computers

Article

Improving Bug Assignment and Developer Allocation in Software Engineering through Interpretable Machine Learning Models

Mina Samir *, Nada Sherief and Walid Abdelmoez

College of Computing and Information Technology, Arab Academy for Science, Technology & Maritime Transport, Alexandria 1029, Egypt; nada.sherief@aast.edu (N.S.); walid.abdelmoez@aast.edu (W.A.)
* Correspondence: mina.abdelnour@student.aast.edu

Abstract: Software engineering is a comprehensive process that requires developers and team members to collaborate across multiple tasks. In software testing, bug triaging is a tedious and time-consuming process. Assigning bugs to the appropriate developers can save time and maintain their motivation. However, without knowledge about a bug's class, triaging is difficult. Motivated by this challenge, this paper focuses on the problem of assigning a suitable developer to a new bug by analyzing the history of developers' profiles and analyzing the history of bugs for all developers using machine learning-based recommender systems. Explainable AI (XAI) is AI that humans can understand. It contrasts with "black box" AI, which even its designers cannot explain. By providing appropriate explanations for results, users can better comprehend the underlying insight behind the outcomes, boosting the recommender system's effectiveness, transparency, and confidence. The trained model is utilized in the recommendation stage to calculate relevance scores for developers based on expertise and past bug handling performance, ultimately presenting the developers with the highest scores as recommendations for new bugs. This approach aims to strike a balance between computational efficiency and accurate predictions, enabling efficient bug assignment while considering developer expertise and historical performance. In this paper, we propose two explainable models for recommendation. The first is an explainable recommender model for personalized developers generated from bug history to know what the preferred type of bug is for each developer. The second model is an explainable recommender model based on bugs to identify the most suitable developer for each bug from bug history.

Keywords: explainability; explainable AI; XAI; recommendation; bugs

check for updates

Citation: Samir, M.; Sherief, N.; Abdelmoez, W. Improving Bug Assignment and Developer Allocation in Software Engineering through Interpretable Machine Learning Models. *Computers* **2023**, *12*, 128. https://doi.org/10.3390/computers12070128

Academic Editor: Paolo Bellavista

Received: 18 May 2023
Revised: 17 June 2023
Accepted: 20 June 2023
Published: 23 June 2023

1. Introduction

Software bugs lead an application to behave unexpectedly. Bugs can be introduced throughout the testing and maintenance phases of the software development lifecycle (SDLC) [1]. Software bugs are a common and persistent problem in the field of software engineering, causing significant challenges and costs for organizations and users alike. Therefore, it is important to better understand the causes, effects, and mitigation strategies associated with software bugs [2], as well impacts on the end user. Bugs can never be entirely eradicated, so no software can be completely bug-free. The testing team can follow best practices to eliminate software bugs. A good management system identifies and fixes most errors before production. If testers and developers work well together, bugs can be discovered and resolved faster [3].

Commonly, issue tracking systems (ITS) [4] are used to create, update, and address reported customer bugs, as well as bugs reported by other personnel within a company. A bug should contain pertinent details regarding the problem encountered. A frequent component of an issue tracking system is a knowledge base holding data on each client, common problem resolutions, the bug's state, developer information, and similar data. Several open-source software projects manage requests professionally through cloud-based bug tracking systems (e.g., Bugzilla, GitHub) [2]. The bug tracking system manages the assignment of each bug to the appropriate developers and classifies it accordingly (e.g., bug, feature, and product component). The developer who handles these assigned bugs is referred to as a bug tracker [5]. As massive numbers of bugs are reported daily in the bug tracking system, it becomes increasingly difficult to manually manage these bug reports on time. Every day, approximately 300 bug reports are discovered or sent to the Eclipse open-source project [6]. These statistics demonstrate the difficulty of the bug triage procedure.

Bug triage is the process of identifying and prioritizing tracker issues. It helps guarantee that reported issues, such as bugs, enhancements, and new feature requests, are managed effectively. Several automated triage systems using the candidate developer's prediction process have been developed [7–9]. The reporter of a defect uses the standard bug report format to make a patch to the tested bugs. It has entries for bug ID, assignee, open date, when the developer begins working on the problem, close date, and when the bug has been completely resolved and closed. The severity of an issue indicates its impact on the system. The priority level of an issue indicates the urgency of fixing it, since the resolution of other defects may rely on its resolution.

In several application sectors, machine learning is considered as a technology of the future [10], ranging from basic applications such as product recommender systems, to automated cancer detection. Many applications make extensive use of recommender systems. Recommendation systems use machine learning algorithms and techniques to provide customers with appropriate recommendations by analyzing data (e.g., past behaviors) and predicting current interests and preferences. Collaborative filtering (CF) has been shown to be one of the most effective methods for generating suggestions based on prior user behavior [11].

However, the newly popular latent representation methods for CF, including both shallow and deep models, are unable to adequately explain to consumers their rating prediction and recommendation outcomes [12]. Some challenging but well-connected topics for implementing more accurate bug fixing techniques have been explored in previous bug fixing research [13,14]. These include topics such as "how to collect information for a developer's skills in software projects", "how to tie various pieces of information to a bug report to assign it to a developer", "how to apply similarity measures to match a bug report with a developer" and "how to use additional hints or heuristics to connect a bug report to a candidate". Almost all past bug fixing studies do indeed include such topics. Unfortunately, bug fixing is still a time-consuming and money-consuming part of software development projects. Fixing software defects is a crucial part of software management.

Furthermore, there is a basic challenge in using machine learning, which is the explainability of the results [15]. Typically, AI algorithms operate as opaque "black boxes" that accept input and produce output without any means of understanding their inner workings. Several previous papers [16] discuss and analyze the various intricacies that are involved in defining explainability and interpretability of neural networks. We support the idea of explainable AI in general, which refers to a collection of methods and algorithms that are intended to increase the dependability and openness of AI systems. Explanations are referred to as supplemental pieces of metadata information that are derived from the AI model and provide insight into the reasoning behind a particular AI decision or the internal workings of the AI model. In the literature review section, many different explainability methodologies that may be used with deep neural networks are discussed. The purpose of explainable artificial intelligence (XAI) is to enable humans to comprehend the logic behind an algorithm's output [12]. Explainability is essential for several reasons:

- It facilitates analysts' timely and simple comprehension of system outputs. Analysts can make better-informed conclusions if they comprehend how the system operates.
- False positives are reduced. Explainability automates a tedious procedure by giving analysts' recommendations and inconsistencies to investigate.

- It provides assurance in the AI diagnosis by explaining the "why." AI can occasionally produce correct outcomes for incorrect reasons. Likewise, AI may and does make mistakes. Explainability means that errors can be understood and trained outside of the system.
- Encourages the adoption and acceptance of AI, since trust via understanding is key.

Appropriate explanations are critical for recommendation systems, as researchers have discovered [12]. This may contribute to the system's enhancement in effectiveness, transparency, and trust. For explainable recommender system, many methods have been developed, most of which are classified as collaborative filtering approach, explain by item-based collaborative filtering, or user-based collaborative filtering [11,12]. However, there is still a research gap related to the explainability of machine learning-based recommendation systems.

Motivated by the challenges mentioned above, and to minimize the costs and time spent on software maintenance, we propose a method for bug triage automation called Assign Bug based on Developer Recommendation (ABDR). Our method's principal goal is to decrease resolving time, cost, and provide explainable results. The proposed method, upon receiving a new issue, suggests developers by retrieving the most appropriate developers based on bug history and developers' profiles as inputs. It is a supervised machine learning technique that generates a list of recommended developers based on previous profiles of the developers. Developer Similarity (DS) uses developer knowledge to fix issues across many features of bugs including product, severity, and component, as well as bug handling time (HT), and effectiveness in fixing several bugs (EB). We also propose an ABDR training model. The model generates developer profiles according to the three abovementioned factors: DS, HT, and EB. We assign each developer and each bug a bug prediction score that reflects the developer's familiarity with resolving the specific bug. Finally, we present two explicable recommendation models. The first is an explainable recommender model for individualized developers created from bug history to determine which type of bug each developer prefers. The second model is an explainable recommender based on bug history to determine the optimal developer for each bug.

The paper is organized as follows: Section 2 introduces the background of prediction models and explainability methods. Section 3 includes a demonstration of related work for assigning bugs to developers using the techniques mentioned in Section 2. Section 4 discusses the proposed model to assign a new bug to the most relevant developer. Section 5 shows the results of the proposed model. Section 6 discusses the research questions and our model's findings. Finally, Section 7 concludes our analysis.

2. Background

In this section, detailed background concerning topics related to this research is discussed. When a new bug is reported, it is the responsibility of the team to decide which developer will be tasked with fixing it. However, if the selected developer is unable to repair the issue, the system must allocate it to a different developer. Delays in resolving bugs [17] are caused by such constant reassignment. We review bug-fixing techniques, bug-fixing processes, machine learning techniques used for bug assessment, and explainability models for machine learning techniques.

2.1. Bug Fixing Techniques

The General Bug Fixing Model is typically implemented with the use of a bug tracking system, which helps the development team to manage the bug fixing process more effectively. This system allows the team to track the progress of each bug and keep all relevant information in one centralized location.

The first step in the process is bug reporting, which involves entering the bug into the tracking system. The bug report typically includes information such as the issue description, steps to reproduce the issue, and any supporting materials such as screenshots or error logs. The bug tracking system also assigns a unique identifier to each bug, which allows the team to easily reference it throughout the bug fixing process.

After the bug is reported, it is typically assigned to a member of the development team for investigation and resolution. This step is facilitated by the bug tracking system, which can assign the bug to a specific team member based on his/her skills. The team members will then investigate the issue and work to find a solution, typically updating the bug report with their progress along the way.

Once a solution is found, the team member will typically submit a fix for the bug, which is then tested and verified by the team. This is typically performed using a testing environment, where the fix can be tested in a controlled environment to ensure that it resolves the issue and does not introduce any new problems.

Finally, once the bug fix is verified, it is closed in the bug tracking system. This indicates that the issue has been resolved and can help the team keep track of which bugs have been addressed and which are still outstanding. The closed bug report typically includes information on the resolution of the issue and any relevant notes or comments.

Overall, the use of a bug tracking system can help streamline the bug fixing process and improve communication between team members. By using this system, development teams can more effectively manage the bug fixing process and ensure that bugs are resolved in a timely and efficient manner, as shown in Figure 1.

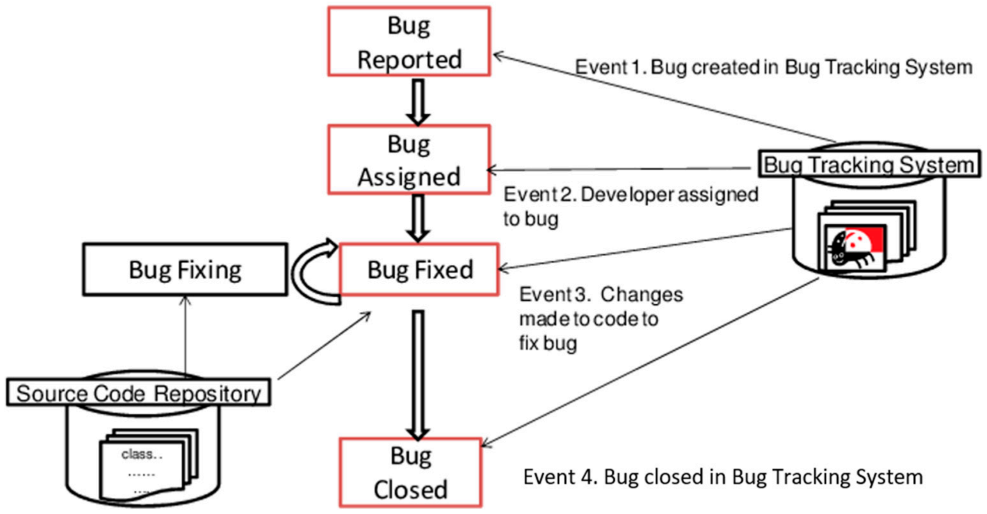
Bug Reported
Event 1. Bug created in Bug Tracking System
Bug Assigned
Bug Tracking System
Event 2. Developer assigned to bug
Bug Fixing
Bug Fixed
Event 3. Changes made to code to fix bug
Source Code Repository
class..
Bug Closed
Event 4. Bug closed in Bug Tracking System

Figure 1. General Bug Fixing Model [18].

The most common description of assigning a developer to a bug is as follows: If you receive a bug report, your task is to determine which developers are the most qualified to address the issue based on their track records of participation with the project [13]. Some approaches aim to reduce the time to fix the problems rather than maximize the knowledge of the potential assignee. For instance, in [19], the authors recommend building a topic model from reported issues. Their technique estimates how long it will take each developer to repair an issue (using the log normal distribution of the three possible combinations of fixer, subject, and severity) and then prioritizes who should be given the report.

A few researchers have explored files and metadata to find connections between developers and the newly discovered issues using location-based techniques [20]. In the beginning, they locate or forecast the presence of issues such as methods or classes. Next, they determine the most qualified programmer to return to work on the items based on the existing connections between programmers and the places in question. Most location-based techniques need extensive version control system documentation. The most prevalent methods were information retrieval (IR) and machine learning (ML). Many recent studies have focused on IR-based activity profiling because it typically results in higher accuracies [21]. Another study [22] used the feature selection method to reduce data volume. The characteristics of the dataset were extracted using a hybrid of the K-Nearest Neighbor and Naive Bayes methods. Finally, the researchers investigated the recommended method using the Mozilla bug tracker.

2.2. Machine Learning Techniques

Supervised and unsupervised learning are two fundamental paradigms in machine learning that have been widely applied in various domains, including software engineering. Supervised learning is a form of machine learning where the model is trained on labeled data, meaning that the input data are associated with known output labels. The goal is to learn a function that can map the input data to the output labels accurately. This type of learning is often used in classification, regression, and prediction tasks. In software engineering, supervised learning has been applied to tasks such as defect prediction, code clone detection, and bug triage, among others.

On the other hand, unsupervised learning is a form of machine learning where the model is trained on unlabeled data, meaning that there are no known output labels for the input data. The goal of unsupervised learning is to identify patterns and structures in the input data without any prior knowledge of the output. Unsupervised learning has found applications in software engineering, including software clustering, anomaly detection, and feature extraction tasks.

Both supervised and unsupervised learning techniques have their strengths and weaknesses and can be applied to different tasks depending on the nature of the data and the problem at hand. However, choosing the appropriate learning technique and model is crucial for achieving accurate and reliable results. Moreover, the interpretation and explanation of these models are equally important to understand their underlying behavior and to trust their outputs.

In this work, deep learning techniques [21] are used in supervised learning because they are particularly suited to handling large and complex datasets with multiple features. In supervised learning, the algorithm learns to map input data to output labels by being trained on a labeled dataset. The following list describes these classifiers in detail:

- Regression [23]: This is a statistical model that classifies binary dependent variables using the logistic function. Despite its simplicity, regression is still commonly employed in prediction to improve performance.
- Root Mean Square Error (RMSE) [24]: This is a commonly used metric in statistics and machine learning for evaluating the accuracy of a regression model. RMSE measures the average distance between the predicted values of a model and the actual values in a dataset. It is calculated as the square root of the average of the squared differences between the predicted and actual values.
- Gradient Boosting Machine (GBM) [25]: This is a powerful machine learning algorithm used for regression and classification problems. GBM is a type of ensemble learning technique, which combines the predictions of several weak models to create a stronger model. GBM works by building a sequence of decision trees, where each tree corrects the errors of the previous tree. The algorithm iteratively adds decision trees to the model, with each subsequent tree fitting the residual errors of the previous tree. The process continues until the specified number of trees is reached, or until a specified level of accuracy is achieved.
- Decision Tree [26]: This method is a type of supervised machine learning, in which it is specified what the input and the output will be in relation to the data used for training. In this method, the data are split in a continuous manner in accordance with a given parameter. In this decision tree, we have two characteristics, which are the parent nodes and the leaves, with the leaves representing the outcome and the parent nodes serving as the criteria.
- Random Forest [27]: This is a collection of decision trees that may also be referred to as an ensemble learning algorithm. In this context, the word “ensemble” refers to a collection or a group; in the context of Random Forest, it refers to the collection of trees.
- XGBoost [28]: The XGBoost algorithm is an effective tool for machine learning. It has only lately been made available and falls under the category of supervised learning. Gradient boosting may be thought of as its fundamental central concept. XGBoost is based on a technique known as parallel tree boosting, which predicts a target based on

the combined results of many weak models. This technique enables Gradient Boosting to achieve both high speed and high accuracy.

Our experiment uses the implementation of the bug prediction classifier described above, developed in the XGBoost model in the R programming language [29]. XGBoost (eXtreme Gradient Boosting) [29] is a popular machine learning algorithm that has been used for bug assignment in software engineering. There are five reasons why XGBoost is a good choice for bug assignment: Firstly, it is an ensemble method that combines the predictions of multiple weak models (i.e., decision trees) to improve the accuracy and robustness of the model. Secondly, it has a regularization parameter that helps prevent overfitting and improves the generalization performance of the model. Third, it can handle both categorical and continuous data, which makes it suitable for bug assignment datasets that may contain a mixture of different types of features. The fourth reason is that it is a highly scalable algorithm that can handle large datasets with millions of instances and features, which is important for bug assignment tasks that may involve many bug reports. Finally, it has a fast implementation and can run on parallel architectures, which makes it a practical choice for real-world bug assignment applications.

Overall, XGBoost is a powerful and flexible machine learning algorithm that is well-suited to bug assignment in software engineering and has been shown to achieve high accuracy and efficiency in several studies.

2.3. *Explainability Methods*

To facilitate a deeper understanding of the topics covered in this paper, the following sections provide a brief explanation of the basic features of XAI from a technological point of view. Therefore, the concept of explainability is linked to the interface that exists between people and those who make decisions [30]. This interface can be understood by people in real time and is an exact reflection of the decision maker. Providing explanations for black-box machine learning techniques such as deep neural networks has become more important in recent years [16].

Explainability models provide ways to understand the reasoning behind the decisions made by machine learning models. These models aim to make machine learning models more transparent and interpretable, enabling us to understand the impact of various factors and features on the model's output.

This section outlines some commonly used methods for model explainability, such as LIME (Local Interpretable Model Agnostic Explanations), Break-down, textual explanations of visual models, and SHAP (SHAPley Additive exPlanations). These techniques aim to provide insights into how machine learning models make decisions and to increase transparency and trust in the decision-making process.

2.3.1. LIME (Local Interpretable Model Agnostic Explanations)

The core idea behind LIME is to provide a simple explanation for a prediction made by a more advanced model, such as a deep neural network, by fitting a simpler local model [31]. On the other hand, LIME [32] creates samples close to the input of interest, evaluates them with the target model, and then approximates the target model in this general neighborhood with a simple linear function. The widespread success of LIME's various implementations across a variety of fields is proof of the method's popular appeal. Using a surrogate model to address the explanation issue is a disadvantage of LIME.

2.3.2. Break-Down

The idea behind this model is to conduct a variable contribution analysis. Researchers first compile a growing list of variables, then examine the values of each variable in turn. Interactions in the model may cause the contributions to change depending on the order in which the variables are considered [33]. If one variable's contribution varies depending on the order in which it appears, then it may be possible to identify this by analyzing various orderings. To discover and illustrate model interactions, the Break-down technique [34] examines the different orders in which they occur. A single order is selected using a feature important to establishing the final attributions. The authors in [35] demonstrate this method for many different datasets. As a result, this method is good for both computational efficiency and interpretation. The Breakdown library [36] is a machine learning library that can be used to analyze the contributions of individual variables in a predictive model's output. The library can be used to construct both binary classifiers and regression models. It also provides a Break-down Plot, which is a graphical representation of how each individual variable contributes to the overall forecast. The plot helps users understand which variables are most influential in making the predictions and can aid in identifying potential bugs or issues in the model.

The Breakdown library works by decomposing the prediction of a model into the contribution of each individual variable. This allows users to see how much each variable is contributing to the prediction and how the variables interact with each other. For binary classification models, the Breakdown library can be used to identify the variables that have the most significant impact on the model's accuracy or precision. This can help users understand which variables to focus on when optimizing the model. For regression models, the Breakdown library can be used to understand which variables are driving the model's predictions and how changes in the variables affect the model's output. Overall, the Breakdown library is a useful tool for understanding the contribution of individual variables in a predictive model and can aid in bug detection and model optimization.

2.3.3. Textual Explanations of Visual Models

Several different machine learning models are used to generate textual descriptions of images. These models have two parts: one that processes the input images (typically a convolutional neural network, or CNN), and one that learns a suitable text sequence (typically a recurrent neural network, RNN). This allows the models to generate textual descriptions of images (RNN). These two components work together to produce picture-descriptive phrases, which are dependent on the fact that a classification assignment has been completed to a satisfactory level. The COCO database made by Microsoft (MS-COCO) was one of the first places where picture descriptions were used as part of a benchmark dataset [37].

2.3.4. SHAP

SHAP, which stands for SHAPley Additive exPlanations, is a visualization tool that may be used to make a machine learning model more explainable by displaying the model's output. By calculating the contribution of each feature to the forecast, it may be used to explain the prediction of any model. It is a grouping of several different tools, such as lime, SHAPely sample values, DeepLift, and QII, amongst others. In [38], SHAP is determined by taking the average of these contributions over all the many orderings that are allowed. The Shapley values have been adapted into this technique to provide an explanation for specific predictions made by machine learning models. At first, Shapley values were suggested to ensure that everyone receives the same number of rewards in cooperative games.

We use The Break-down method in bug assignments to help identify which features or variables of a model are the most important contributors to a particular bug. By analyzing the impact of each feature in the model, the Break-down method can provide a better understanding of the relationship between the input variables and the output. Additionally, the Break-down method is computationally efficient and can provide easy-to-understand visualizations, making it a useful tool for developers and other stakeholders involved in the bug fixing process.

3. Related Work

The use of machine learning techniques in software engineering must be accompanied by rigorous evaluation and verification to ensure their usefulness and applicability in practice.

3.1. Machine Learning Techniques in Bug Fixing

Reference [12] is a survey paper that reviews recent developments in explainable recommendation systems, including both supervised and unsupervised learning techniques. The authors discuss the importance of explainability in recommendation systems and the challenges of achieving it, such as the need for transparency, interpretability, and user trust. The authors provide a comprehensive overview of the different approaches used in explainable recommendation systems, including rule-based systems, matrix factorization, deep learning, and hybrid methods. The authors also discuss the different methods used to evaluate the effectiveness of explainable recommendation systems, such as user studies, surveys, and performance metrics. They highlight the importance of evaluating not only the accuracy and effectiveness of the system but also its interpretability and user satisfaction.

3.1.1. Supervised Learning Techniques

The research described in [39] used supervised learning techniques to develop a release-aware bug triaging method that considers developers' bug-fixing loads. The authors used a labeled dataset of bug reports from the Eclipse project, which included information on the severity of the bug, the component it belongs to, and the developer who fixed it. The authors used a logistic regression model to predict the probability of a bug being fixed by a particular developer, given its severity and component. They also developed a release-aware method that considers the expected bug-fixing load of each developer and assigns bugs to developers accordingly. To evaluate the effectiveness of their method, the authors compared it with several baseline methods and conducted a sensitivity analysis to evaluate the impact of different factors on the model's performance. The results showed that their method outperformed the baseline methods in terms of several metrics, including accuracy, F1 score, and AUC-ROC. The authors also conducted a detailed analysis of the results to identify factors that contribute to the model's performance, such as the severity of the bug, the component it belongs to, and the expected bug-fixing load of each developer. They also discussed the potential limitations of their approach and suggested directions for future research, such as incorporating more features including developer experience and workload, and testing on datasets from other software projects.

The core idea of paper [40] relates to the impact on the interpretation of defect models of correlated metrics, which are used to predict software defects. The authors show that correlated metrics can have a significant impact on the performance of defective models, and they propose a method for identifying and addressing these correlations. The paper includes an evaluation of the method on several real-world datasets, demonstrating its effectiveness in improving the performance of defective models.

The purpose of study [41] was to investigate the impact of tangled code changes on defective prediction models. Tangled code changes are changes that affect multiple code locations at once, and the authors show that these changes can have a significant impact on the performance of defect prediction models. The paper proposes a method for detecting tangled code changes and incorporating them into defect prediction models, and it includes an evaluation of the method on several real-world datasets.

Another study [42] aimed to address the issue of manual categorization of bug reports, which can be a time-consuming and error-prone process. The authors proposed an automated approach based on supervised learning techniques using LSTM networks. LSTM networks are a type of recurrent neural network (RNN) that is well-suited for processing sequential data such as text. The authors used a dataset of bug reports from the Apache Software Foundation, which consisted of over 12,000 bug reports labeled into six categories. They preprocessed the text data by tokenizing, stemming, and removing stop words, and then trained the LSTM model on the preprocessed data. The authors evaluated the model's performance using various metrics, including precision, recall, and F1 score, and compared it with several baseline models. The results showed that the LSTM model outperformed the baseline models and achieved an F1 score of 0.727, indicating a significant improvement in categorization accuracy. The authors also conducted a sensitivity analysis to evaluate the impact of different parameters, such as the number of LSTM layers and the embedding dimension, on the model's performance. The authors of [35] present a method for predicting fault-proneness in software using Random Forests. They show that Random Forests can outperform other machine learning algorithms, such as neural networks and support vector machines, in terms of prediction accuracy. The paper also discusses the use of feature selection and cross-validation to improve the performance of the model. The method is evaluated on several real-world datasets, demonstrating its effectiveness in predicting fault-proneness in software.

The authors of [36] investigated whether the chronological order of data used in just-in-time (JIT) defect prediction models affects their performance. The authors partially replicated a previous study and evaluated the impact of different training and testing data sets on the performance of the model. The results show that the chronological order of the data can have a significant impact on the performance of JIT defect prediction models, and the authors provide recommendations for improving the accuracy of these models. The study highlights the importance of considering the temporal order of data in software defect prediction.

The authors of paper [8] used supervised learning techniques, specifically deep learning, to develop an automated bug triaging system called DeepTriage. They used a labeled dataset of bug reports from the Eclipse project, which included information on the severity of the bug, the component it belongs to, and the developer who fixed it. The authors used a deep learning model based on a convolutional neural network (CNN) and a long short-term memory (LSTM) network to predict the probability of a bug being fixed by a particular developer, given its severity and component. They also developed an approach to address the class imbalance problem in the dataset, where some developers are responsible for fixing a significantly larger number of bugs than others. To evaluate the effectiveness of their approach, the authors compared it with several baseline methods and conducted a sensitivity analysis to evaluate the impact of different factors on the model's performance. The results showed that their approach outperformed the baseline methods in terms of several metrics, including accuracy, F1 score, and AUC-ROC. The authors also conducted a detailed analysis of the results to identify factors that contribute to the model's performance, such as the importance of the component and developer information in predicting bug triage, and the impact of different hyperparameters on the model's performance. They also discussed the potential limitations of their approach, such as the need for large amounts of labeled data and the potential biases in the dataset. Overall, the study [6] demonstrates the potential of supervised learning techniques, specifically deep learning, in developing effective bug triaging systems. The authors provide a detailed analysis of their approach and discuss potential avenues for future research, such as incorporating more features, for example, developer experience and workload, and testing on datasets from other software projects.

3.1.2. Unsupervised Learning Techniques

The research reported in [11] used unsupervised learning techniques, specifically constrained matrix factorization, to improve the explainability of recommender systems. The authors proposed a novel approach to incorporate user constraints, such as explicit preferences and implicit feedback, into the matrix factorization process to generate more interpretable recommendations. The authors used a dataset of movie ratings from the MovieLens dataset and evaluated their approach using several metrics, including prediction accuracy, diversity, and novelty. They also conducted a user study to evaluate the explainability of the recommendations generated by their approach, compared with a baseline method. The authors discussed the potential benefits of their approach in generating more explainable recommendations, which could improve user trust and satisfaction with the system. They also discussed the potential limitations of their approach, such as the need for additional constraints and the impact of the sparsity of the dataset on the performance of the approach. Overall, the study demonstrates the potential of unsupervised learning techniques, specifically constrained matrix factorization, in improving the explainability of recommender systems. The authors provide a detailed analysis of their approach and discuss potential avenues for future research, such as incorporating more user constraints and testing on datasets from other domains.

In conclusion, supervised learning allows for precise and accurate predictions as it leverages labeled data to train models and make informed decisions. With clear and explicit target values provided during the training phase, supervised learning algorithms can effectively learn patterns and relationships in the data, leading to reliable predictions. In addition, supervised learning offers interpretability and explainability. Since the training data are labeled, it becomes easier to understand the factors influencing predictions or classifications. This transparency allows users to comprehend the underlying logic of the model's decisions, enhancing trust and facilitating debugging and error analysis. While unsupervised learning techniques have their merits, such as identifying hidden patterns and clustering similar instances without labeled data, they may lack the precision and interpretability of supervised learning. The absence of labeled data makes it challenging to assess the accuracy of predictions or provide explanations for the discovered patterns.

3.2. Explainability Models for Recommender-Based ML Techniques

Paper [43] applied the Textual Sentence Explanations approach to increasing amounts of user-generated material, such as e-commerce user reviews and social media user contributions. This information is extremely valuable for predicting more extensive user preferences and may be used to deliver fine-grained and more credible suggestion explanations to convince customers or to assist consumers in making more educated choices. Based on this idea, a few models have been developed recently to explain recommendations by using different kinds of textual information. These models usually produce a textual phrase that explains the suggestion.

In [38], the authors propose a novel framework called SHAP (SHapley Additive exPlanations) to explain the output of any machine learning model. SHAP computes Shapley values, which is a method for assigning contributions to each feature in a prediction. The paper also presents the integration of SHAP with deep neural networks and Random Forests. The results presented in the experimental evaluation section show that the proposed framework is effective in providing interpretable and meaningful explanations for NIDS decisions. The authors use several metrics to evaluate the performance of the framework, including accuracy, precision, recall, and F1 score, which provide a comprehensive evaluation of the framework's effectiveness. One limitation of the paper is that it does not explicitly address the scalability of the proposed framework. As mentioned earlier, SHAP can be computationally expensive, especially for large datasets and complex models. It would be interesting to see how the proposed framework performs in larger-scale experiments or real-world settings where speed and efficiency are critical.

Paper [44] presents a technique for characterizing the complexity of neural networks using Fisher-Shapley randomization. The paper uses SHAP values to identify the most important features in a model and then applies Fisher–Shapley randomization to evaluate the perturbation effects of these features on the model's output.

The research presented in [45] investigates a technique for evaluating the predictive uncertainty of machine learning models under dataset shift. The paper uses SHAP values to identify the most important features in a model and then evaluates the robustness of the model's predictive uncertainty to changes in those features.

4. Proposed Model

Previous studies approached the subject of triaging as a classification problem and concluded that manual triaging was the most effective method. Based on this paper's Related Work section, three research questions that have not been answered yet are:

1. How can considering multiple factors, such as the severity and priority of the bug, the expertise of the developer, and the complexity of the bug, improve the performance of bug triaging using machine learning?

2. How can machine learning techniques be used to assign bugs to developers based on their expertise?
3. How can explainable artificial intelligence (XAI) techniques be applied to enhance the transparency and interpretability of machine learning models for bug triaging, thereby helping developers comprehend the decision-making process of the models and providing suggestions on how to improve them?

Within the scope of this study, an optimization strategy for the bug triaging process using the Eclipse dataset is proposed. The strategy incorporates two methods, namely, the Break-down method and the SHAP technique, to improve the bug fixing time and standardize developer practices. In addition to the Break-down method, the proposed optimization strategy also integrates the SHAP technique. SHAP is a model-agnostic interpretability method that quantifies the contribution of each feature in a predictive model. By applying SHAP to the bug triaging process, developers can gain a better understanding of the impact of individual features on the fixing time.

The proposed method is designed to leverage supervised learning techniques to assign a new bug to the most relevant developer based on the bug's features and the developer's expertise, as determined from bug history performance. By utilizing a supervised learning approach with the XGBoost technique, the model can learn from historical data and make predictions on new, unseen data. This can lead to more accurate bug assignments and help reduce the workload of developers by ensuring that bugs are assigned to the most appropriate individual. As seen in Figure 2, the procedure consists of three distinct phases as following:

1. Preparation stage: In this stage, the bug data are preprocessed to extract relevant features and reduce noise. The bug reports are filtered to include only those that have been resolved, verified, or closed, and only six variables are selected for each bug, namely "PRODUCT", "COMPONENT", "OP_SYS", "SEVERITY", "PLATFORM", and "PRIORITY".
2. Machine learning stage: In this stage, a machine learning model is trained using historical bug data to predict the most relevant developers for a new bug based on its features. The model uses a combination of gradient boosting and k-nearest neighbor algorithms to make predictions.
3. Recommendation stage: In this stage, the trained machine learning model is used to recommend the most relevant developers for a new bug. The model calculates a relevance score for each developer based on their expertise and past bug handling performance. The developers with the highest relevance scores are then presented as recommendations for the new bug.

By using a combination of machine learning and bug history performance data, the proposed method aims to improve the accuracy and efficiency of the bug triaging process by recommending the most relevant developers for each new bug.

Stage 1: Preparation. In stage one, the new bug is received, and its features are extracted or derived to match with historical bug data. The relevant metadata for each bug, such as the product, component, operating system, severity, platform, and priority, are selected and used for further processing in the next stages. The preprocessing and feature extraction steps are necessary to reduce noise and make the bug data more suitable for the machine learning model to learn from. The following subsections present an expanded explanation of what occurs in Step One.

- Receiving a new bug: The first step in stage one is to receive a new bug that needs to be assigned to a developer for fixing. The bug may come from various sources, such as user reports.
- Extracting metadata: After receiving the new bug, the relevant metadata for the bug is extracted or derived. This metadata includes information such as the product, component, operating system, severity, platform, and priority of the bug.

- Preprocessing: The extracted metadata is preprocessed to remove any noise and inconsistencies in the data. This is achieved by text reduction and cleaning methods, such as removing stop words, converting text to lowercase, and removing punctuation. This helps reduce the dimensionality of the data and improves the accuracy of the model.
- Feature extraction: After preprocessing, the relevant features for the bug are extracted from the metadata. This involves selecting the most relevant variables that are likely to influence the developer assignment decision. In this study, only six variables were used: product, component, operating system, severity, platform, and priority.
- Matching with historical data: The extracted features for the new bug are then matched with the historical bug data to find the most similar bugs in the dataset. This is achieved using various similarity measures, such as cosine similarity or Jaccard similarity.
- Retrieving relevant developers: Once the most similar bugs in the historical data are identified, the developers who fixed those bugs are retrieved. This is achieved by analyzing the change logs and version control systems to find the developers who were responsible for fixing the bugs.

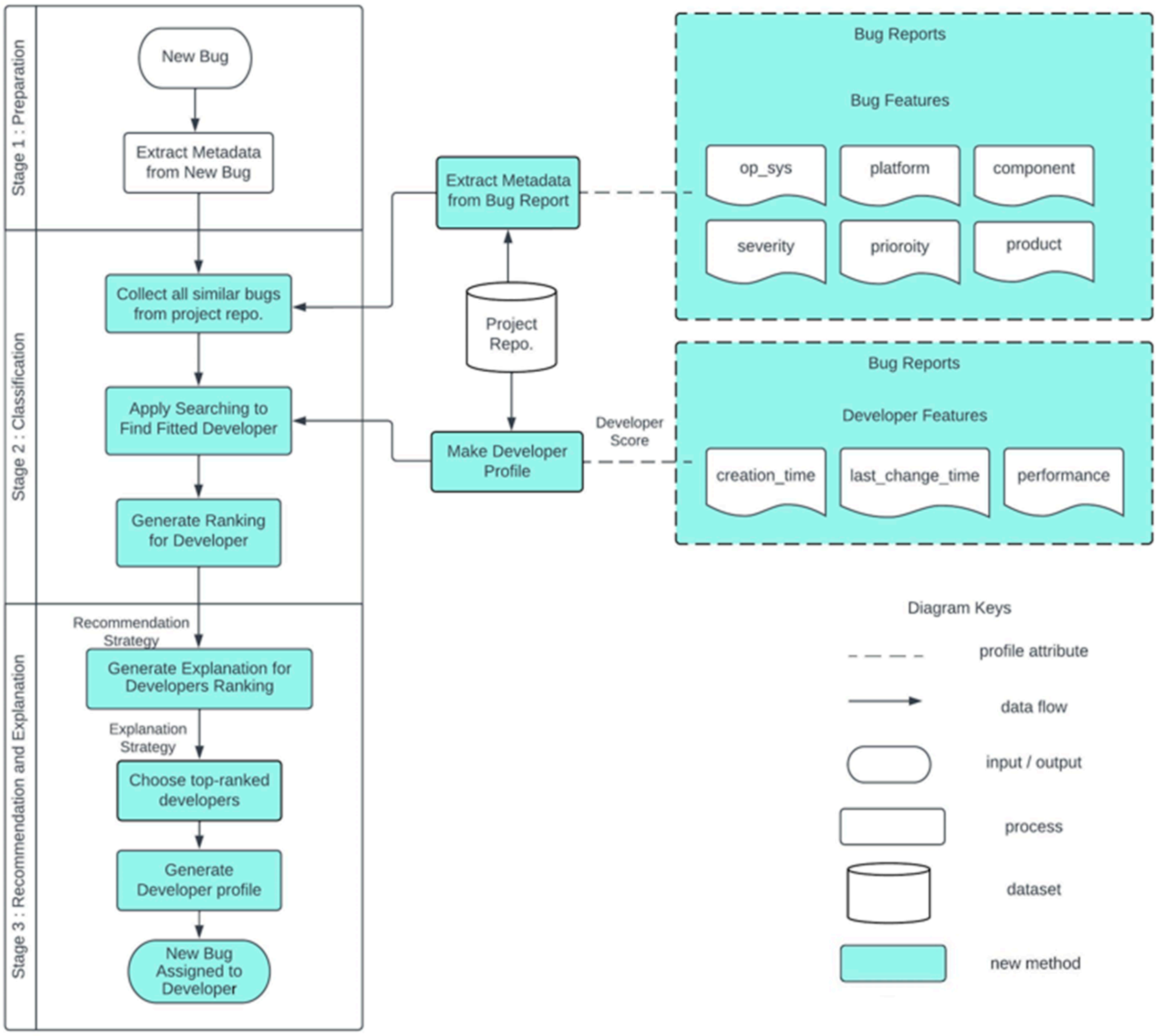

Figure 2. ABDR recommender model.

Overall, stage one involves receiving a new bug, extracting, and preprocessing its metadata, selecting relevant features, and matching with historical bug data to retrieve relevant developers. This lays the groundwork for the machine learning model to learn and make accurate developer assignment decisions in the subsequent stages.

The history of bugs in the Eclipse content includes 208,862 bugs that are described across 47 variables [46] in Table 1. To make the analysis more manageable and easier to interpret, we use some text reduction and cleaning methods. Specifically, we will only consider bug reports that are marked as "CLOSED", "VERIFIED", or "RESOLVED".

Table 1. Bug history of Eclipse.

No.	Variable	No.	Variable	No.	Variable
1	Alias	17	creator_detail.real_name	33	product
2	assigned_to	18	deadline	34	qa_contact
3	assigned_to_detail.email	19	depends_on	35	qa_contact_detail.email
4	Id	20	dupe_of	36	qa_contact_detail.id
5	assigned_to_detail.name	21	flags	37	qa_contact_detail.name
6	name	22	groups	38	qa_contact_detail.real_name
7	blocks	23	id	39	resolution
8	cc	24	is_cc_accessible	40	see_also
9	cc_detail	25	is_confirmed	41	severity
10	classification	26	is_creator_accessible	42	status
11	component	27	is_open	43	summary
12	creation_time	28	keywords	44	target_milestone
13	creator	29	last_change_time	45	url
14	creator_detail.email	30	op_sys	46	version
15	creator_detail.id	31	platform	47	whiteboard
16	creator_detail.name	32	priority		

Additionally, we reduce the number of variables for each bug to six, focusing only on the most relevant factors that contribute to bug resolution. These variables include "PRODUCT", "COMPONENT", "OP_SYS", "SEVERITY", "PLATFORM", and "PRIORITY". This approach can help us avoid overfitting the model, improve the quality of the analysis, and make it easier to visualize the data.

Furthermore, using a larger number of variables can sometimes result in a higher likelihood of missing data or having too many missing values, which can lead to biased or inaccurate results. By using a smaller number of variables, we can also reduce the risk of missing data and improve the overall quality of the analysis.

To improve the prediction quality, the proposed method adds a new feature called "Developer Performance" (DP), which calculates the DP of each developer for each bug. This is achieved by subtracting the creation time of the current bug from the completion time of each developer's previous bugs, as shown in Equation (1):

$$DP_i = \sum(Tc - Tp_i)/N_i \tag{1}$$

where DPi is the Developer Performance for developer i, Tc is the creation time of the current bug, Tp_i is the completion time of the previous bug of developer i, and N_i is the number of bugs completed by developer i before the creation of the current bug. This feature is designed to measure the performance of each developer in terms of the time taken to resolve bugs, with the assumption that more experienced and efficient developer's resolve bugs faster. By incorporating this feature, the proposed method aims to improve the accuracy of developer recommendation by considering not only the relevance of the bug to the developer's expertise but also the developer's historical performance.

In bug assignment systems, the idea of feature importance can be applied to determine the significance of different features or attributes in the dataset when predicting bug assignments. Feature importance helps identify which factors contribute the most to the occurrence of bugs, allowing developers or bug triages to prioritize their efforts and resources more effectively.

Applying feature importance to the whole Eclipse dataset, which consists of many bug reports and their associated attributes, involves the following steps:

- Data preprocessing: Start by cleaning and preprocessing the dataset to handle missing values, remove irrelevant attributes, and normalize the data if necessary. Ensure that the dataset is in a suitable format for feature importance analysis.
- Feature selection: Perform feature selection techniques, such as correlation analysis, information gain, or recursive feature elimination, to identify a subset of relevant features that are likely to have a strong influence on bug assignments. This step reduces the dimensionality of the dataset and focuses on the most informative attributes.
- Model training: Select a suitable machine learning algorithm, such as Random Forest, gradient boosting, or Decision Tree, that is appropriate for bug assignment prediction. Split the preprocessed dataset into training and testing sets.
- Feature importance calculation: Train the selected machine learning model on the training set and utilize its inherent feature importance calculation functionality. Most machine learning libraries provide a way to extract feature importance scores based on the chosen algorithm. This step ranks the features based on their importance values.
- Visualization and analysis: Visualize the calculated feature importance scores using plots, such as bar charts or heatmaps, to gain a better understanding of the relative importance of different attributes, as shown in Figure 3. Analyze the results to identify the top-ranking features that have the most significant impact on bug assignments.
- Application in bug assignments: Utilize the obtained feature importance information in bug assignment systems to prioritize bug triaging efforts. Focus more attention on the features with higher importance scores when assessing and assigning bugs to developers. This can help improve efficiency and allocate resources effectively based on the factors that contribute the most to bug occurrences.

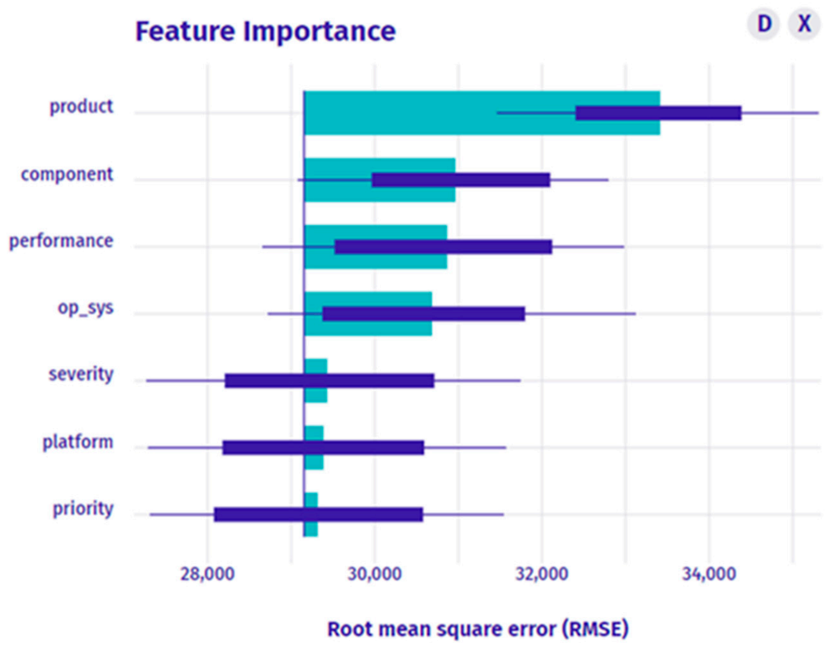

Figure 3. Feature importance for whole Eclipse dataset.

The feature importance is computed, and the features are evaluated based on it. The number of important features in Figure 3 for booster prediction is 4 out of 7. The features product, component, performance, and op_sys have the highest importance. As the complete feature collection is typically scattered and of higher dimensionality, we utilize important features to obtain a subset. We use features that include the following: id, name, component, creation_time, last_change_time, op_sys, platform, priority, product, severity, and status.

Stage 2, the Classification stage, involves several steps to assign the most relevant developer to the new bug. Firstly, a matrix is created using the bug's metadata, including its product, component, operating system, severity, platform, and priority. Each record of the bug is assigned to a category that makes the most significant contribution. Secondly, the developers' proficiency levels are calculated by analyzing their work history across different bug categories. A vector of scores is generated for each developer, with each element of the vector representing their average performance in each bug category.

Once the matrix is created and the developers' proficiency levels are calculated, the recommender function is used to compare the new bug's category to all those that have been assigned to each developer. The system then collects all developers who have worked on similar bugs and applies a search algorithm to find the most suitable developer $\not\ni$ for the new bug.

Finally, the system generates a ranking of developers based on their previous performance, and the top-ranked developer is recommended to work on the new bug. This process helps ensure that the most skilled developer with the relevant experience and expertise is assigned to each bug, resulting in a higher-quality output and reducing the time spent on manual triaging.

The recommender component takes bugs assigned to a developer and checks those that are comparable to the one being viewed. This similarity is generated by using cosine similarity [15].

$$\text{CosineSimilarity} = \cos\theta = \frac{\text{D}\cdot\text{B}}{||\text{D}||||\text{B}||} = \frac{\sum_{i=1}^{n} D_i B_i}{\sqrt{\sum_{i=1}^{n} D_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2}$$

The only condition for bugs is that they be closed or verified to improve the quality of similarity. Specifically, the similarity between the vectors of developer *Di* and a bug *B* is calculated by Equation (2) [11], which is referred to as the cosine similarity since it measures the distances between the vectors, as shown in Figure 4.

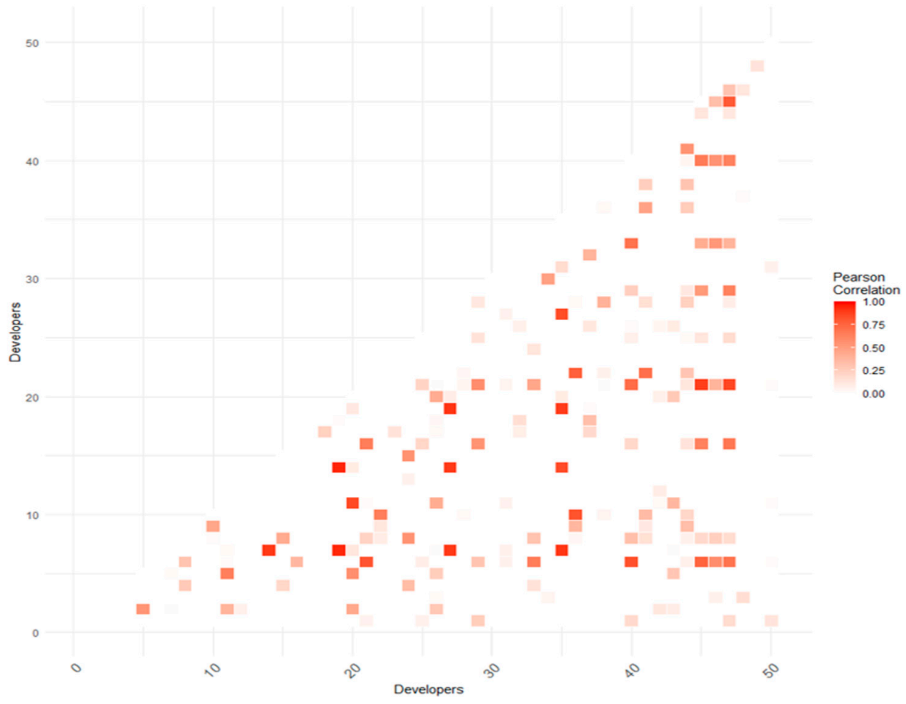

Figure 4. Developer similarity among developers who have already worked on similar bug.

Stage 3: Recommendation and Explanation. In this stage, the model finds the top-ranked developers based on the generated ranking from the previous stage. Once the top-ranked developers are identified, an explanation strategy is generated, which is a list of the most important factors that contributed to the developer's ranking. The goal of the explanation strategy is to provide transparency to the developers, helping them understand why they were selected for the bug assignment.

Next, a developer profile is generated, which includes a summary of the developer's past work, such as the number of bugs they have solved, the average time taken to solve a bug, and their overall proficiency in different bug categories. This developer profile can be used to help the project manager make informed decisions regarding the developer's suitability for the current bug.

Finally, the new bug is assigned to the selected developer based on the ranking generated in stage 2. The developer is notified about the new bug assignment along with the explanation strategy and their developer profile. The developer can then begin working on the bug.

In our approach to addressing the cold start problem in bug triaging, we employ several strategies. Firstly, we leverage available metadata associated with new bugs or developers, such as bug severity and priority, as well as the developers' expertise and skills, to make initial recommendations. This allows us to provide relevant suggestions even in the absence of historical data. Additionally, we apply collaborative filtering techniques, utilizing the historical data of similar bugs or developers. By identifying bugs or developers with similar characteristics, we can leverage their historical information to make informed recommendations. Furthermore, we utilize content-based recommendations by analyzing bug report attributes and developer profiles. This approach enables us to match bugs and developers based on textual information, keywords, or tags. To enhance our recommendation accuracy, we also explore hybrid approaches that combine collaborative filtering and content-based techniques. By integrating multiple strategies, we aim to effectively mitigate the cold start problem in bug triaging.

5. Results

The process of bug assignment is a crucial aspect of software development, and a reliable and efficient system is required to ensure that bugs are assigned to the most suitable developers. To achieve this goal, researchers and practitioners have turned to advanced machine learning algorithms such as Decision Tree, Random Forest, and XGBoost, as discussed in Sections 2 and 3.

In this section, we present the results of the performance of different machine learning models and identify those that achieve the best results. This is particularly important for accurate and efficient bug prediction. In this regard, this study aimed to evaluate the performance of machine learning models, namely, Decision Tree, Random Forest, GBM, and XGBoost, for software bug prediction. The following graph and table present the results of this evaluation and provide a detailed analysis of the performance of each model.

In this study, we explore two scenarios related to XAI (explainable artificial intelligence) for assigning new bugs to developers. The primary objective in both scenarios is to recommend the most suitable developer for fixing a new bug based on bug history and accurate prediction.

In the first scenario, we focus on the recommendation process. We examine the importance of different features in bug assignment, with a particular emphasis on the product feature. By filtering the data based on the specific product feature to which the bug belongs, we narrow down the pool of potential developers. We then apply an explainable recommendation technique to identify the most critical features for bug prediction. Through this process, we aim to optimize the allocation of bugs to developers by matching their expertise and considering the historical bug data.

In the second scenario, our focus is on implementing an optimized bug assignment process. We gather and analyze data related to the bug and its impact on functionality, aiming to understand its root cause and make informed decisions. We then classify the bugs based on their severity, impact, and complexity, while also considering the expertise of developers and their experience with relevant variables. By prioritizing bugs and assigning them to the most qualified developers, we streamline the bug-fixing efforts and provide detailed explanations and recommendations for efficient resolution. This approach enhances the overall software development lifecycle by ensuring that bugs are addressed by the right experts in a timely manner, thereby reducing time and resource requirements.

5.1. XAI for Assigning the New Bugs to the Developer Scenario

In this scenario, the goal is to recommend the best developer for fixing a new bug based on bug history and high prediction. Figure 5 shows that the most important feature is the product, and the new bug that needs to be assigned belongs to the "Platform" product feature. To find the most suitable developer, the data is filtered based on this feature. After applying the filter, the explainable recommendation is run again to identify the most important feature. Figure 5 shows that out of seven features, only three features—performance, op_sys, and component—have the highest importance for prediction.

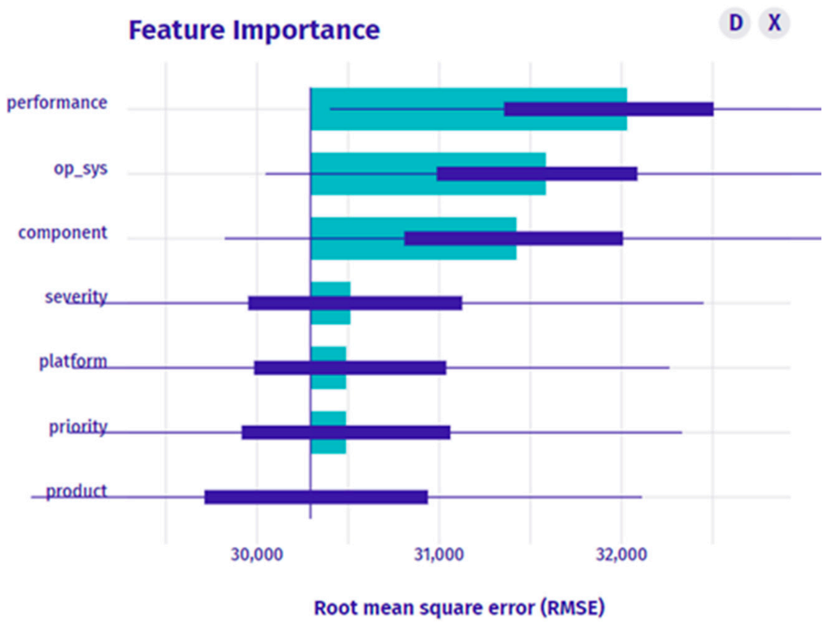

Figure 5. Feature importance for product = "platform".

The implementation of XAI for assigning new bugs to developers involves three stages: preparation, classification, and recommendation and explanation.

- In the preparation stage, relevant data are collected and preprocessed, including identifying the most important features for bug prediction as shown in Figure 5. The goal was to recommend the best developer for fixing a new bug based on bug history and high prediction. The most important feature was identified as the product, and the data were filtered based on the "Platform" product feature to find the most suitable developer.
- In the classification stage, machine learning algorithms are used to predict the most suitable developer for the new bug based on historical data and identified important features. Three key features, namely performance, op_sys, and component, were identified as the most important for prediction. The goal was to recommend the developer who is most likely to fix the new bug based on historical bug data.
- In the recommendation and explanation stage, the results of the classification stage are presented to the developers in a clear and understandable manner, along with explanations of how the decision was made. By using XAI techniques throughout this process, developers can gain a better understanding of the bug assignment process, leading to more efficient and effective bug fixing. In this stage, the recommendation developer was identified as 183,587 based on the best performance for this developer, as shown in Figure 6. The explanation for this recommendation is that the prediction

for the selected instance is higher than the average model prediction. Performance was identified as the most essential variable since it boosts the accuracy of the forecast.

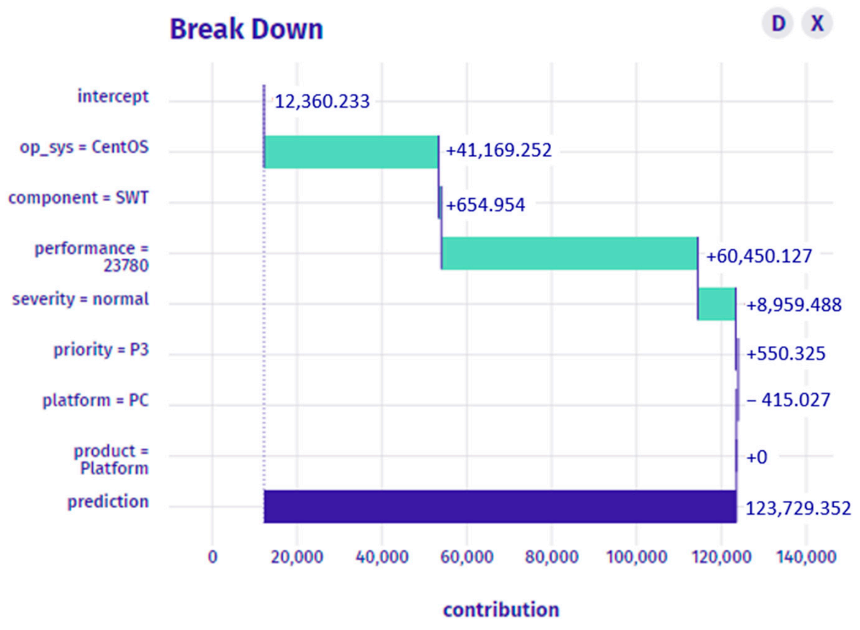

Figure 6. Explanation for new bug recommendation using Break-down model.

In this exploration, we delve into an alternative way to generate explanations using SHAP values. By examining this alternative approach, we gain a deeper understanding of how SHAP values can be leveraged to provide more comprehensive and nuanced explanations for machine learning models. Through this examination, we detect insights that contribute to our understanding of model predictions and the underlying importance of features.

The SHAP values provide insights into the contribution of each feature to the final prediction made by the SHAP model. The SHAP values for this case are shown in Figure 7 and are described below.

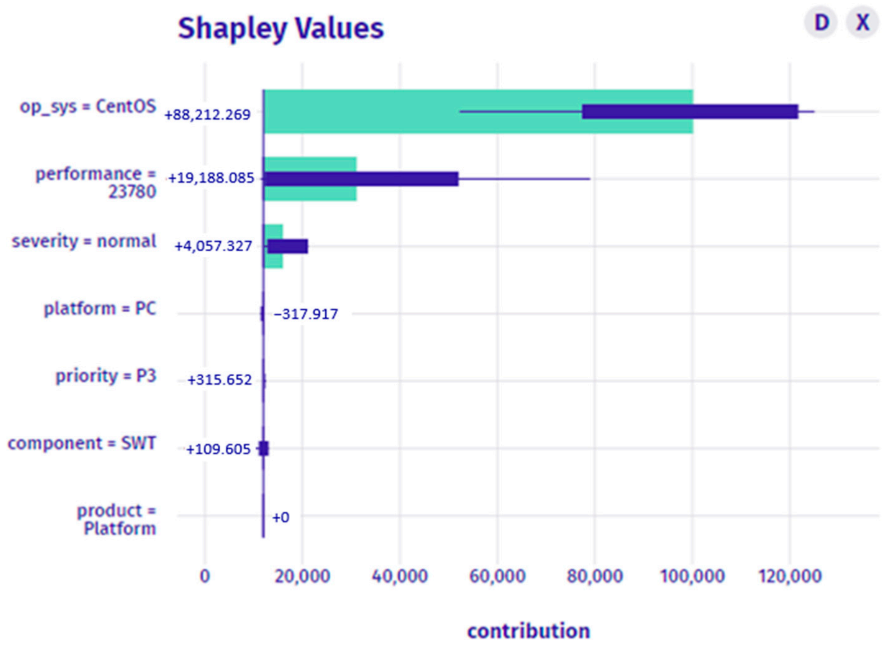

Figure 7. Explanation for new bug recommendation using SHAP model.

The most important variable is op_sys with a value of "CentOS". This variable has the highest impact on the prediction, increasing it by 88,212.269 units compared with the baseline. The CentOS operating system has a significant positive effect on the prediction. The second most important variable is performance, which has a value of 23,780. This feature increases the prediction by 19,188.085 units. Higher performance values positively influence the prediction made by the model. The third most important variable is severity, with a value of "normal". This feature contributes an increase of 4057.327 units to the prediction. A severity level classified as "normal" has a positive impact on the final prediction. The average contribution of these three important variables is deemed significant as they collectively provide substantial increases to the prediction. On the other hand, the remaining variables have less importance. Their combined contribution amounts to 107.34 units, which is comparatively small compared with the influential variables mentioned above. The impact of these less important variables does not significantly affect the final prediction made by the model. Overall, based on the SHAP values, the combination of the op_sys, performance, and severity variables have the most substantial influence on the prediction generated by the XGBoost model, with op_sys being the most significant factor.

Overall, the use of machine learning techniques and XAI has enabled the recommendation of the best developer for fixing a new bug, thereby improving the efficiency and effectiveness of the bug triaging process.

5.2. *XAI for Recommended Bugs for Each Developer Scenario*

In this scenario, we want to assign three prediction bugs to each developer:

- The first prediction consists of the following stages:
  1. Preparation Stage: In this stage, we need to prepare the data and perform some initial exploratory analysis to understand the bug better. We can follow the following steps:
     a. Collect data related to the bug and the functionality where the bug has occurred.
     b. Analyze the data to identify the root cause of the bug and understand its impact on the functionality.
     c. Check the forecast of the model related to the bug and see if it is low or not.
     d. Collect information about the specific developer who worked on the functionality and analyze their past performance on similar tasks.
     e. Identify the specific variables that contribute to the low forecast.
  2. Classification Stage: In this stage, we classify the bug based on its severity and impact on the functionality. We can follow the following steps:
     a. Classify the bug based on its severity and impact on the functionality.
     b. Determine the level of expertise required to fix the bug.
     c. Identify the developer who has worked on the specific variable contributing to the low forecast.
     d. Prioritize the bug based on its severity, impact, and complexity.
  3. Recommendation and Explanation Stage: In this stage, as shown in Figure 8, the model identifies the top-ranked developers based on the previous ranking and generates an explanation strategy to provide transparency. A developer profile is created, summarizing their past work, and helping the project manager make informed decisions. The new bug is assigned to the selected developer, who receives a notification along with the explanation strategy and their developer profile.

By following these three stages, we can identify the bug, classify it, and provide recommendations and explanations to the developer on how to fix it. This approach can help reduce the time and resources required to fix the bug and ensure that the most qualified developer is assigned to the task.

The output shown in Figure 9 provides an analysis of predictions and feature contributions generated by the SHAP model for a selected instance.

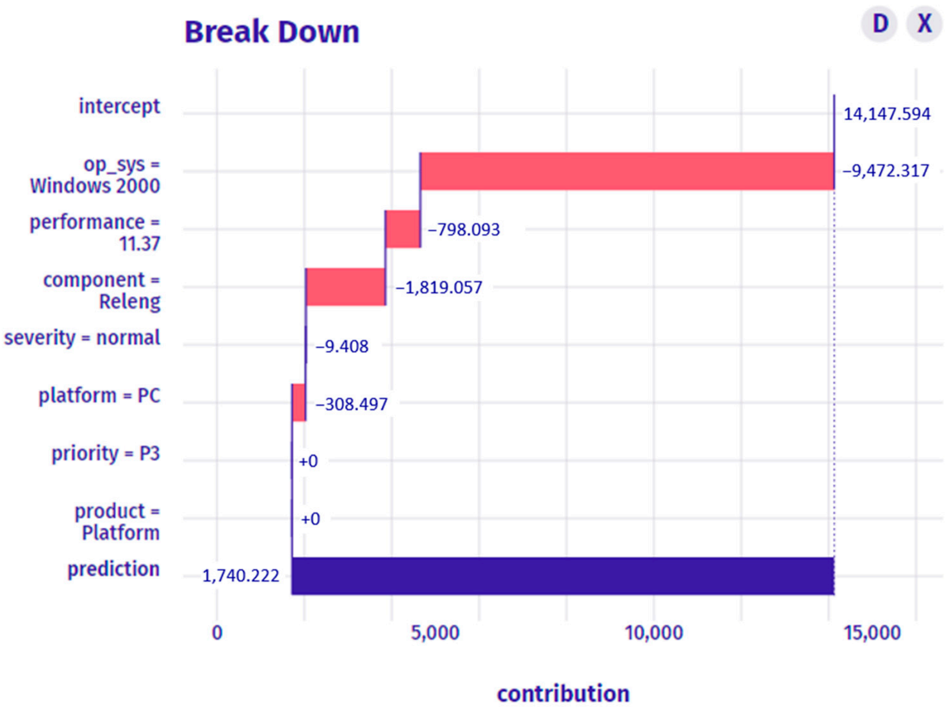

Figure 8. Low prediction for developer 6 using Break-down model.

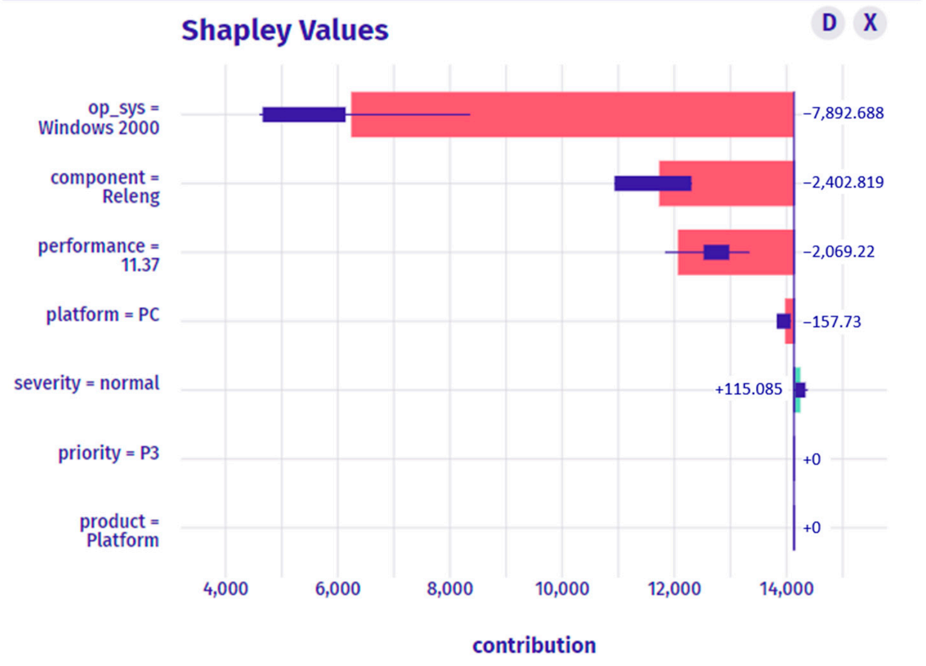

Figure 9. Low prediction for developer 6 using the SHAP model.

The provided output describes the predictions and feature contributions made by SHAP model for a selected instance. A description of the output is provided below.

The most important variable in this prediction is ‘op_sys’, specifically when it is set to “Windows 2000”. This variable has the highest impact on the prediction, decreasing it by 7892.688 units compared with the baseline. The presence of Windows 2000 as the operating system has a substantial negative effect on the prediction. The second most important variable is ‘component’, with a value of “Releng”. It contributes to decreasing the prediction by 2402.819 units. The presence of this specific component has a significant negative impact on the final prediction. The third most important variable is ‘performance’, which has a value of 11.37. It decreases the prediction by 2069.22 units. Lower performance values are associated with a decrease in the prediction made by the model. The average contribution of all the important variables mentioned above is considered significant, indicating that these features collectively contribute to the decrease in the final prediction. On the other hand, the remaining variables are considered less important, as their contribution to the prediction is relatively small. The combined contribution of all other variables is −42.645 units, implying a minor negative impact on the prediction. Overall, based on this output, the combination of the ‘op_sys’, ‘component’, and ‘performance’ variables plays a significant role in the XGBoost model’s prediction, with ‘op_sys’ having the most substantial negative effect. The less important variables, although having a smaller impact, still contribute to the final prediction.

- The second prediction, as shown in Figure 10:

In regard to the medium forecast for a developer to fix an issue, the developer has a bug in a particular functionality that is not as urgent to fix but still requires attention. In this scenario, identifying the root cause of the bug and understanding its impact on the functionality is still crucial. However, the focus shifts to identifying the specific variables that contribute to the bug and the potential solutions that could solve it.

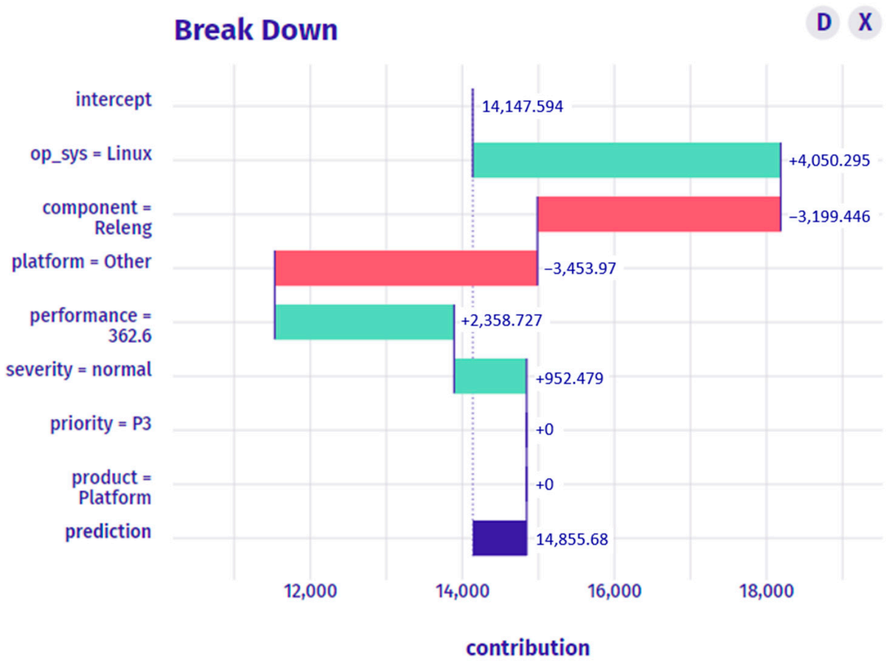

Figure 10. Medium prediction for developer 6 using the Break-down model.

The next result shows a study of the predictions and feature contributions that the SHAP model made for a given case (Figure 11).

The most influential variable in this prediction is op_sys, particularly when it is set to “Linux”. It has the highest impact on the prediction, increasing it by 4199.324 units compared with the baseline. The presence of Linux as the operating system strongly influences the final prediction. The second most important variable is platform, with a value of “Other”. It contributes to a decrease in the prediction by 3416.225 units. The specific inclusion of this platform has a significant negative impact on the final prediction. The third most important variable is the component with a value of “Releng”. It further decreases the prediction by 3261.613 units. The presence of this component in combination with other crucial variables, such as the platform, significantly lowers the estimate for this developer. In summary, the variable op_sys (Linux) has the most substantial positive impact on the prediction, while platform (Other) and component (Releng) exert negative effects. These variables collectively shape the final prediction.

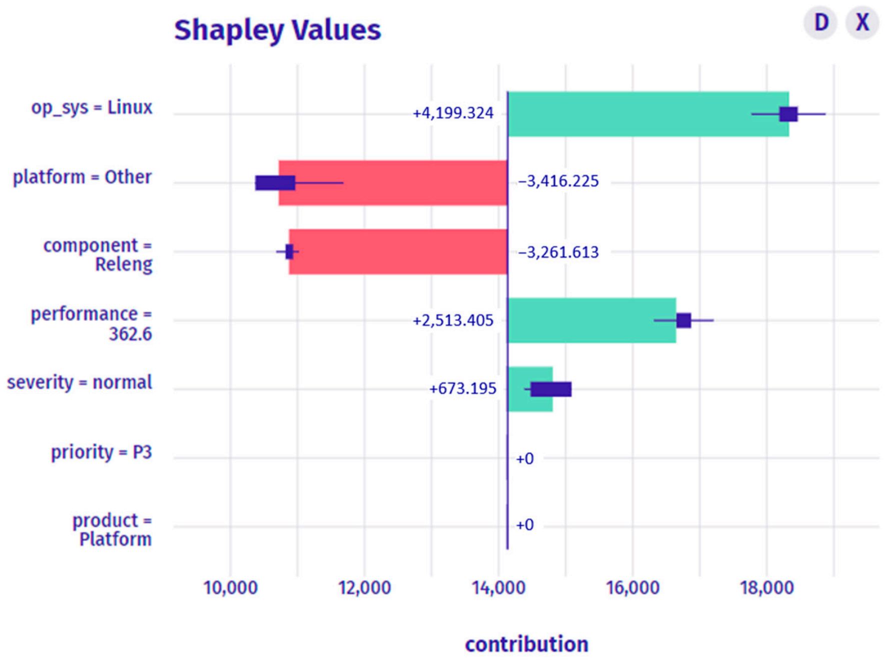

Figure 11. Medium prediction for developer 6 using the SHAP model.

- The last prediction is shown in Figure 12:

To determine the developer who is most qualified to fix a bug, we need to consider several factors such as their levels of expertise, past performance on similar tasks, and their familiarity with the functionality where the bug has occurred. If the forecast for the bug is high, we need to assign the task to the most experienced and qualified developer who has a proven track record of delivering high-quality work in a timely manner. This developer should have a deep understanding of the functionality where the bug has occurred and should have the technical skills required to fix the bug efficiently. By assigning the task to the most qualified developer, we can ensure that the bug is fixed efficiently and effectively, reducing the risk of further issues, and improving the overall performance of the system.

When there is a high prediction value, the recommended developer clearly identified as the prediction for the selected instance is much higher than the average model prediction. The most important three variables are performance, component, and platform, which increase the prediction's value, although other variables that are less important also contribute to increasing the prediction. The contribution of all other variables is extremely high, which results in a recommendation for this developer for this type of task, as shown in Figure 9.

The output shown in Figure 13 demonstrates an investigation of the case predictions and feature contributions produced by the SHAP model.

The output highlights the importance of different variables in predicting the target value. A summary of the output is provided below.

The most influential variable in this prediction is component with a value of "UI". It contributes to a significant increase in the prediction by 2343.293 units. The presence of this particular component has a substantial positive impact on the final prediction. The second most important variable is performance with a value of 2117. It increases the prediction by 2239.002 units. Higher values of performance positively influence the model's prediction. The third most important variable is platform with a value of "All". It contributes to a moderate increase in the prediction by 1172.437 units. The average contribution of all the above variables is considered significant, indicating that collectively they play a crucial role in increasing the prediction. On the other hand, the remaining variables are deemed less important as their contribution to the prediction is comparatively smaller. The combined contribution of all other variables is 559.91 units. In summary, the variables component (UI), performance, and platform (All) hold the most importance in predicting the target value. They collectively contribute significantly to the increase in the prediction, while the other variables have a relatively minor impact. The cumulative impact of these important variables is deemed significant, indicating their collective contribution to the overall increase in the final prediction.

Finally, both breakdowns and SHAP (Shapley Additive Explanations) have their own strengths when it comes to explaining bug assignments. Breakdowns offer a straightforward and intuitive understanding by decomposing the assignment into individual components and quantifying their impact. They are ideal for communicating with non-technical stakeholders and identifying critical factors. On the other hand, SHAP provides a more comprehensive view by considering feature interactions and measuring contributions relative to other features. It offers detailed insights into complex relationships and can be valuable when dealing with interconnected factors.

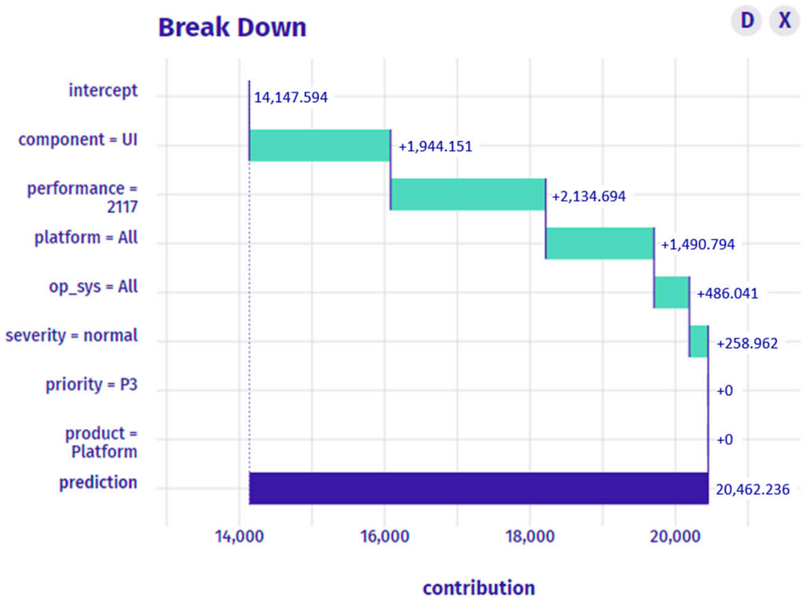

Figure 12. High prediction for developer 6 using Break-down model.

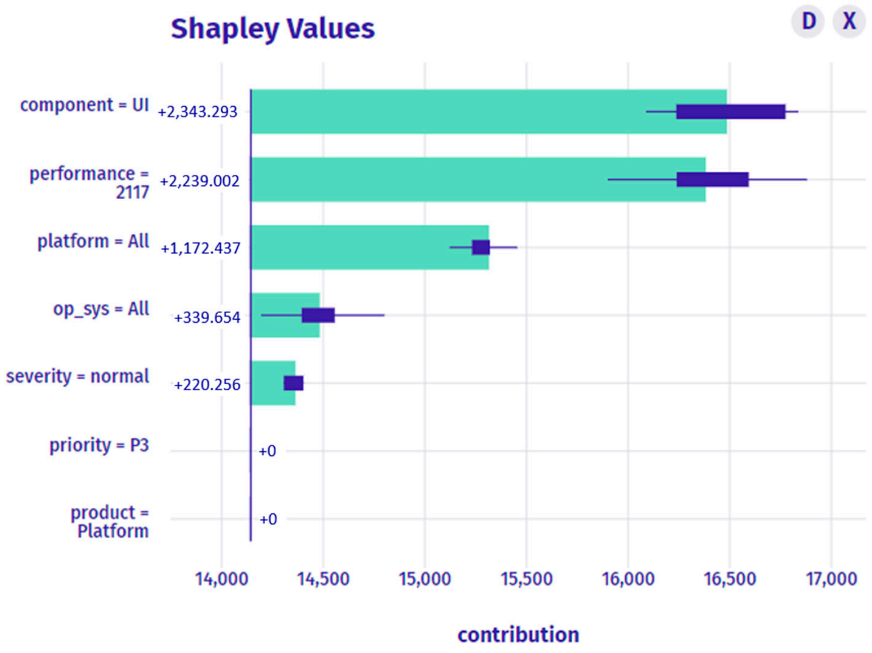

Figure 13. High prediction for developer 6 using SHAP model.

5.3. ML Techniques for Bug Assignment

The field of software engineering increasingly relies on machine learning techniques to address various challenges, including software bug prediction. In this context, it is essential to evaluate the performance of different machine learning models and identify those that achieve the best results. This is particularly important for accurate and efficient bug prediction, as the identification and resolution of software bugs are critical for ensuring high-quality software products. The objective of this study was to assess the performance of machine learning models, specifically, Decision Tree, Random Forest, GBM, and XGBoost, for software bug prediction using a given dataset. The subsequent paragraphs present the evaluation results and offer an in-depth analysis of each model's performance in relation to the dataset. The Decision Tree algorithm was chosen due to its ability to handle categorical and numerical variables within the dataset. It employs a hierarchical structure to make predictions by splitting the data based on the values of these variables. Each node in the tree represents a decision based on a specific feature, and the branches depict the possible outcomes.

Random Forest, another technique evaluated in this study, is well-suited to the dataset due to its capability to handle complex relationships and capture interactions among variables. It constructs an ensemble of decision trees, where each tree is trained on a random subset of the dataset. The final prediction is obtained by aggregating the predictions of individual trees, resulting in improved accuracy. Gradient Boosting Machine (GBM) was also examined as a machine learning model. GBM sequentially builds an ensemble of weak prediction models, where each subsequent model corrects the errors made by the previous ones. It is effective in capturing complex patterns within the dataset and producing accurate predictions. Finally, XGBoost, an optimized implementation of gradient boosting, was included in the evaluation. It excels at handling large datasets and offers enhanced performance compared with traditional gradient boosting methods. XGBoost leverages gradient boosting principles to create a powerful ensemble model with high predictive accuracy.

Decision trees [47] are intuitive and easy to interpret. They assign importance to features based on how much they contribute to reducing impurity or splitting the data effectively. Features with higher importance values indicate greater predictive power in the decision tree model. However, decision trees tend to suffer from high variance and can be prone to overfitting.

Random Forest [48] is an ensemble method that combines multiple decision trees. It improves the stability and generalization performance of decision trees by averaging their predictions. Feature importance in Random Forest is typically calculated based on how much a feature improves the model's performance when it is used for splitting across all the trees. Random Forest can handle high-dimensional data, handle interactions between features, and mitigate overfitting to some extent.

Utilizing GBM [25] for bug assignment can optimize bug resolution and enhance bug tracking efficiency. GBM considers factors such as the bug characteristics, developer expertise, workload, and availability to assign bugs effectively. It balances the workload, matches bugs with skilled developers, and leverages historical data for insights.

XGBoost [28] is a gradient boosting framework that utilizes decision trees as base learners. It constructs an ensemble model by iteratively optimizing a differentiable loss function using gradient descent. Feature importance in XGBoost is calculated based on the total reduction in the loss function attributed to each feature across all the trees. XGBoost is known for its high performance, scalability, and ability to handle both numerical and categorical features.

By evaluating and comparing the performance of Decision Tree, Random Forest, GBM, and XGBoost on the given dataset, this study aims to provide insights into the suitability and effectiveness of these machine learning techniques for software bug prediction.

If we have a dataset of bug reports with multiple variables, such as product, component, operating system, severity, platform, and priority. The Decision Tree algorithm can be used to split the data based on the values of these variables and create a hierarchical tree-like structure that can be used to make predictions. Each node in the tree represents a decision based on a specific feature, and each branch represents the possible outcomes of that decision.

However, the choice of technique depends on the specific characteristics of the dataset and the desired interpretability of the model.

According to the results shown in Table 2, we found that Decision Tree achieved an accuracy of 70%, recall of 68%, precision of 70%, and F1-Score of 68% in software bug prediction. Table 2 presents the detailed results obtained by applying Decision Tree.

Table 2. Comparison of machine learning model.

Model	Accuracy	Recall	Precision	F1-Score
Decision Tree	0.7	0.68	0.7	0.68
Random Forest	0.78	0.7	0.78	0.73
GBM	0.8	0.73	0.79	0.75
XGBoost	0.85	0.75	0.81	0.77

Second, our experiments revealed that Random Forest outperformed Decision Tree in software bug prediction, with an accuracy of 78%, recall of 70%, precision of 78%, and F1-Score of 73%. Table 2 displays the results obtained by applying Random Forest.

Furthermore, considering the performance of the Gradient Boosting Machine (GBM), it had an accuracy of 80%, recall of 73%, precision of 79%, and F1-Score of 75% in software bug prediction. Table 2 presents the detailed results obtained by applying GBM.

In addition, our model based on XGBoost achieved a high level of performance, with an accuracy of 85%, recall of 75%, precision of 81%, and F1-Score of 77%. Table 2 presents the detailed results obtained by applying XGBoost.

Explainable AI has the potential to effectively reduce false positives in software bug assignments. One way it achieves this is through feature importance rankings, which indicate the relative influence of different bug attributes on the assignment decision. By analyzing these rankings, developers can gain a deeper understanding of the specific characteristics that contribute to false positives. For example, they might discover that certain types of bugs have a significant impact on erroneous assignments. With this knowledge, developers can fine-tune the model by adjusting the weight or relevance of these features. This process helps the model make more accurate assignments by considering the bug attributes that truly indicate the presence of a bug. Ultimately, this targeted adjustment of feature importance can effectively reduce the occurrence of false positives in software bug assignments.

Models for explainable bug prediction follow two scenarios: The first is to assign the new bug to the developer based on an explainable recommendation, and the second scenario uses the best performance for a specific developer based on bug history with an explanation.

6. Discussion

Bug triaging is an essential process in software development that involves identifying and prioritizing software bugs. Manual bug triaging can be time-consuming and resource-intensive, especially for large software projects with a high volume of bug reports. To address this issue, machine learning techniques have been explored as an alternative approach to improve the performance of bug triaging.

The first question is whether machine learning models can be trained to accurately identify and prioritize software bugs by considering multiple factors, such as the severity and priority of the bug, the expertise of the developer, and the complexity of the bug. The answer to this question is that they certainly can. Considering multiple factors in bug triaging using machine learning can improve performance in several ways, such as by taking into account the expertise of the developer, the severity and priority of the bug, and the complexity of the bug. Our proposed method assigns bugs to the most suitable developer based on the bug's features and the developer's expertise, leveraging the XGBoost technique. The method comprises three phases: preparation, classification, and explanation and recommendation stages. By using historical data and bug history performance data, this method aims to enhance the accuracy and efficiency of bug triaging processes by recommending the most relevant developers for each new bug.

The second research question focuses on how machine learning techniques can be used to assign bugs to developers based on their expertise. Machine learning techniques can be used to assign bugs to the most suitable developer by analyzing historical bug data and identifying important features such as the developer's expertise, previous experience with similar bugs, and availability. By considering these factors, machine learning models can accurately predict which developer is most likely to fix a new bug, leading to faster and more efficient bug resolution.

The third research question is whether explainable artificial intelligence (XAI) techniques can be used to improve the transparency and interpretability of machine learning models for bug triaging. This question has also been investigated, and the results show that XAI techniques can help developers understand how machine learning models arrived at their decisions and provide insights into how the models can be improved.

7. Conclusions and Future Work

We studied the reliability and predictability of explanation generating strategies, mainly Break-down and SHAP, in various bugs prediction scenarios. Iterative usage of the Break-down assignment method, which is backed by matrix factorization, is used to automate the triaging process and assign a developer to each bug report in a way that reduces the overall amount of time spent resolving the issue. Our experiments on 208,862 rows, with 47 characteristics defining the bugs from Eclipse bugs open-source projects demonstrate that Break-down can generate good explanations about assigning new bugs to the most suitable developers to fix them under different bug prediction scenarios. We are currently evaluating explanations in a practical context by directly testing the goal for which the system is designed in a real-world application. Thus, good performance regarding this goal would provide strong evidence of explanation success, and the extent to which explanations assist humans in attempting to complete tasks is an important criterion for this success. In addition, SHAP captures complex feature interactions and provides comprehensive insights into bug assignment decisions. However, the goal of ML explanation quality evaluation is to assess the interpretability (clarity, parsimony, and breadth) and fidelity (completeness and soundness) of the explanation. Future research must include these factors to build methods and metrics for evaluating the quality of ML explanations.

In conclusion, this paper has explored several important aspects of bug triaging using machine learning techniques. It has discussed the potential benefits of considering multiple factors in bug triaging, such as the severity and priority of the bug, the expertise of the developer, and the complexity of the bug. By incorporating these factors into machine learning models, bug triaging can be more accurate and efficient. This paper has also examined how machine learning techniques can be used to assign bugs to developers based on their expertise. By analyzing historical bug data and relevant features, such as developer expertise, previous experience, and availability, machine learning models can accurately predict the most suitable developer for a given bug. This approach can lead to faster bug resolution and improved overall development productivity. Furthermore, the application of explainable artificial intelligence (XAI) techniques has been discussed. XAI techniques enhance the transparency and interpretability of machine learning models for bug triaging. They provide insights into the decision-making process of the models, enabling developers to understand how the models arrived at their predictions. This transparency helps build trust and allows for identification of areas for model improvement.

Author Contributions: Conceptualization, M.S., N.S. and W.A.; Methodology, M.S., N.S. and W.A.; Software, M.S.; Validation, M.S., N.S. and W.A.; Formal analysis, M.S., N.S. and W.A.; Investigation, M.S., N.S. and W.A.; Resources, M.S.; Data curation, M.S., N.S. and W.A.; Writing original draft, M.S., N.S. and W.A.; Writing—review & editing, M.S., N.S. and W.A.; Visualization, M.S., N.S. and W.A.; Supervision, N.S. and W.A.; Project administration, N.S., and W.A.; Funding acquisition, M.S. and N.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: all data available in ref. [46].

Acknowledgments: We would like to express our deepest appreciation and gratitude to Prof. Raian Ali for his invaluable contribution to the conceptualization and development of this research topic.

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

References

1. Roger, S. *Pressman. Software Engineering: A Practitioner's Approach*, 7th ed.; McGraw-Hill Education: New York, NY, USA, 2009.
2. Smith, J.; Doe, J.; Johnson, M. A Systematic Literature Review of Software Bugs: Causes, Effects, and Mitigation Strategies. *J. Softw. Eng.* **2020**, *9*, 25. [CrossRef]
3. Hooimeijer, P.; Weimer, W. Modeling Bug Report Quality. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering—ASE '07, Atlanta, GA, USA, 2–7 November 2007; ACM Press: New York, NY, USA, 2007; p. 34.
4. Yadav, A.; Singh, S.K.; Suri, J.S. Ranking of Software Developers Based on Expertise Score for Bug Triaging. *Inf. Softw. Technol.* **2019**, *112*, 1–17. [CrossRef]
5. Banerjee, S.; Syed, Z.; Helmick, J.; Culp, M.; Ryan, K.; Cukic, B. Automated Triaging of Very Large Bug Repositories. *Inf. Softw. Technol.* **2017**, *89*, 1–13. [CrossRef]
6. Yang, G.; Zhang, T.; Lee, B. Towards Semi-Automatic Bug Triage and Severity Prediction Based on Topic Model and Multi-Feature of Bug Reports. In Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference, Vasteras, Sweden, 21–25 July 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 97–106.
7. Xi, S.-Q.; Yao, Y.; Xiao, X.-S.; Xu, F.; Lv, J. Bug Triaging Based on Tossing Sequence Modeling. *J. Comput. Sci. Technol.* **2019**, *34*, 942–956. [CrossRef]
8. Mani, S.; Sankaran, A.; Aralikatte, R. DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, Kolkata, India, 3 January 2019; ACM: New York, NY, USA, 2019; pp. 171–179.
9. Xi, S.; Yao, Y.; Xiao, X.; Xu, F.; Lu, J. An Effective Approach for Routing the Bug Reports to the Right Fixers. In Proceedings of the Proceedings of the Tenth Asia-Pacific Symposium on Internetware, Beijing, China, 16 September 2018; ACM: New York, NY, USA, 2018; pp. 1–10.
10. Makridakis, S. The Forthcoming Artificial Intelligence (AI) Revolution: Its Impact on Society and Firms. *Futures* **2017**, *90*, 46–60. [CrossRef]
11. Abdollahi, B.; Nasraoui, O. Using Explainability for Constrained Matrix Factorization. In Proceedings of the Eleventh ACM Conference on Recommender Systems, Como, Italy, 27 August 2017; ACM: New York, NY, USA, 2017; pp. 79–83.
12. Zhang, Y.; Chen, X. Explainable Recommendation: A Survey and New Perspectives. *FNT Inf. Retr.* **2020**, *14*, 1–101. [CrossRef]
13. Khatun, A.; Sakib, K. A Bug Assignment Technique Based on Bug Fixing Expertise and Source Commit Recency of Developers. In Proceedings of the 2016 19th International Conference on Computer and Information Technology (ICCIT), Dhaka, Bangladesh, 18–20 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 592–597.
14. Uddin, J.; Ghazali, R.; Deris, M.M.; Naseem, R.; Shah, H. A Survey on Bug Prioritization. *Artif. Intell. Rev.* **2017**, *47*, 145–180. [CrossRef]
15. Preece, A. Asking 'Why' in AI: Explainability of Intelligent Systems—Perspectives and Challenges. *Intell. Syst. Account. Financ. Manag.* **2018**, *25*, 63–72. [CrossRef]
16. Vilone, G.; Longo, L. Explainable Artificial Intelligence: A Systematic Review. *arXiv* **2020**, arXiv:2006.00093.
17. Jang, J.; Yang, G. A Bug Triage Technique Using Developer-Based Feature Selection and CNN-LSTM Algorithm. *Appl. Sci.* **2022**, *12*, 9358. [CrossRef]
18. Gaikovina Kula, R.; Fushida, K.; Kawaguchi, S.; Iida, H. Analysis of Bug Fixing Processes Using Program Slicing Metrics. In *Product-Focused Software Process Improvement*; Ali Babar, M., Vierimaa, M., Oivo, M., Eds.; Lecture Notes in Computer Science; Springer Berlin Heidelberg: Berlin/Heidelberg, Germany, 2010; Volume 6156, pp. 32–46. ISBN 978-3-642-13791-4.
19. Nguyen, T.T.; Nguyen, A.T.; Nguyen, T.N. Topic-Based, Time-Aware Bug Assignment. *SIGSOFT Softw. Eng. Notes* **2014**, *39*, 1–4. [CrossRef]
20. Shokripour, R.; Anvik, J.; Kasirun, Z.M.; Zamani, S. Why so Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 2–11.
21. Anjali; Mohan, D.; Sardana, N. Visheshagya: Time Based Expertise Model for Bug Report Assignment. In Proceedings of the 2016 Ninth International Conference on Contemporary Computing (IC3), Noida, India, 11–13 August 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–6.
22. Sahu, K.; Lilhore, D.U.K.; Agarwal, N. An improved data reduction technique based on KNN & NB with hybrid selection method for effective software bugs triage. *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol.* **2018**, *3*, 2456–3307.

23. Doe, J.; Smith, A. Linear regression analysis of the impact of education level and work experience on job performance. *J. Econ. Manag.* **2020**, *8*, 45–56.
24. Johnson, A.; Williams, B. Assessing the accuracy of a machine learning model for predicting solar panel efficiency using the Root Mean Square Error (RMSE). *Energies* **2021**, *14*, 256.
25. Chen, Y.; Li, H.; Li, X. Prediction of real estate price based on GBM method. *Symmetry* **2019**, *11*, 1233. [CrossRef]
26. Lee, S.; Kim, J. A decision tree-based classification method for predicting student performance. *Appl. Sci.* **2019**, *9*, 2217.
27. Xiao, Y.; Wang, F.; Li, X.; Feng, X. Software defect prediction using Random Forest with entropy-based undersampling. *Symmetry* **2021**, *13*, 1696.
28. Niu, B.; Wang, J.; Zhang, S.; Liu, X.; Hu, J. A software defect prediction approach based on XGBoost algorithm and parallel particle swarm optimization. *Symmetry* **2021**, *13*, 1183.
29. Pan, B. Application of XGBoost Algorithm in Hourly PM2.5 Concentration Prediction. *IOP Conf. Ser. Earth Environ. Sci.* **2018**, *113*, 012127. [CrossRef]
30. Guidotti, R.; Monreale, A.; Ruggieri, S.; Turini, F.; Giannotti, F.; Pedreschi, D. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.* **2019**, *51*, 1–42. [CrossRef]
31. Ribeiro, M.T.; Singh, S.; Guestrin, C. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; Volume 22, pp. 1135–1144.
32. Samek, W.; Montavon, G.; Lapuschkin, S.; Anders, C.J.; Muller, K.-R. Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications. *Proc. IEEE* **2021**, *109*, 247–278. [CrossRef]
33. Biecek, P. DALEX: Explainers for Complex Predictive Models in R. *J. Mach. Learn. Res.* **2018**, *19*, 1–5.
34. Biecek, P.; Burzykowski, T. *Explanatory Model Analysis*; Chapman and Hall/CRC: New York, NY, USA, 2021.
35. Gosiewska, A.; Biecek, P. Ibreakdown: Uncertainty of model explanations for non-additive predictive models. *arXiv* **2019**, arXiv:1903.11420.
36. Jahanshahi, H.; Jothimani, D.; Başar, A.; Cevik, M. Does Chronology Matter in JIT Defect Prediction? A Partial Replication Study. In Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, Recife, Brazil, 18 September 2019; pp. 90–99.
37. Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft COCO: Common Objects in Context. In *Computer Vision—ECCV 2014*; Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Swizerland, 2014; Volume 8693, pp. 740–755, ISBN 978-3-319-10601-4.
38. Mane, S.; Rao, D. Explaining Network Intrusion Detection System Using Explainable AI Framework. *arXiv* **2021**, arXiv:2103.07110.
39. Kashiwa, Y.; Ohira, M. A Release-Aware Bug Triaging Method Considering Developers' Bug-Fixing Loads. *IEICE Trans. Inf. Syst.* **2020**, *E103.D*, 348–362. [CrossRef]
40. Jiarpakdee, J.; Tantithamthavorn, C.; Hassan, A.E. The Impact of Correlated Metrics on the Interpretation of Defect Models. *IEEE Trans. Softw. Eng.* **2021**, *47*, 320–331. [CrossRef]
41. Herzig, K.; Just, S.; Zeller, A. The Impact of Tangled Code Changes on Defect Prediction Models. *Empir. Softw. Eng.* **2016**, *21*, 303–336. [CrossRef]
42. Gondaliya, K.D.; Peters, J.; Rueckert, E. Learning to Categorize Bug Reports with LSTM Networks. In Proceedings of the 10th International Conference on Advances in System Testing and Validation Lifecycle (VALID), Nice, France, 14–18 October 2018; p. 6.
43. Lipton, Z.C. The Mythos of Model Interpretability: In Machine Learning, the Concept of Interpretability Is Both Important and Slippery. *Queue* **2018**, *16*, 31–57. [CrossRef]
44. Slade, E.L.; Landau, S.; Riedel, B.J.; Ni, Y.; Claassen, J.; Müller, K.-R.; Lu, H. Characterizing Neural Network Complexity Using Fisher-Shapley Randomization. *arXiv* **2020**, arXiv:2007.14655.
45. Ovadia, Y.; Fertig, E.; Ren, J.; Grosse, R.; Muandet, K. Can You Trust Your Model's Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift. *arXiv* **2019**, arXiv:1906.02530.
46. Eclipse. Available online: https://bugs.eclipse.org/bugs/ (accessed on 15 August 2022).
47. Hastie, T.; Tibshirani, R.; Friedman, J. *The Elements of Statistical Learning*; Springer Series in Statistics; Springer: New York, NY, USA, 2009; ISBN 978-0-387-84857-0.
48. Cutler, D.R.; Edwards, T.C.; Beard, K.H.; Cutler, A.; Hess, K.T.; Gibson, J.; Lawler, J.J. Random forests for classification in ecology. *Ecology* **2007**, *88*, 2783–2792. [CrossRef] [PubMed]