

Software Bug Prediction Using Git Repository History

Mr. Sangana Koti Reddy¹, Valeti Manjulatha², Hannah Jemima Vishkabai³, Thanusree Vemulapalli⁴, Sravani Gugulothu⁵

¹Assistant Professor, Dept. of Computer Science and Engineering, HITAM, Telangana, India

²³⁴⁵Student, Dept. of Computer Science and Engineering, HITAM, Telangana, India

Abstract-Modern software projects undergo continuous changes through feature development, maintenance, and bug fixes. As the number of files and commits increases, testing every component with the same level of attention becomes difficult and time-consuming. This study presents an approach for predicting software bug risk from Git repository history. Historical file-level metrics, including code churn, lines added, lines deleted, total commits, functional commits, commit quality, number of developers, and days since the last commit, are analyzed to understand development and modification patterns. Random Forest, Decision Tree, and Logistic Regression models are used to estimate the risk associated with individual software files. The system helps identify files that may require greater attention during testing and code review. The models are evaluated using accuracy, precision, recall, and F1-score. The results show that repository history can support more focused testing and maintenance decisions.

KeyWords: Software Defect Prediction, Machine Learning, Git Repository History, Predicted Defect Risk, Code Churn, Commit Quality, Random Forest, Decision Tree, Logistic Regression, Software Quality, Predictive Analytics.

1.INTRODUCTION

Software projects change continuously as developers add features, modify existing code, fix issues, and perform maintenance. As a project grows, maintaining software quality becomes more challenging because defects can be introduced at different stages of development. These defects may affect system reliability, increase maintenance effort, and create additional work for developers and testers. For this reason, identifying files that may need greater testing attention before deployment is an important software engineering task. Traditional testing often treats a large number of files as equally important. In practice, however, the development history of files can be very different. A file that has undergone substantial code changes or has been repeatedly modified may deserve more attention than a relatively stable file. Testing every component with the same priority can therefore require considerable time and resources.

Software Defect Prediction (SDP) uses information from past software development to estimate which components may be more likely to experience defects.

Machine Learning methods are useful for this task because they can learn patterns from historical repository data and use those patterns when making predictions for individual files.

1.1 Background and Software Defect Prediction

Machine Learning algorithms can analyze repository metrics and identify patterns related to software bug risk. In this study, the main metrics include code churn, lines added, lines deleted, total commits, functional commits, commit quality, number of developers, and days since the last commit. Together, these variables describe the modification history and development activity of individual files and are used as inputs to the prediction models. With the widespread use of version control systems, historical software repositories have become valuable sources of information for defect prediction. Git repositories maintain detailed records of software development activities, including commits, file modifications, developers, timestamps, and code changes. These historical records provide valuable insights into how software components evolve over time.

Based on these features, the models can predict whether a software file has a higher probability of containing defects.

1.2 Need for Git Repository-Based Defect Prediction

This study addresses the problem by analyzing Git repository history and applying Machine Learning techniques at the file level. Historical information such as code churn, lines added, lines deleted, total commits, functional commits, commit quality, developer involvement, and modification recency is extracted and processed. These features are then used to train models that estimate the risk associated with individual software files.

Analyzing Git repository history manually can be difficult and time-consuming, particularly for large software projects containing numerous files and commits. Therefore, an automated approach is required to collect repository information, extract meaningful metrics, and identify files that require greater testing attention.

The proposed system addresses this problem by analyzing Git repository history and applying Machine Learning techniques to predict defect-prone software files. Historical

metrics such as commit frequency, code churn, number of contributors, and file modification patterns are extracted and processed. These features are then used to train Machine Learning models, including Random Forest, Decision Tree, and Logistic Regression.

Instead of locating the exact bug within the source code, the proposed system predicts the defect risk associated with individual software files. The prediction results can help developers prioritize testing, code reviews, and maintenance activities. By focusing more attention on files with a higher predicted defect risk, software development teams can improve resource utilization, reduce testing effort, and enhance overall software quality.

Historical metrics such as code churn, lines added, lines deleted, total commits, functional commits, commit quality, number of developers, and days since the last commit are extracted from the Git repository history. These metrics provide information regarding the modification behavior and development activity of individual software files.

The extracted features are used to train Machine Learning models, including Random Forest, Decision Tree, and Logistic Regression. The system generates a predicted bug risk category along with the probability of a software file being defect-prone. These prediction results can help developers prioritize testing, code reviews, and maintenance activities.

2. LITERATURE REVIEW

Software Defect Prediction has become an important research area in software engineering because identifying defect-prone components at an early stage can help reduce testing effort and improve software quality. Previous research has explored different approaches using static code metrics, code changes, software repositories, data mining, and Machine Learning techniques.

With the increasing availability of version control systems such as Git, researchers have started analyzing historical repository information, including commits, code modifications, developer contributions, and change patterns. These historical metrics provide useful information regarding the evolution of software files and can be used to predict software defects.

2.1 Software Defect Prediction Using Software Metrics

Early research in Software Defect Prediction primarily focused on analyzing software metrics to identify components that may contain defects. Menzies et al. [5] demonstrated the use of data mining techniques and

static code attributes for developing software defect prediction models.

Software metrics provide information regarding the characteristics and complexity of software components. However, software systems continuously evolve through commits, feature additions, maintenance activities, and bug fixes. Therefore, static metrics alone may not completely represent the dynamic nature of software development.

Nagappan and Ball [6] investigated the relationship between code churn and software defects. Their study demonstrated the importance of analyzing code changes for predicting defect density. Code churn is particularly relevant to the proposed system because it represents the amount of code added and deleted during software development.

2.2 Git History and Commit-Based Software Defect Prediction

Git repositories maintain detailed records of software development activities and provide valuable historical information for software quality analysis. Commit history can provide information regarding code modifications, developers, timestamps, and changes made to individual software files.

Ranaivoson and Badri [1] investigated software fault severity prediction using Git history metrics and commits. Their research demonstrates the usefulness of Git repository information and commit-related metrics for analyzing software faults. The study supports the use of historical repository features for predictive software quality analysis.

Yokomori and Inoue [2] performed an empirical analysis of Git commit logs to investigate potential inconsistencies in software development history. Their work highlights the importance of analyzing commit information and development patterns stored within Git repositories.

Kim et al. [7] also demonstrated the usefulness of historical software information for predicting software faults. Similarly, Zimmermann et al. [8] explored the prediction of software defects using historical project data. These studies show that repository history can provide valuable information for identifying files and components that require greater attention during software development.

2.3 Machine Learning Approaches for Software Defect Prediction

Machine Learning techniques have been widely used to analyze software data and predict defect-prone components. These algorithms identify relationships between historical software metrics and defect occurrence.

Breiman [3] introduced the Random Forest algorithm, which is an ensemble learning technique based on multiple Decision Trees. Random Forest is widely used for classification problems because it can handle multiple input features and identify complex relationships within datasets.

Data mining and classification techniques provide the foundation for analyzing large software datasets. Han, Kamber, and Pei [4] discussed fundamental concepts and techniques related to data mining, classification, data preprocessing, and predictive analysis. These concepts are relevant to the proposed research because the system involves extracting historical repository data, preprocessing the dataset, and applying classification algorithms.

Lessmann et al. [9] compared different Machine Learning classification models for Software Defect Prediction and highlighted the importance of selecting suitable algorithms and evaluation techniques.

Machine Learning models such as Random Forest, Decision Tree, and Logistic Regression have been widely applied to classification problems and are used in the proposed system to predict software defect risk.

2.4 Change Metrics and Historical Repository Analysis

Change-related metrics provide important information regarding the development history of software files. Features such as code churn, number of commits, developer contributions, and modification frequency can indicate the level of development activity associated with a file.

Kamei et al. [10] conducted a large-scale study on change-based software quality assurance and demonstrated the importance of historical change information for identifying risky software changes. Lessmann et al. [9] evaluated different defect prediction approaches and provided a comparison of methods used for software quality prediction. The analysis of historical repository metrics allows researchers to understand how software components evolve over time. Files with frequent modifications, significant code changes, or contributions from multiple developers may require additional testing and code review.

The proposed system uses similar historical information from Git repositories, including code churn, lines added, lines deleted, total commits, functional commits, commit quality, developer contributions, and days since the last commit.

2.5 Research Gap

Existing research demonstrates that software metrics, code churn, Machine Learning algorithms, and Git repository history can be effectively used for Software Defect Prediction. However, applying these approaches to provide a practical file-level risk prediction system remains challenging.

Many studies focus on predicting whether a component is defective or analyzing fault severity. The proposed research focuses on analyzing the historical development behavior of individual software files and generating an understandable prediction in the form of predicted bug risk and bug probability.

The proposed system extracts Git repository features related to code changes, commit activity, commit quality, developer contributions, and modification recency. Machine Learning algorithms, including Random Forest, Decision Tree, and Logistic Regression, are applied to analyze these features.

Unlike approaches that focus only on identifying the presence of a defect, the proposed system aims to provide a risk-based prediction that can assist developers in prioritizing software testing and maintenance activities. This can help development teams focus greater attention on files with a higher predicted probability of software defects.

3. METHODOLOGY

The proposed research follows a quantitative and experimental methodology. Historical software development data is collected from Git repositories and converted into numerical features representing software development activities. The processed dataset is then used to train and evaluate Machine Learning models.

The methodology consists of data collection, feature extraction, data preprocessing, model training, defect prediction, and performance evaluation.

The overall workflow of the proposed system is as follows:

**Git Repository → Data Collection → Feature Extraction
→ Data Preprocessing → Model Training
→ Defect Prediction → Performance Evaluation**

3.1 Research Method

The proposed research follows a quantitative research methodology because the system analyzes numerical and measurable software development metrics. Historical Git repository data is collected and converted into numerical features that are used as input for Machine Learning models.

An experimental approach is used to train and evaluate multiple classification algorithms. The performance of Random Forest, Decision Tree, and Logistic Regression models is compared using standard evaluation metrics. The objective of the methodology is to determine whether historical Git repository metrics can effectively predict defect-prone software files.

3.2 Git Repository Data Collection

The Git repository serves as the primary source of data for the proposed system. Historical information related to commits and file modifications is collected from the repository.

The data collection process extracts information such as commit history, modified files, developers, timestamps, lines added, and lines deleted. Each commit provides information regarding changes made to one or more software files.

Git-based tools and Python libraries such as GitPython and other repository mining techniques can be used to collect and process historical repository information. The collected data is organized at the file level to analyze the development and modification history of individual software files.

3.3 Feature Extraction

Historical Git repository metrics provide useful information about how individual software files evolve during development. Code churn represents the overall amount of change in a file, while lines added and lines deleted provide more detailed information about the modifications made during development.

Total commits and functional commits represent the development activity associated with each file. Commit quality provides an additional measure based on the proportion of functional commits relative to the total number of commits. The Developers metric represents the number of developers who contributed to a particular file, while Days Since Last Commit indicates the recency of file modification.

Together, these metrics describe code changes, development activity, developer involvement, and modification recency. The Machine Learning models use these historical patterns to estimate the predicted bug risk associated with individual software files.

Table -1: Features Used for Software Defect Prediction

Feature	Description
File	Name or path of the software file being analyzed
Code Churn	Total amount of code changed in a file
Lines Added	Total number of lines added
Lines Deleted	Total number of lines deleted
Total Commits	Total commits associated with the file
Functional Commits	Meaningful or functional commits

Commit Quality (%)	Percentage of functional commits relative to total commits
Developers	Number of developers who contributed to the file
Days Since Last C	Days since the file was last modified
Predicted Bug Risk	Predicted risk category
Bug Probability (%)	Probability of defect risk

preprocessing is performed before training the Machine Learning models.

The preprocessing process includes:

- Removing duplicate records.
- Handling missing values.
- Converting feature values into appropriate numerical formats.
- Removing unnecessary features.
- Selecting relevant features for prediction.
- Scaling numerical features where required. Feature scaling can be used to transform numerical values into a common range. This helps prevent features with larger numerical values from having an excessive influence on the Machine Learning models. The processed dataset is then prepared for model training and testing.

The features related to code changes, commit history, developer contributions, and modification recency provide useful information regarding the evolution of software files. These metrics are analyzed by the Machine Learning models to identify patterns associated with software defect risk.

3.4 Code Churn and Commit Quality Calculation

Code churn represents the amount of code changed during software development. It is calculated using the number of lines added and lines deleted from a software file.

Code Churn = Lines Added + Lines Deleted

A high code churn value indicates that a file has undergone significant modifications. Frequently changing files may require additional testing and code review because continuous modifications can increase the possibility of introducing defects.

Commit quality is another important metric used in the proposed system. It represents the proportion of functional or meaningful commits among the total commits associated with a file.

Commit Quality (%) = (Functional Commits / Total Commits) × 100

A higher commit quality percentage indicates that a greater proportion of the commits associated with the file are considered meaningful or functional. This metric helps analyze the overall quality of development activity within the Git repository.

3.5 Data Preprocessing

The raw data collected from Git repositories may contain duplicate records, missing values, inconsistent formats, and irrelevant information. Therefore, data

3.6 Machine Learning Models

The proposed system uses three Machine Learning algorithms for Software Defect Prediction:

- Random Forest
- Decision Tree
- Logistic Regression

The models are trained using the processed Git repository features and evaluated using the testing dataset.

3.6.1 Random Forest

Random Forest is an ensemble Machine Learning algorithm that consists of multiple Decision Trees. Each Decision Tree generates a prediction, and the final prediction is determined using the combined results of the trees.

Random Forest can handle multiple input features and identify complex relationships between software development metrics. It can also reduce the risk of overfitting compared with a single Decision Tree.

In the proposed system, Random Forest is used to predict whether a software file is likely to be defect-prone based on its historical Git repository metrics.

3.6.2 Decision Tree

A Decision Tree is a Machine Learning algorithm used for classification and prediction. It divides the dataset into different branches based on feature conditions and makes predictions by following a sequence of decisions.

Decision Trees are easy to interpret and can help identify important features associated with defect prediction.

In the proposed system, the Decision Tree model analyzes Git repository metrics and classifies software files according to their predicted defect status.

3.6.3 Logistic Regression

Logistic Regression is a Machine Learning algorithm commonly used for classification problems. It estimates the probability of an input belonging to a particular

class.

In the proposed system, Logistic Regression is used to predict whether a software file belongs to the defect-prone or non-defect-prone category based on its historical repository features.

Logistic Regression provides a useful baseline model for comparing the performance of other Machine Learning algorithms.

3.7 Model Training and Testing

The processed dataset is divided into training and testing datasets. The training dataset is used to train the Machine Learning models, while the testing dataset is used to evaluate their performance.

The models learn patterns from historical Git repository metrics during the training process. After training, the models make predictions on the testing data.

The performance of Random Forest, Decision Tree, and Logistic Regression is compared to determine the effectiveness of each algorithm.

The dataset was divided into **80% training data and 20% testing data**.

3.8 Performance Evaluation Metrics

The performance of the Machine Learning models is evaluated using Accuracy, Precision, Recall, and F1-score.

- **Accuracy**

Accuracy represents the overall percentage of correct predictions made by the model.

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

- **Precision**

Precision represents the proportion of correctly predicted defect-prone files among all files predicted as defect-prone.

$$\text{Precision} = TP / (TP + FP)$$

- **Recall**

Recall measures the ability of the model to correctly identify positive instances associated with software bug risk.

$$\text{Recall} = TP / (TP + FN)$$

- **F1-Score**

The F1-score provides a balance between Precision and Recall.

$$\text{F1-Score} = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

Where:

- TP represents True Positive.
- TN represents True Negative.
- FP represents False Positive.
- FN represents False Negative.

4. RESULTS

The experimental results indicate that Git repository history can be useful for software bug risk prediction. The extracted metrics capture different aspects of development activity and file modification history, allowing the models to identify patterns associated with higher predicted risk.

4.1 Dataset Description

The dataset used in this study was developed using historical information extracted from the Git repository of the JUnit 5 project. The collected data represents the modification and development history of individual software files.

The dataset includes Git repository metrics related to code changes, commit activity, developer contributions, and modification history. These metrics were extracted and processed to analyze the historical development behavior of software files and predict their defect risk. The dataset consists of **3,247 software file records**. The final prediction dataset contains **11 columns**, including the file identifier, Git-based metrics, and prediction results.

For the purpose of analysis, files classified as **Moderate Risk and High Risk** were considered predicted defect-prone files, while files classified as **Low Risk** were considered predicted non-defect-prone files.

- **Total number of records: 3,247**
- **Total number of columns: 11**
- **Number of predicted defect-prone files: 759**
- **Number of predicted non-defect-prone files: 2,488**
- **Dataset source: JUnit 5 Git Repository**

The distribution of the dataset indicates that **23.38% of the analyzed files were predicted as defect-prone**, while **76.62% were predicted as non-defect-prone**.

4.2 Dataset Feature Analysis

The extracted dataset contains Git repository metrics that represent different aspects of software development activity and file modification history.

These features were used to analyze patterns associated with software defect risk.

The dataset includes **Code Churn**, which represents the total amount of code changed in a software file. It is derived from the number of **Lines Added** and **Lines Deleted** during the development process. Files with significant code changes may require additional testing and review.

The **Total Commits** feature represents the number of commits associated with a software file. **Functional Commits** represent commits identified as meaningful or functional, while **Commit Quality (%)** indicates the proportion of functional commits relative to the total number of commits.

The **Developers** feature represents the number of developers who contributed to or modified a particular software file. Multiple developer contributions may indicate increased development activity and complexity.

The **Days Since Last Commit** feature represents the number of days since the software file was last modified. This feature provides information regarding the recency of development activity.

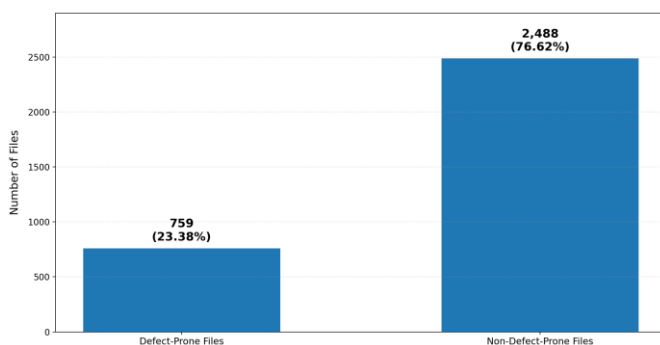


Fig -1: Distribution of Defect-Prone and Non-Defect-Prone Files

Fig. 1 illustrates the distribution of predicted defect-prone and predicted non-defect-prone files in the dataset. Out of the total 3,247 software files analyzed, 759 files (23.38%) were classified as predicted defect-prone files, including Moderate Risk and High Risk categories. The remaining 2,488 files (76.62%) were classified as predicted non-defect-prone files under the Low Risk category. The distribution indicates that the

majority of files have a low predicted defect risk; however, a significant proportion of files require additional attention during software testing, code review, and maintenance activities.

4.3 Random Forest Results

The Random Forest model was trained using the selected Git repository features. The model analyzed patterns associated with commits, code churn, developer activity, and file modifications to predict defect-prone files.

The performance of the Random Forest model was evaluated using Accuracy, Precision, Recall, and F1-score.

Table -2: Random Forest Performance

Evaluation Metric	Result
Accuracy	0.8631
Precision	0.8909
Recall	0.9414
F1-Score	0.9155

The results indicate the effectiveness of the Random Forest model in identifying defect-prone software files based on historical Git repository information.

4.4 Decision Tree Results

The Decision Tree model was trained using the same dataset and selected features. The model classified software files by creating decision rules based on the characteristics of the input features.

Table -3: Decision Tree Performance

Evaluation Metric	Result
Accuracy	0.8200
Precision	0.8850
Recall	0.8867
F1-Score	0.8859

The Decision Tree results demonstrate the ability of the model to classify software files based on their historical development characteristics.

4.5 Logistic Regression Results

Logistic Regression was implemented as a classification model to predict the probability of a software file being defect-prone.

Table -4: Logistic Regression Performance

Evaluation Metric	Result
Accuracy	0.8277
Precision	0.8401
Recall	0.9648
F1-Score	0.8982

The performance of Logistic Regression provides a baseline for comparing the results obtained from Random Forest and Decision Tree models.

4.6 Model Performance Comparison

The performance of all three Machine Learning models was compared using Accuracy, Precision, Recall, and F1-score.

Table -5: Comparison of Machine Learning Models

Model	Accuracy	Precision	Recall	F1-Score
Random Forest	0.8631	0.8909	0.9414	0.9155
Decision Tree	0.8200	0.8850	0.8867	0.8859
Logistic Regression	0.8277	0.8401	0.9648	0.8982

The experimental results show that the Random Forest model achieved the best overall performance among the three evaluated Machine Learning models. It obtained the highest Accuracy of **0.8631**, Precision of **0.8909**, and F1-score of **0.9155**. The higher F1-score indicates that the model achieved a good balance between Precision and Recall.

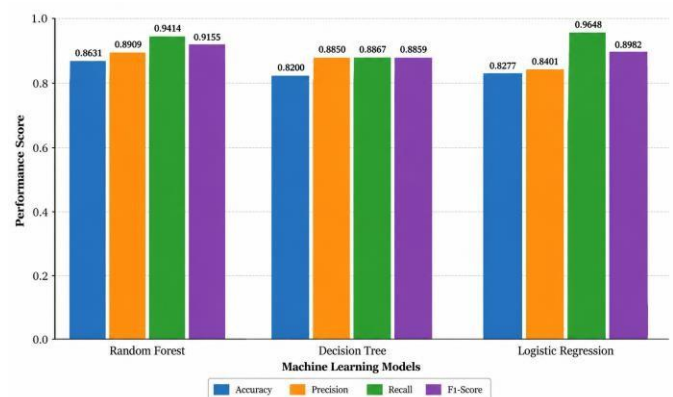
Logistic Regression achieved the highest Recall of **0.9648**, indicating that it identified a larger proportion of the positive class. However, its Precision and Accuracy were lower than those of the Random Forest

model, which resulted in a comparatively lower overall F1-score.

The Decision Tree model achieved an Accuracy of **0.8200** and an F1-score of **0.8859**. Although its performance was satisfactory, it did not perform as well as the Random Forest model.

Based on the combined evaluation of Accuracy, Precision, Recall, and F1-score, Random Forest demonstrated the strongest overall performance for the implemented Software Bug Prediction system.

Chart -1: Performance Comparison of Machine Learning Models



5. DISCUSSION

The experimental results indicate that historical Git repository metrics can support software bug risk prediction. The extracted metrics represent different aspects of code changes, commit activity, developer involvement, and modification recency. These patterns help the Machine Learning models estimate the risk associated with individual software files. Features such as commit frequency and code churn represent the level of modification activity associated with a software file. Files that undergo frequent modifications may experience a greater possibility of introducing defects because continuous changes can increase software complexity and maintenance requirements.

The number of contributors and developer activity also provide useful information regarding the development process. Files modified by multiple developers may experience integration issues or inconsistencies, potentially increasing the probability of defects.

The performance comparison of Random Forest, Decision Tree, and Logistic Regression provides insights into the effectiveness of different Machine Learning algorithms for the proposed dataset.

5.1 Performance Comparison and Analysis

The experimental results demonstrate that all three Machine Learning models were capable of predicting software defect risk using historical Git repository metrics. However, differences were observed in their overall performance.

The Random Forest model achieved the highest Accuracy of **0.8631**, Precision of **0.8909**, and F1-score of **0.9155**. These results indicate that Random Forest provided the best overall balance between correctly identifying potential defect-prone files and minimizing incorrect predictions.

Logistic Regression achieved the highest Recall of **0.9648**, indicating that the model was highly effective in identifying instances associated with potential software defect risk. However, its Accuracy of **0.8277** and F1-score of **0.8982** were lower than those achieved by the Random Forest model.

The Decision Tree model achieved an Accuracy of **0.8200**, Precision of **0.8850**, Recall of **0.8867**, and F1-score of **0.8859**. Although its overall performance was lower than Random Forest, the Decision Tree model produced comparatively balanced results and has the advantage of being easier to interpret.

Considering all four evaluation metrics together, Random Forest produced the strongest overall results among the three models. Its higher Accuracy and F1-score indicate that it provided the most balanced performance for the dataset used in this study. Therefore, Random Forest was selected as the best-performing model for software bug risk prediction using the extracted Git repository metrics.

5.2 Importance of Historical Git Metrics

Historical Git repository metrics provide useful information about how individual software files evolve during development. Code churn represents the overall amount of change in a file, while lines added and lines deleted provide more detailed information about the modifications made during development.

Total commits and functional commits represent the development activity associated with each file. Commit quality provides an additional measure based on the proportion of functional commits relative to the total number of commits. The Developers metric represents the number of developers who contributed to a particular file, while Days Since Last Commit indicates the recency of file modification.

Together, these metrics describe code changes,

development activity, developer involvement, and modification recency. The Machine Learning models use these historical patterns to estimate the predicted bug risk associated with individual software files.

5.3 Practical Significance of the Proposed System

The proposed system can assist software developers and testing teams in prioritizing their activities. Instead of allocating equal testing effort to every software file, developers can focus more attention on files that are predicted to have a higher defect risk.

The system can contribute to:

- Prioritization of software testing.
- Improved allocation of testing resources.
- Identification of frequently modified files.
- Improved code review planning.
- Efficient software maintenance.
- Improved software quality and reliability.

The proposed approach is particularly useful for software projects containing large numbers of files and frequent commits.

6. CONCLUSION

Software defects can affect reliability, maintenance effort, development cost, and the overall user experience. For large projects, giving every file the same level of testing attention is not always practical. Identifying files that may require additional review can therefore support more efficient testing and maintenance.

This study presented a Software Bug Prediction approach based on Git repository history and Machine Learning. Historical development data was analyzed at the file level using metrics such as code churn, lines added, lines deleted, total commits, functional commits, commit quality, number of developers, and days since the last commit.

The extracted metrics were used to train and evaluate Random Forest, Decision Tree, and Logistic Regression models. The models were evaluated using Accuracy, Precision, Recall, and F1-score.

Among the three models, Random Forest achieved the best overall performance, with an Accuracy of **0.8631** and an F1-score of **0.9155**. Logistic Regression achieved the highest Recall of **0.9648**, while the Decision Tree model showed comparatively lower overall performance.

The developed system focuses on estimating the predicted bug risk of individual software files rather than identifying the exact location of a bug in the source code. The prediction results can help developers and testing teams prioritize files for testing, code review, and maintenance.

Overall, the study indicates that historical Git repository metrics can provide useful information for file-level software bug risk prediction. By identifying files with a higher predicted risk, the approach can support more focused testing and better use of development resources. Future Scope

The proposed work can be extended in several ways. The system can be evaluated using larger datasets collected from multiple software repositories to examine its performance across different projects. Additional Machine Learning and Deep Learning models can also be explored and compared with the models used in this study.

Future work may include the analysis of commit messages and other textual repository information using Natural Language Processing techniques. The system could also be integrated with Continuous Integration and Continuous Deployment environments to support automated software quality monitoring.

An interactive dashboard and explainability techniques can further help developers understand the predicted bug risk and the factors that influence the predictions.

REFERENCES

- [1] H. Ranaivoson and M. Badri, "Software Fault Severity Prediction Using Git History Metrics and Commits," *Journal of Software*, vol. 17, no. 2, pp. 36–47, 2022.
- [2] R. Yokomori and K. Inoue, "An Empirical Analysis of Git Commit Logs for Potential Inconsistency in Code Clones," *arXiv preprint arXiv:2409.08555*, 2024.
- [3] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [4] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Morgan Kaufmann, 2011.
- [5] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.
- [6] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pp. 284–292, 2005.
- [7] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pp. 489–498, 2007.
- [8] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pp. 9–15, 2007.
- [9] 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.
- [10] Y. Kamei et al., "A Large-Scale Empirical Study of Just-in-Time Quality Assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.