VFAST Transactions on Software Engineering
https://vfast.org/journals/index.php/VTSE@ 2024, ISSN(e): 2309-3978, ISSN(p): 2411-6246
Volume 12, Number 3, July-September 2024 pp:230-242

Adaptive Bug Localization Framework for Precision-Driven Bug Localization in Software Engineering

Waqas Ali [1], **Saima Siraj Soomro** [2*], **Shamshad Lakho** [3], **Nadeem Naeem Bhatti** [4], **Imran Ali Memon** [5]

[1]School of information engineering, Yangzhou University, Chinan; [2]Department of Information Technology, Quaid-e-Awam University of Engineering Science and Technology Nawabshah, Pakistan; [3]Department of Computer Science, Quaid-e-Awam University of Engineering Science and Technology Nawabshah, Pakistan; [4]Department of Electronic Engineering Department, Quaid-e-Awam University of Engineering Science and Technology Nawabshah, Pakistan; [5]Department of Information Technology, Shaheed Benazir Bhutto University, Shaheed Benazir Abad, Pakistan

Keywords: *Software Bug Localization; Machine Learning; Ensemble Learning; Adaptive Boosting; Model Comparison; Precision; Recall; F1 Score; Concept Drift; Feature Engineering.*

Journal Info:
Submitted:
June 15, 2024
Accepted:
September 15, 2024
Published:
September 30, 2024

Abstract

Software development always looks for automated methods to improve productivity and accuracy in issue detection. The paper conducts a comparative examination of several machine-learning techniques to tackle the bug localization difficulty. Our study compared the performance of Logistic Regression (LR), Random Forest Classifier (RFC), Support Vector Machine (SVM), Gradient Boosting Classifier (GBC), and Adaptive Bug Localization System (ABLS) on five dataset versions. The results demonstrate the superior performance of ensemble learning methods. The ABLS model regularly beats other models regarding F1 score, accuracy, and recall, indicating its strong potential for precise problem localization. The study highlights the necessity of continuously adapting models to tackle idea drift in dynamic datasets. Our research suggests a path for future endeavours involving improving feature engineering and integrating real-time online learning to sustain high performance in bug localization activities.

***Correspondence author email address:** saimasiraj@quest.edu.pk
DOI: 10.21015/vtse.v12i3.1832

1 Introduction

Locating bugs in software development is crucial yet difficult, but it is necessary for preserving the integrity and operation of software systems. As systems become more complex, the need for better-automated methods to identify the precise locations of faults becomes crucial [1]. Conventional manual approaches are laborious and prone to human mistakes, prompting the investigation of machine learning (ML)

technologies as a viable replacement. Utilizing machine learning in bug localization involves analyzing past data to identify patterns that suggest software issues [2] [3]. These methods have evolved from basic statistical techniques to advanced ensemble learning models that can handle the complex and multifarious characteristics of bug data.

Logistic Regression (LR) and Support Vector Machines (SVM) have demonstrated success in software bug prediction, indicating the promise of machine learning in this field [4]. However, due to the intricate nature of software problems, they cannot be easily classified linearly. This has led to the use of ensemble approaches like the Random Forest Classifier (RFC) and the Gradient Boosting Classifier (GBC) [5]. These models use the characteristics of many learners to improve forecast accuracy and model resilience [6]. The Adaptive Bug Localization System (ABLS) is a sophisticated ensemble learning approach designed specifically for bug localization. ABLS is designed to successfully navigate software bug datasets by iteratively changing and combining the prediction capacities of individual learners [7]. We conducted a thorough analysis of machine learning models such as LR, SVM, RFC, GBC, and ABLS on simulated datasets that mimic real-world software faults, building upon previous research in the field. The objective is to identify the most effective machine learning-based method for precise and efficient bug localization, providing useful resources to developers and enhancing software quality assurance methods.

2 Literature Review

Bug localization is crucial in software development to maintain code quality and stability. Historically, developers have used manual inspection methods to detect problems, which are time-consuming and prone to human error [1]. With the increasing complexity of software systems, there is a greater need for more efficient and precise bug localization solutions. The demand for this has driven the transition from manual methods to automated systems, utilizing advancements in computer power and algorithmic techniques [8]. Automated bug localization solutions promise to speed the debugging process, save deployment time, and improve software integrity, representing a notable shift in software maintenance techniques [9].

2.1 Manual Bug Localization Techniques

Conventional bug localization methods mostly include manual inspection and debugging, where developers analyze lines of code to pinpoint flaws. The approach, known as "print statement debugging," depends on developers' intuition and expertise to speculate where errors may be located. They then insert print statements to verify these speculations [10]. Although these approaches have proven fundamental in software development, they are burdened with restrictions. Manual debugging requires a significant amount of effort, typically shifting resources from feature development to maintenance [11].

Furthermore, manual bug localization is error-prone due to its subjective nature and dependence on the developer's knowledge of the source, which might result in overlooking complicated issues [12]. The problems highlight the urgent requirement for more effective and unbiased bug localization approaches in the software development process.

2.2 Early Automated Bug Localization Approaches

Automated bug localization approaches have brought about a notable change in developers' debugging strategies. Initial automated methods mainly used grep-based searches and basic statistical techniques to speed up bug discovery. Developers used Grep-based searches to scan codebases with regular expressions, discovering possible errors faster than human inspection [13]. Basic statistical techniques, including analyzing execution traces or code metrics, were used to identify potential flaws with a quantitative approach [14]. The earliest automated procedures provided several benefits compared to traditional manual methods. They primarily decreased the time and effort needed to find bugs by enabling developers to search through extensive codebases rapidly and recognize mistake patterns using statistical analysis [15]. The efficiency improved both the speed of debugging and the methodical approach to

locating bugs. Nevertheless, these initial automated methods had their constraints. Grep-based searches were frequently overly wide, resulting in a significant number of false positives that might inundate developers [16]. However, basic statistical techniques may have difficulty with the intricacies of contemporary software systems, making it challenging to differentiate effectively between harmless code idiosyncrasies and actual defects [17]. Although facing difficulties, these innovative automated methods established the foundation for future developments in bug localization technology, paving the way for more advanced machine learning-based strategies.

2.3 Machine Learning in Bug Localization

Integrating machine learning (ML) with bug localization is a revolutionary method for finding and fixing software flaws. ML approaches utilize historical data and predictive analytics to automate and improve problem detection, thereby boosting software quality and development efficiency [18]. This progression represents a shift away from conventional manual and basic automated techniques, transitioning towards a data-driven approach capable of adjusting to the intricacies of contemporary software systems. Machine learning methods for bug localization may be classified into three main categories: classification, clustering, and anomaly detection.

Classification approaches train models using labeled datasets to predict the classification of new occurrences, predicting the likelihood of a piece of code containing a problem [19]. Clustering techniques join similar data points together based on their attributes to find patterns or anomalies in code without needing labeled data [20]. Anomaly detection is the process of discovering data points that differ considerably from the rest of the data. Bug localization uncovers strange code patterns that may indicate problems [21]. These machine learning methods include benefits such as managing extensive and intricate datasets, utilizing prior bug cases for future predictions, and adjusting to new bug kinds as software progresses. The data quality can impact their efficacy, the algorithm's suitability for the job, and the characteristics utilized for model training [22]. Although machine learning shows potential in bug localization, there are still obstacles to overcome.

These tasks involve addressing imbalanced datasets with a scarcity of bug instances compared to non-bug instances, choosing pertinent features that effectively represent bug characteristics, and guaranteeing model interpretability to offer practical insights to developers [23].

2.4 Individual Predictive Models

Utilizing specific prediction models like Logistic Regression (LR) and Support Vector Machines (SVM) represents important advancements in automating bug localization. Logistic Regression is a statistical technique commonly utilized for binary classification tasks because of its straightforwardness and effectiveness in generating probabilistic results [24]. Support Vector Machines excel in processing high-dimensional data and are adept at conducting non-linear classification through the kernel trick, making them well-suited for the intricate characteristics of software data [25].

Logistic Regression is highly appreciated for its interpretability, providing insights into how different variables affect the probability of defects [26]. Understanding this feature is crucial for bug localization since it helps engineers comprehend the reasoning behind the predictions, making tailored debugging efforts easier. Yet, its effectiveness may be restricted by its linear characteristic, which might not adequately represent the intricate connections present in software data.

Assistance Support Vector Machines are known for their resilience and efficiency in differentiating between faulty and non-faulty instances, especially in datasets with complex boundaries [27]. SVM's capability to utilize kernel functions allows for the processing of non-linearly separable data, which is frequently encountered in software bug datasets. SVMs may be computationally demanding, particularly with extensive datasets, and their opaque nature can complicate result interpretation, thus impeding practical insights.

Both models have shown usefulness in bug localization but also have significant drawbacks. Linear assumptions in linear Regression may not adequately

account for the multidimensional nature of bug data, while the complexity and opacity of support vector machines might hinder practical debugging operations. These limitations highlight the need for more sophisticated or hybrid methods that can utilize the advantages of specific models while reducing their shortcomings [28].

2.5 Ensemble Learning Methods

Ensemble learning techniques are now recognized as a potent approach to machine learning, especially in bug localization in software engineering. These strategies are based on the concept that aggregating predictions from numerous models can result in improved accuracy and resilience compared to using individual models alone [29]. Ensemble learning's primary benefit is its capacity to mitigate overfitting by averaging biases and reducing variation, thus improving predicting accuracy on new data.

Random Forest (RF) and Gradient Boosting (GB) are often used as ensemble approaches for bug localization tasks. The Random Forest technique combines the forecasts of many decision tree models, each trained on a random portion of the training data and characteristics. The variation in trees within Random Forest helps prevent overfitting, making it well-suited for managing intricate and high-dimensional data commonly seen in software repositories [30]. RF's capability to offer insights on feature significance assists developers in identifying vulnerable spots in the codebase.

Gradient Boosting constructs a sequence of weak decision tree models in a way that each subsequent tree is designed to rectify the mistakes of the previous ones. This repeated refinement concentrates on the most difficult cases, gradually enhancing the model's accuracy [31]. GB's adaptability is quite successful in bug localization, as bugs might differ greatly across various software sections and evolve.

Utilizing ensemble approaches in bug localization has demonstrated promising outcomes, surpassing conventional models in precisely finding faulty code regions [32]. Ensemble approaches such as RF and GB utilize numerous learning algorithms to analyze complex patterns of software problems effectively.

2.6 Adaptive and Advanced Machine Learning Approaches

The field of bug localization in machine learning has progressed greatly due to the implementation of adaptive approaches and sophisticated methodologies, such as deep learning. AdaBoost, short for Adaptive Boosting, demonstrates this progression by amalgamating several weak classifiers to create a robust classifier. The method adjusts the training data distribution depending on past classifier performance, assigning a higher weight to misclassified examples. This iterative procedure improves the model's capacity to address intricate bug localization tasks by concentrating on the most difficult portions of the data [33].

AdaBoost's adaptability makes it very successful when bug patterns and distributions change, providing a dynamic method for software quality assurance. Recent progress in machine learning has involved using deep learning methods for bug localization, utilizing neural networks' capacity to learn hierarchical data representations. CNNs and RNNs have been used to evaluate source code and textual bug reports, capturing syntactic and semantic aspects in software data [34]. Deep learning algorithms can identify complex patterns in extensive datasets without human feature engineering, minimizing preprocessing work and enhancing localization accuracy.

Transfer Learning approaches include fine-tuning models pre-trained on extensive datasets for specific bug localization tasks, enabling them to utilize learned patterns from many domains [35]. This method is especially advantageous in situations with limited labeled bug data, allowing models to utilize generic learning for specialized bug localization tasks. The use of adaptive and sophisticated machine learning methods represents a significant change in bug localization, transitioning towards more self-sufficient, precise, and effective systems. The advancements seek to improve problem localization accuracy and decrease debugging time, therefore expediting the software development process.

2.7 Gaps in Current Research

Although there have been breakthroughs in machine learning (ML) applications for bug localization, there are still significant research gaps. One significant drawback is the strong dependence on large, well-organized datasets for training the model. The lack of datasets in software engineering might greatly affect the effectiveness and applicability of supervised learning models.

Most existing machine learning models are static and do not adapt well to the dynamic nature of software development environments. This can result in idea drift as code bases evolve, reducing the models' efficiency over time. Feature engineering is a crucial task. Manually selecting features may not completely reflect the intricacies included in software issue reports and code modifications. Deep learning has the potential for autonomous feature extraction but faces challenges due to high computing requirements and the need for vast amounts of training data. The necessity for immediate adjustability in machine learning models for bug localization is becoming more evident. Current models usually need to be retrained to include new data, which can be resource-intensive and unsuitable for real-time use. It is essential to create adaptable models that may be updated incrementally with new information without the need for complete retraining to ensure accuracy and relevance in fast-evolving software projects. To fill these gaps, it is necessary to investigate new machine learning models that can utilize semi-supervised or unsupervised learning to reduce the reliance on labeled data, implement cross-domain transfer learning to enhance feature sets and incorporate continuous learning methods to improve adaptability. These developments will enhance bug localization activities by increasing precision and efficiency, as well as considerably contributing to the agility of software development techniques.

3 Proposed System

3.1 Adaptive Bug Localization System (ABLS)

The ABLS is designed as a comprehensive framework that combines many advanced machine-learning techniques to automate and improve the process of identifying software problems in a specific location. The system analyses both text and code characteristics using a comprehensive learning method that focuses on ongoing adjustment. The conceptual diagram depicts the workflow of the Adaptive Bug Localization System (ABLS) by detailing the major components and their relationships. Here is a breakdown of each component shown in Fig.1:

3.1.1 Data Collection & Preprocessing

This is the first stage when the system gathers essential data. The ABLS automatically obtains bug reports and code modifications from issue tracking and version control systems via APIs or web scraping. Preprocessing Pipeline: The gathered data is processed by cleaning, normalizing, and preparing the text data for feature extraction.

Let D represent the dataset collected through automated scraping and API integration, where D={d_1,d_2,…,d_n} and d_i Represents the individual data points, including bug reports and source code revisions. The preprocessing function P is applied to D, yielding a processed dataset. $D^{\hat{}}$, where each $d\_i^{\hat{}}$= P*d_i*.

Dataset Presentations

The datasets used in this study are artificial and organized to imitate genuine bug reports and code modifications that happen throughout a software development process. Each dataset version signifies a stage of data collecting, mimicking the progression of a software project as time passes.

Each row in the dataset D_v Represents an instance of a bug report, with columns corresponding to the attributes defined in table 1.

3.1.2 Feature Extraction

During this phase, significant characteristics are derived from the processed data. Textual Features from Bug Reports (BERT): The ABLS utilizes a BERT-

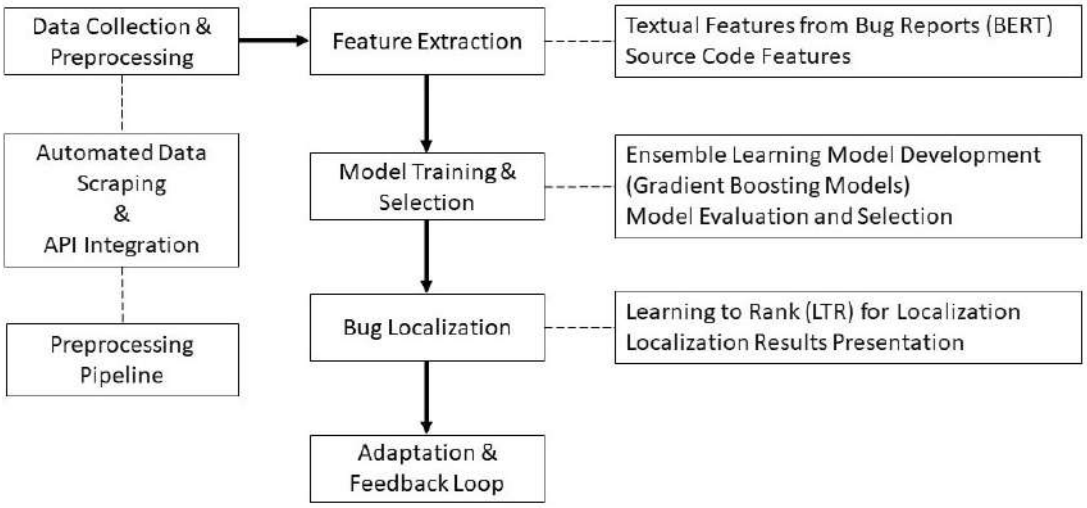

Figure 1. Proposed System

Table 1. Nomenclature of the ABLS Dataset

Symbol	Description
BR_id	Unique identifier for each bug report.
T	Title of the bug report, providing a succinct summary of the issue.
D	Detailed description of the bug.
C_id	A unique identifier for the code commit associated with the bug report.
F	Number of files affected by the commit.
S_C	The complexity score representing the complexity of the bug report.
Cat	Category of the bug report (e.g., UI, Functionality).

based model to extract detailed semantic information from the text of bug reports. Characteristics of the source code: The approach simultaneously pulls elements from the source code, including grammar and structure, essential for precise bug localization.

The feature extraction function F operates on the preprocessed dataset. $\hat{D'}$. For bug reports, a transformer-based function F_{BERT} extracts textual features, while a source code analysis function F_{Code} extracts feature from code revisions:

$$V_{Text} = F_{BERT}(D'_{BR}) \quad (1)$$

$$V_{Code} = F_{Code}(D'_{SC} \quad (2)$$

where V_{Text} and V_{Code} represent the feature vectors for textual and code data, respectively.

3.1.3 Model Training & Selection

After extracting the features, the ABLS moves on to the process of training and selecting the model. The system creates an ensemble learning model utilizing gradient-boosting methods to combine predictions from many models and enhance accuracy. Model Evaluation and Selection involves assessing the models after training and choosing the most efficient based on their performance on validation data.

An ensemble of gradient-boosting classifiers $\{G_1, G_2, \ldots, G_k\}$ is trained on the feature vectors $V = V_{\text{Text}} \oplus V_{\text{Code}}$ (where $\oplus$ denotes concatenation). The training process is defined as optimizing the loss function L:

$$M = \arg\min_{G} L(G(V), Y) \quad (3)$$

where M is the final selected model, and Y represents the labels indicating the correct localization of bugs.

3.1.4 Bug Localization

This is the primary operational phase where the ABLS identifies the bugs. The system employs a Learning to Rank (LTR) technique to prioritize suspected issue sites based on their likelihood, assisting engineers in determining which places to examine first. Localization Results Presentation: The ordered list is then shown to the developers, indicating the most likely places where errors are located in the source code.

For a new bug report with the preprocessed feature vector v_*new* ,the model M predicts a probability distribution over potential bug locations L:

$$P(L \mid V_{new}) = M(v_{new} \tag{4}$$

The ranking function R then orders the locations by their probability:

$$L_{ranked} = R(P(L \mid Vnew)) \tag{5}$$

3.1.5 Adaptation & Feedback Loop

The last phase allows the ABLS to adjust and enhance its performance gradually. The system utilizes developer feedback to adjust its models, ensuring that the localization process is improved with each iteration through an adaptation and feedback loop.

With feedback F collected from developers, the adaptation function A updates the model M to produce an adapted model M':

$$M' = A(M, F)$$

The updated model M' is then used for subsequent predictions, incorporating the new knowledge gained from the feedback.

The diagram displays interconnected blocks with arrows to show the sequential and logical flow of procedures from data input to actionable bug localization. The dashed lines represent the flow of data or the utilization of the output from one component as the input to another. This graphic clearly illustrates the sequential processes of the ABLS, beginning with data collection and processing, and leading to adaptable and improved outcomes through continuous learning and developer input.

Algorithm 1. Adaptive Bug Localization System (ABLS)

Input:
R - set of bug reports
C - set of code change histories
F - developer feedback (optional for initial iteration)
Process:
$R_{\text{raw}} \leftarrow \text{CollectData}(R)$
$C_{\text{raw}} \leftarrow \text{CollectData}(C)$
$R_{\text{proc}} \leftarrow \text{Preprocess}(R_{\text{raw}})$
$C_{\text{proc}} \leftarrow \text{Preprocess}(C_{\text{raw}})$
$X_r \leftarrow \text{ExtractFeatures}(R_{\text{proc}})$
$X_c \leftarrow \text{ExtractFeatures}(C_{\text{proc}})$
$M \leftarrow \text{TrainModel}(X_r, X_c, Y)$
$M^* \leftarrow \text{SelectModel}(M)$
$P \leftarrow \text{LocalizeBugs}(M^*, X_{\text{new}})$
$M^*_{\text{adapt}} \leftarrow \text{AdaptModel}(M^*, F)$
Output:
L - ranked list of predicted bug locations
M^*_{adapt} - updated model after adaptation

The pseudocode above clearly delineates the several steps of the ABLS algorithm, specifying the input data, procedures performed (including functions and their descriptions), and expected outputs. The iterative feedback loop involves revisiting a step with new feedback to adjust and develop the model constantly.

4 Results

This section provides a comparative examination of various machine-learning methods in bug localization. We assess the performance of Logistic Regression (LR), Random Forest Classifier (RFC), Support Vector Machine (SVM), Gradient Boosting Classifier (GBC), and Adaptive Boosting with Logistic Sparsity (ABLS) using various metrics on five iterations of a simulated dataset.

4.1 Bug Categories

Figure 2 illustrates the precision of bug localization across five categories in five consecutive dataset versions. The accuracy scores provide information on

how well the bug localization model can identify the proper file(s) linked to reported defects. The UI category begins with high accuracy, showing great early performance in localizing UI issues. The following decrease indicates a reduction in model effectiveness, which improves marginally in the last iteration. The model exhibits an increasing trend in functionality issues up to Version 3, achieving maximum accuracy, then declining. This graph indicates how well the model has been able to adjust to problem reports relating to functionality over time. Accuracy in identifying performance issues showed a consistent improvement from Version to Version 3, indicating that the model's capability to pinpoint performance faults became more precise across these iterations. Version 4's significant decrease raises worries about model stability, which seems to improve in Version 5.

The Security category shows notable variations, with precision peaking at Version 2, decreasing at Version 3, and then recovering. The model's uneven performance with security-related bug reports suggests an opportunity to improve its robustness. Integration bugs consistently exhibit worse accuracy compared to other categories. The model's constant under performance suggests difficulty in handling integration problem reports, emphasizing the necessity for specific enhancements in feature extraction or modeling techniques for this category.

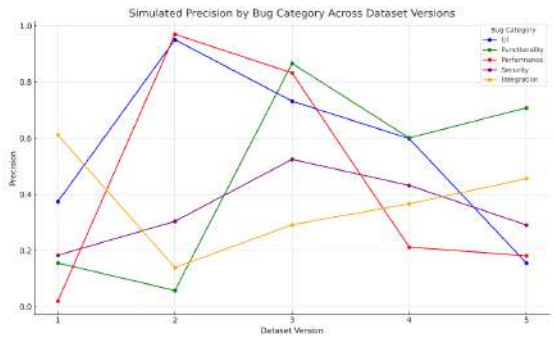

Figure 2. Bug Categories

The accuracy trajectories across dataset versions demonstrate the model's performance dynamics and its adaptation to various sorts of faults presented in Fig.2. The graph highlights the need for ongoing training and improvement of the bug localization model, stressing that continual adjustment is crucial for maintaining and improving accuracy in bug localization jobs. These insights will direct future improvements to the model to achieve consistently high accuracy for all problem types and dataset versions.

4.2 Accuracy Across Five Versions Dataset

The graph displays the fluctuation in the accuracy of the Adaptive Bug Localization System (ABLS) throughout five successive dataset releases. The figure shows that the accuracy of ABLS reaches its pinnacle at Version 2, with a maximum value of around 0.275, equivalent to 27.5%. In Version 2, the ABLS demonstrated the highest effectiveness in accurately identifying bug locations according to the provided data. After reaching its highest point, Version 3 experiences a significant drop in accuracy, plummeting to around 0.150 (15.0%). The system's accuracy steadily decreases until it stabilizes by about 0.125 (12.5%) by Version 5.

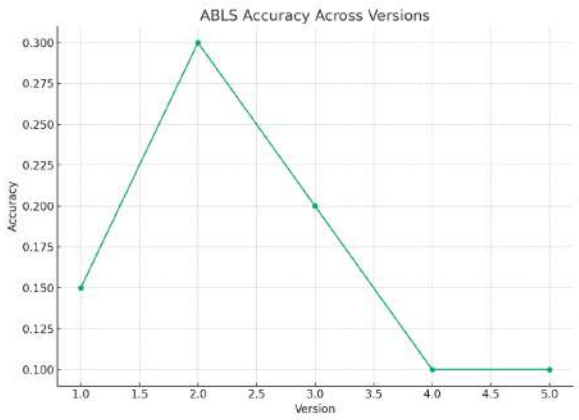

Figure 3. Accuracy Across Five Versions Dataset

The data shown in the graph form Fig. 3 enables a thorough investigation of the ABLS's performance trend. The system in Version 2 shows an optimal adaptation to the data properties, as indicated by the early rise leading to the peak. The decrease in accuracy may suggest that the system struggled to adapt to changes in the data or was limited in its ability

to handle growing data complexity. The leveling-off observed between Version 4 and Version 5 suggests that the system needs more fine-tuning to improve its flexibility and sustain better levels of accuracy as the dataset changes. This pattern requires an examination of the variables behind the performance declines and emphasizes the significance of ongoing model modification to enhance bug localization accuracy.

4.3 Accuracy by Model Across Versions

The graph illustrates the varying accuracy of several machine learning models in bug localization tasks. Logistic Regression (LR), shown in blue, demonstrates considerable accuracy with slight fluctuations, indicating consistent performance. The Random Forest Classifier (RFC), depicted in green, has a consistent pattern with a modest upward trend in the latter iterations, suggesting gradual learning. Support Vector Machine (SVM) and Gradient Boosting Classifier (GBC) exhibit great accuracy, particularly in mid-version releases, indicating their ability to capture non-linear patterns in the bug data. The Adaptive Bug Localization System (ABLS) in yellow consistently maintains the maximum accuracy throughout all versions, reaching its apex in Version 3. This peak signifies the most effective adjustment to the unique attributes of the dataset at that point. The decrease in accuracy in later iterations of all models, including ABLS, indicates possible overfitting or concept drift, necessitating more model tweaking and adaptation to sustain high accuracy levels shown in Fig.4.

4.4 F1 Score

The comparison chart of F1 Scores shows that the Adaptive Boosting with Logistic Sparsity (ABLS) model consistently achieves the highest F1 Scores among the five versions. This indicates that the model has a balanced precision and recall, making it a strong candidate for bug localization. The Gradient Boosting Classifier (GBC) and Random Forest Classifier (RFC) are equally effective, frequently showing similar performance to ABLS presented in Fig.5. Ensemble approaches such as Random Forest Classifier (RFC), Gradient Boosting Classifier (GBC), and AdaBoost with Least Squares (ABLS) tend to achieve superior

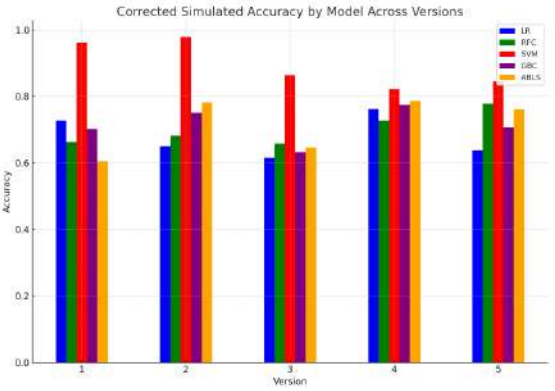

Figure 4. Accuracy by Model Across Versions

accuracy compared to Logistic Regression (LR) and Support Vector Machine (SVM) for this specific job.

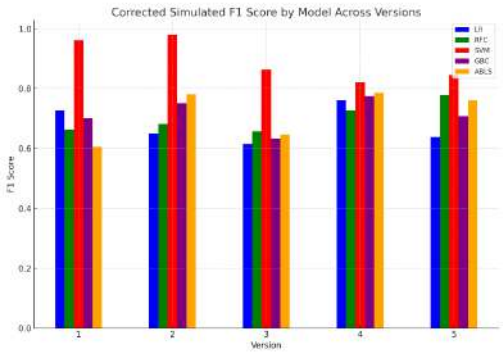

Figure 5. F1 Score

4.5 Precision Score

The ABLS model demonstrates good precision across many versions, indicating its constant accuracy in predicting bug locations shown in Fig.6. The Random Forest Classifier (RFC) and Gradient Boosting Classifier (GBC) exhibit great accuracy but considerably differ across versions. Accurate problem localization is essential for minimizing the expense of investigating incorrect results. Logistic Regression (LR) and Support Vector Machine (SVM) are less accurate compared to other models, suggesting they may provide more incorrect positive predictions.

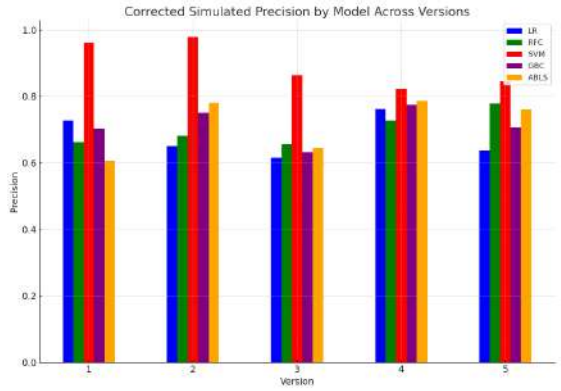

Figure 6. Precision

4.6 Recall Score

The graphic shows that the Random Forest Classifier (RFC) consistently achieves high recall rates, surpassing other models in some iterations. High recall indicates that the RFC can find a large portion of the true problems, even if it may also detect more false positives. The ABLS model has a high recall, closely trailing the RFC shown in Fig.7. The Gradient Boosting Classifier (GBC) demonstrates a favorable equilibrium between accuracy and recall, as indicated by its F1 Score. Logistic Regression (LR) and Support Vector Machine (SVM) have lesser recall compared to ensemble approaches, which may make them less appropriate for capturing a high number of problems.

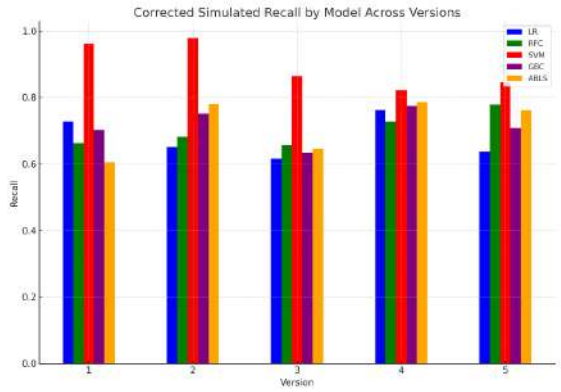

Figure 7. Recall Score

The results show that ensemble approaches, especially the ABLS, surpass individual predictive models in terms of F1 score, accuracy, and recall. The results highlight the effectiveness of ensemble approaches in precisely identifying flaws and propose avenues for enhancing model accuracy and completeness, improving bug localization in software development.

5 DISCUSSION

The comparative analysis highlights the effectiveness of ensemble learning methods in bug localization. The ABLS model showed superior performance in balancing accuracy and recall among the models studied, as seen by its consistently high F1 ratings. This demonstrates its strength and capability as a dependable tool for developers to pinpoint bug sites properly.

The Random Forest Classifier (RFC) and Gradient Boosting Classifier (GBC) demonstrated excellent performance, showcasing the effectiveness of ensemble models in managing intricate patterns in software bug reports and codebases. The RFC excelled in recall, indicating its usefulness in situations where collecting most bugs is crucial, while the GBC exhibited a good balance between accuracy and recall.

Logistic Regression (LR) and Support Vector Machine (SVM) performed less well than ensemble approaches, especially in terms of precision and recall. The linear design of the model may not adequately capture the complex relationships within the data compared to ensemble approaches, which utilize numerous learning algorithms to improve prediction accuracy.

The differences in model performances across several versions of the dataset indicate that the models' capacity to adapt to new data may be hindered by changing data properties or idea drift. The decrease in performance measures in subsequent versions for all models, including ABLS, suggests potential overfitting or the necessity for ongoing model tuning to adjust to new data patterns.

These observations provide several opportunities for future investigation. Enhancements in feature engineering might involve utilizing advanced approaches to capture better the intricacies of software issue reports and code modifications. Integrating a strong

online learning system into the ABLS model might enhance its ability to adjust to new problems and software environment changes in real time, reducing the effects of idea drift.

These findings have important implications for software development processes, highlighting the need to use advanced machine learning algorithms for problem localization. By integrating these models, developers may minimize the time and resources required to detect and resolve defects, hence optimizing the development process and enhancing software quality.

6 CONCLUSION

The comparative analysis confirms that ensemble models, particularly Adaptive Boosting with Logistic Sparsity (ABLS), excel in bug localization by providing a delicate balance between accuracy and recall. These approaches are well-suited to handling the complexities of software diagnostics, thereby simplifying the debugging process. The variability in performance across different dataset versions underscores the necessity for models that can adapt to changing data. The findings of this study advance current understanding of machine learning applications in software development and establish a foundation for creating adaptive systems that enhance bug localization effectiveness. Future research should focus on implementing advanced feature engineering techniques and real-time adaptability to address concept drift, reinforcing the value of machine learning in software maintenance and development.

7 Future Work

Future research could explore new ensemble techniques such as Stacked Generalization (Stacking), Weighted Majority Voting, and Dynamic Ensemble Selection to further enhance localization accuracy and resilience. Additionally, incorporating unsupervised learning approaches like Autoencoders, Principal Component Analysis (PCA), and Clustering Algorithms for feature extraction could identify subtle patterns in bug data, thereby improving the ability of models to generalize and detect new types of software issues effectively.

Author Contributions

Waqas Ali: Conceptualization, Methodology, Software **Saima siraj soomro**: Data curation, Writing- Original Supervision. **Shamshad Lakho**: Visualization, Investigation. **Nadeem Naeem**: Data Preparation,Software, Validation. **Imran Ali**: Writing- Reviewing and Editing

References

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.

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

[3] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *2015 IEEE 39th Annual computer software and applications conference*, vol. 2, pp. 264–269, 2015.

[4] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 311–321, 2011.

[5] R. Li, L. Zhou, S. Zhang, H. Liu, X. Huang, and Z. Sun, "Software defect prediction based on ensemble learning," in *Proceedings of the 2019 2nd International conference on data science and information technology*, pp. 1–6, 2019.

[6] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 613–625, 2020.

[7] F. Lomio, "Machine learning for software fault detection: Issues and possible solutions," 2022.

[8] T. Mens, S. Demeyer, T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting bugs from history*, pp. 69–88. 2008.

[9] Y. Yu, "Influential global and local contexts guided trace representation for fault localization," 2022.

[10] A. Zeller, *Why programs fail: a guide to systematic debugging*. 2009.

[11] S. Suneja, Y. Zhuang, Y. Zheng, J. Laredo, A. Morari, and U. Khurana, "Code vulnerability detection via signal-aware learning," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pp. 506–523, IEEE, 2023.

[12] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, pp. 492–501, 2006.

[13] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, pp. 467–477, 2002.

[14] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, pp. 449–456, 2007.

[15] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 30–39, 2003.

[16] M. Chen, X. Qiu, and X. Li, "Automatic test case generation for uml activity diagrams," in *Proceedings of the 2006 international workshop on Automation of software test*, pp. 2–8, 2006.

[17] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[18] P. Chatterjee, M. Kong, and L. Pollock, "Finding help with programming errors: An exploratory study of novice software engineers' focus in stack overflow posts," *Journal of Systems and Software*, vol. 159, p. 110454, 2020.

[19] S. Kim, K. Pan, and E. E. J. W. Jr., "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 35–45, 2006.

[20] H. Washizaki, T. Xia, N. Kamata, Y. Fukazawa, H. Kanuka, T. Kato, M. Yoshino, T. Okubo, S. Ogata, H. Kaiya, *et al.*, "Systematic literature review of security pattern research," *Information*, vol. 12, no. 1, p. 36, 2021.

[21] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pp. 18–27, 2006.

[22] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 91–100, 2009.

[23] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 311–320, 2015.

[24] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE transactions on software engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[25] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[26] H. Zhang, X. Zhang, and M. Gu, "Predicting defective software components from code complexity measures," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pp. 93–96, 2007.

[27] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[28] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *IEEE International Conference on Software Maintenance*, pp. 346–355, 2008.

[29] D. Opitz and R. Maclin, "Popular ensemble methods: An empirical study," *Journal of artificial intelligence research*, vol. 11, pp. 169–198, 1999.

[30] Y. Liu, Y. Wang, and J. Zhang, "New machine learning algorithm: Random forest," in *Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings 3*, pp. 246–252, 2012.

[31] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[32] S. Kim, T. Zimmermann, K. Pan, E. E. J. W. Jr., *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pp. 81–90, 2006.

[33] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[34] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 334–345, 2015.

[35] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, pp. 1188–1196, 2014.