

Article

Not peer-reviewed version

Improving Bug Assignment and Developer Allocation in Soft-ware Engineering through Interpretable Machine Learning Models

[Mina Samir Shenouda](#)*, [Nada H. Sherief](#), [Walid M. Abdelmoez](#)

Posted Date: 22 May 2023

doi: 10.20944/preprints202305.1488.v1

Keywords: Explainability; Explainable AI; XAI; Recommendation; Bugs



Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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 (M.S.);

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 the suitable developer to new bug by analyzing the history of developers' profiles and analyzing 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 can't 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. In this paper, we propose two explainable models for recommendation. The first one 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 generate the best developer for each bug from bug history.

Keywords: Explainability; Explainable AI; XAI; Recommendation; Bugs.

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. To better understand the causes, effects, and mitigation strategies associated with software bugs [2], as well as by the end user. Bugs can never be entirely eradicated. 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 to 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 [7,8,9]. 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 to, open date; when the developer begins working on the problem; and closed date; 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 to 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 a more accurate bug fixing technique have been explored in previous bug fixing research, 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 [13]. 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 [14] discuss and analyze the various intricacies that are involved in defining explainability and interpretability of neural networks. We are on board with 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 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:

- 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.
- 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 out 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 almost 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 in 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 method proposes that, upon getting a new issue, it suggests developers by retrieving the most appropriate developers based on bug history and developer's profile as an input. It is a supervised machine learning technique that generates a list of recommended developers based on a previous profile of the developer. Developer Similarity (DS) that use developer knowledge to fix issues across many features of bugs like product, severity, and component. Also, bug handling time (HT), and Effectiveness in fixing several bugs (EB). We propose ABDR training model. The model generates developer profile according to three mentioned factors: DS, HT, and EB. We assign each developer and each bug a bug prediction score based on the 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 mentioned techniques in section 2. Section 4 discusses the experimental study of our approach. Section 6 presents models used for explainable bug recommendations. Finally, Section 7 concludes our approach.

2. Background

In this section, a detailed background for related topics 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. The delay in resolving bugs [15] is caused by the constant reassignment procedure. We will be reviewing 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 their skills or workload. 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 done 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 to 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 to 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.

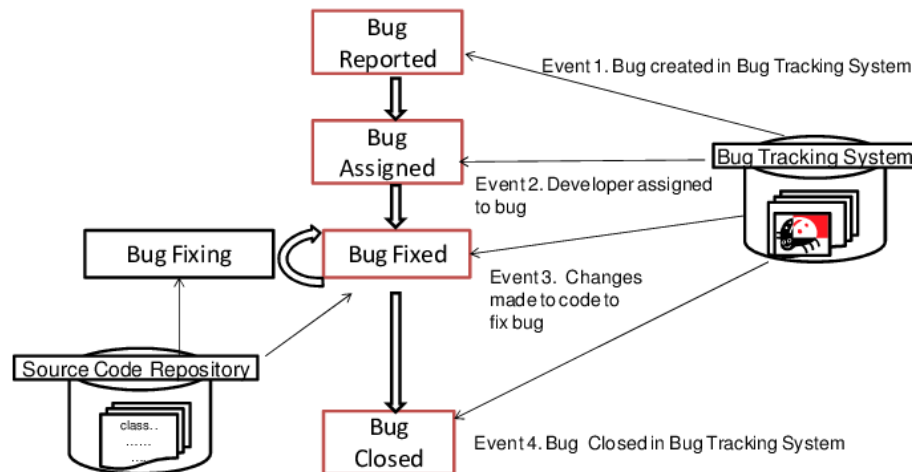


Figure 1. General Bug Fixing Model [16].

The most common description of new bug goes like this: If you get 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 [17]. Some approaches aim to reduce the time-to-fix the problems rather than maximize the knowledge of the potential assignee. For instance, in [18], 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 of the researchers explored the files and metadata to find connections between developers and the newly discovered issues using location-based techniques [19]. In the beginning, they locate or forecast the presence of issues like 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 focused on IR-based activity profiling because it typically results in higher accuracies [20]. This study [21] used the feature selection method to cut down on data volume. The characteristics of the dataset are extracted using a hybrid of the K-Nearest Neighbor and Naive Bayes methods. Finally, researchers looked at 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 is 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. This type of learning is often used in clustering, anomaly detection,

and feature extraction tasks. In software engineering, unsupervised learning has been applied to tasks such as software clustering, anomaly detection, and feature extraction, among others.

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 [20] 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 lists describes these classifiers in detail:

- **Regression [22]:** It 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) [23]:** 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) [24]:** 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 [25]:** It is a method known as Supervised Machine Learning, in which you specify what the input and the output will be related to the data that is trained. In this method, the data is split in a continuous 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 [26]:** it 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 [27]:** The XGBoost algorithm is an effective tool for machine learning. It has only lately been made available. This falls under the category of Supervised Learning. Gradient boosting may be thought of as its fundamental central concept. XG Boost 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 XG Boost 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 [28]. XGBoost (eXtreme Gradient Boosting) [28] is a popular machine learning algorithm that has been used for bug assignment in software engineering. There are 5 reasons why XGBoost is a good choice for bug assignment: first 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. The second reason has a regularization parameter that helps to prevent overfitting and improves the generalization performance of the model. Third one 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 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. Last reason 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 for 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 [29]. 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 like deep neural networks has become more important in recent years [14].

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, break-down, textual explanations of visual models, and SHAP. 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 [30]. On the other hand, LIME [31] 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 [32]. 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 [33] 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 [34] demonstrate this method for many different datasets. As a result, this method is good for both computational efficiency and interpretation. The Breakdown library [35] 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. 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 will 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; 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 [36].

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 [37], 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 make sure that everyone gets the same number of rewards in cooperative games.

We will 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. Therefore, 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

Research [38] 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.

The research [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 to 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, such as developer experience and workload, and testing on datasets from other software projects.

The core idea for this paper [40] investigates the impact of correlated metrics on the interpretation of defect models, 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 defect models.

The purpose of this study [41] is 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.

The 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. The 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 to 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. This paper [34] presents a method for predicting fault-proneness in software using random forests. The authors 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.

This study [35] looks at whether the chronological order of data used in just-in-time (JIT) defect prediction models affects their performance. The authors partially replicate a previous study and evaluate 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 paper [8] used supervised learning techniques, specifically deep learning, to develop an automated bug triaging system called DeepTriage. 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 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 to 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, such as developer experience and workload, and testing on datasets from other software projects.

3.2. Unsupervised Learning Techniques

The research [43] 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 to 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. Also, supervised learning offers interpretability and explainability. Since the training data is 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.

3.3. Explainability Models for Recommender-Based ML Techniques

Lipton et al. [44] applied the Textual Sentence Explanations approach to an increasing amount 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 made

recently to explain recommendations by using different kinds of textual information. These models usually come up with a textual phrase that explains the suggestion.

In [37], This paper proposes 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 experimental evaluation section presents the results of the evaluation, which 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.

This paper [45] 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 effect of perturbing those features on the model's output.

This research [46] presents 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.

Previous studies approached the subject of triaging as a classification problem and concluded that manual triaging was the most effective method. Based on the 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. It displays the impact that the break-down method, which is backed by matrix factorization and the Differential Evolution algorithm, has on improving the fixing time and demonstrates the benefits that it offers in terms of standardizing the developer.

4. Proposed Model

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 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 is 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.

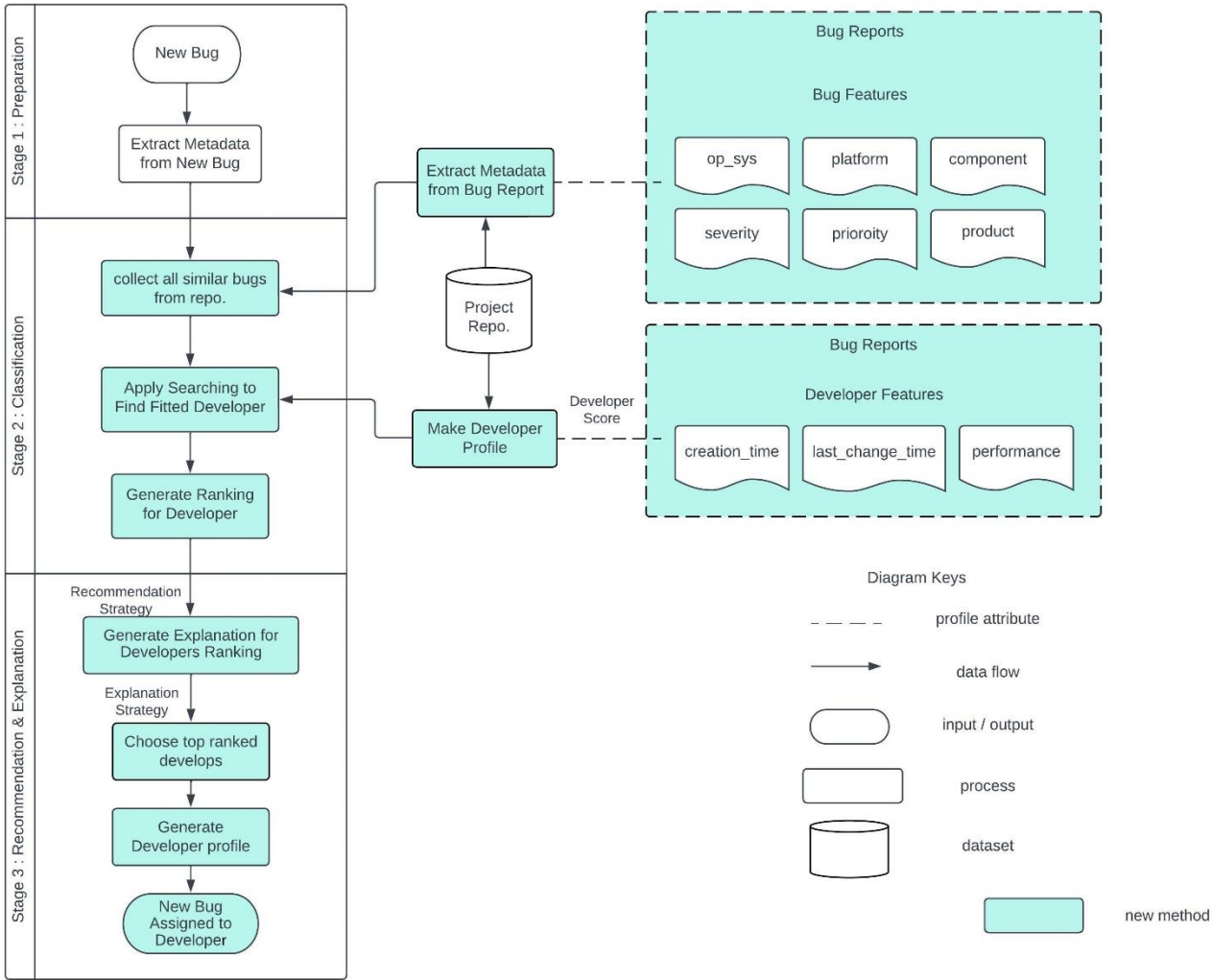


Figure 2. ABDR recommender model.

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. Presented in the following subsections is an expanded explanation of what happens 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 done by performing text reduction and cleaning methods, such as removing stop words, converting text to lowercase, and removing punctuation. This helps to reduce the dimensionality of the data and improve 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 done 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 done by analyzing the change logs and version control systems to find the developers who were responsible for fixing the bugs.

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 from and make accurate developer assignment decisions in the subsequent stages.

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

Additionally, we will reduce the number of variables for each bug to 6, 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):

$$DP_i = \sum(Tc - Tp_i) / N_i \quad (1)$$

where DP_i 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 developers will complete 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.

Table 1. Bugs history for 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		

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 get 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.

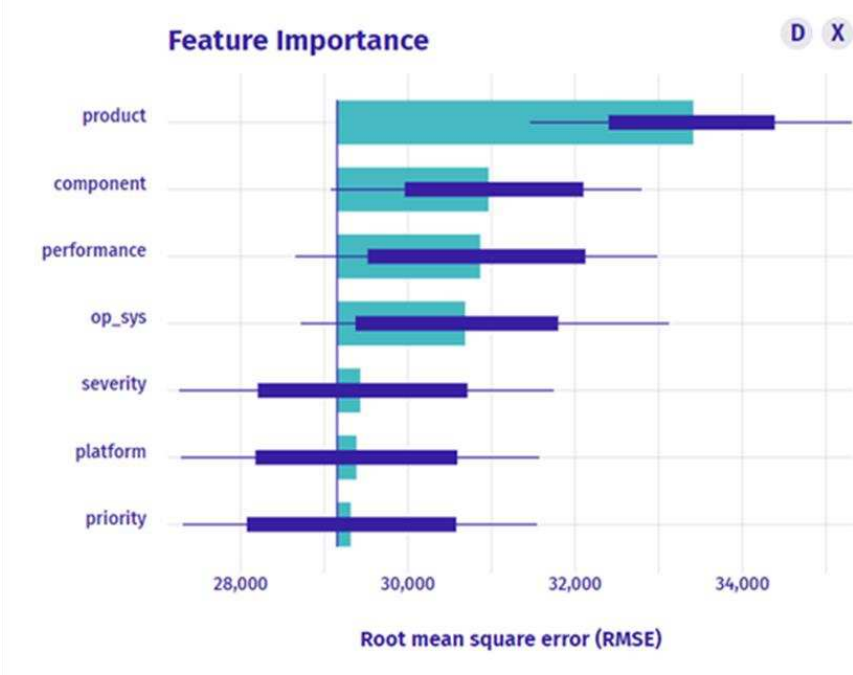


Figure 3. Feature Importance for whole eclipse dataset.

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 developer's proficiency is 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 developer's proficiency is 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 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 to 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 [13].

The only condition for bugs is that they be closed or verified to improve the quality of similarity.

$$\text{Cosine Similarity} = \cos \theta = \frac{D \cdot B}{||D|| ||B||} = \frac{\sum_{i=1}^n D_i B_i}{\sqrt{\sum_{i=1}^n D_i^2} \sqrt{\sum_{i=1}^n B_i^2}} \quad (2)$$

In specific, the similarity between the vectors of developer D_i and a bug B is calculated from the (Equation (2)[11]), which is referred to as the cosine similarity since it measures the distances between the vectors as shown in Figure 4.

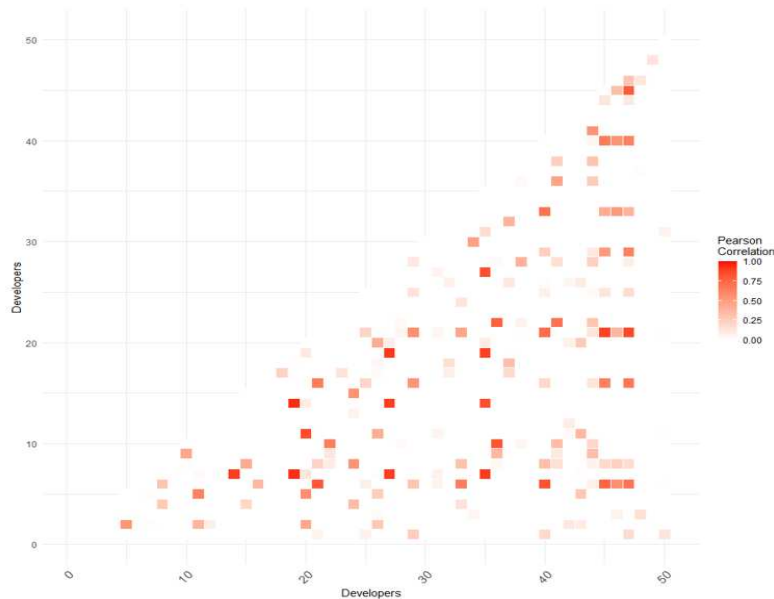


Figure 4. Developer similarity those developers who have already worked on similar bug.

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 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 will employ several strategies. Firstly, we will leverage available metadata associated with new bugs or developers, such as bug severity, priority, expertise, and skills, to make initial recommendations. This will allow us to provide relevant suggestions even in the absence of historical data. Additionally, we will 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 will utilize content-based recommendations by analyzing bug report attributes and developer profiles. This approach will enable us to match bugs and developers based on textual information, keywords, or tags. To enhance our recommendation accuracy, we will 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 it requires a reliable and efficient system 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 section 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 three popular machine learning models, namely Decision Tree, Random Forest, 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 the 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, 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 7 features, only 3 features - performance, op_sys, and component - have the highest importance for prediction.

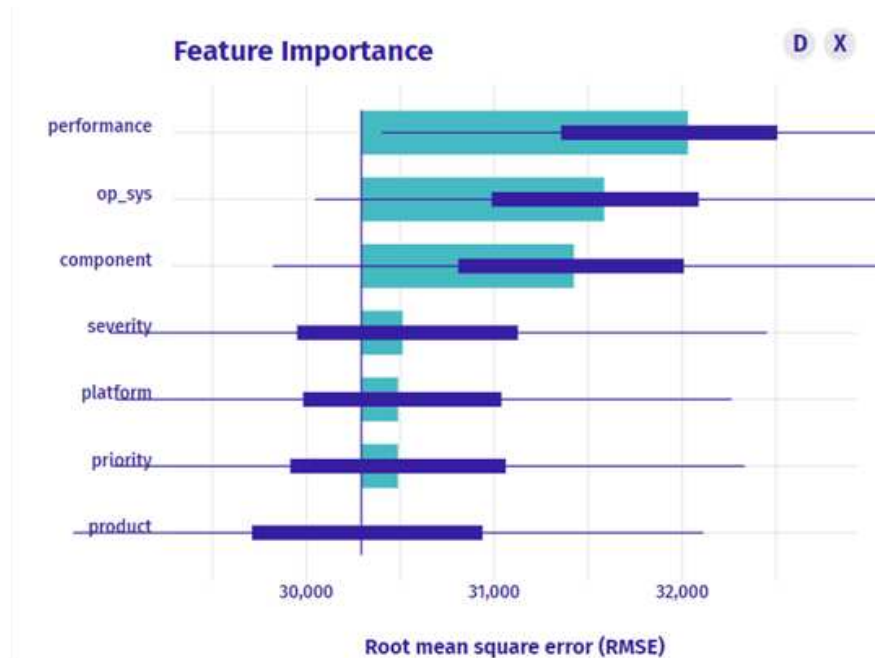


Figure 5. Feature importance for product = "platform".

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

- In the preparation stage, relevant data is 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 was 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 & explanation stage, the bug assignment process moves on to the recommendation and explanation stage. During this 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 183587 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.

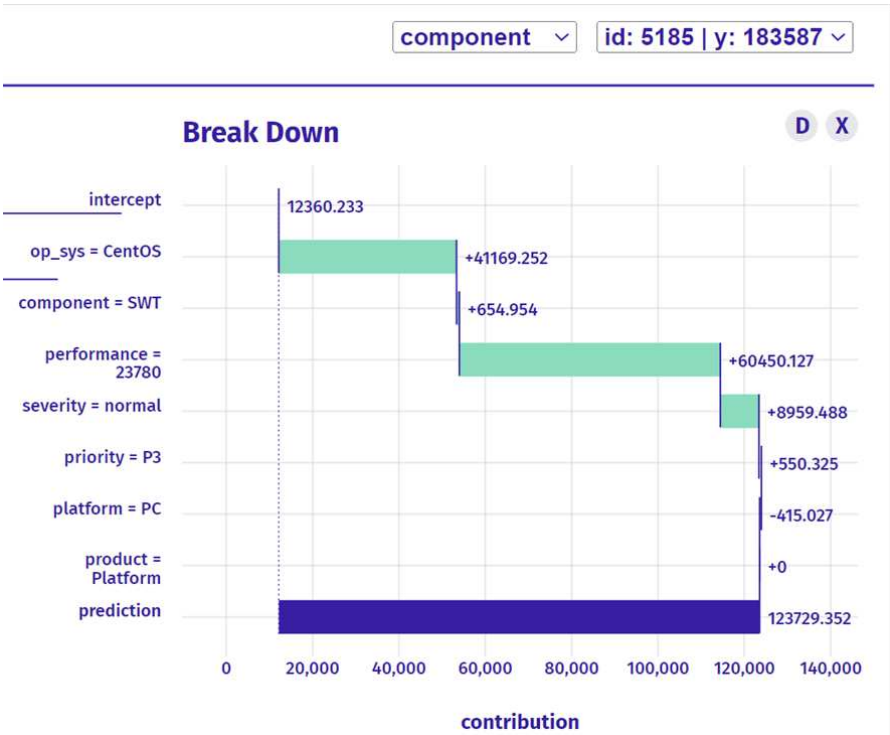


Figure 6. Explanation for new bug recommendation.

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 Recommend Bugs for Each Developer Scenario

In this scenario, want to determine three prediction bugs to each developer:
In the first prediction as shown in Figure 7:

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's 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 will 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. Explanation Stage:
- In this stage, we will 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.

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.

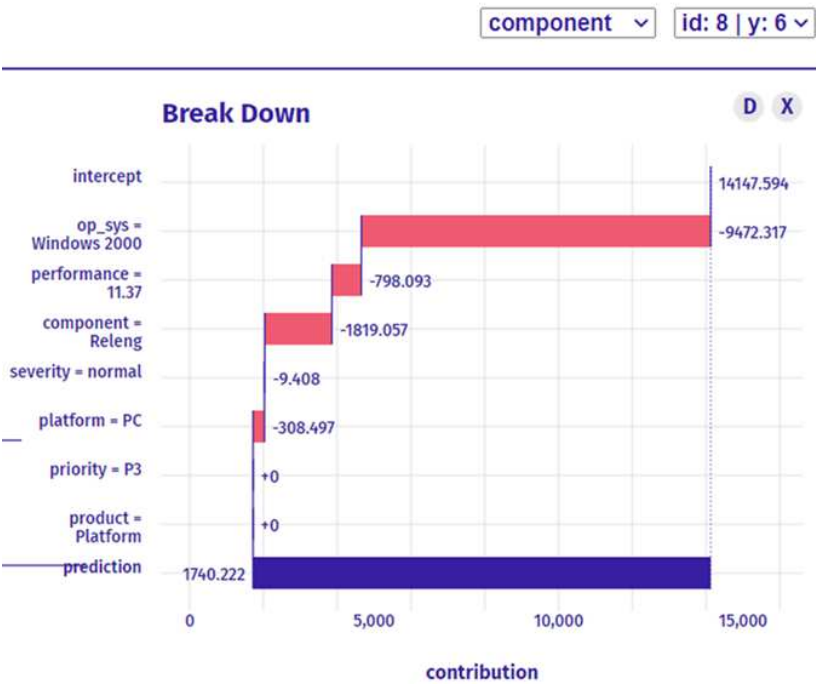


Figure 7. Low prediction for developer 6.

In the second prediction as shown in Figure 8:

let me address the medium forecast for a developer to fix the issue. In the context of a medium forecast, 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.

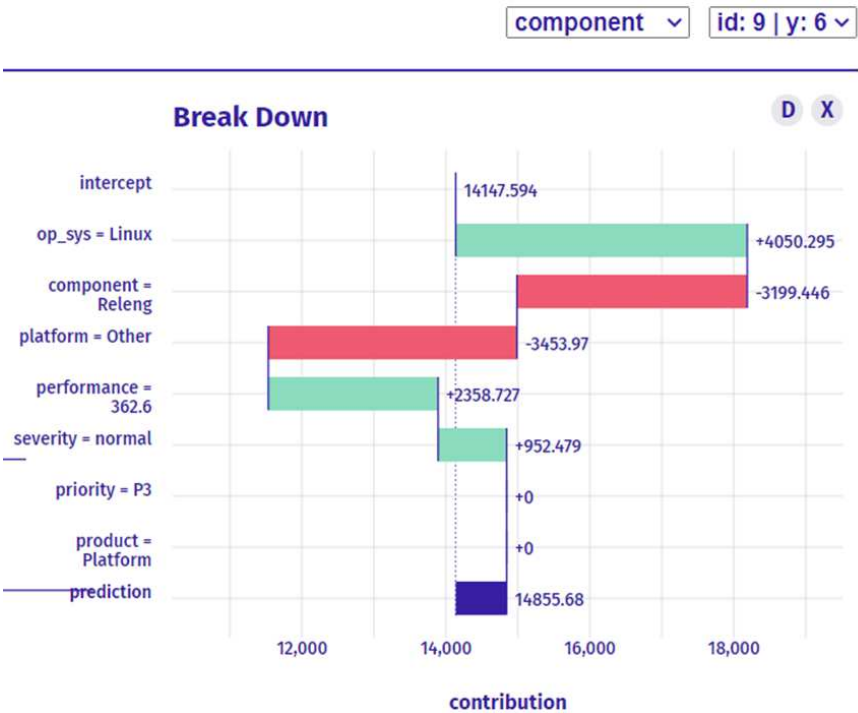


Figure 8. Medium prediction for developer 6.

In last prediction as shown in Figure 9:

To determine the developer who is most qualified to fix the bug, we need to consider several factors such as their level 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.

In high prediction, this is the recommended developer. that the prediction for the selected instance is much higher than the average model prediction. The most important three variables are performance, component, and platform increase the prediction with high value. Although, other variables with less important are increase the prediction. The contribution of all other variables is extremely high with recommend this developer for this type of task as shown in figure 9.

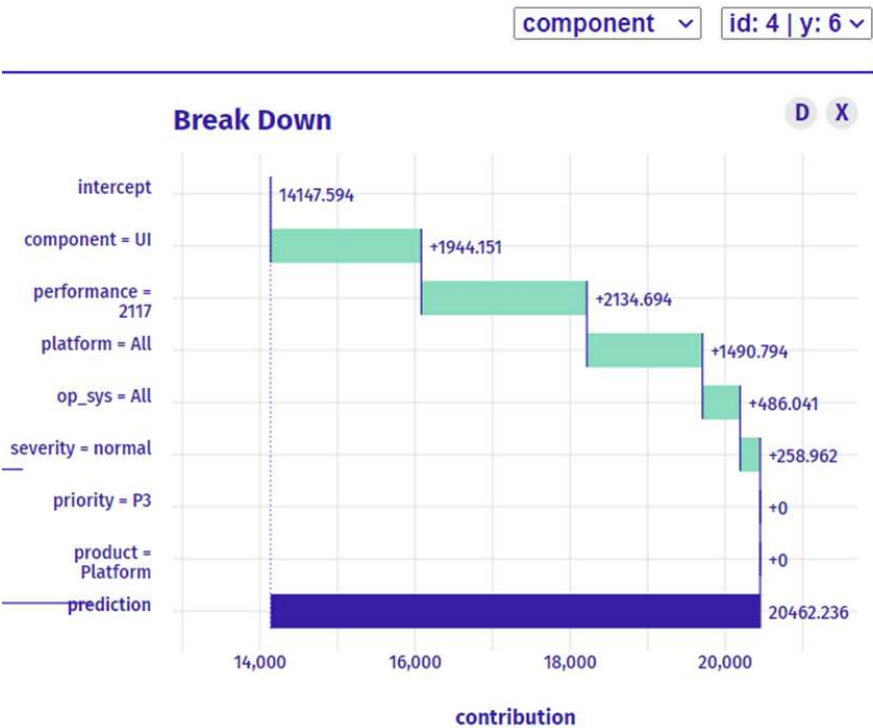


Figure 9. High prediction for developer 6.

5.3. ML Techniques for Bug Assignment

The field of software engineering has increasingly relied 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. In this regard, this study aimed to evaluate the performance of three popular machine learning models, namely Decision Tree, Random Forest, and XGBoost, for software bug prediction. The following paragraphs present the results of this evaluation and provide a detailed analysis of the performance of each model.

let's say we have a dataset of bug reports with multiple variables, such as product, component, operating system, severity, platform, and priority. The decision tree algo-rithm 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.

Overall, these algorithms are better suited for bug assignment than simpler meth-ods like linear regression or logistic regression because they can handle more complex relationships between variables and produce more accurate predictions.

Table 2. Comparison of machine learning model.

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

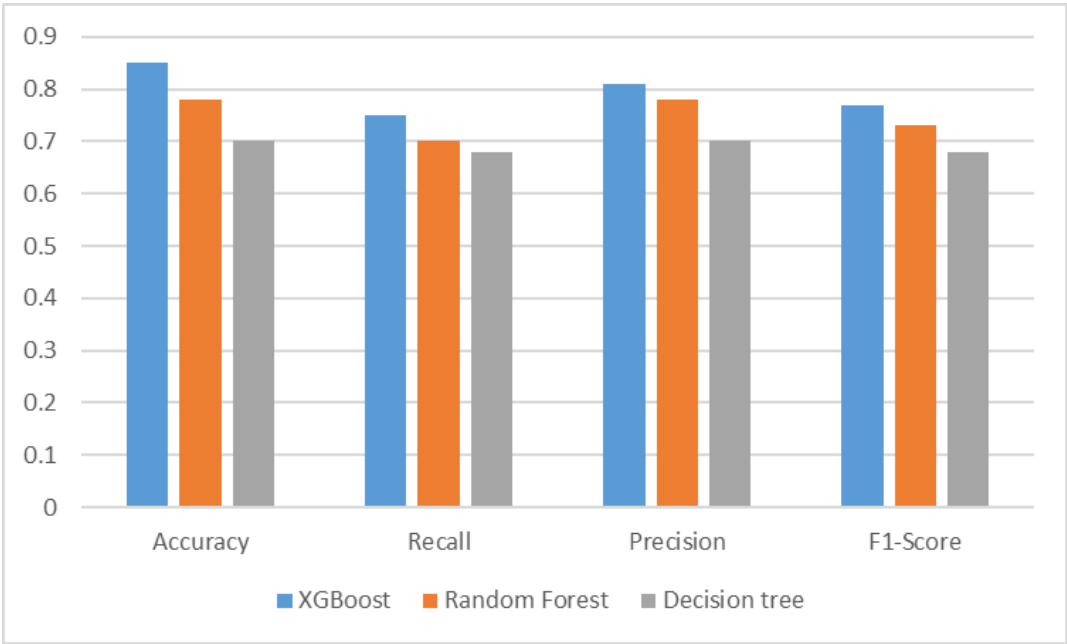


Figure 10. Comparison of machine learning model.

According to the results shown in Figure 10, we found that decision tree achieved an accuracy of 70%, precision of 68%, recall of 70%, and F1-Score of 68% in software bug prediction. Table 2 presents the detailed results obtained by applying decision tree.

Furthermore, our experiments revealed that random forest outperformed decision tree in software bug prediction, with an accuracy of 78%, precision of 70%, recall of 78%, and F1-Score of 73%. Table 2 displays the results obtained by applying random forest.

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

Models for Explainable Bug prediction may be classified into two scenarios: The first is to assign the new bug to the developer based on an explainable recommendation, and the second one is 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 certainly! 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 assign 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 whether machine learning techniques How can machine learning techniques be used to assign bugs to developers based on their expertise. machine

learning techniques can be used to assign bugs to the best 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 have shown 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 study the reliability and predictability of explanation generating strategies, mainly BreakDown, 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 enhances 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 BreakDown can generate good explanations for assigning new bug to best developer can fix it under different bug prediction scenarios. We are currently evaluating explanations in a practical context. It directly tests the goal for which the system is designed in a real-world application; thus, performance regarding this goal provides strong evidence of explanation success. That is the extent to which explanations assist humans in attempting to complete tasks is an important criterion for this. 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.

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. <https://doi.org/10.3390/jse9020025>.
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*; ACM Press: Atlanta, Georgia, USA, 2007; p. 34.
4. Yadav, A.; Singh, S.K.; Suri, J.S. Ranking of Software Developers Based on Expertise Score for Bug Triaging. *Information and Software Technology* 2019, 112, 1–17, doi:10.1016/j.infsof.2019.03.014.
5. Banerjee, S.; Syed, Z.; Helmick, J.; Culp, M.; Ryan, K.; Cukic, B. Automated Triaging of Very Large Bug Repositories. *Information and Software Technology* 2017, 89, 1–13, doi:10.1016/j.infsof.2016.09.006.
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*; IEEE: Vasteras, Sweden, July 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, doi:10.1007/s11390-019-1953-5.
8. Mani, S.; Sankaran, A.; Aralikatte, R. DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging. In *Proceedings of the Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*; ACM: Kolkata India, January 3 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 Internetworking*; ACM: Beijing China, September 16 2018; pp. 1–10.
10. Makridakis, S. The Forthcoming Artificial Intelligence (AI) Revolution: Its Impact on Society and Firms. *Futures* 2017, 90, 46–60, doi:10.1016/j.futures.2017.03.006.

11. Abdollahi, B.; Nasraoui, O. Using Explainability for Constrained Matrix Factorization. In Proceedings of the Proceedings of the Eleventh ACM Conference on Recommender Systems; ACM: Como Italy, August 27 2017; pp. 79–83.
12. Zhang, Y.; Chen, X. Explainable Recommendation: A Survey and New Perspectives. *FNT in Information Retrieval* 2020, 14, 1–101, doi:10.1561/15000000066.
13. Preece, A. Asking ‘Why’ in AI: Explainability of Intelligent Systems – Perspectives and Challenges. *Intell Sys Acc Fin Mgmt* 2018, 25, 63–72, doi:10.1002/isaf.1422.
14. Vilone, G.; Longo, L. Explainable Artificial Intelligence: A Systematic Review, 2020.
15. Jang, J.; Yang, G. A Bug Triage Technique Using Developer-Based Feature Selection and CNN-LSTM Algorithm. *Applied Sciences* 2022, 12, 9358, doi:10.3390/app12189358.
16. 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, 2010; Vol. 6156, pp. 32–46 ISBN 978-3-642-13791-4.
17. 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); IEEE: Dhaka, Bangladesh, December 2016; pp. 592–597.
18. Nguyen, T.T.; Nguyen, A.T.; Nguyen, T.N. Topic-Based, Time-Aware Bug Assignment. *SIGSOFT Softw. Eng. Notes* 2014, 39, 1–4, doi:10.1145/2557833.2560585.
19. 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); IEEE: San Francisco, CA, USA, May 2013; pp. 2–11.
20. 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); IEEE: Noida, India, August 2016; pp. 1–6.
21. 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. *International Journal of Scientific Research in Computer Science, Engineering, and Information Technology*, 2018, vol. 3, no. 5, pp. 2456–3307.
22. 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.
23. 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.
24. Chen, Y.; Li, H.; Li, X. Prediction of real estate price based on GBM method. *Symmetry* 2019, 11, 1233.
25. Lee, S.; Kim, J. A decision tree-based classification method for predicting student performance. *Appl. Sci.* 2019, 9, 2217.
26. Xiao, Y.; Wang, F.; Li, X.; Feng, X. Software defect prediction using Random Forest with entropy-based undersampling. *Symmetry* 2021, 13, 1696.
27. 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. doi.org/10.3390/sym13071183.
28. Pan, B. Application of XGBoost Algorithm in Hourly PM2.5 Concentration Prediction. *IOP Conf. Ser.: Earth Environ. Sci.* 2018, 113, 012127, doi:10.1088/1755-1315/113/1/012127.
29. 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, doi:10.1145/3236009.
30. Ribeiro, M.T.; Singh, S.; Guestrin, C. “Why Should I Trust You?”: Explaining the Predictions of Any Classifier. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, 22, pp. 1135–1144.
31. 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, doi:10.1109/JPROC.2021.3060483.
32. Biecek, P. DALEX: Explainers for Complex Predictive Models in R. *J. Mach. Learn. Res.* 19(84), 2018, 1–5
33. 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, 2014; Vol. 8693, pp. 740–755 ISBN 978-3-319-10601-4.

34. Gosiewska, A.; Biecek, P. IBREAKDOWN: UNCERTAINTY OF MODEL EXPLANATIONS FOR NON-ADDITIVE PREDICTIVE MODELS. arXiv preprint arXiv:1903.11420, 2019.
35. Jahanshahi, H.; Jothimani, D.; Başar, A.; Cevik, M. Does Chronology Matter in JIT Defect Prediction? A Partial Replication Study. In Proceedings of the Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering; September 18 2019; pp. 90–99.
36. 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, 2014; Vol. 8693, pp. 740–755 ISBN 978-3-319-10601-4.
37. Mane, S.; Rao, D. Explaining Network Intrusion Detection System Using Explainable AI Framework. arXiv 2021, arXiv:2103.07110.
38. Zhang, Y.; Chen, X. Explainable Recommendation: A Survey and New Perspectives. FNT in Information Retrieval 2020, 14, 1–101, doi:10.1561/15000000066.
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, doi:10.1587/transinf.2019EDP7152.
40. Lan Guo; Yan Ma; Cukic, B.; Singh, H. Robust Prediction of Fault-Proneess by Random Forests. In Proceedings of the 15th International Symposium on Software Reliability Engineering; IEEE: Saint-Malo, Bretagne, France, 2004; pp. 417–428.
41. Hooker, S.; Erhan, D.; Kindermans, P.-J.; Kim, B. A Benchmark for Interpretability Methods in Deep Neural Networks. ACM Comput. Surv. 2018, 51, 1–42.
42. Gondaliya, K.D.; Peters, J.; Rueckert, E. Learning to Categorize Bug Reports with LSTM Networks. 2018, 6.
43. Abdollahi, B.; Nasraoui, O. Using Explainability for Constrained Matrix Factorization. In Proceedings of the Proceedings of the Eleventh ACM Conference on Recommender Systems; ACM: Como Italy, August 27 2017; pp. 79–83.
44. Lipton, Z.C. In Machine Learning, the Concept of Interpretability Is Both Important and Slippery. machine learning 28. 2022
45. 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 preprint arXiv:2007.14655, 2020.
46. Ovadia, Yair; Fertig, Emanuel; Ren, Jie; Grosse, Roger; Muandet, Krikamol. "Can You Trust Your Model's Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift." arXiv preprint arXiv:1906.02530 (2019).
47. Eclipse Available online: <https://bugs.eclipse.org/bugs/> [Accessed on 15 August 2022].

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.