between code and natural language comprehension tasks, reporting a 90% accuracy.

Emotions and perceived productivity Recent research also investigated the relationship between physiological measures and developers' productivity. Radevski et al. [17] proposed a framework for continuous monitoring of developers' perceived productivity based on brain electrical activities. Müller and Fritz [18] used a combination of different biometric measurements to predict self-assessed progress and *interruptibility* of developers while programming. They also demonstrated that it is possible to recognize developers' emotions using a combination of EEG-, eye-, and heart-related metrics with an accuracy of 71%. The progress experienced by developers was predicted at a similar rate, but using a different set of biometrics (i.e., EDA signal, skin temperature, brainwave frequency, and pupil size). The findings of the study by Müller and Fritz [18] have been confirmed and extended by a recent replication performed by Girardi et al. [22].

They investigated the range and triggers of emotions that software developers experience while programming and observed a positive correlation between emotional valence — i.e. the (un)pleasantness of the emotion stimuli — and the self-reported progress, in line with previous studies [18], [23]. Furthermore, they demonstrate how emotion recognition is feasible using a minimal set of biometric sensors for EDA and heart-related metrics. Their machine-learning classifier can detect valence and arousal — i.e. the emotional activation, ranging from excited to relaxed — with an accuracy of 71% and 68%, respectively. Girardi et al. [24] performed further investigation on the use of biometrics for emotion recognition at the workplace. In their field study, they experiment with a minimal set of non-invasive biometric sensors including EDA and hearth-related metrics as collected by the Empatica E4 wristband during the daily working activities of professional developers from five different companies. While promising, the performance of their models is not yet robust enough for practical usage, thus calling for further data collection and individual fine-tuning of emotion models.

Identification of code quality concerns Müller and Fritz [25] also investigated the use of EDA, EEG, and heart-related biometrics for real-time identification of code quality concerns, i.e., low-quality code containing bugs. The authors provide evidence that biometrics can outperform traditional code-related metrics to identify quality issues in a codebase. Differently from their work, in this paper (i) we define a framework that aims at generalizing the aspects beyond biometrics by also including behavioral aspects; (ii) we focus on bugs, while they focus on code quality concerns (which are more generic); (iii) we run a controlled experiment, while they conducted a field study.

III. Conceptual Framework

We develop a conceptual framework for the analysis of factors influencing the probability of introducing a bug. The framework is built upon the evidence provided by relevant literature through an informal literature review. Starting from a seed set of papers in the field we were aware of, we snowballed and identified other relevant papers based on which we determined a set of *factors*. Then, we operationalize each factor by introducing a set of metrics that we use as candidate predictors for our machine learning study (see Section IV-A).

A. Modeling the Factors

Building upon the evidence provided by relevant literature, we develop a conceptual framework for the analysis of factors influencing the probability of introducing a bug. Figure 1 depicts the framework and summarizes the metrics which we used to operationalize them, as explained in Section III-B. We identify three families of factors: *behavioral* factors, describing what the developers do during the task, *psychophysical* factors, describing the physical, cognitive and emotional condition of developers during the task, and *control* factors, describing incidental characteristics, including the context in which developers perform the task and their experience.

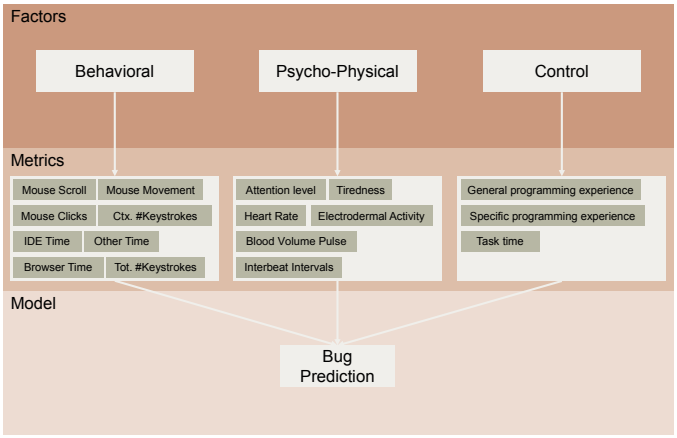

Fig. 1: Conceptual framework of developer-related factors influencing the introduction of bugs.

1) ***Behavioral Factors****:* We conjecture that what developers *do* during a task can provide some important hints that could allow predicting the possible introduction of a bug while completing a development task. Developers perform several activities while they write code: they type, they browse the web, they possibly get distracted by notifications. To capture such pieces of information, we focus on the developers' *activity* and on their *actions*.

We capture the high-level **activity** that the developer is performing (browsing the web, writing code, or something else). For example, if a developer uses the browser more than the IDE, it might be the case that he/she is experiencing some problem in understanding the problem at hand or how to use a specific technology.

The low-level **actions** performed by the developer while completing a task may be indicative of their cognitive effort. Basic actions of developers include typing, moving the pointer, and scrolling. The typing behavior may provide important clues: For example, typing incessantly may indicate that the developer is not spending enough time checking the code or at thinking at the solution, as it has been observed in previous work [26]. While moving the mouse and scrolling do not result in the production of source code, they may provide interesting pieces of information as well: For example, frequent scrolling may indicate lack of visual attention, which, in turn, might be a symptom of a lack of focus [27].

2) ***Psychophysical Factors****:* Constraints on time, quality, and cost can introduce time pressure and stress, and reduce the emotional well-being at work. [28]–[31]. Physiological aspects could help in detecting the state in which the programmer is, which may impair the completion of a task. The psychophysical state may both depend on the task and external factors (*e.g.,* developer's private life or work conditions): developers may feel frustrated if they cannot understand a specific piece of code or they may be stressed because an important deadline is approaching. Such kinds of conditions may have an impact on the performance. We identified three psychophysical factors: *stress*, *affective states*, and *concentration*.

Stress can be broadly defined as *an actual or anticipated disruption of homeostasis or an anticipated threat to well-being* [32]. In 1993, the authors of one of the first works in the literature on stress in Software Engineering [33] propose a new perspective on how this factor may affect the behavioral dynamics in the information system development. They state that stress - with high probabilities - is one of the main corrosive factors for a software developer. Stressor-related information from all major sensory systems is conveyed to the brain, which recruits neural and neuroendocrine systems (effectors) to minimize the net cost. The physiological response to stress involves an efficient and highly conserved set of interlocking systems and aims to maintain physiologic integrity even in the most demanding of circumstances. The autonomic nervous system provides the most immediate response to stressor exposure — through its sympathetic and parasympathetic arms, which provoke rapid alterations in physiological states through neural innervation of end organs. Stress can be detected from biomedical low-invasive sensors. Among others, heart-related measures [34], [35] or their temporal trend [36] were leveraged. It is generally accepted that the activities of the Autonomic Nervous System, which consists of the Sympathetic (SNS) and Parasympathetic nervous systems (PNS), are reflected in the low-frequency (LF) and high-frequency (HF) bands in heart rate variability (HRV) [35]. Some studies support the use of several different physiological factors: Vinkers *et al.* [37] use the body temperature to detect stress, while Suess *et al.* [38] observed that an anomalous respiratory activity (*e.g.,* hyperventilation) may indicate stress as well. Finally, changes in the electrodermal activity due to sweating is also leveraged for the stress assessment [39].

Affective states refer to the emotional response triggered by an internal (e.g. feeling of failure) or external event (e.g. peers ask for help). In line with previous work on emotions in software development, we model emotions along continuous dimensions [22], [24], [40], [41]. In particular, we adopt the Russell's Circumplex model [42], which describes emotions in terms of valence (pleasantness of the stimuli) and arousal (level of activation) dimensions. Recent work shows how unpleasant emotions correlates to developers' unhappiness, causing the production of software with lower quality than expected [40]. High arousal, instead, could be a symptom of time pressure and might negatively impacts the quality and the efficiency of the development activities [43]. Conversely, happy software developers achieve better performance [44]. The link between positive emotions and self-assessed productivity was further confirmed by recent lab [18], [45] and field [24] studies.

Concentration concerns the ability to think carefully about something the developers are doing and nothing else. Motivated by the fact that navigating through code and searching for relevant information requires a lot of developer time, Ahrens *et al.* [46] use eye tracking to record and transfer developers' attention during software maintenance. The output is offered to the developers in form of a heat map of attention levels. Results showed that this attention representation helped some of the participants for orientation and code finding purposes. However, the majority rated them as barely helpful or even not helpful thus calling for future research in this direction. Focus is also related to concentration, and it has been recently studied by Soto *et al.* [47]. The authors performed an experiment involving 14 professional knowledge workers in their workplace over an eight-week with a large variety of biometric measurements (e.g., Heart Rate, Skin Temperature, Respiration Rate). The results showed that the focus can be predicted with an overall accuracy equal to 67%. The concept of staying focused was also reported in recent work by Pilzer *et al.* [48]. The authors conducted a formative study with 18 professionals in which they examined their computer-based and eye-gaze interaction with the window environment and devised a relevant model of open windows. The results showed that the devised model was able to predict the relevance of open windows with an accuracy of 72.7%. Finally, interruptibility also relates to concentration. According to Zuger *et al.* [49], knowing a person's interruptibility allows optimizing the timing of interruptions and minimizing disruption. The authors conducted a two-week field study with 13 professional software developers to investigate a variety of computer interactions such as heart-, sleep-, and physical activity-related data. Their analysis showed that computer interaction data is more accurate in predicting interruptibility at the computer than biometric data (74.8% vs. 68.3% accuracy), and that combining both yields the best results (75.7% accuracy).

*3) **Control Factors**:* The performance of a developer may depend on aspects which are not directly of interest to the study's aims, but that could still influence the outcomes. Specifically, we focus on two factors that previous work showed some correlation with the probability of introducing bugs [50]:

Developers' experience. All else being equal, novice developers may introduce bugs more often than expert developers [50], [51]. This could not always be true because there are many factors that may contribute to the production of source code. What is often the case however is that novice developers take more time to complete the given task. Therefore, considering the protocol of our experiment, the same time is given to every participant. This could provoke that less experienced developers may introduce more bugs.

Task Time. The time at which a task is performed may influence the performance [50]: some developers may work better in the afternoon (or, even, in the night), while others may prefer working in the morning.

B. Measuring the Factors

In the following, we describe the metrics we use to operationalize the factors in our framework.

1) Behavioral Features: To capture the users' *behavior* while developing (*i.e.,* keystrokes, mouse movements, active window at a given time), we use the software *activity-tracker* [52].

Specifically, we capture the *activity* performed by the developer by measuring the time a window is active and on the participant's laptop. Then, we use some simple rules to determine the type of activity performed by the developer.

The activity-tracker tool records the time intervals in which each window received focus and the process that is running behind the window. We focus on two main categories of activities: browsing the web and working in the IDE. Therefore, we define three features that capture the *usage context* (UC): (i) **Browser Time (UC$_b$)**, *i.e.,* the percentage of time spent in a browser window; (ii) **IDE Time (UC$_i$)**, *i.e.,* the percentage of time spent in the IDE used for the task; (iii) **Other Time (UC$_o$)**, *i.e.,* the percentage of time spent in other windows, such as the desktop.

To capture the low-level *actions* made by the developer, we keep into account all the interactions that a user may have with the computer. Specifically, we use the following features: (i) **Global Number of keystrokes (KS)**, which we measure by simply counting the number of keys pressed by the developer during the whole task; (ii) **Contextual Number of keystrokes (KS$_c$)**, which we measure by counting the number of keys pressed by the developer in three different contexts, *i.e.,* (while using the IDE (KS$_i$), while using a web browser (KS$_b$), and while using other programs (KS$_o$); (iii) **Mouse Cursor Movement (MCM)**, that we measure by tracking the length of the path traveled by the mouse cursor during the task (in number of pixels); (iv) **Mouse Clicks (MC)**, *i.e.,* the number of clicks made by the developer during the task; (v) **Mouse Scroll (MS)**, which we measure by tracking the number of pixels scrolled in any direction during the task.

2) Psychophysical Features: To capture *psychophysical* aspects, we use wearable devices that record biometric signals while developers are performing the tasks. In line with previous studies [18], [22], [24], we use the Empatica E4 wristband [53] to capture the heart- and skin-related metrics. Empatica is a wearable device (in the form of a wrist-worn bracelet) that offers real-time physiological data acquisition. As for brain-related metrics, we use the BrainCo helmet [54]. It is equipped with a single frontal electrode for EEG acquisition and also offers aggregated metrics, such as attention and meditation.

First, we need the signal s in a period in which the developer is completely relaxed: this allows us to calibrate the features for the specific person and to avoid that specific physiological conditions (*e.g.,* a naturally slower HR) make the feature only valid for a single developer or a specific category of developers. Given a signal s obtained during the task and the same signal s_r obtained while relaxing, we first remove outlier samples, possibly due to movement artifacts, by using a robust outlier detection technique [55]. Then, we compute two types of features. First, we extract generic descriptive statistics: we compute the mean and the variance of the signals: we do this to give the machine learning model a rough idea of the state of the developer. If a developer is particularly stressed on a specific day, their average heart rate may be higher than usual, on average, during the whole task. Then, we compute 10 features that describe the relative distribution of the signals: given a signal that ranges between a minimum value m and a maximum value M, we divide the range $[m, M]$ in 10 equally-sized bins and, for each of them, we compute the percentage of the samples of the signals that fall in the specific bin.

This allows the model to have an idea of the distribution of the signal. For example, if the signal ranges between 1 and 10, the first bin (b_1) will contain samples in the range $[1, 2)$, b_2 will be related to the range $[2, 3)$, and so on. We extract the features previously described in two separate contexts: (i) for the whole task, and (ii) just for the last minute of activity. The data about the whole task of the signal gives the model a broad idea of the shape of the signal. On the other hand, the data about the last minute could help get more interesting information: such data may help capture a state of relief or stress at the end of the task.

To capture the *stress* level, we keep into account the following aspects:

Heart Rate (HR). This simple measure allows us to have a rough idea of the psychophysical state of the developer; a high HR possibly indicate anxiety, while a low HR indicates calm. Given the HR signal, we extract the features for the whole task, *i.e.,* HR$_{mean}$, HR$_{var}$, and HR$_j$ (for j between 1 and 10), and the features for the last minute, *i.e.,* HR$^{last}_{mean}$, HR$^{last}_{var}$, and HR$^{last}_j$ (for j between 1 and 10) as previously described.

Electrodermal Activity (EDA). This measure—together with heart related information— can be a reliable indicator of the stress level [56]. Given the EDA signal, we divide it in phasic (EDAP) and tonic (EDAT) using the algorithm proposed by Greco *et al.* [57]. Therefore, given such two signals, we extract the features for the whole task, *i.e.,* EDAP$_{mean}$/EDAT$_{mean}$ EDAP$_{var}$/EDAT$_{var}$, EDAP$_j$/EDAT$_j$ (for j between 1 and 10), and the features for the last minute, *i.e.,* EDAP$^{last}_{mean}$/EDAT$^{last}_{mean}$ EDAP$^{last}_{var}$/EDAT$^{last}_{var}$, EDAP$^{last}_j$/EDAT$^{last}_j$ (for j between 1 and 10). To estimate the number of times in which the stress starts to increase, we introduce an additional feature that extracts the number of times that the first derivative of the signal is positive and greater than an ϵ which allows to be tolerant to small errors by the sensor. We compute such a feature for both the signals, both for the whole task (EDAP$_{incr}$ and EDAT$_{incr}$) and for the last minute (EDAP$^{last}_{incr}$ and EDAT$^{last}_{incr}$). We use $\epsilon = 0.01$.

Interbeat Intervals (IBI). The heart activity is an indicator of stressful situations. Therefore, we used also a finer observation of Heart Rate Variability (HRV), such as the IBI information. The variation between successive heartbeats is low whether the system of a human is in more of a fight-or-flight state. The variation between beats is high if one is in a more comfortable state [58]. In particular, we used aggregated data such as the mean and the variance [59].

Blood Volume Pulse (BVP). BVP represents the phasic change in blood volume that corresponds to each heart-beat interval [60]. According to Greene *et al.* [59], BVP is one of the available measures for stress. We involved this measure—available from the smartwatch—in our analysis.

While such features are interesting to capture on their own, they may influence more or less the actual result depending on how hard the developer was working while in a stressed state. For example, a state of stress occurring while trying to understand the problem may impact less the end result than a state of stress occurring while writing code.

Therefore, we computed a set of features that weights keystrokes by the stress-related signals previously described. We call such features **Weighted keystrokes (WKS$_c$)**: for each 1-minute period, we measure the stress-related signals, we normalize and average them, and we use the resulting value as a weight for the number of keystrokes typed in that period. Finally, we sum all such weighted keystrokes to obtain the final WKS$_c$ value. We do this using the signals related to heart rate (WKS$_{HR}$), electrodermal activity (WKS$_{EDAT}$ and WKS$_{EDAP}$), interbeat intervals (WKS$_{IBI}$), and blood-volume pulse (WKS$_{BVP}$). To capture the *concentration* level, we rely on two factors:

Attention level (ATT). We use an attention measure provided by the BrainCo device. Such a device uses a proprietary algorithm to combine several brain wave signals to determine the attention level of the user at a given time. Such a measure ranges between 0 (completely relaxed) and 100 (very concentrated). Also for such a signal, we compute the metrics as previously described. Specifically, we compute: ATT$_{mean}$, ATT$_{var}$, ATT$_j$ (for j between 1 and 10), ATT$^{last}_{mean}$, ATT$^{last}_{var}$, ATT$^{last}_j$ (for j between 1 and 10).

Tiredness (TIR). We ask the developers to self-assess their tiredness on a Likert scale from 1 (not tired) to 9 (completely tired) both before and after each the task. Therefore, we use two features based on such a self-assessment: TIR$_{before}$ and TIR$_{after}$. The presence of both the features allows the model to implicitly estimate also to what extent the task tired the developer. We also use a binary feature, TIR$_{first}$, that indicates whether the task was the first of the day (*true*) or the developer completed another complex task before the task at hand (*false*).

Finally, we use the SAM [61] questionnaire[1] to capture the *emotional state* of the developers. We do this both *before* and *after* each the task. SAM provides that each characteristic of the PAD model (pleasure, arousal, and dominance) is represented through the use of a graphic character arranged along a discrete scale. As for the pleasure, the SAM starts from an initial configuration formed by a smiling and happy figure up to a frowning and unhappy figure. As for the excitement, instead, the SAM starts from a figure of a sleepy character with closed eyes until the excitement, represented by the same character with open eyes. The dominance scale shows the SAM ranging from a very small figure — which aims to represent a feeling of not having the situation under control — to a very large figure, which represents a feeling of control or power. We use three features measured on a 9-point Likert scale [62].

3) Control Features: We conjecture that there are several ways in which *developers' experience* can be captured. In this case, we focused on the programming experience and we asked the developers to report the number of years of programming experience in two contexts: *general programming experience* (regardless of the programming language) (PE$_{gen}$), and *specific programming experience* (on the specific programming language used to complete the task) (PE_{spec}).

[1] The Self-Assessment Manikin (SAM) is a technique of non-verbal pictorial evaluation that specifically tests the enjoyment, excitement, and dominance associated with the affective response of a person to a wide range of stimuli.

Finally, we model the time of the day at which a task is completed (*task time*) as either as either *AM* (from 7 AM to 12 PM) or *PM* (from 12 PM to 7 PM).

4) Combined Features: We opted for the creation of a new set of features with the aim of increasing the knowledge of the data set and improving the classification performances. The new set of features has been defined by weighting the number of typed keys by all the features measured using Empatica device (including the temperature). Consider the number of keys $K_{s,e}$ typed in a time slot $[s, e]$. Consider, then, a given Empatica feature $f_{s,e}$ computed in the same time interval. We compute the number of keystrokes weighted by f using the following formula:

$$K^f = \sum_{(s,e)\in KS}^{n-1} K_{s,e} norm(f_{s,e})$$

where KS contains the couples of start and end time s and e for which the key tracker recorded the number of keystrokes (a registration for each minute of activity), and $norm$ is a function that normalizes the f feature between 0 and 1 by simply using the formula $norm(x) = \frac{x-min}{max-min}$. The features f we consider are all the ones recorded by Empatica, *i.e.,* the Blood Volume Pulse BVP, the Interbeat Interval IBI, the Hearth Rate HR, and the two EDA features (*i.e.,* phasic and tonic). We also included three additional features in which we measure the number of keystrokes divided by the environment in which the participant was (IDE, web browser, or other).

As a first step, we measure the signals and the metrics we need to compute the features. Then, we extract the feature as previously described. Given a set of labeled data for which we have both (i) the features we use, and (ii) the task result (*correct* or *buggy*) we selected the best features and train a classifier, and then we use the related model to predict the outcome.

IV. Controlled Experiment

A. Research Questions

The *goal* of our study is to understand if developer-based features, which are measured with non-invasive sensors, allow to achieve better results than a state-of-the-art model. Our study is steered by the following research questions:

- RQ_1 *Do developer-based features allow to achieve better results than code-based features?* With this research question we want to assess to what extent the developer-based features in our framework can be used to predict if the developer will introduce at least a bug in the code. Furthermore, we want to compare the performance achieved by a model leveraging developer-based features with the performance achieved using code-based features that are used in the literature;
- RQ_2 *To what extent combining developer- and code-based features improve the performance of bug prediction models?* With this research question we want to understand if a model obtained combining our new features with the code-related features of the baseline used to answer RQ_1 allows achieving a higher classification accuracy.

B. Experiment Design

To answer our research questions, we conducted a controlled experiment in which we invited 22 developers to complete 4 programming tasks. Our study includes both *subjects* (developers) and *objects* (tasks to be performed). As for the subjects, we involved a group of software developers composed of both students at the University of Molise, Italy (PhD candidates, master, and bachelor students) and professional developers. In total, we involved 22 developers. As for bachelor students, we only involved those that passed the Java exam in the first year of the degree course in Computer Science at the University of Molise, Italy. To avoid involving young developers with limited experience, we posed an additional constraint for the second-year students (the youngest involved in our study), *i.e.,* we only invited the ones that passed the exam with the highest score (30/30). We had to discard two of the invited developers as we lost part of the biometric signals due to a memory problem with one of the devices. Overall, our pool of participants include 20 developers. More than half of the participants (13) are young developers (bachelor students at the 2^{nd} or 3^{rd} year), while five and three are master students and PhD students, respectively, and one is working in the industry. 36.4% of the developers have at least 5 years of programming experience.

As for the objects, we first selected four problems from *LeetCode* [63], an online platform commonly used by developers to exercise for coding interviews. LeetCode allows to access a wide range of problems and provides a mechanism for validating a solution: given the source code of the solution, the platform runs several test cases (depending on the problem) and reports the number of failed tests. A solution is *accepted* if all the test cases pass. Furthermore, for each problem, LeetCode reports the acceptance rate, *i.e.,* the percentage of submissions by the community that was accepted/rejected by the platform. We used the acceptance rate to distinguish between *easy* and *hard* problems to include in our study. We selected two *easy* problems, i.e., with an acceptance rate greater than 70%. and two *hard* problems, i.e. with an acceptance rate between 50% and 60%. Among the candidate problems, we selected the ones that, according to our preliminary assessment, could be completed in about half an hour. We run a pilot study with two participants to assess whether (i) the tasks were feasible, and (ii) the given time was sufficient. Both the participants were able to complete the tasks on time and declared that the tasks were feasible. At this stage, we did not check if their solutions were correct.

Starting from the selected problems, we defined four tasks, two implementation tasks, and two bug-fixing tasks. The implementation tasks consisted in solving the problem from scratch. We randomly chose two of the problems selected from LeetCode to define such tasks, one easy and one hard. The bug-fixing tasks required the developers to read a partial solution to the problem that contained at least a bug and to fix all the bugs. Two of the authors developed a partial solution for the two remaining problems. We report in Table I the list of the tasks we used.

TABLE I: Tasks selected from LeetCode for the controlled experiment.

Difficulty	Type	Problem name	Acceptance rate
Hard	Implementation	Roman to Integer	55.1%
Hard	Bug Fixing	Camelcase Matching	56.1%
Easy	Implementation	Split a string in balanced strings	82.4%
Easy	Bug Fixing	Robot return to origin	73.2%

We controlled the following variables: (i) **Task type**: Each developers was asked to complete two bug-fixing and two implementation tasks; (ii) **Task difficulty**: Each developer was asked to complete two easy and two hard tasks; (iii) **Time of the day**: We made sure that each developer performed exactly two tasks in the morning and two tasks in the afternoon.

In line with this design, we defined eight groups, and we divided the developers into such groups to avoid any of the previously described variables influenced the results. In dividing the developers into groups, we balanced them based on their education level (*i.e.,* we avoided groups with all Master students and groups with all first-year Bachelor students). To mitigate the risks due to fatigue, which could negatively affect the performance, the developers completed two tasks in a session and two tasks in another session, scheduled on a different day. The order of the tasks in each group, however, was always the same. The participants were asked to complete the tasks by using the Java programming language. They were provided with a laptop with IntelliJ IDEA Community Edition and the software needed for tracking their activities already installed. Also, they were asked to wear the two devices needed to capture their psychophysical-related signals (*i.e.,* Empatica and BrainCo). We submitted questionnaires to the participants both before/after the whole experiment and before/after each task. Finally, since some psychophysical metrics required baseline measurements, we asked the developers to watch a relaxing video for two minutes before starting the task, to acquire baseline biometrics. This is in line with consolidated practice in research using biometrics in software developemnt [18], [24], [45].

C. Data Collection

We collected (i) the biomedical signal data, (ii) the activity tracker recordings, (iii) the contextual information, and (iv) responses given by the developers to the questionnaires. These data was used to compute the metrics we defined in section III-B. We labeled each completed task according to the outcome as either *no-buggy* or *buggy*, leveraging LeetCode. To do so, we run the solution provided to each problem, and we checked if the platform reported any bugs. In presence of failure for at least a test case, we marked the instance as *buggy*, while we labeled it *no-buggy* otherwise. It is worth mentioning that, for some problems, LeetCode might report a negative result only because the proposed solution does not meet some pre-defined performance requirements. In our case, however, we specifically excluded tasks for which such requirements

(and, thus, tests) were present so that a failure indicates that at least a *functional* problem exist.

D. Machine Learning Process

We trained and tested several machine-learning classifiers in different settings, as explained in the following. We did this for the different sets of features we take into account (*i.e.,* developer-based, code-based, and combined). Our pipeline consists of three pre-processing steps and a final model building step.

Step 1: oversampling. The first step consisted in using a oversampling technique, SMOTE [64], for generating synthetic instances to balance the training dataset.

Step 2: correlation. As a second step, we ran a correlation analysis between all the pairs of features, and we discarded the ones with Pearson correlation greater than 0.95.

Step 3: feature selection. To further eliminate features that would not contribute to the prediction, in the third step we performed feature selection using a wrapper technique, which selects the best subset of features for a specific machine-learning classifier based on the accuracy achieved by using them. We experimented with seven algorithms: Random Forest (RF) [65], Logistic Regression (LR) [66], Support Vector Machine (SVM) [67], AdaBoost [68], Stochastic Gradient Descent (SGD) [69], Passive Aggressive Classifier (PAC) [70], Extra Trees Classifier (ETC) [71].

Step 4: model building. After having selected the most relevant features, we trained and tested several machine-learning classifiers. Specifically, we made this step configurable with 11 machine-learning algorithms: Random Forest (RF) [65], Multilayer Perceptron (MP) [72], Logistic Regression (LR) [66], Support Vector Machine (SVM) and Linear SVM [67], K Nearest Neighbours (KNN) [73], Gaussian Naive Bayes (GNB) [74], Stochastic Gradient Descent (SGD) [69], Decision Tree (DT) [75], Bagging Classifier (BC) [76], and Gradient Boosting Classifier (GBC) [77].

For each model, we tested all the possible combinations of configurations. The first two steps could only be used or not used (2 configurations each), while the third step had 8 configurations (7 algorithms, plus a configuration in which it was not used), and the fourth step had 11 configurations. In total, for each model, we trained and tested 352 machine-learning models. Given a starting model and the resulting 352 machine-learning models, we picked the one which achieves the best results in terms of overall accuracy.

To avoid overfitting, we adopted a *Leave One Subject Out* Cross-Validation scheme (LOSO-CV) for training and testing each machine-learning model, in line with previous research [18], [24], [45]. The data were splint into n folds, one for each subject. Then, we used each of such folds — composed of all the four instances of a given subject — as test set and the union of the remaining folds as training set. As a result, the data related to a single subject appears once in the test set and *n-1* times in the training set. It is worth noting that this is a challenging scenario since the machine learning technique can not learn any peculiarities of a specific subject. We used the implementations of all the mentioned algorithms available in the Keras toolkit [78].

To answer RQ_1, we compare a model leveraging only developer-based features with a state-of-the-art model leveraging code-based features [11]. We could choose among many possible bug prediction approaches available in the literature. The most recent approaches, however, are designed to work for big Object-Oriented programs for which a revision history is available. In our case we don't have such an information. For this reason, we could not use process metrics and we could only consider product metrics. Also, we had to exclude most of the CK metrics [79] since we have at most three classes in a program, while most of them are implemented in a single class or even method. In this study, we used as the baseline the approach introduced by Nagappan *et al.* [11], who leverages a set of metrics that could be computed also at method level. Specifically, we considered the following metrics: *number of classes*, *number of functions*, *number of executable lines*, *number of parameters*, *number of arcs in control flow graph*, *number of basic blocks in control flow graph*, *FanIn* (*i.e.,* number of calling functions), *FanOut* (*i.e.,* number of functions called), and *Cyclomatic Complexity*. We implemented the scripts for computing such metrics and we release them in our replication package.

For both the models, we used the previously described pipeline. We evaluate the performance of all models in terms of accuracy, precision, recall, and F1-score. Given the best performing machine-learning model selected for each of the two compared models (developer-based and code-based), we report the rank of the features in terms of number of times they are selected in the 20 folds. We only report the features selected in at least 10 folds, meaning that these values have been selected at least for half of the participants of this study.

To answer RQ_2, we first compute and report the overlap metrics between our developer-based model and the code-based model. To do this, we compute the percentage of instances correctly classified (i) only by using a developer-based model (*OnlyD*), only by using a code-based model (*OnlyC*), by both the models (*Common*). Specifically, given the set of instances correctly classified by the developer-based model, D, ad the set of instances correctly classified by the code-based model, C, we computed the metrics as follows:

- *OnlyD* = $\frac{|D \setminus C|}{|D \cup C|}$
- *OnlyC* = $\frac{|C \setminus D|}{|D \cup C|}$
- *Common* = $\frac{|C \cap D|}{|D \cup C|}$

As a second step, we define a combined model containing all the features from both the models compared in RQ_1 and tested it using the previously described procedure. Similarly to what we did to answer RQ_1, we compare the results achieved with the results obtained using the single models.

1) Replication Package: To foster future research, we provide a comprehensive replication package [80], which includes (i) the detailed protocol we used for the experiment, (ii) the anonymized raw data, and (iii) the scripts we used to perform the analyses described in this paper.

TABLE II: Performances achieved by the best Developer-based (D), the Code-based (C), and the combined (C+D) models.

Model	Class	Precision	Recall	F1-Score	Accuracy
D	*buggy*	0.77	0.85	0.81	0.76
	no-buggy	0.74	0.62	0.68	
C	*buggy*	0.94	0.69	0.8	0.79
	no-buggy	0.67	0.94	0.78	
C+D	*buggy*	0.84	0.90	0.87	0.84
	no-buggy	0.83	0.75	0.79	

V. Results

In this section, we report the results of our study. First, we answer our research questions, and then we discuss some interesting cases we found. The full data set used for the analysis of the results is composed of 80 data points, each related to a task performed by a developer. The actual number of tasks is divided into 48 *buggy* and 32 *no-buggy* tasks. To have an idea about the performance that can be achieved with trivial classifiers, we tested a *constant* classifier—*i.e.,* a classifier always predicting the majority class— and a random classifier — *i.e.,* a classifier which randomly predicts one of the two classes. The first one correctly classifies the outcome of 48 out of 80 tasks (60% accuracy), while the second one achieves 49.9% average accuracy over 1000 repetitions.

1) RQ_1: Developer-Based vs Code-Based Models: The best pipeline configuration for the developer-based model was obtained by (i) not using oversampling, (ii) using correlation analysis, (iii) using feature selection with a LR classifier, and (iv) using the GBC classifier. The best configuration for the code-based model, instead, was obtained by (i) using both oversampling and (ii) correlation analysis — even if the latter was not influent —, (iii) not using feature selection, and (iv) using a SVM classifier.

We report the results achieved for the two models in Table II. The developer-based model correctly classifies the outcome 61 out of 80 tasks. Of these, 41 out of 48 were correctly classified as *buggy*, and 20 out of 32 were correctly classified as *no-buggy*. The code-based model, instead, correctly classified the outcome of 63 out of 80 tasks as *buggy* or *no-buggy*. Of these, 33 out of 48 and 30 out of 32 were correctly classified as *buggy* and *no-buggy* tasks, respectively.

The most predictive features resulting from feature selection and evaluated in the LOSO cross-validation scheme are the following. First, we have the features selected in all the folds. Such features include: (i) HR-based values (such as the mean and the variance and the InterBeat Intervals, during all the time taken by participants to solve the assigned tasks and also during the last 60 seconds before the final delivery; (ii) the number of pressed keys, considered both alone and combined with the EDA Phasic, and the number of mouse clicks. Other features, instead, have been selected very frequently, but not always, such as SAM values (19 times out of 20) and the programming experience, both generic and in Java (18 times out of 20).

For the Code-based model, instead, the best results have been obtained when no features selection method was applied (*i.e.,* all the features were used in all the folds). This is reasonable, considering that this set of features is relatively small.

In terms of overlap metrics, we observed a considerable percentage of instances (about 34.7%) that can be correctly classified only considering either a developer-based (16%) or a code-based model (∼18.7%). Indeed, only ∼65.3% of the instances can be correctly classified by both the models. This suggests that it could be beneficial combining the features in a combined model.

Summary of RQ_1. A developer-based bug prediction model achieves comparable accuracy with respect to the code-based state-of-the-art baseline.

2) RQ_2: Effectiveness of a Combined Model: The best pipeline configuration for the combined model was obtained by not using neither (i) oversampling nor (ii) correlation analysis, (iii) using feature selection with SVM, and (iv) using a RF classifier. We report the results achieved for such a model in Table II. The model correctly classified 67 out of 80 tasks as *buggy* or *no-buggy* (83.8% accuracy). Of these, 43 out of 48 and 24 out of 32 were correctly classified as *buggy* and *no-buggy* tasks, respectively.

We now discuss the features selected during the process. The programming experience (generic and Java) and the heart rate variance during the last 60 seconds before the delivery of the coding tasks were selected 20 out of 20 times. Other relevant features, selected 19 out of 20 times, are (i) features related to the number of methods and parameters involved to solve the tasks, (ii) the subjective assessment of tiredness before and after the completion of a task, and (iii) the keyboard activity evaluated according to the heart rate. We provide more details about the feature importance in the replication package.

Summary of RQ_2. The combined model outperforms the best-performing model (+5% accuracy compared to the code-based model), achieving 84% accuracy.

VI. Discussion

A. Follow-up analysis and Lessons Learned

1) An Example of Wrong Classification: We analyzed more in-depth the classifications for the participants for whom the developer-based bug prediction model provided the worst results (*i.e.,* lowest accuracy) to understand what did not work and what could be the future research directions. To this aim, we took a closer look at the subject for which we observed a lower accuracy.

Looking at the features, we found that the subject typed less than 1,000 keys in all the tasks and shows average values of EDA phasic higher than other participants. We tried to train and test a rule-based classifier (JRip [81]) to understand, specifically, which features contributed to the misclassification based on the defined rules: we found that the $EDAP_{10}$ of the participant was beyond the threshold for classifying it as *Buggy*, according to this classifier, even when he/she correctly completed the task.

TABLE III: Evaluation of individual information sources.

Source	Type	Invasiveness	Accuracy
Activity Tracker	Program	None	78%
Empatica	Wristband	Medium	66%
Questions	Questionnaire	Low	66%
BrainCo	Headband	High	64%

This possibly means that the participant was stressed for other reasons but was able to correctly complete the task anyway. According to the answers to the pre/post task questionnaires, the developer reported a higher level of SAM compared to the median of all the other developers (except for one task). This roughly indicates that he/she was slightly sadder than the other participants. Also, it can be noticed that his/her SAM levels reported before and after the task were always the same (except for a case in which one of the values changed by 1).

2) Individual Information Sources: Some biometric sensors may be perceived as uncomfortable to wear in a normal working environment. Also, asking questions to a developer before and after a task may be perceived as intrusive and may limit the possible applicability of our approach in practice. We perform a further investigation to understand which information sources are more important and which ones can be safely avoided, without impacting the model accuracy.

Table III reports a summary of the accuracy achieved by each individual detector. These results suggest that the activity tracker is a fundamental information source, and it could even be used alone. Actually, when considering the features from such a device alone, we achieve an accuracy level similar to the one achieved by the code-based model (79%) and higher than the best developer-based model selected (76%). This happens because, despite in our experiment we test several feature selection techniques, none of them is perfect, and all of them end up selecting a sub-optimal set of features. This is a well-known issue in machine learning which is evident when a large number of features are available (curse of dimensionality). However, it is very likely that this issue affects the developer-based model and the combined-model the most, since they have significantly more features than the code-based model. On the other hand, the biometric parameters are less suitable to be used as standalone data generators for such applications. The less suited one is the BrainCo-based model, showing a global accuracy only slightly higher than the constant classifier (64% vs 60%).

Given our results, the main lesson learned is that **EEG-like features are likely unsuitable for predicting the presence of bugs**. Of course, this observation should be put in perspective and might not generalize to all the tasks and contexts. Still, our results clearly suggest that the effectiveness achieved through these features measured with the specific equipment we used is very low. Further studies are needed to possibly confirm this finding. However, we point out that future researchers interested in human factors for bug prediction should probably be careful when taking into account such an information source for two reasons.

First, it appears to add a small amount of information. Second, it can be only acquired with invasive equipment, at the moment, which (as we observed while performing the experiment) might bother the subjects, above all in the long run.

B. Limitations

Most of the threats to the validity of our results are related to the **external validity**. The results may be mostly valid for our sample of participants and the specific tasks we chose. We tried to minimize this threat by (i) involving both more and less experienced junior developers, and (ii) choosing both easy and difficult programming tasks. Moreover, it is worth noting that our results only refer to relatively small programs and short time spans.

The biggest threat to the **internal validity** of our study is the monitoring of participants: This might have implicitly influenced their behaviors since they may have felt observed. We minimized this threat by using a between-group design to avoid learning effects, assigning the same amount of tasks from set A and B to both groups, and giving all instructions in written form. Also, some biomedical metrics are influenced by the individual, as well as environmental factors, such as lighting. We counteracted these confounding factors by having a controlled setup with fixed screen brightness, room temperature, and lighting.

A possible threat to **construct validity** regards the methodology used to define the theoretical framework (Section III) on which we base the specific metrics we measure in our experiment. Such a framework is based on an informal literature review, mainly based on the papers we are aware of in this field. More specifically, We used snowballing from such studies to define a set of metrics organized in cohesive categories. It is possible, however, that other factors or even whole categories have an important role in predicting bugs.

A threat to **conclusion validity** might be related to the fact that some of the predictors we used to capture the aspects of interest strongly rely on the measurements provided by the devices. To mitigate such a limitation, we tried to involve only reliable instrumentation. Indeed, we opted for (i) the Empatica E4 wristband because it has received the certification of medical-grade wearable device (CE-medical certified in the EU [53]) and (ii) a helmet produced by BrainCo [54], a company that grew out of the Center for Brain Science at Harvard and the Mcgovern Institute for Brain Research at MIT to conduct R&D in wireless EEG brain wave detector.

C. Ethical Impact Statement

As far as potential application including AI-based modules are proposed in research, *ethical issues* are becoming an important concern that need to be discussed. We acknowledge the potential misuse of sensor-based detection classifiers when embedded in technology to monitor people's behavior. We do not advocate in favor of the implementation of a monitoring technology that might have an impact on privacy.

Conversely, we advocate in favor of using sensor-based to support developers in gaining self-awareness on possible errors introduced in the code, like the plethora of tools available (*e.g.,* linters). In the context of software development, the classifier output could be shared with the colleagues on a voluntary basis, *e.g.,* to request a code-review activities in case of a predicted high likelihood of bug introduction.

As for the risk of deceptive applications and model failure, we are aware that our classifier requires further validation on a larger dataset as it might not be robust enough yet to be deployed for daily use without running the risk of incorrect classification on new unseen data. Nevertheless, the effect of misclassifying buggy code is limited to loosing users' confidence that the classifier can be helpful. As for risks connected to privacy, the data collection protocol we use to collect data was carefully explained to the participants at the beginning of the experiment. According to the rules at the university that hosted the data collection, participants were requested to sign a consent form where they give consent to the anonymous storage and treatment of the experimental data.

VII. Conclusion

We introduced a developer-based model that, given *behavioral*, *psycho-physical*, and *control* factors, is able to detect if bugs were introduced while completing a programming task. We conducted a controlled experiment, based on a rigorous protocol, to assess the effectiveness of our developer-based model, and we used a code-based model as a baseline.

Our results show that our developer-based model achieves a similar performance compared to the state-of-the-art code-based model. We achieve higher accuracy (84%) by training a combined developer- and code-based model. Our results can have several implications. First, specialized training sessions could be envisioned to (i) make developers aware of the conditions in which they risk to introduce bugs while writing code, and (ii) train them to avoid such conditions in the first place. Second, our model can be used by developers to get warnings about the possible introduction of bugs before they make a commit, like other tools based only on the source code. This could allow them to more carefully check the code or ask for more rigid code reviews.

Future replications of our study may involve a larger and more diverse pool of participants. We also plan to evaluate the predictive power of finer-grained behavioral features, for example for distinguishing productive use of the browser (*e.g.,* for reading API documentation) and distractions (*e.g.,* social networks).

References

[1] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 78–88.

[2] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 2013, pp. 279–289.

[3] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 414–423.

[4] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.

[5] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 13–23.

[6] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 432–441.

[7] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2017.

[8] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 194–218, 2017.

[9] B. S. Alqadi and J. I. Maletic, "Slice-based cognitive complexity metrics for defect prediction," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 411–422.

[10] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques," *Expert Systems with Applications*, vol. 144, p. 113085, 2020.

[11] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452–461.

[12] D. Fucci, G. Scanniello, S. Romano, and N. Juristo, "Need for sleep: The impact of a night of sleep deprivation on novice developers' performance," *IEEE Trans. Software Eng.*, vol. 46, no. 1, pp. 1–19, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2834900

[13] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, 2012.

[14] Y. Qu, J. Chi, and H. Yin, "Leveraging developer information for efficient effort-aware bug prediction," *Information and Software Technology*, vol. 137, p. 106605, 2021.

[15] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *Journal of Systems and Software*, vol. 169, p. 110691, 2020.

[16] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "A Look into Programmers' Heads," *IEEE Transactions on Software Engineering*, 2018.

[17] S. Radevski, H. Hata, and K. Matsumoto, "Real-time Monitoring of Neural State in Assessing and Improving Software Developers' Productivity," in *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE Press, 2015, pp. 93–96.

[18] S. Müller and T. Fritz, "Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 688–699.

[19] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using Psycho-physiological Measures to Assess Task Difficulty in Software Development," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 402–413.

[20] C. Parnin, "Subvocalization-Toward Hearing the Inner Thoughts of Developers," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 197–200.

[21] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile, "A replication study on code comprehension and expertise using lightweight biometric sensors," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 311–322. [Online]. Available: https://dl.acm.org/citation.cfm?id=3339126

[22] D. Girardi, N. Novielli, D. Fucci, and F. Lanubile, "Recognizing Developers' Emotions while Programming," in *42nd Int. Conf. on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea*, 2020, pp. –.

[23] D. Graziotin, X. Wang, and P. Abrahamsson, "Do feelings matter? on the correlation of affects and the self-assessed productivity in software engineering," *Journal of Software: Evolution and Process*, vol. 27, no. 7, pp. 467–487, 2015. [Online]. Available: https://doi.org/10.1002/smr.1673

[24] D. Girardi, F. Lanubile, N. Novielli, and A. Serebrenik, "Emotions and perceived productivity of software developers at the workplace," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[25] S. Müller and T. Fritz, "Using (bio) Metrics to Predict Code Quality Online," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 452–463.

[26] R. C. Thomas, A. Karahasanovic, and G. E. Kennedy, "An investigation into keystroke latency metrics as an indicator of programming performance," in *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, 2005, pp. 127–134.

[27] M. A. Just and P. A. Carpenter, "A theory of reading: from eye fixations to comprehension." *Psychological review*, vol. 87, no. 4, p. 329, 1980.

[28] D. Graziotin, X. Wang, and P. Abrahamsson, "How do you feel, developer? an explanatory theory of the impact of affects on programming performance," *PeerJ Computer Science*, vol. 1, p. e18, 2015.

[29] ——, "Software developers, moods, emotions, and performance," *arXiv preprint arXiv:1405.4422*, 2014.

[30] M. V. Mäntylä, K. Petersen, T. O. Lehtinen, and C. Lassenius, "Time pressure: a controlled experiment of test case development and requirements review," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 83–94.

[31] K. Rajeswari and R. Anantharaman, "Development of an instrument to measure stress among software professionals: Factor analytic study," in *Proceedings of the 2003 SIGMIS conference on Computer personnel research: Freedom in Philadelphia–leveraging differences and diversity in the IT workforce*, 2003, pp. 34–43.

[32] Y. M. Ulrich-Lai and J. P. Herman, "Neural regulation of endocrine and autonomic stress responses," *Nature reviews neuroscience*, vol. 10, no. 6, pp. 397–409, 2009.

[33] D. Wastell and M. Newman, "The behavioral dynamics of information system development: A stress perspective," *Accounting, Management and Information Technologies*, vol. 3, no. 2, pp. 121–148, 1993.

[34] H.-G. Kim, E.-J. Cheon, D.-S. Bai, Y. H. Lee, and B.-H. Koo, "Stress and heart rate variability: A meta-analysis and review of the literature," *Psychiatry investigation*, vol. 15, no. 3, p. 235, 2018.

[35] W. von Rosenberg, T. Chanwimalueang, T. Adjei, U. Jaffer, V. Goverdovsky, and D. P. Mandic, "Resolving ambiguities in the lf/hf ratio: Lf-hf scatter plots for the categorization of mental and physical stress from hrv," *Frontiers in physiology*, vol. 8, p. 360, 2017.

[36] C. Schubert, M. Lambertz, R. Nelesen, W. Bardwell, J.-B. Choi, and J. Dimsdale, "Effects of stress on heart rate complexity—a comparison between short-term and chronic stress," *Biological psychology*, vol. 80, no. 3, pp. 325–332, 2009.

[37] C. H. Vinkers, R. Penning, J. Hellhammer, J. C. Verster, J. H. Klaessens, B. Olivier, and C. J. Kalkman, "The effect of stress on core and peripheral body temperature in humans," *Stress*, vol. 16, no. 5, pp. 520–530, 2013.

[38] W. M. Suess, A. B. Alexander, D. D. Smith, H. W. Sweeney, and R. J. Marion, "The effects of psychological stress on respiration: a preliminary study of anxiety and hyperventilation," *Psychophysiology*, vol. 17, no. 6, pp. 535–540, 1980.

[39] H. F. Posada-Quintero, J. P. Florian, A. D. Orjuela-Cañón, and K. H. Chon, "Electrodermal activity is sensitive to cognitive stress under water," *Frontiers in physiology*, vol. 8, p. 1128, 2018.

[40] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, "What happens when software developers are (un)happy," *Journal of Systems and Software*, vol. 140, pp. 32 – 47, 2018.

[41] M. Mäntylä, B. Adams, G. Destefanis, D. Graziotin, and M. Ortu, "Mining valence, arousal, and dominance: Possibilities for detecting burnout and productivity?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 247–258. [Online]. Available: https://doi.org/10.1145/2901739.2901752

[42] J. Russell, "A circumplex model of affect," *Journal of Personality and Social Psychology*, vol. 39, pp. 1161–1178, 1980.

[43] M. Kuutila, M. Mäntylä, U. Farooq, and M. Claes, "Time pressure in software engineering: A systematic literature review," *CoRR*, vol. abs/1901.05771, 2019.

[44] D. Graziotin, X. Wang, and P. Abrahamsson, "Are happy developers more productive?" in *International Conference on Product Focused Software Process Improvement*. Springer, 2013, pp. 50–64.

[45] D. Girardi, N. Novielli, D. Fucci, and F. Lanubile, "Recognizing developers' emotions while programming," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 666–677.

[46] M. Ahrens, K. Schneider, and M. Busch, "Attention in software maintenance: an eye tracking study," in *2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. IEEE, 2019, pp. 2–9.

[47] M. Soto, C. Satterfield, T. Fritz, G. C. Murphy, D. C. Shepherd, and N. Kraft, "Observing and predicting knowledge worker stress, focus and awakeness in the wild," *International Journal of Human-Computer Studies*, vol. 146, p. 102560.

[48] J. Pilzer, R. Rosenast, A. N. Meyer, E. M. Huang, and T. Fritz, "Supporting software developers' focused work on window-based desktops," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13.

[49] M. Züger, S. C. Müller, A. N. Meyer, and T. Fritz, "Sensing interruptibility in the office: A field study on the use of biometric and computer interaction sensors," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–14.

[50] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 153–162.

[51] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*. Springer Science & Business Media, 2013.

[52] "Personal analytics," https://github.com/HASE-UZH/PersonalAnalytics, [Online].

[53] "Empatica e4," https://www.empatica.com/en-eu/research/e4/, [Online].

[54] "Brainco," https://brainco.tech/technology/, [Online].

[55] P. J. Rousseeuw and A. M. Leroy, *Robust regression and outlier detection*. John wiley & sons, 2005, vol. 589.

[56] A. Affanni, "Wireless sensors system for stress detection by means of ecg and eda acquisition," *Sensors*, vol. 20, no. 7, p. 2026, 2020.

[57] A. Greco, G. Valenza, A. Lanata, E. P. Scilingo, and L. Citi, "cvxeda: A convex optimization approach to electrodermal activity processing," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 4, pp. 797–804, 2015.

[58] E. Buccelletti, E. Gilardi, E. Scaini, L. Galiuto, R. Persiani, A. Biondi, F. Basile, N. G. Silveri *et al.*, "Heart rate variability and myocardial infarction: systematic literature review and metanalysis," *Eur Rev Med Pharmacol Sci*, vol. 13, no. 4, pp. 299–307, 2009.

[59] S. Greene, H. Thapliyal, and A. Caban-Holt, "A survey of affective computing for stress detection: Evaluating technologies in stress detection for better health," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 44–56, 2016.

[60] E. Peper, R. Harvey, I.-M. Lin, H. Tylova, and D. Moss, "Is there more to blood volume pulse than heart rate variability, respiratory sinus arrhythmia, and cardiorespiratory synchrony?" *Biofeedback*, vol. 35, no. 2, 2007.

[61] M. Bradley, "Pj: Measuring emotion: The self-assessment manikin (sam) and the semantic differential," *Journal of Experimental Psychiatry Behavior Therapy*, vol. 25, no. 1, pp. 4–59, 1994.

[62] J. D. Morris, "Observations: Sam: the self-assessment manikin; an efficient cross-cultural measurement of emotional response," *Journal of advertising research*, vol. 35, no. 6, pp. 63–68, 1995.

[63] "Leetcode," https://leetcode.com, [Online].

[64] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[65] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[66] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[67] V. Cortes, "Cortes c., vapnik v," *Support-vector networks, Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[68] R. E. Schapire, "Explaining adaboost," in *Empirical inference*. Springer, 2013, pp. 37–52.

[69] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[70] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive aggressive algorithms," 2006.

[71] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[72] D. Morariu, R. Creţulescu, and M. Breazu, "The weka multilayer perceptron classifier," *International Journal of Advanced Statistics and IT&C for Economics and Life Sciences*, vol. 7, no. 1, 2018.

[73] J. Goldberger, G. E. Hinton, S. Roweis, and R. R. Salakhutdinov, "Neighbourhood components analysis," *Advances in neural information processing systems*, vol. 17, 2004.

[74] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Updating formulae and a pairwise algorithm for computing sample variances," in *COMPSTAT 1982 5th Symposium held at Toulouse 1982*. Springer, 1982, pp. 30–41.
[75] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees. belmont, ca: Wadsworth," *International Group*, vol. 432, no. 151-166, p. 9, 1984.
[76] M. Skurichina and R. P. Duin, "Bagging for linear classifiers," *Pattern Recognition*, vol. 31, no. 7, pp. 909–930, 1998.
[77] X. Shi, J.-F. Paiement, D. Grangier, and P. S. Yu, "Gbc: Gradient boosting consensus model for heterogeneous data," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 7, no. 3, pp. 161–174, 2014.
[78] "Keras," https://keras.io, [Online].
[79] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[80] G. Laudato, S. Scalabrino, N. Novielli, F. Lanubile, and R. Oliveto, "Replication package of "Predicting bugs by monitoring developers during task execution"," https://10.6084/m9.figshare.16667710, [Online].
[81] W. W. Cohen, "Fast effective rule induction," in *Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995, pp. 115–123.