International Journal of Advances in Data and Information Systems
Vol. 6, No. 1, April 2025, pp. 90~106
ISSN: 2721-3056, DOI: 10.59395/ijadis.v6i1.1368

Predicting Software Defects at Package Level in Java Project Using Stacking of Ensemble Learning Approach

Nabila Athifah Zahra[1], Amalia Anjani Arifiyanti[2], Dhian Satria Yudha Kartika[3]
[1,2,3] Department of Information System, Veteran National Development University, East Java, Indonesia

Article Info

Article history:

Received Feb 13, 2025
Revised Apr 12, 2025
Accepted Apr 29, 2025

Keywords:

Software Defects
Prediction
Classification
Ensemble
Java

ABSTRACT

Compared to manual and automated testing, AI-driven testing provides a more intelligent approach by enabling earlier prediction of software defects and improving testing efficiency. This research focuses on predicting software defects by analizing CK software metrics using classification algorithms. A total of 8924 data points were collected from five open-source Java projects on Github. Due to class imbalanced, undersampling was applied during preprocessing along with data cleaning and normalization. Tthe final dataset is consisting of 1314 instances (746 clean and 568 buggy). The predictive model is developed in two stages: base learner (level-0) using AdaBoost, Random Forest RF), Extra Trees (ET), Gradient Boosting (GB), Histogram-based Gradient Boosting (HGB), XGBoost (XGB), and CatBoost (CAT) algorithms, and meta-learner (level-1) that optimizes the results using ensemble stacking techniques. The stacking model achieved ROC-AUC score of 0.8575, outperforming all individual classifiers and effectively distinguishing defective from non-defective software components. The comparison of performance improvements between the base model (tree-based ensemble) and stacking was statistically validated using paired t-tests. All p-values were below 0.05, confirming the significance of Stacking's superior performance, with the largest gain observed against Gradient Boosting (+0.0411, p = 0.0030). The confussion matrix of stacking model is the most optimal model because it has high of True Positive and True Negative, while False Positive and False Negative values are relatively low. These findings affirm that ensemble stacking yields a more robust and balanced classification system, enhancing defect prediction accuracy and enabling earlier issue detection in the Software Development Life Cycle (SDLC).

Corresponding Author:
Nabila Athifah Zahra,
Department of Information System,
Universitas Pembangunan Nasional Veteran Jawa Timur,
Jl Raya Rungkut Madya, Surabaya, Indonesia.
Email: 21082010053@student.upnjatim.ac.id

1. INTRODUCTION

Software testing is one of the essential phases of the software *development life cycle (SDLC)* stage. The quality and stability of the software is a crucial thing that needs to be considered in software development because, from this stage, it will be known about errors, defects, or vulnerabilities of a system [1]. Software testing has stages of the process that are carried out systematically and planned, otherwise known as the *Software Testing Life Cycle (STLC).* STLC refers to specific stages ranging from requirements analysis, *test* planning, *test case* generation, *test* environment setup, and test implementation [2]. In the testing phase, the tester will observe the running of the system with the aim of finding the problems, failures or *errors.* Failure is defined when the running system is different from the expected conditions based on the *requirements of* a system [3].

Several types of software testing are most often used namely functional and structural testing. Functional testing, also known as *black-box* testing, is testing the functional features by observing the input and output results of the software without knowing the structure of the software code. Meanwhile, structural testing, also known as *white-box* testing, is software testing that analyzes and examines internal structures, such as code implementation, data flow, and possible failures in software. Software testing is very important because it not only has an impact on performance but also on the company's reputation for financial losses[4].

One of *Software Bug* incident happened to the Uber app in France. The bug caused customer trip data to be tracked even though they had left the application. This issue led to a 45-million-dollar lawsuit against the company. In the context of software development, the cost of *bug fixing* often increases exponentially depending on what phase of development the bug is found. According to the *Cost of Quality* principle*,* finding and fixing a bug in the production or post-release phase requires 30-100x more resources than if the bug was identified pre-release. In 2018, software company Tricentis claimed that 606 reports of software bugs caused the company to lose 1.7 million dollars [5]. The incident also caused 5-20% of users to lose trust in the company [6]. These incidents imply that *software defects* will cause losses both financially and reputationally. Therefore, the software testing phase is considered a crucial and very important phase.

There are several types of testing, one of which is definitely used is manual functional testing. Manual software testing is considered to require more time and resources, and there is still a possibility of *human error,* so the results are less efficient. There is a need to streamline the testing process by applying *automation testing, machine learning* (ML), and *artificial intelligence (AI*) [1]. *Machine Learning* is a part of AI that is able to learn data and improve performance without explicit programming. This is the foundation of *AI-driven.* The use of AI in software testing makes software more reliable, efficient and effective by utilizing automation and machine learning [7].

Currently, research on *AI-driven* testing is growing in both academia and industry. Bug prediction approaches are starting to focus on prediction rather than detection. This prediction approach aims to identify potential bugs in the early stages of the SDLC phase, in contrast to traditional bug detection, which is usually done after bugs appear or often uses automated *testing* tools. With the model's capability to predict software defects, testers can more efficiently manage resources, prioritize testing, and improve the quality of products [1]. With software defect prediction, testing resources can be optimized by directing focus to bug-prone areas. This approach will minimize the cost of software repair and modification after release.

Several studies on Software Defect Prediction are relevant to this research. One of them is research titled *Software Defect Prediction Using Ensemble Learning: An ANP-Based Evaluation Method,* which discusses the prediction of *software defects* using an *ensemble learning* approach [8]. This research aims to evaluate the performance of classification algorithms in *Software Defect Prediction* (SDP) by comparing the performance of *single classifiers* (SMO, MLP, KNN, and Decision Tree) with *Ensemble Methods* (*Bagging, Boosting, Stacking,* and *Voting*). This research uses 11 datasets of Java and C++ *software defect* projects taken from public repositories. This data includes software analytics matrices such as complexity and code size.

One of the *machine learning* model approaches used to predict *software defects* is classification or egression using *software code metrics.* This approach aims to identify software modules that are prone to errors or bugs based on the analysis of *source code metrics*. *Software metrics* generated from *source code* extraction are used to build predictive models. *Software metrics* that are commonly used in predicting *software defects* are McCabe Metrics, Halstead Metrics, and *Static Code Metrics* [9]. This approach is also done using *Chidamber and Kemerer Metrics* or CK matrix to build a prediction model for *software bugs* [5].

Several studies that discuss *software defect prediction* use Java-based projects as research objects because they have OOP-based *software metrics*. Java is an object-oriented programming language that has standard development documentation that makes it easy to use. Java has the scalability of a software solution that is able to provide strong performance and has scalability, which is very important for companies.

Table 1. The Popularity of Java Project

Nov 2024	Nov 2023	Change	Programming Language	Ratings	Change
1	1		Python	22.85%	+8.69%
2	3	^	C++	10.64%	+0.29%
3	4	^	Java	9.60%	+1.26%
4	2	v	C	9.01%	-2.76%
5	5		C#	4.96%	-2.67%
6	6		JavaScript	3.71%	+0.50%
7	13	^	Go	2.35%	+1.16%
8	12	^	Fortran	1.97%	+0.67%
9	8	v	Visual Basic	1.95%	-0.15%
10	9	v	SQL	1.94%	+0.05%

Java's popularity is evidenced by the fact that 90% of Fortune companies use Java [10]. Based on a survey conducted by TIOBE Programming Community Index in 2024, which is shown in Figure 1, Java is ranked 3rd in the most popular programming language with a *rating of* 9.6%. The popularity of Java in November 2024 increased by 1.26% compared to November 2023 [11]. According to a survey conducted by Eclipse in 2019, Java became the most important programming language in the realm of *Artificial Intelligence (AI), Internet of Things (IoT)*, and big data. In the field of AI, Java is used for the development of *Machine Learning* solutions*, Neural Networks,* genetic programming, and multi-robot systems. Therefore, in this research, Java is chosen as the object of focus because it has a high level of use or adoption in the enterprise. Java also has historical relevance in previous research on *software defects.* By using Java-based projects as the object*,* this research will utilize *Object Oriented Programming-based* software metrics in Java to produce a more predictive model of *software defects*.

Based on previous research, one of the methods used to predict *software defects* is using the *ensemble* method*.* The *ensemble learning* method, or the combination of several base models, has proven to be able to improve the accuracy of the model compared to approaches that only use one algorithm (*Single Classifier)* [12]*.* One of the ensemble learning techniques used in this research is the *Stacking technique. This method* combines predictions from several *base* models (*base classifiers)* to produce models that have a higher level of accuracy. *An ensemble* approach to predicting *software defects* has also been made using 10 NASA MDP public datasets and using 13 different performance measures [13]. The ensemble method used in the study was *stacking,* which resulted in an accuracy of 92.53%. The *ensemble* method approach for predicting *software defects* has also been carried out using Random Forest, Extremely Randomized Trees and XGBoost algorithms as a baseline classifier and using ensemble techniques in the form of a Stacking Classifier (STC) to produce the best accuracy rate [14]. When compared with other ensemble methods, the *stacking* method is considered superior because of its ability to combine the strengths of various models by prediction results and training *meta-learner* models to optimize the final prediction.

Based on the background description above, it is found that *software defect* prediction research is urgently needed to minimize the loss or impact of the discovery of post-release *software defects*. By predicting bugs before the *software* is released, the cost of *bug fixing* will also be reduced. Moreover, with the existence of *software analytics*, this technique can be implemented in various *object-oriented programming-based* project development. Based on this background, the aim of this research is to determine the prediction of software defects using the *ensemble learning* method. Seven algorithms will be used to create a *base model* in this study*,* including Adaptive Boosting, Random Forest, Extra Trees, Gradient Boosting, Histogram Gradient Boosting, XGBoost, and Categorical Boosting. The ensemble stacking technique was also used to make the model's final prediction. The output produced in this is a *machine learning* model to predict *software defects*.

2. RESEARCH METHODOLOGY

Based on Figure 2**,** this research begins with collecting software defect data. The data collected is historical data related to Java project software defects obtained from *open source* projects on the Github *repository platform* and the results of extracting *software metrics* from the *project*. After the data is collected, the next step is to *pre-process* the data to clean, normalize, and transform the data so that it is ready to be

used in model training [15]. Next, the classification stage is carried out, the training model is trained through two levels, the *base learner (level-0)* and *the learner model (level-1).* In the *base learner* training, *tree-based ensemble* algorithms used include AdaBoost, Random Forest (RF), Extra Trees (ET), Gradient Boosting (GB), Histogram-based Gradient Boosting (HGB), XGBoost (XGB), and CatBoost (CAT) [12]. During data training, hyperparameter optimization is also performed to optimize the value of the *base learner* model .

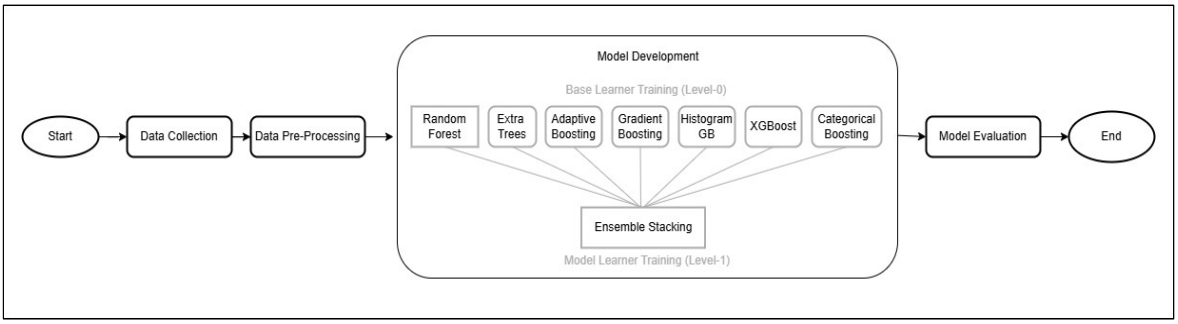

Figure 2. Flow of Research Methodology

Furthermore, the optimized model will be used at Level-1 as *input* for the *Stacking Ensemble* process. At this stage, several basic models are combined to produce the final prediction using the stacking ensemble technique with the Logistic Regression algorithm as a meta-classifier to get the final prediction value [16]. In the *testing* layer, the performance of the trained model will be tested. Testing is done using *testing data* as input for the trained model.

Furthermore, the prediction process is carried out to validate the performance of the model with the final result in the form of classification between *defective* and *undefective.* The output of the testing layer ensures that the model performs as expected on test data that has never been seen before. Once the model is trained, performance evaluation is performed to measure the performance of the model with relevant evaluation metrics, such as *accuracy*, precision, roc_auc, f1score, and *recall* [5].

1.1 Data Collection

The data collection process begins with the selection of open source Java projects with the provision of having more than 1000 commits [17]. The project must also have documentation or bug history on its modules because it will be used in the data labeling process. Furthermore, after selecting the project, software metrics extraction will be carried out using the CK Metrics Calculator. CK Metrics Calculator is a software analytics tool specifically for projects based on Object Oriented Programming (OOP) Java language[18]. This tool will calculate the code matrix at the package level in Java projects using static analysis, so it does not require compiled code [18].

The resulting analysis coverage is CK Metrics (Chidamber and Kemerer Metrics Suite) which consists of class-level metrics such as Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), and Number of Methods (NOM) [19]. There are also method-level metrics such as Cyclomatic Complexity and Line of Codes (LOC). The result of the extraction process is raw software metrics data. This data is quantitative information that reflects the quality and characteristics of the software project being analyzed. The dataset used for the training data of the software defect prediction model in this research is software analysis matrix data from java-language open source projects obtained from public repositories, namely Github. In this research, the dataset used comes from 5 Java-language open source projects that have bug trackers documentation, in Github [20]. The data shows the results of softwaare metrics extraction from 5 open-source Java projects, namely JFreeChart, Closure-Compiler, Commons-Math, Commons Lang, and Mockito, with a total of 8924 packages.

1.2 Data Preprocessing

In the data pre-processing stage, several steps must be taken to manage the results of data collection. The process starts with software metrics feature selection, where relevant features are selected according to the purpose of the analysis, such as class, method, or code complexity metrics [15]. Next, data cleaning is performed to ensure the quality of the dataset, including removing duplicate data, handling missing values, and correcting errors. After that, in the data acquisition stage, the data is divided into training data and testing data. The process continues with data balance analysis, which aims to check the distribution of data for balance, thus avoiding bias and ensuring representative analysis results.

1.3 Software Metrics Feature Selection

Software metrics feature selection is a crucial step in the software defect dataset creation process. The goal is to identify important attributes relevant to the analysis needs while filtering out unnecessary attributes from the raw data [22]. This step helps to increase the efficiency of the analysis and reduce the complexity of the processed data. In the raw data, some attributes are irrelevant or do not contribute significantly to the identification of *software defects*. Therefore, the selected features need to be aligned with research or recognized standards. In this context, feature selection is done by referring to CK metrics (Chidamber & Kemerer metrics) [5]. CK metrics include important metrics such as Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), and others, which are proven to provide significant insights into code quality and potential defects. This research shows that the use of CK metrics can help identify patterns in code that are potential sources of bugs while improving the accuracy of *defect* prediction. Feature selection not only serves to improve efficiency but also has a significant impact on the quality of analysis results. For example, research shows that datasets focused on important attributes have a higher prediction accuracy rate than datasets that use all attributes without selection [5].

1.3.1 Data Splitting

The *data splitting* scheme in this study divides the dataset into two parts, namely 80% for the training process and 20% for test data (holdout). The training dataset (X_train, y_train) will be divided into two parts, namely (X_train_base, y_train_base) as much as 50% of the training set to train the base model. Moreover, (X_train_meta, y_train_meta) as much as 50% also to train the meta-model [16]. The first division aims to set aside holdout data for final validation, while the second division aims to create the base model and meta-model in the stacking ensemble

1.3.2 Data Cleaning

Data cleaning is an important step to ensure the quality of the dataset to be used in software metrics analysis [23]. At this stage, the collected raw data is thoroughly examined to identify and address issues such as duplicate data, *missing values*, data *outliers*, and errors in data format. Data cleaning aims to improve the accuracy of the analysis results by removing elements that may negatively affect the results.

1.3.3 Data Balance Analysis

Data balance analysis aims to evaluate the distribution of classes in a dataset, such as the ratio between *buggy* and *non-buggy* data. Unbalanced data distribution can cause the analysis model to be biased towards the majority class, thus reducing the predictive ability of the minority class. This imbalance often occurs in classification problems where one class has more samples than the other. Some techniques used to deal with this problem are oversampling the minority class or *undersampling* the majority class[9].

1.4 Data Labeling Pipelines

The labeling pipeline mechanism focuses on creating a reliable dataset of real bugs for Java programs by identifying actual bug fixes from version control history [21]. The process begins by scanning the version control logs of open-source Java projects to identify commits that fix bugs. This identification is usually done by checking whether a commit references a bug ID from a bug tracker or if the bug tracker links back to a commit. Only commits that involve changes to the source code (excluding documentation or configuration files) are considered valid bug fixes.

1.5 Base Model Training

In this stage, the dataset that has been divided into *training data* will be trained using the *tree-based ensemble* algorithm. Tree-based ensembles are one of the methods in *ensemble learning* that combines several *decision trees* to improve prediction performance. This research uses *tree-based ensemble* algorithms including AdaBoost, Random Forest (RF), Extra Trees (ET), Gradient Boosting (GB), Histogram-based Gradient Boosting (HGB), XGBoost (XGB), and CatBoost (CAT) [12]. This method has been widely used in software defect prediction due to its ability to handle complex data and provide results that can be interpreted well. The algorithm method for creating a *baseline model* for predicting *software defects* is based on the references in the following table.

Each of these algorithms has a different approach to model building, with the main goal of reducing bias and variance in predictions. For example, AdaBoost prioritizes improving accuracy by giving more weight to data that is difficult to predict. Meanwhile, Random Forest and Extra Trees use a random feature selection approach to build a variety of more stable decision trees. addition, Gradient Boosting and XGBoost focus on incrementally improving the model by minimizing previous prediction errors. The model training process is done by parameter tuning the various hyperparameters for each

algorithm, in order to find the best combination that improves model performance on the data used [12].

Table 2. Reference of Tree-Based Ensemble Algorithm

Algorithm	Reference
AdaBoost	[12]; [24]; [25]
Random Forest (RF)	[14]; [12]; [15]; [24]
Extre Trees (ET)	[12]; [14]
Gradient Boosting (GB)	[12]; [25]
Histogram-based Gradient Boosting (HGB)	[12]; [26]
XGBoost (XGB)	[14]; 12]; [19]; [27]
CatBoost (CAT)	[12]; [24]

1.5.1 Random Forest

Random Forest is part of the *ensemble learning* algorithm, which is an algorithm that utilizes many models to get more accurate prediction results than just using decision trees [23]. The main concept of naming "Random" in Random Foret is that sampling is done randomly from the training data set and *feature subsets* are always considered when separating *nodes*. For example, if there are *p* total features then only *m* features are randomly selected for *data splitting.*

1.5.2 Extra Trees

Extra Trees (ET) is an algorithm that is almost similar to the Random Forest algorithm, which both create *decision trees* to make the final prediction by combining the results of all decision trees. What is different between ExtraTrees and Random Forest is that each decision tree is trained using the entire dataset and nodes or features are selected randomly [12].

1.5.3 Adaptive Boosting

AdaBoost or Adaptive Boosting is a classification algorithm that works iteratively by training *weak learners* such as *decision trees* on a dataset and weighting each training instance based on its classification. *Instances* that are difficult to classify will be the focus of greater attention in subsequent iterations. The final prediction is calculated by integrating the results of all *base classifiers* using a *weighted majority vote* approach*,* where each *base classifier* will contribute based on its performance. The advantage of the AdaBoost algorithms that it has the ability to adaptively and iteratively correct classification errors so that it can produce a more accurate model than just one base model [12].

1.5.4 Gradient Boosting

Gradient Boosting (GB) is a generalization of the AdaBoost ensemble method that allows the use of various loss functions. Unlike AdaBoost, GB utilizes the gradient to build a new base classifier instead of the weight of the misclassified instances [25]. Although GB improves efficiency in building the base classifier, it has the disadvantage sub-optimal memory usage and processing time.

1.5.5 Histogram Gradient Boosting

Histogram-Based Gradient Boosting (HGB) is an *ensemble boosting* method that uses feature histograms to select the best *split* efficiently and reliably. Compared to Gradient Boosting (GB), HGB has an advantage in terms of processing speed, making it more optimal for handling large and complex datasets [26].

1.5.6 XGBoost

XGBoost (Extreme Gradient Boosting) is a machine learning *ensemble* algorithm designed to improve the performance of gradient-boosted decision trees algorithms through a faster, parallel, and distributed approach. Through model tuning, parameter regulation, and memory usage efficiency, XGBoost can significantly reduce computation time. Essentially, XGBoost is used to minimize the *loss function* by adding a *weak classifier* [28]. XGBoost offers additional capabilities such as handling data with missing values (Sparse Aware), parallel structure to improve performance, and the ability to work with additional data on the trained model [29].

1.5.7 Categorical Boosting

CatBoost (Categorical Boosting) is a meta model for classification. This algorithm has two main characteristics. First, CatBoost effectively handles categorical features using a *one-hot encoding* technique. Second, it uses *oblivious decision trees* as the base classifier, where each level of the tree uses the same *splitting* criteria across *nodes*. This symmetrical tree structure helps minimize *overfitting*

and speeds up training time. CatBoost has a reliable performance in performing classification compared to other algorithms [25].

1.5.8 Stacking Ensemble

Stacking is a heterogeneous ensemble model that combines predictions from multiple base classifiers through a meta-classifier to produce a final prediction model. The training dataset is divided into two parts: one for training the base classifier and another for training the meta-classifier. Each base classifier is trained using the entire training dataset with different learning algorithms. The prediction results of the base classifier are used as input to train the meta-classifier which will generate the final prediction by combining the outputs of all the base classifiers. The stacking algorithm consists of three main steps, namely base classifier training, new dataset creation, and meta-classifier training [28]. This approach allows stacking to utilize the combined power of various learning algorithms to improve the accuracy of the prediction model [8]. The following is the formula used in the stacking technique shown in formula (1).

$$y_{base} = [y_1, y_2, \ldots . y_{n-1}] \quad (1)$$

Description:
y_{base}=final model
y_{i-n}=value of each model

1.6 Model Performance Evaluation

After all the model training is completed and the accuracy and performance values of the *ensemble learning* method are obtained, the next step is to evaluate the model. Several evaluations were conducted to assess the performance of the *ensemble* prediction model, including *accuracy, precision, Recall, f1score, and roc_auc score* . Accuracy (2) is calculated as the ratio of the number of correct predictions to the total data, but it is less effective if the classes are not balanced. Precision (3) measures the proportion of correct positive predictions out of all positive predictions, while Recall (4) measures how many positive instances were successfully identified out of the total true positive instances. F1 Score (5) which is the harmonic mean between Precision and Recall, is used to balance the error between False Positive (FP) and False Negative (FN), especially in unbalanced datasets. ROC-AUC (6) is used to assess the model's ability to distinguish between positive and negative classes by calculating the area under the ROC curve. The higher the AUC value (close to 1), the better the model is at distinguishing classes, while a value of 0.5 indicates the model's performance is no better than a random guess [5]. The following is the formula for calculating the model performance evaluation.

$$Accuracy = \frac{TN+TP}{TP+TN+FP+FN} x100\% \quad (2)$$

$$Precision = \frac{TP}{TP+FP} x100\% \quad (3)$$

$$Recall = \frac{TP}{TP+FN} x100\% \quad (4)$$

$$F1\ Score = 2\ x\ \frac{precision\ x\ recall}{precision+recall}\ x\ 100\% \quad (5)$$

$$ROC\text{-}AUC = \int_0^1 TPR(FPR)\ d\ FPR = \int_0^1 TPR\left(FPR^{-1}(x)\right)dx \quad (6)$$

Note: $TP = True\ Positive, TN = True\ Negative, FP = False\ Positive, FN = False\ Negative,$
$TPR = True\ Positive\ Rate, FPR = Flase\ Positive\ Rate$

3. RESULTS AND DISCUSSION

The results and discussion chapter will discuss the implementation of the program based on the stages of the research methodology, starting from data collection, data preprocessing, prediction model building**,** to its implementation.

3.1. Data Collection

Data collection is done by exploring open-source Java projects in the public repository, GitHub. There are several prerequisites for choosing a Java project that will be used as a dataset, which must have documentation and bug history in its modules. Through the bug history in the project modules, the data labelling process will be carried out in each module into the buggy and clean classes. In addition, the project must also have more than 1000 commit. Extraction or retrieval of software metrics data on the selected project is done through the CK Metrics Calculator tool developed by Mauricio[18]. This tool can extract a Java project into software metrics attributes. There are three architectural components used in this tool, including CK, runner, and MetricsExecutor. CK is the main component that manages the metrics collection process, starting from metrics initialization, file division based on memory capacity, managing execution in each directory, and optimizing Java file analysis. The *software metrics* data is in numeric form with the number as shown in table 3 below.

Table 3. Software metrics collection results

Project Name	Total Commit	Total Package
Jfreechart	4.226	1048
Closure-compiler	19.514	2844
Commons-math	7.229	1846
Commons-lang	8.419	1097
Mockito	6.237	2089
	Total	8924

3.2. Data Preprocessing

Before training the data, pre-processing will be carried out on the dataset collected in the previous stage. This pre-processing stage includes feature selection, data splitting, data balance analysis, and missing value checking. After analyzing the data balance, it was found that the distribution of the dataset was not balanced. Data imbalance between the two classes will allow overfitting to occur. Therefore, to overcome the data imbalance, several imbalanced data handling scenarios are carried out. There are several scenarios of handling imbalanced datasets used in the research including SMOTE (Synthetic Minority Oversampling Techniques), SMOTE-Tomek Links, and stratified undersampling techniques. The results when training the data show that the stratified undersampling technique produces a more optimal evaluation performance value compared to the other techniques. After *stratified undersampling* the data, the amount of new data obtained is 1314, with the distribution of clean classes reaching 746 and *buggy* classes totaling 568.

3.3. Data Labeling

Table 4. The Results of Labeling Software Metrics Dataset

Project Name	Number of Clean	Number of Bugs
Jfreechart	80	27
Closure-compiler	194	369
Commons-math	140	66
Commons-lang	81	63
Mockito	251	43
Total	746	568

Based on table 3 above, presents the outcome of labeling process applied accros 5 Java open sources projects choosen in previous section. This step of labeling pipelines is identifying bug-fixing commits by mining commit logs. Each project listed such as Jfreechart, Closure-compiler, etc underwent the systematic process of identifying bug fixing commits. The labeling pipeline successfully differentiated between of version code that are clean and those labeld as containing real bugs, which are associated with valid bug-fixing patches. For instance, the Closure-compiler project shows the highest number of identified bugs (369), which suggest that the version control and bug tracking records for this project are rich and well-maintened. Although, a project like Jfreechart with only 27 labeeled bugs, might have had fewer explicit bug references or less testable commit history.

3.4. Data Exploration

This research applies the Exploratory Data Analysis methodology to gain a deeper understanding of the dataset associated with the variables that determine whether or not a Java *package* project has *defects*. This stage is very important to identify patterns, correlations, and characteristics of a variable used in code quality analysis. This research uses multivariate correlation analysis to understand the relationship between variables that affect the complexity and potential defects in the code. Table 3 shows the statistical analysis results of each variable.

Table 5. Quantitative Analysis of Software Metrics Variables

Variable	Min	Mean	Max
Cbo	0.00	4.88	1334.00
Dit	1.00	1.48	39.00
fanin	0.00	1.98	405.00
Fanout	0.00	4.89	134.00
Lcom	0.00+ee	4.41	1.49
Noc	0.00	0.16	226.00
Loc	1.00	77.88	5584.00
Rfc	0.00	11.66	545.00
Wmc	0.00	15.62	1758.00
totalMethodsQty	0.00	8.45	1733.00
protectedMethodsQtys	0.00	1.15	366.00
publicMethodsQty	0.00	5.65	1726.00
privateMethodsQty	0.00	0.84	233.00
finalFieldsQty	0.00	0.90	143.00
protectedFieldsQty	0.00	0.07	54.00
publicFieldsQty	0.00	0.20	189.00
privateFieldsQty	0.00	1.35	142.00
bugs	0.00	0.06	1.00

An explorative analysis of the software metrics dataset revealed various characteristics that affect code complexity and quality. One of the main aspects of concern is Coupling Between Objects (CBO), which has an average value of 4.88 with a range up to 1334.00. This high maximum value indicates that some classes are highly dependent on other classes, which can increase system complexity and make code maintenance difficult. In addition, the Depth of Inheritance Tree (DIT) has an average of 1.48, with some classes reaching a depth of up to 39.00. Inheritance hierarchies that are too deep can complicate understanding the code structure and make debugging and testing difficult. In addition to the inheritance and dependency factors between classes, the analysis also showed that fan-in and fan-out have a major influence on the connectedness between system components. An average fan-in of 1.98 indicates that methods in a class tend to be called by several other classes, while a fan-out that has an average of 4.89 and a maximum of 134.00 indicates that a class is highly dependent on many other classes. This extensive dependency increases the risk of code instability, as a small change to one class can have a significant impact on other components it is associated with.

In addition, the lack of cohesiveness of methods (LCOM) metric, with an average of 4.41, indicates that some classes have methods that are not very cohesive, meaning they work on different parts of the class attributes. Low cohesion may indicate the need for refactoring to improve code modularity. On the other hand, the number of lines of code in a class (LOC) varies from 1 to 5584, with an average of 77.88, indicating that most classes are relatively small in size. However, some classes are too large, which may hinder code readability and maintainability. Other complexity metrics, such as Weighted Methods per Class (WMC), show that some classes have a very high number of methods, with a maximum value of 1758.00 and an average of 15.62. The more methods in a class, the greater the chance of errors due to increased complexity. This case is reinforced by the accessibility pattern of methods in the class, where public methods (publicMethodsQty) have the highest average of 5.65, while private methods (privateMethodsQty) are only about 0.84.

Classes that expose too many public methods can be more vulnerable to external changes and increase the risk of errors. Analysis of the bugs variable shows that most of the classes in the dataset do not have bugs, but there is a small percentage of classes that are prone to errors. This analysis indicates that classes with high complexity, as indicated by high Cbo, Wmc, and fan-out, tend to have a greater

risk of containing bugs. By understanding this pattern, the next step is to identify complexity reduction strategies, such as code refactoring, increasing modularity, and managing dependencies between classes to improve overall software quality.

3.5. Correlation and Significance

The correlation matrix in Figure 3 below measures how strong the relationship is between two variables. The correlation matrix ranges from -1 to 1. A positive value indicates a direct relationship when one variable increases the other also increases. Whereas a negative value indicates that there is an opposite relationship when one variable increases, the other will accordingly. From the matrix, it can be seen that some matrices have a high correlation value with each other. For example, publicMethodsQty has a very high correlation with privateMethodsQty (0.94), which indicates that the number of public and private methods tend to increase simultaneously within a class. Similarly, totalMethodsQty has a high correlation with protectedMethodsQty and privateMethodsQty, which makes sense since total methods are the sum of different types of methods in a class.

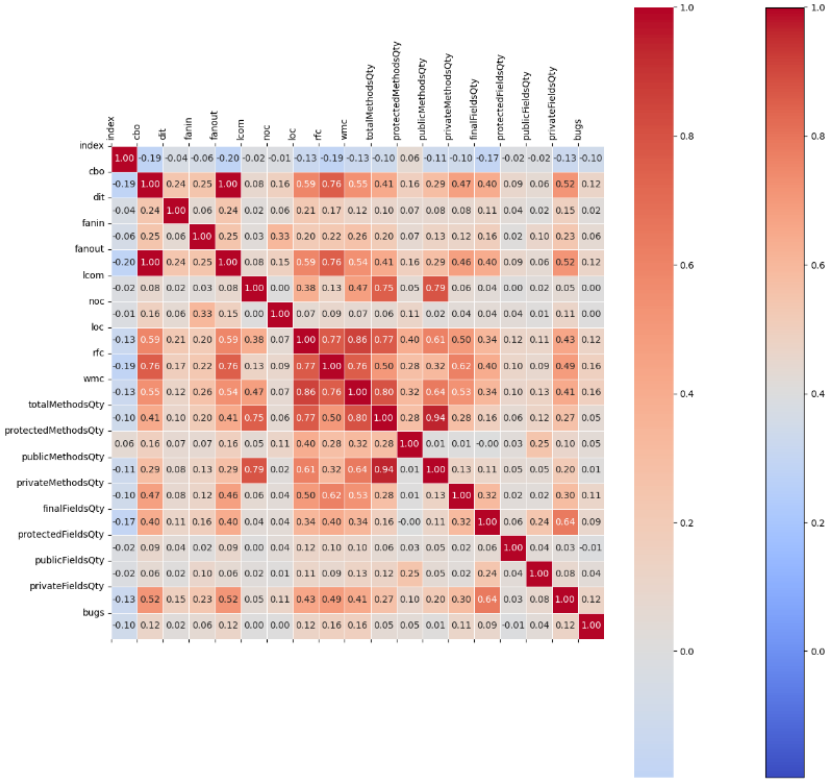

Figure 3. Inter-variable Correlation Matrix

Metrics such as cbo (Coupling Between Objects) and wmc (Weighted Methods per Class) also show a fairly high correlation (0.76), indicating that classes with high complexity tend to be more connected to other classes. In addition, lcom (Lack of Cohesion of Methods) has a positive correlation with fanout and fanin, meaning classes with less cohesive methods often have more dependencies on other classes. Meanwhile, the bugs variable has low correlations with most code metrics, although there is a slight relationship with wmc (0.16) and cbo (0.12), which could suggest that the more complex and connected a class is, the more likely it is to have bugs. Overall, these correlations provide insight into how code characteristics relate to each other. This can help developers understand how factors such as complexity, cohesion, and number of methods can affect code quality and the potential for bugs.

3.6. Model Development

This research develops a prediction model to predict *software defects* in Java package projects. The ensemble model build with *tree-based algorithm* as the base model, including Adaptive Boosting, Random Forest, Extratrees, Histogram XGBoost, and Categorical Boosting. Furthermore, the results of the data base model training will be integrated through ensemble stacking techniques with Random Forest Regressor as a meta model to optimize prediction quality. This approach is expected to be able

to optimize the accuracy, precision, roc_auc, f1score and recall values which become the matrix for determining model performance.

3.7. Random Forest

Random Forest uses hyperparameters that are tuned to achieve a balance between accuracy and generalization. n_estimators=40 ensures a sufficient number of trees to improve prediction stability without excessive computational cost. max_depth=4 limits the depth of the tree to avoid overcomplexity and overfitting. min_samples_leaf=5 prevents the tree from being too specific to the training data. criterion='gini' is used to measure the impurity of nodes in optimal data splitting. random_state=1 ensures consistent and reproducible results. This implementation of hyperparameter optimization makes Random Forest more reliable in handling data variability with stable performance [30].

3.8. Extra Trees

The Extra Trees algorithm uses hyperparameters that are tuned for a balance between accuracy and efficiency. n_estimators=30 ensures prediction stability without excessive computational cost. max_depth=8 limits the depth of the tree to prevent overfitting, while min_samples_leaf=4 prevents too specific splits. criterion='gini' is used for effective data splitting, and implementing gini for the criterion paramater. This combination makes Extra Trees more resilient to data variance while maintaining high accuracy.

3.9. Adaptive Boosting

AdaBoost or Adaptive Boosting is a classification algorithm that works iteratively by training *weak learners* such as *decision trees* on a dataset and giving weight to each training instance based on its classification. In the Adaptive Boosting algorithm, the hyperparameter tuning process is carried out using five parameters including estimator = AdaBoostClassifier with n_estimators = 50 and learning_rate =0. 1.

3.10. Gradient Boosting

Gradient Boosting uses several hyperparameters that have been adjusted to specific values to achieve a balance between bias and variance in the model. n_estimators=100 was chosen because this number is sufficient to capture patterns in the data without causing excessive overfitting. If it is too large, the model can become too complex and lose its generalization ability. learning_rate=0.01 was used because this value provides a good balance between stable convergence and learning speed. min_samples_leaf=5 was chosen to ensure that each leaf of the tree has at least two samples, which helps reduce the chance of the model overfitting the training data. max_depth=3 was used as this depth is sufficient to capture non-linear patterns in the data without making the model too complex. loss='exponential' was chosen to give more weight to hard-to-classify observations, similar to the approach used in Adaboost, so that the model focuses more on hard-to-correct errors.

3.11. Histogram Gradient Boosting

Histogram Gradient Boosting is a boosting algorithm that groups data into histograms to improve computational efficiency, especially on large datasets. In the hyperparameter tuning process, several key parameters are used to control the performance of the model. max_iter=50 specifies the maximum number of iterations or number of trees to be created, which affects the extent to which the model learns from the data without causing overfitting. learning_rate=0.01 controls the speed, where larger values speed up convergence but can sacrifice accuracy if too high. min_samples_leaf=5 sets the minimum number of samples that should be present in each leaf of the decision tree, which helps reduce overfitting by ensuring that each data division has enough observations. max_depth=3 sets the maximum depth of the decision tree, which affects the complexity of the model as well as the balance between bias and variance.

3.12. XGBoost

In the hyperparameter tuning process, several key parameters are used to optimize the model. estimators=50 sets the number of trees in boosting, where a larger number can improve accuracy but also increases the risk of overfitting. max_depth=4sets the maximum depth of the decision tree, which affects the complexity of the model as well as the balance between bias and variance. learning_rate=0.01 controls the speed, where larger values speed up convergence but can sacrifice accuracy if too high. With this combination of parameters, XGBoost is able to provide strong performance with a good balance between speed and accuracy in the machine learning process.

3.13. Categorical Boosting

CatBoost is a gradient boosting algorithm optimized to handle categorical features efficiently. In the hyperparameter tuning process, several key parameters are used to improve model performance. n_estimators=50 determines the number of trees used in boosting, this value is optimal because the time consumption during training is very low. loss_function='Logloss' is used for binary classification by measuring the probability of prediction error. learning_rate=0.1 controls the learning speed, with higher values speeding up convergence but may miss the optimal solution. depth=5 determines the maximum depth of the decision tree, which affects the complexity of the model. min_data_in_leaf=1 sets the minimum number of samples in each leaf to prevent overfitting. random_seed=1 ensures consistent and reproducible results. Finally, logging_level='Silent' reduces log output during training to make the process more concise. With this combination of parameters, CatBoost can produce more accurate and efficient models in classification tasks.

3.14. Stacking

The Stacking Ensemble Learning technique aims to improve classification performance by combining multiple base models and using meta models to produce more accurate final predictions. In this approach, several machine learning algorithms are used as base models, including RandomForestClassifier, ExtraTreesClassifier, AdaBoostClassifier, GradientBoostingClassifier, HistGradientBoostingClassifier, XGBClassifier, and CatBoostClassifie. Each of these models has different characteristics in handling data patterns, so by combining them, the system can utilize the advantages of each model.

Stacking Program Code

```
base_model_preds= []
for model in base_models:
   model.fit(X_train_base, y_train_base)
   pred = model.predict(X_train_meta)
   base_model_preds.append(pred)

stacking_dataset= np.column_stack(base_model_preds)
meta_model = RandomForestRegressor()
meta_model.fit(stacking_dataset, y_train_meta)
```

Figure 4. Stacking Program Code

In the first part of the code in Figure 4, a `base_models` list is created to store all the models used in the initial stacking stage. Then, an empty list `base_model_preds` is prepared to store the prediction results of each base model. In the `for` loop, each model in `base_models` is trained using the `X_train_base` and `y_train_base` datasets. After training, the model is used to make predictions on the `X_train_meta` dataset, and the prediction results are stored in a list `base_model_preds`. This process allows each model to provide an initial estimate of the class of the metadataset. After all the base models have provided predictions, the next step is to combine all the predictions into one stacked dataset using `np.column_stack(base_model_preds)`. This dataset has dimensions that match the amount of data in `X_train_meta` but with features derived from the predictions of the various base models. Thus, this dataset is no longer the original features but a representation of the decisions of the various base models that have been trained previously. The final step in stacking is to train the meta-model, which in this case is Logistic Regression. This model receives the stacked dataset (`stacking_dataset`) as input and is trained using `y_train_meta`. The meta-model acts as a decision maker that learns from the prediction patterns of the base models to make a more accurate final decision. With this approach, the system is able to reduce the bias of one particular model and increase generalization in classification.

3.15. Model Performance Evaluation

After performing the model building stages, the following are the results of evaluating the model performance of the training base model and the final model using stacking.

Based on the model performance evaluation results in Table 6, several important things can be obtained by comparing the performance of various models based on Accuracy, Precision, Recall, F1-Score, and ROC AUC metrics. From the table, stacking has the best performance compared to other models, with the highest Accuracy 0.8669, highest Precision 0.8712, highest Recall 0.8000, highest F1-Score 0.8341, and highest roc_auc score 0.8575. This result shows that stacking is able to improve

classification performance well compared to other basic models. In further analysis, the highest Accuracy was achieved by stacking with a value of 0.8669, while the lowest was Histogram Gradient Boosting with a value of 0.8098. This number indicates that stacking is able to combine the strengths of several base learner models to produce more accurate predictions. At the same time, Histogram Gradient Boosting has the lowest Accuracy, possibly because this model is too sensitive to data that is difficult to classify and tends to experience overfitting on noise in the dataset.

Table 6. Training Results of SDP Base Model and Learner Model

Matrix Evaluation	Random Forest	Extra Trees	Adaptive Boosting	Gradient Boosting	Histogram Gradient Boosting	XGBoosting	Categorical Boosting	Stacking
Accuracy	0.8365	0.8403	0.8441	0.8212	0.8098	0.8250	0.8403	0.8669
Precision	0.8190	0.8541	0.8415	0.8181	0.8061	0.8200	0.8469	0.8712
Recall	0.7818	0.7454	0.7727	0.7363	0.7181	0.7400	0.7545	0.8000
F1Score	0.8000	0.7961	0.8056	0.7751	0.7596	0.7809	0.7980	0.8341
Roc_auc	0.8288	0.8269	0.8340	0.8093	0.7969	0.8139	0.8282	0.8575

In the precision metric, stacking also shows the highest value of 0.8712, while Histogram Gradient Boosting has the lowest Precision of 0.8061. The high Precision of stacking indicates that this model produces fewer false positives than other models, which means it is better at avoiding the misclassification of negative classes. In contrast, the low Precision of adaptive boosting indicates that this model more often misclassifies negative classes as positive. The highest Recall is also achieved by stacking with a value of 0.8000, while the lowest is Histogram Gradient Boosting with a value of 0.7181. This number shows that stacking is able to capture almost all instances of the positive class, which is very important in scenarios where a mistake in detecting the positive class can have a big impact, such as in fraud detection. The low Recall of adaptive boosting indicates that the model more often fails to detect the positive class, resulting in many false negatives. In terms of F1-Score, stacking again shows the highest performance with a value of 0.8341, while adaptive boosting has the lowest value of 0.7586. The high F1-Score of stacking shows that this model has a good balance between Precision and Recall, which means that it is able to detect positive classes well while keeping the number of false positives low. In contrast, the low F1-Score in adaptive boosting indicates that the model is not optimal enough in handling the imbalance between Precision and Recall.

In the ROC AUC metric, stacking has the highest value of 0.8575, while Histogram Gradient Boosting has the lowest value of 0.7969. The high ROC AUC of categorical boosting and stacking indicates that these two models have a good ability to distinguish between positive and negative classes at various classification thresholds. In contrast, the low ROC AUC in Histogram Gradient Boosting indicates that this model has poorer discrimination ability than the other models, making it more difficult to distinguish between positive and negative classes well. The superiority of stacking in almost all evaluation metrics indicates that it is able to combine the strengths of various weak classifiers and utilize meta-learner models to produce more accurate decisions. It can mitigate the weaknesses of a single model that may be overly biased towards specific patterns in the data. In addition, stacking is also able to lift the performance of weak models by combining the outputs of several models so that the advantages of another model can compensate for the weaknesses of one model. In terms of bias and variance, stacking reduces the bias that may occur in decision tree-based models such as random forest or extra trees. It reduces the high variance in boosting models such as XGBoost or adaptive boosting. In this way, stacking can produce more stable and accurate predictions.

In addition, stacking is more resistant to noise in the dataset because it uses a variety of different approaches to handle the data, making it more difficult for noise to cause significant classification errors. Based on the evaluation results of these models, stacking is the best model to use for classification in this system, as it has the highest Accuracy, Precision, Recall, F1-Score, and ROC AUC compared to other models. Based on these results, the use of stacking is highly recommended for this classification scenario, especially if the main goal is to maximize Accuracy and ensure a balance between Precision and Recall.

3.16. Statistical Test

The statistical test of this research is based on ROC AUC (Receiver Operating Characteristic - Area Under Curve) as the primary performance metric. ROC AUC is widely recommended in the

domain of software defect prediction due to its ability to effectively handle class imbalance, such as the disparity between bug and non-bug instances, and its independence from classification thresholds. Unlike accuracy, which may be misleading in imbalanced datasets, ROC AUC offers a comprehensive measure of a model's ability to distinguish between defective and non-defective instances, making it a more reliable and informative metric in this context.

To enhance the robustness of the evaluation, the statistical testing is conducted using k-fold cross-validation with k = 5. This method partitions the dataset into five folds, where each model is iteratively trained on four folds and tested on the remaining one. The ROC AUC scores obtained from each fold serve as the input for the statistical tests. This cross-validation strategy reduces the risk of overfitting and ensures that the comparative analysis reflects consistent model performance across different data partitions.

Tabel 8. Shapiro Wilk Normality Test

Algorithm	Wilk Test	P-Value	Interpretation
RF	0.8645	0.2449	Normal
ET	0.9329	0.6160	Normal
AdaBoost	0.9240	0.5559	Normal
GBoost	0.9779	0.9232	Normal
HGB	0.9303	0.5983	Normal
XGBoost	0.9472	0.7169	Normal
CatBoost	0.7883	0.0648	Normal
Stacking	0.9283	0.5845	Normal

To validate the hypothesis disccussed in previous section, this research conduct Shapiro-Wilk Normality Test and Paired T-Test. These statistical tests are conducted to assess the normality of data distribution and to determine whether the performance difference among the algorithms are statistically significant. The shapiro Wilk test examines whether the performance score for each algorithm follow a normal distribution, which is an critical assumption for applying the paired t-test. As shown in the table 8, all algorithms have p-values greater than 0.05, indicating that their performance distributions do not significantly deviate from normal. Therefore, it can be interpret that the performance scores of each algorithm are normally distributed, validating the use of he paired t-test in the further step.

Tabel 9. Comparing Stacking with Other Algorithms with Paired T-Test

Algorithm	Mean Difference	T-Test	P-Value	Interpretation
RF	+0.0122	3.7384	0.0201	Significant
ET	+0.0391	8.7400	0.0009	Significant
AdaBoost	+0.0187	4.2677	0.0130	Significant
GBoost	+0.0411	6.4345	0.0030	Significant
HGB	+0.0387	5.7926	0.0044	Significant
XGBoost	+0.0197	2.8450	0.0466	Significant
CatBoost	+0.0338	5.4142	0.0056	Significant

The paired t-test was conducted to statistically compare the performance of the Stacking ensemble method against each of the other algorithms. T-test indicated the size of difference relative to the variation in the data. The larger the t-statistic, the more likely the means are different. Based the results on table 9, shows that each model comparisons yield p-values below 0.05, indicating that the differences in performance between stacking and each of the other models are statistically significant. However, statistical significance alone does not imply that stacking performs better—it is also essential to examine the direction of the performance differences.

For instance, the largest mean difference is observed between Stacking and Gradient Boosting (+0.0411), supported by a strong t-statistic of 6.4345 and a p-value of 0.0030, confirming that the improvement is statistically significant. Similarly, comparisons with Extra Trees (+0.0391), Histogram Gradient Boosting (+0.0387), and CatBoost (+0.0338) also show notable performance gains, each with p-values well below the 0.01 threshold, further supporting the robustness of the results.

Even the smallest observed mean difference, between Stacking and Random Forest (+0.0122), yields a t-statistic of 3.7384 and a p-value of 0.0201, which still meets the standard criterion for statistical significance ($p < 0.05$). This pattern holds across all comparisons: despite varying magnitudes of improvement, every p-value falls below 0.05, confirming that the performance improvements of the Stacking model are not due to random chance.

From the results determine that Stacking outperforms all the individual algorithms tested, with the differences being statistically significant. These findings are valuable as they support the use of ensemble methods like Stacking to combine the strengths of multiple base learners, leading to improved predictive performance. Moreover, since the normality assumption is satisfied, the statistical inference made from the paired t-test is reliable. This emphasizes that model selection should not only rely on average performance but also consider statistical validation to ensure robust conclusions.

3.17. Confusion Matrix

Confusion Matrix reflects four main metrics including True Positive (predicted true, actual true), True Negative (predicted false, actual false), False Positive (predicted true, actual false), and False Negative (predicted false, actual false). which give an indication of how the model classifies positive and negative classes in the test data. The following in table 6 is the result of the calculation of *confusion metrics* from all base learner algorithms and learner models.

Table 7. Confusion Matrix of Base Model and Learner Model SDP

Confusion Matrix	RF	ET	AdaBoost	GBoost	HGB	XGBoost	CatBoost	Stacking
True Positive	86	82	85	81	79	82	83	88
True Negative	134	139	137	135	134	135	138	140
False Positive	19	14	16	18	19	18	15	13
False Negative	24	28	25	29	31	28	27	22

Based on Table 7, the Stacking model shows the best performance, with the highest number of True Positives of 88, compared to other models such as Random Forest (86), AdaBoost (85),CatBoost (83), XGBoost (82), Extra Trees (82), Gradient Boosting (81), and the lowest is Histogram Gradient Boosting (79). This results indicates that Stacking is able to recognize positive classes better than other models. The ability to detect True Negative (TN) is also highest in the Stacking model (140), which means that this model is more accurate in classifying negative data correctly, followed by Extra Trees and CatBoost, which each have TN of 139 and 138. In contrast, the Random Forest and Histogram Gradient Boosting model has the lowest TN (134), indicating that it more often misclassifies negative data as positive.

In addition, the classification error measured by False Positive (FP) and False Negative (FN) shows that the Random Forest and Histogram Gradient Boosting model has the highest error rate, with FP of 19. This result indicates that the model more often misclassifies negative data as positive (False Positive) and fails to recognize the positive class (False Negative) correctly. In contrast, the Stacking model has the lowest misclassification rate, with an FP of only 13 and an FN of 22, which means it makes the fewest errors in identifying both classes.

Based on this analysis, Stacking is the optimal model for handling classification because it has high TP and TN and low FP and FN, resulting in better accuracy than other models. The Extra trees model is also a strong choice, as it performs close to Stacking with a low error rate. If computational complexity is a consideration, models such as AdaBoost and CatBoost can be an alternative, as they still have a good balance between accuracy and efficiency. In contrast, the Histogram Gradient Boosting model shows the weakest performance, as it has the highest number of misclassifications in both FP and FN, making it less recommended in this classification scenario.

4. CONCLUSION

The implementation of AI-driven testing in software testing provides a more efficient and accurate solution compared to manual testing. By utilizing machine learning (ML)-based approaches—particularly bug prediction through the Stacking Ensemble Model—this study demonstrates that software defects can be identified at earlier stages of the Software Development Life Cycle (SDLC). This early detection enables better resource management, reduces post-release maintenance costs, and enhances overall software quality. A total of 8,924 data points were collected from five different Java projects. Since the dataset was imbalanced, an undersampling technique was applied to address class distribution issues and improve model performance. During data preprocessing, steps such as data cleaning, normalization, and undersampling were carried out. After these processes, the remaining dataset consisted of only 1,314 data points, with a distribution of 746 clean instances and 568 buggy

instances. This research applies the Stacking Ensemble approach to improve the accuracy of software defect prediction using software metrics from open-source projects on GitHub. The model is developed in two stages: base learner (level-0) using AdaBoost, Random Forest (RF), Extra Trees (ET), Gradient Boosting (GB), Histogram-based Gradient Boosting (HGB), XGBoost (XGB), and CatBoost (CAT) algorithms, and a learner model (level-1) that optimizes the results with ensemble stacking techniques. Based on the model evaluation results, the Stacking Model outperforms other individual models. With an ROC-AUC score of 0.8575, the model proves to be more reliable in distinguishing between defective and non-defective software. Additionally, the stacking approach improves the performance of weaker classifiers such as Histogram Gradient Boosting. Therefore, the findings of this study confirm that combining multiple models through ensemble stacking yields a more robust and balanced classification system compared to using a single model. A paired t-test confirmed that the Stacking model's performance improvements over individual models are statistically significant, with all p-values below 0.05. The most notable gain was against Gradient Boosting (+0.0411, $p = 0.0030$), and even the smallest improvement remained significant. These results, supported by normality assumptions, validate the robustness and reliability of the Stacking approach. By leveraging various ensemble techniques, the model effectively minimizes prediction errors and enhances the accuracy of software defect detection. Future development of AI-driven testing models may involve testing on more diverse datasets and applying more comprehensive hyperparameter optimization to further improve performance. Moreover, integrating this predictive model into the Continuous Integration/Continuous Deployment (CI/CD) pipeline can support automation in the testing process and significantly improve software development efficiency.

ACKNOWLEDGEMENTS

The author would like to thank the research supervisor at the department of information systems, faculty of computer science, Universitas Pembangunan Nasional Veteran Jawa Timmur for valuable support to complete this research.

REFERENCES

[1] C. Deming, M. A. Khair, S. R. Mallipeddi, and A. Varghese, "Software Testing in the Era of AI: Leveraging Machine Learning and Automation for Efficient Quality Assurance," *Asian J. Appl. Sci. Eng.*, vol. 10, no. 1, pp. 66–76, 2021, doi: 10.18034/ajase.v10i1.88.

[2] G. Singh, "A Study on Software Testing Life Cycle in Software Engineering," *Int. J. Soft Comput. Eng.*, vol. 9, no. 2, pp. 1–5, 2018.

[3] O. J. Amman Paul, *Introduction to Software Testing*. Cambridge Press, 2017. [Online]. Available: https://books.google.co.id/books?hl=id&lr=&id=bQtQDQAAQBAJ&oi=fnd&pg=PR9&dq=Introduction+to+Software+Testing&ots=fA6P213_pQ&sig=vTKZfwwNJMsYUzib1KSgQ-TjRDI&redir_esc=y#v=onepage&q&f=false

[4] V. H. S. Durelli *et al.*, "Machine learning applied to software testing: A systematic mapping study," *IEEE Trans. Reliab.*, vol. 68, no. 3, pp. 1189–1212, 2019, doi: 10.1109/TR.2019.2892517.

[5] U. Subbiah, M. Ramachandran, and Z. Mahmood, "Software engineering approach to bug prediction models using machine learning as a service (MLaaS)," *ICSOFT 2018 - Proc. 13th Int. Conf. Softw. Technol.*, no. Icsoft, pp. 879–887, 2019, doi: 10.5220/0006926308790887.

[6] R. R. Saputra, E. Setiawan, and A. Ambarwati, "Manajemen Risiko Teknologi Informasi Menggunakan Metode OCTAVE Allegro pada PT . Hakiki Donarta Surabaya," vol. 17, no. 1, pp. 1–10, 2019.

[7] A. Pandey, S. Maddula, G. P. Kumar, S. K. Shailendra, and K. Mudaliar, "A Comprehensive Analysis of Ensemble-based Fault Prediction Models Using Product , Process , and Object-Oriented Metrics in Software Engineering," no. December 2023, 2024, doi: 10.5281/zenodo.10464708.

[8] A. O. Balogun, A. O. Bajeh, V. A. Orie, and A. W. Yusuf-Asaju, "Software Defect Prediction Using Ensemble Learning: An ANP Based Evaluation Method," *FUOYE J. Eng. Technol.*, vol. 3, no. 2, 2018, doi: 10.46792/fuoyejet.v3i2.200.

[9] R. Malhotra, S. Chawla, and A. Sharma, *Software defect prediction using hybrid techniques: a systematic literature review*, vol. 27, no. 12. Springer Berlin Heidelberg, 2023. doi: 10.1007/s00500-022-07738-w.

[10] D. Gray, "Why Does Java Remain So Popular?," *blogs.oracle.com*, 2019. https://blogs.oracle.com/oracleuniversity/post/why-does-java-remain-so-popular

[11] M. Crouse, "TIOBE Index for November 2024: Top 10 Most Popular Programming Languages," 2024. https://www.techrepublic.com/article/tiobe-index-language-rankings/ (accessed Nov. 25, 2024).

[12] A. Alazba and H. Aljamaan, "Software Defect Prediction Using Stacking Generalization of Optimized Tree-Based Ensembles," *Appl. Sci.*, vol. 12, no. 9, 2022, doi: 10.3390/app12094577.

[13] Y. Peng, G. Kou, G. Wang, W. Wu, and Y. Shi, "Ensemble of software defect predictors: An AHP-based evaluation method," *Int. J. Inf. Technol. Decis. Mak.*, vol. 10, no. 1, pp. 187–206, 2011, doi: 10.1142/S0219622011004282.

[14] M. A. Ihsan Aquil, "Predicting Software Defects using Machine Learning Techniques," *Int. J. Adv. Trends Comput.*

Sci. Eng., vol. 9, no. 4, pp. 6609–6616, 2020, doi: 10.30534/ijatcse/2020/352942020.
[15] M. Ali *et al.*, "Software Defect Prediction Using an Intelligent Ensemble-Based Model," *IEEE Access*, vol. 12, pp. 20376–20395, 2024, doi: 10.1109/ACCESS.2024.3358201.
[16] and B. H. Wang, Wenfeng, Jingjing Zhang, "Meta-learning with Logistic Regression for Multi-classification," in *New Approaches for Multidimensional Signal Processing: Proceedings of International Workshop*, Singapore: Springer Singapore, 2022. doi: https://doi.org/10.1007/978-981-16-8558-.
[17] J. Xu, F. Wang, and J. Ai, "Defect Prediction with Semantics and Context Features of Codes Based on Graph Representation Learning," *IEEE Trans. Reliab.*, vol. 70, no. 2, pp. 613–625, 2021, doi: 10.1109/TR.2020.3040191.
[18] Maurício Aniche, "Java Code Metrics Calculator (CK Metrics)," 2015.
[19] G. Esteves, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Understanding machine learning software defect predictions," *Autom. Softw. Eng.*, vol. 27, no. 3–4, pp. 369–392, 2020, doi: 10.1007/s10515-020-00277-4.
[20] G. Gay and R. Just, "Defects4J as a Challenge Case for the Search-Based Software Engineering Community," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 12420 LNCS, no. Section 2, pp. 255–261, 2020, doi: 10.1007/978-3-030-59762-7_19.
[21] C. S. . Pasareanu and D. Marinov, "Just, R., Jalali, D., & Ernst, M. D. (2014, July). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 international symposium on software testing and analysis (pp. 437-440).," pp. 2–5, 2014.
[22] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, "Complexity vs. performance," no. 119, pp. 384–397, 2017, doi: 10.1145/3131365.3131372.
[23] I. Q. U. Fatwa Ramdani, *Pengantar Data Science*. Jakarta: Bumi Aksara, 2022.
[24] E. Ronchieri, M. Canaparo, and M. Belgiovine, *Software Defect Prediction on Unlabelled Datasets: A Comparative Study*, vol. 12250 LNCS. Springer International Publishing, 2020. doi: 10.1007/978-3-030-58802-1_25.
[25] A. A. Ibrahim, R. L. Ridwan, M. M. Muhammed, R. O. Abdulaziz, and G. A. Saheed, "Comparison of the CatBoost Classifier with other Machine Learning Methods," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 11, pp. 738–748, 2020, doi: 10.14569/IJACSA.2020.0111190.
[26] Guryanov, "Histogram-Based Algorithm for Building Gradient Boosting Ensembles of Piecewise Linear Decision Trees," *Anal. Images, Soc. Networks Texts*, vol. 11832, pp. 39–50, 2019, doi: https://doi.org/10.1007/978-3-030-37334-4_4.
[27] G. Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Predicting Software Defects with Explainable Machine Learning," *ACM Int. Conf. Proceeding Ser.*, 2020, doi: 10.1145/3439961.3439979.
[28] Z. Faska, L. Khrissi, K. Haddouch, and N. El Akkad, "A robust and consistent stack generalized ensemble-learning framework for image segmentation," *J. Eng. Appl. Sci.*, vol. 70, no. 1, pp. 1–20, 2023, doi: 10.1186/s44147-023-00226-4.
[29] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, vol. 13-17-Augu, pp. 785–794, 2016, doi: 10.1145/2939672.2939785.
[30] S. K. Palaniswamy and R. Venkatesan, "Hyperparameters tuning of ensemble model for software effort estimation," *J. Ambient Intell. Humaniz. Comput.*, vol. 12, no. 6, pp. 6579–6589, 2021, doi: 10.1007/s12652-020-02277-4.