Predictive Exploration of Method-Level Bugs : Java and Weka API Collaboration

Thesis submitted

in partial fulfilment of the requirements

for the award of the degree of

Master of Technology

in

Data science

By

SATTI SONIYA

21VV1D8801

Under the Esteemed Guidance of

Dr. B. TIRIMULA RAO

Assistant Professor & HOD

Department of Information Technology

Department of Information Technology

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

GURAJADA,VIZIANAGARAM , 535003, A.P.

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY
GURAJADA,VIZIANAGARAM

CERTIFICATE

This is to certify that the dissertation titled **Predictive Exploration of Method-Level Bugs : Java and Weka API Collaboration** submitted by **SATTI SONIYA** under the Enrollment number **21VV1D8801**, to the JNTU-GV College of Engineering, Vizianagaram, for the award of the degree of **Master of Technology** in **Data science**, is a bona fide record of the research work done by her under our supervision during the year 2022-2023.

The results embodied have not been submitted to any other Institute or University for the award of any degree or diploma.

Signature of Project Guide	**Signature of Head of the Department**
Dr. B. TIRIMULA RAO	Dr. B. TIRIMULA RAO
Assistant Professor & HOD	Assistant Professor & HOD
Department of Information Technology	Department of Information Technology
JNTU-GV College of Engineering	JNTU-GV College of Engineering

DECLARATION

I, **SATTI SONIYA** (Reg. No: 21VV1D8801) declare that this submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original source. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature

SATTI SONIYA

21VV1D8801

Date :

Place :

ACKNOWLEDGEMENT

The acknowledgement transcends the reality of formality when I express deep gratitude and respect to all those people behind the screen who inspired and helped us in completion of this project work.

I take the privilege to express my heartfelt gratitude to project supervisor Dr. B. TIRIMULA RAO, Assistant Professor & Head of the Department of Information Technology, College of Engineering Vizianagaram,JNTU-GURAJADA VIZIANAGARAM for his valuable suggestions and constant motivation that greatly helped me in the successful completion of the project.

I express my sincere thanks to Dr. K. SRIKUMAR, Principal of College of Engineering Vizianagaram, JNTU-GURAJADA VIZIANAGARAM, for providing me with a good infrastructure and environment to carry out this project.

I am thankful to all Teaching and Non-Teaching staff of Information Technology Department, College of Engineering Vizianagaram, JNTU-GURAJADA VIZIANAGARAM for their direct and indirect help provided to me in completing the project. Finally, I am extremely thankful to my parents and friends for their constant help and moral support.

Abstract

In recent decades, extensive research has been dedicated to predicting software bugs across varying levels of granularity including file level, package level, and module level. Surprisingly, little attention has been given to the prediction of bugs at the method level. This project first collects 18 resulting open-source datasets that have conducted method-level bug predictions using code metrics and historical measures then assesses the effectiveness of various prediction model types. Finally, perform studies of each metric's ability to forecast, and then use the results to further examine the streamlined prediction models. This work addresses the Efficiency of Method-Level Predictions, Model Performance and Metric Variance, This work leverages the Weka tool for its analytical capabilities. The interaction of Weka's features with a Java plugin facilitates enabling seamless model development and evaluation. Moreover, the integration of the G-mean metric provides an additional dimension for assessing prediction model performance.
Keywords: Method-level bug prediction , Code metrics, History measures

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Bug

A bug is a flaw, mistake, or failure in the program or system being developed that leads to unanticipated outcomes.

1.1.1 what is software bug prediction?

The modules that are defect-prone and need thorough testing are identified using software defect prediction. In this manner, it is possible to use the testing resources effectively while yet violating the limitations.

1.1.2 Why software defect prediction is important?

To boost software reliability, software defect prediction has grown to be an important area of study. The usage of program defect predictions helps developers to spot prospective issues and to make the most use of testing resources to increase program dependability.

1.1.3 Context

Researchers have been working to forecast the presence of software defects for many years. The studies attempt to forecast problems in various software structures, ranging from single files (Units) and groups of files (Packages) as well as larger systems (Modules). Very few research have examined the possibility of predicting issues with methods, which are even smaller structures.

1.2 Different levels of bugs predictions

Bug predictions can be done at various levels like File level, package level, Module level, and Method level

1.2.1 File-level bug prediction:

Bug predictions at the file level concentrate on specific source code files inside a project.

1.2.2 Package-level bug prediction:

This makes it easier to spot packages that are more likely to have issues in them.

1.2.3 Module-level bug prediction:

Within a software system, modules represent logical units of functionality. At the module level, bug predictions concentrate on evaluating each module's level of complexity, interdependencies, and previous bug data.

1.2.4 Method-level bug prediction:

Within a module, methods are discrete operations or steps that carry out specific responsibilities. At the method level, bug predictions are made by looking at variables including method complexity, code metrics, parameter usage, and previously reported bugs for each method. The likelihood of defects in particular approaches can be predicted by looking at these parameters.

Predicting software units likely to contain bugs is the task of bug prediction, which could increase the effectiveness of bug solutions. Bug prediction has been a popular academic topic for several years. Numerous approaches and methods have been proposed by researchers for bug prediction. These methods and technologies typically draw on a variety of types of metrics (such as historical measurements and code metrics), and are used at several degrees of granularity, including the file level, the package level, Level of modules, etc[1].

Although method-level bug prediction could offer more detailed information and help with code reviews and inspections, there hasn't been much research into and analysis of method-level bug prediction models. Investigating method-level bug prediction models is the purpose of this work. More precisely, describe how to create bug prediction models based on metrics after collecting a wide range of method-level code metrics and history measurements. Further analysed the prediction importance of each measure to gain a better understanding of how much it influences the forecasts. Finally, we looked into ways to make the creation of prediction models simpler[1]

Initially, we collected 18 open-source datasets that conducted method-level bug predictions using both code metrics and historical measures.

Next, we applied either method-level code metrics or historical metrics to effectively anticipate bug-prone methods. Interestingly, models utilizing historical metrics demonstrated superior performance. Furthermore, models incorporating both types of metrics exhibited the highest level of performance.

In the prediction of bug-prone methods, distinct metrics or measures exhibit varying degrees of predictive efficacy. Notably, historical measures displayed superior predictive power compared to other metrics.

To assess the applicability of our work in constructing prediction models employing a diverse array of modeling methods, we further applied four additional techniques: Random Forest, Naive Bayes, Logistic Regression, Decision Tree, and SVM algorithms. These methods are easily implementable and widely utilized in bug prediction, and they demonstrated promising predictive capabilities as well. This total work is done using the integration of Java with weka

1.3 Integrating Weka API with Java

Weka, a comprehensive machine learning library, seamlessly integrates with Java, making it a powerful tool for developers and data scientists alike. By leveraging the Weka API (Application Programming Interface), one can harness the full potential of Weka's diverse set of algorithms within Java applications.

1.3.1 Foundation in Java

Since Weka is primarily written in Java, it naturally lends itself to Java-based projects. This compatibility ensures that the transition from Weka to Java is seamless, allowing developers to exploit the strengths of both environments.

1.3.2 Seamless Library Inclusion

To initiate a project incorporating Weka, developers can conveniently import the necessary Weka libraries into their Java environment. This is achieved by adding the Weka JAR (Java Archive) files to the project's build path, enabling the Java code to access the Weka classes and methods.

1.3.3 Loading and Preparing Data

Weka supports an array of data formats, including popular ones like ARFF and CSV files. This versatility allows developers to work with data from diverse sources. The Instances class serves as the conduit for loading and handling data. Through it, data can be read, organized, and manipulated as needed for the machine learning tasks at hand.

1.3.4 Algorithm Selection and Configuration

Choosing an appropriate algorithm is paramount to achieving optimal results. Weka offers a wide selection of algorithms, each designed to address specific machine learning tasks such as classification, regression, and clustering. Developers must carefully consider the nature of their data and the objectives of their project when selecting and configuring an algorithm.

1.3.5 Training the Model

Once the algorithm is selected and properly configured, it's time to train the model. This step involves feeding the algorithm with the training data so that it can learn the underlying patterns and relationships. The model-building process culminates in a trained model that is ready to make predictions.

1.3.6 Making Predictions

With a trained model in place, developers can apply it to new, unseen data. This process, known as inference or prediction, allows the model to generate output based on input it has not encountered before. This is a critical step in deploying machine learning models for real-world applications.

1.3.7 Evaluating Model Performance

Assessing the performance of a machine learning model is crucial for ensuring its effectiveness. Weka provides a range of evaluation metrics and tools that allow developers to rigorously analyze the model's performance on test data. This step helps identify areas for improvement and informs decisions about model refinement.

1.3.8 Error Handling and Exception Management

Robust error handling is essential in any software development project. Incorporating mechanisms to gracefully handle exceptions and errors ensures that the application remains stable and reliable, even in the face of unexpected events.

The seamless integration of Weka with Java empowers developers to harness the extensive capabilities of both environments. By following the steps outlined above, practitioners can build robust and effective machine learning applications that leverage Weka's rich set of algorithms within the familiar framework of Java. This integration opens up a world of possibilities for creating intelligent, data-driven solutions across various domains.

To develop, train, test, and assess method-level bug prediction models: First, code metrics and history measures were used in this work, which was conducted on a dataset of 18 large-scale software projects.We will use the datasets for our discussions are acquired from github.com. Second, the labelled data samples (i.e., methods) were used to determine whether they are bug-prone or not. To create prediction models, we thirdly used a supervised machine learning technique. Finally, we assessed how well these derived prediction models performed at the method level after training and validating the perdition models.

History measures:

Using historical metrics to predict bugs has become a widespread practice. These metrics capture the alteration history of code segments as recorded in the revision history. various historical measures from three distinct perspectives, Firstly, to signify the alterations in lines of a method, promptly computed a series of history measures (H1-H5). These encompass metrics like the number of lines of code that have been added, deleted, or modified within a method, how frequently a method has undergone modification, and the number of authors involved in a method's alterations. Secondly, examined the alteration records of the Abstract Syntax Tree (AST) tokens for each method, leading to the derivation of history measures (H6-H11). These encompass factors such as the count of statements, comments, parameters, etc., that have undergone modification. Lastly, computed a set of relative historical measures (H12–H19) to provide a comprehensive view.

Table 1.1: List of method-level History measures

Index	Measure name
H1	Added LOC
H2	Deleted LOC
H3	Changed LOC
H4	Number of Changes
H5	Number of Authors
H6	Number of Modified Statements
H7	Number of Modified Expressions
H8	Number of Modified Comments
H9	Number of Modified Return type
H10	Number of Modified Parameters
H11	Number of Modified Prefix
H12	Added LOC/LOC
H13	Deleted LOC/LOC
H14	Added LOC/Deleted LOC
H15	Changed LOC/Number of Changes
H16	Number of Modified Statements/ Number of Statements
H17	Number of Modified Expressions/ Number of Statements
H18	Number of Modified Comments/ LOC
H19	Number of Modified Parameters/ Number of Parameters

Code metrics: Over time, bug prediction has extensively relied on code metrics, which provide insights into the characteristics of source code. compute a range of code metrics at the method level from four distinct perspectives. Firstly, a set of code metrics (C1-C7) based on the size of each method, considering it as a textual sequence. This encompassed metrics such as lines of code (LOC), comment lines of code, and the count of blank lines. Secondly, derived metrics (C8-C16) that gauge the volume of each method based on the number of tokens extracted from the method's Abstract Syntax Tree (AST). This includes factors like the number of statements, parameters, and the Halstead metrics [17], derived from the count of operators and operands. Thirdly, assessed the complexity of a method by examining its control flow, encompassing metrics such as maximum nesting levels, number of paths, and cyclomatic complexity. Lastly, to ascertain the level of coupling between a method and other methods, we computed the Fan-in and Fan-out of the method. All of these aforementioned code metrics have been extensively employed in bug prediction practices.

Table 1.2: List of method-level code metrics

Index	Metric name
C1	Lines of Code
C2	Number of Comment lines
C3	Number of all lines
C4	Number of Blank lines
C5	Number of Declare lines
C6	Number of Executable lines
C7	Number of Parameters
C8	Number of Statements
C9	Number of Declare Statements
C10	Number of Executable Statements
C11	Halstead-Vocabulary
C12	Halstead-Length
C13	Halstead-Difficulty
C14	Halstead-Volume
C15	Halstead-Effort
C16	Halstead-Bugs
C17	Cyclomatic complexity
C18	Number of Path
C19	MaxNesting
C20	Fan-in
C21	Fan-out

Calculating Code Metrics and History Measures : To get code metrics and history measurements Fine-grained Code Metrics and History Measures Extractor (FCHE) is a tool developed by Ran Moa, Shaozhi Wei, Qiong Feng, and Zengyang Li [1] to automatically generate the aforementioned history measures and code metrics. (2) Given a project's revision history, first used an opensource tool, ChangeDistiller, to capture fine-grained information of each commit, such as the changed methods in each commit, the related change types (added, deleted, updated, etc.), the specific changed code pieces, and the date of a change, etc. Then, using FCHE to mine and parse all the fine-grained data, automatically construct the proposed historical measures.

Chapter 2

Literature Survey

To learn more about this work, I looked back on the studies and advancements the writers had made in the past. In this section, we outline the observations .

R. Mo, W. Su, Q. Feng, and Z. Li [1] determined the total number of bug-prone methods across all files as well as the specific number of bug-prone methods. And they found approaches that were prone to bugs.by mining a project's bug reports and revision history. Upon a commits,message includes a bug ID, thus they believe this commit was made for that bug.Fix for bugs. the files associated with the commit are then all labelled as bug-prone. Using the fine-grained change data for each file, examine in more detail the procedures in this file have been modified, and these techniques are deemed to be bug-prone methods.

Data Mining Static Code Attributes to Learn Defect Predictors Tim Menzies; Jeremy Greenwald; Art Frank [3] demonstrate that such arguments are unimportant since the method of constructing predictors from the qualities, as opposed to the specific attributes employed, is of utmost importance. Additionally, we demonstrate that, in contrast to earlier pessimism, such defect predictors are demonstrably effective and, when applied to the analysed data, produce predictors with a mean probability of detection of 71% and a mean false alarm rate of 25%. The predictors might be helpful for prioritising an inspection of code that is resource-bound.

Numerous studies have been done on fault-proneness as a quality element. Software engineers can design system evolutions by using software module fault-proneness predictions. The Raed Shatnawi [7] strategy may be jeopardised if the prediction models are biassed

or have poor forecast accuracy. The data imbalance, in which the majority of modules are faultless but the minority of modules is simply faulty, is one significant issue that might affect the prediction performance. In order to oversample the minority, Raed Shatnawi [7] proposes using the fault content (i.e., the number of defects in a module).Raed Shatnawi [7] then applied this technique to a sizable object-oriented system.

In practical applications, classification models can be created by periodically going through the steps of preprocessing data, creating models with various specialised algorithms, and evaluating the resulting models until a classification model of sufficient quality is produced. The model created in this way can be trusted to a certain extent because it has been tested before being used to make predictions on new data. Weka is an extremely potent machine learning tool that can perform all the operations mentioned above.[10]

Dan Ungureanu-Anghel, Paul Arseni-Ailoi, and Raul Robu [10] Utilising the Weka API, the suggested application will be able to rebuild the chosen classification model, store and load it in/from a binary file, and then make predictions using a user interface that is precisely suited to the utilised data set. The proposed application makes predicting the end beneficiary easier.

They demonstrate how to create a custom categorization application using the Java programming language and the Weka API. The Cleveland heart disease data set, which was collected from the UCI Machine Learning Repository, served as the foundation for the authors' custom application for the classification of patient data that may have heart disorders, which they used to validate the suggested method [10]

Software systems, according to Yasutaka Kamei and Emad Shihab [2], are becoming more and more crucial in our lives while also becoming more complex. The rising complexity of software systems makes it more challenging to guarantee their quality. As a result, a sizable portion of current research concentrates on how to prioritise software quality assurance efforts.

Software defect prediction, where predictions are made to determine where future flaws may arise, is one field of work that has been gaining more attention for more than 40 years. Since then, the field of software defect prediction has seen numerous studies and successes. However, there are still a lot of difficulties in the subject of software defect prediction.[2]

A variety of strategies were put out by Emanuel Giger, Marco D'Ambros, Martin

Pinzger, and Harald C. Gall [4] to create efficient bug prediction models that consider many facets of the software development process. Such models produced accurate predictions, directing programmers to the areas of their system where a high percentage of faults are likely to occur. However, the majority of those methods foresee file-level issues. This frequently forces developers to put out a lot of work to thoroughly check every method in a file until a flaw is found.

Chapter 3

Analysis

3.1 Dataset

In order to build, train, and test bug prediction models at the method level and evaluate which prediction model performs well on trying data at the method level bug prediction. For this work, we implement the models and evaluate the performance of the method-level bug prediction based on 18-large scale open source datasets the datasets are available at github.com.

The 18 datasets have different samples 42 attributes, as well as a final categorization of either "bug-prone" or "Not bug-prone".

Each dataset is described by a combination of the following attributes about the project: Code metrics (Lines of Code,Number of Comment lines,Number of all lines ,Number of Blank lines ,Number of Declare lines ,Number of Executable lines , Number of Parameters ,Number of Statements,Number of Declare Statements,Number of Executable Statements ,Halstead-Vocabulary,Halstead-Length ,Halstead-Difficulty ,Halstead-Volume ,Halstead-Effort ,Halstead-Bugs ,Cyclomatic complexity ,Number of Path ,MaxNesting ,Fan -in ,Fan-out)

History measures(Added LOC, Deleted LOC ,Changed LOC ,Number of Changes ,Number of Authors , Number of Modified Statements ,Number of Modified Expressions ,Number of Modified Comments ,Number of Modified Return type ,Number of Modified Parameters ,Number of Modified Prefix ,Added LOC/LOC ,Deleted LOC/LOC ,Added LOC/Deleted

LOC ,Changed LOC/Number of Changes ,Number of Modified Statements/ Number of Statements ,Number of Modified Expressions/ Number of Statements ,Number of Modified Comments/ LOC ,Number of Modified Parameters/ Number of Parameters)

3.2 Prediction model development

To assess the importance of our work for developing prediction models at the method level, we further applied five machine learning approaches to the problem of bug prediction. Random forest, Naive Bayes, Logistic, and Decision Tree support vector machine techniques, which are simple to use and frequently used in bug prediction and might produce a promising prediction at various granularity levels. Weka api integrated with Java, which contains a set of machine learning algorithms for data mining tasks, was used in this work to conduct the predictions.

3.3 Training and Validating Models

Dataset Loading and Preparation: Loading a dataset from a specified file path. It then ensures that the target class is properly set for classification. This is crucial as the algorithm needs to know which attribute it's trying to predict. The dataset is then divided into multiple subsets, or folds, to facilitate cross-validation.

3.3.1 Classifier Initialization

The chosen algorithm for capable of making predictions based on a series of conditions

3.3.2 Training Process

The classifier is trained using a portion of the dataset by using a train test split that is (80% for training and 20% for testing) or (70% for training and 30% for testing). This step is essential for the algorithm to learn the underlying patterns within the data. The goal is to develop a predictive model that can generalize well to unseen instances.

Experiment Parameters Sets essential parameters for the experiment. This includes the number of repetitions and the number of folds for cross-validation performed on training data . These parameters govern how the model is trained, tested, and evaluated.

Cross-Validation Then perform cross-validation. This process involves training and evaluating the model multiple times. Each iteration contributes to a more comprehensive assessment of the model's performance, ensuring it can handle a variety of scenarios.

Epochs and folds are fundamental concepts in machine learning, particularly in the context of training and evaluating models.

3.3.3 Epochs:

Definition:

An epoch is one complete pass through the entire training dataset. In other words, it's the number of times the learning algorithm sees the entire dataset.

Significance:

During each epoch, the model's parameters (like weights in a neural network) are updated in an attempt to minimize the chosen loss function. This iterative process helps the model learn patterns in the data and improve its performance.

Usage:

Too Few Epochs: If the model is trained for too few epochs, it might not have seen enough of the data to learn meaningful patterns. The model may underfit, performing poorly even on the training data.

Too Many Epochs: On the other hand, if the model is trained for too many epochs, it might start to memorize the training data, leading to overfitting. This means it becomes too specialized to the training data and performs poorly on unseen data.

3.3.4 Cross-Validation Folds:

Definition:

Cross-validation is a technique used to assess the performance of a machine learning model. Folds refer to the subdivisions of the dataset created during this process.

Significance:

In k-fold cross-validation, the original dataset is divided into k equal-sized subsets (or folds). The model is then trained on k-1 of these folds and tested on the remaining one. This process is repeated k times, each time using a different fold as the test set.

Usage:

Evaluation: Cross-validation provides a robust estimate of a model's performance because it ensures that every data point is used for testing exactly once. This reduces the possibility of overfitting to a particular subset of the data.

Hyperparameter Tuning: It's especially useful for hyperparameter tuning. By training and evaluating the model on different subsets of the data, cross-validation helps ensure that the performance metrics are representative of the model's ability to generalize to unseen data.

Data Scarcity: In cases where data is limited, cross-validation can make the most out of the available data by ensuring that every data point is used for both training and testing at some point.

Remember, choosing the right number of epochs and the appropriate type of cross-validation are crucial aspects of building robust and generalizable machine learning models. It's a balance between training long enough to learn meaningful patterns, but not so long that the model starts to memorize the training data. Similarly, cross-validation helps in getting a reliable estimate of a model's performance and aids in making informed decisions about its hyperparameters.

3.4 Performance metric

To calculate the performance of the each model we use the performance metrics like Accuracy, precision, F-measure, AUC, MCC and G-mean.

3.4.1 Performance Metric Collection

After each iteration, various performance metrics are calculated. These metrics are critical for evaluating the model's effectiveness. They include precision, F1-score, Matthews correlation coefficient, area under the ROC curve, and geometric mean. These metrics collectively provide a detailed understanding of the model's predictive power.

3.4.2 Epoch-wise Analysis

For each epoch, the code computes the average mean and standard deviation of the evaluation metrics. This analysis provides insights into how the model's performance evolves over time. It helps to identify trends or potential areas for improvement during the training process.Once all epochs are completed, the code presents the final evaluation. It showcases the overall average mean and standard deviation scores across all epochs. This summary serves as a comprehensive assessment of the model's performance, considering the entire training process.

3.4.3 Testing Set Evaluation

Finally, the trained model is evaluated using a separate testing set. This is crucial for assessing how well the model generalizes to new, unseen data. The code computes various metrics, including accuracy, precision, F1-score, and others, to provide a final evaluation of the model's predictive capabilities.

3.5 prediction models

To assess the importance of our work for developing prediction models at the method level. Random forest, Naive Bayes, Logistic, and Decision Tree support vector machine techniques, which are simple to use and frequently used in bug prediction might produce a promising prediction at various granularity levels. Here is the explanation of machine learning models and their overview key components of each model.

3.5.1 Random Forest

Random Forest is an ensemble learning algorithm used for both classification and regression tasks. It's particularly powerful and widely used in machine learning. Here's a comprehensive explanation of the Random Forest algorithm:

Overview: Random Forest is a type of ensemble learning method that combines the predictions of multiple individual decision trees to improve overall predictive accuracy and control overfitting.

Key Components: Decision Trees: Random Forest is built upon the foundation of decision trees. Decision trees are models that make decisions based on asking a series of questions about the features of the data. Each question leads to a new branch in the tree, eventually resulting in a prediction.

Bootstrapping (Bagging): Random Forest employs a technique known as bootstrapping. It involves creating multiple random subsets of the training data (with replacement). Each of these subsets is used to train an individual decision tree. This process creates diversity among the trees.

Feature Selection: In addition to using different subsets of the data, Random Forest also uses a random subset of features for each split when constructing the decision trees. This adds another layer of randomness and diversity, making the individual trees more distinct.

Voting (Classification) or Averaging (Regression): For classification tasks, the predictions of the individual trees are combined through a voting process. The class that receives the most votes is chosen as the final prediction. For regression tasks, the predictions are averaged.

How Random Forest Works:

Training Phase:

Random Forest begins by creating a predefined number of decision trees. Each tree is trained using a different subset of the data (sampled with replacement) and a random subset of features for each split. During tree construction, the algorithm recursively chooses the best feature and value to split the data based on certain criteria (e.g., Gini impurity

for classification, mean squared error for regression).

Prediction Phase:

When making predictions, new data is passed down each tree in the forest. Each tree provides a prediction. For classification, the forest's prediction is the class with the most votes from all the trees. For regression, it's the average of all tree predictions.

Advantages of Random Forest:

High Accuracy: Random Forest is known for producing highly accurate predictions, often outperforming individual decision trees.

Reduced Overfitting: The ensemble nature of Random Forest helps reduce overfitting, a common issue with individual decision trees.

Feature Importance: Random Forest provides a measure of feature importance, which helps in understanding which features are most influential in making predictions.

Robustness to Outliers and Noisy Data: Random Forest is less susceptible to outliers and noisy data compared to other algorithms.

Handles Large Datasets: It can efficiently handle large datasets with many features and instances.

Parallel Processing: The training of individual trees in a Random Forest can be done in parallel, making it computationally efficient.

3.5.2 Logistic

Logistic Regression is a fundamental machine learning algorithm used for binary classification tasks, where the goal is to predict the probability of an instance belonging to one of two classes. Despite its name, logistic regression is actually a statistical model for regression analysis, but it's commonly used for classification purposes.

Overview: Logistic Regression models the relationship between the dependent binary variable and one or more independent variables. It does this by estimating probabilities using the logistic function, also known as the sigmoid function. The sigmoid function maps any real-valued number to a value between 0 and 1, making it suitable for estimating probabilities.

Key Components:

Logistic Function (Sigmoid): The logistic function is defined as $P(Y=1)=1/\ 1+e^{\ z}$

, where z is a linear combination of the features and their associated weights.

Parameters (Weights and Bias): Logistic Regression assigns a weight to each feature, indicating its influence on the prediction. Additionally, there's a bias term that adjusts the prediction.

Training (Maximum Likelihood Estimation): During training, the model learns the optimal weights and bias by maximizing the likelihood function. This involves finding the parameters that make the observed data most probable.

How Logistic Regression Works:

Training Phase:

Logistic Regression starts by assigning random weights and a bias term to each feature. It then calculates the predicted probabilities using the logistic function. The model's predictions are compared to the actual labels, and the error (the difference between the predicted and actual values) is computed.

Cost Function (Cross-Entropy Loss):

The cost function measures how well the model is performing. In logistic regression, the cost function is typically the cross-entropy loss, which penalizes large errors more heavily.

Gradient Descent:

The algorithm then uses gradient descent to adjust the weights and bias, aiming to minimize the cost function. This involves iteratively updating the parameters in the direction that decreases the cost.

Prediction Phase:

Once the model is trained, it can make predictions on new data. The logistic function is applied to the linear combination of features and weights to get the probability of belonging to the positive class.

Advantages of Logistic Regression:

Interpretability:

Logistic Regression provides interpretable coefficients, making it easy to understand the impact of each feature on the prediction.

Efficiency:

It is computationally efficient and can handle large datasets with a large number of features.

Probabilistic Output:

The model produces probabilities, allowing for a nuanced understanding of the confidence in the predictions.

No Assumption of Linearity:

While the model is called "regression," it can capture non-linear relationships through feature engineering.

3.5.3 Decision tree

The Decision Tree algorithm is a popular machine learning technique used for both classification and regression tasks. It works by creating a tree-like structure where each internal node represents a feature, each branch represents a decision rule, and each leaf node represents an outcome. The algorithm aims to split the data in a way that maximizes the purity of the resulting subsets.

Overview:

A Decision Tree is built iteratively by making decisions at each node based on the values of features in the dataset. It's a top-down approach, starting from the root node and recursively partitioning the data until it reaches a stopping criterion, such as a pure subset or a maximum depth.

Key Components:

Root Node:

The topmost node of the tree that represents the entire dataset. It is associated with the feature that best splits the data.

Internal Nodes:

These nodes represent features and corresponding decision rules. They split the data into subsets based on the feature's values.

Branches:

Branches emanating from internal nodes represent the possible values of the associated feature.

Leaf Nodes:

Leaf nodes are the terminal nodes that provide the final prediction or output. They do not have any further branches.

Decision Rules:

At each internal node, a decision rule is applied based on the feature's value. For example, "if feature A 5, go left; else, go right".

How Decision Trees Work:

Splitting Criteria:

The algorithm starts with the root node, which contains the entire dataset. It evaluates different features to find the one that best splits the data into more homogeneous subsets.

Information Gain or Gini Index:

The algorithm uses metrics like Information Gain (for classification) or Gini Index to measure the impurity or disorder of a dataset. It selects the feature that results in the highest reduction of impurity after the split.

Recursive Partitioning:

Once a feature and split point are chosen, the data is divided into subsets based on the values of that feature. The process is repeated recursively for each subset until a stopping criterion is met.

Stopping Criteria:

The algorithm stops partitioning under certain conditions, such as reaching a maximum depth, achieving pure subsets, or when further splits do not significantly improve purity.

Prediction:

When a new instance is fed into the tree, it traverses the branches based on the values of its features until it reaches a leaf node. The output of that leaf node is the prediction.

Advantages of Decision Trees:

Interpretability:

Decision Trees are highly interpretable and can be visualized, allowing for easy understanding of the decision-making process.

Handling Non-linearity:

They can model complex relationships and interactions between features without assuming linearity.

Feature Importance:

Decision Trees can provide insights into which features are most important for making predictions.

Handling Categorical Data:

They can naturally handle categorical features without the need for one-hot encoding.

3.5.4 Bayes

The Bayes algorithm, specifically referring to the Naive Bayes classifier, is a simple yet effective machine learning algorithm used primarily for classification tasks. It is based on Bayes' theorem, which describes the probability of an event based on prior knowledge of conditions that might be related to the event. Naive Bayes is considered "naive" because it makes a strong assumption of independence between features, which may not always hold in real-world scenarios.

How Naive Bayes Works:

Bayes' Theorem:

The algorithm starts with Bayes' theorem, which calculates the conditional probability of an event given prior knowledge. In the context of classification, it's used to calculate the probability of a class given a set of features.

Feature Independence Assumption:

Naive Bayes assumes that the features used for classification are conditionally independent given the class label. This means that the presence or absence of a particular feature does not influence the presence or absence of any other feature.

Training:

During the training phase, the algorithm learns the likelihood probabilities and prior probabilities from the training data. It calculates the likelihood of each feature value given each class and the prior probability of each class. **Classification:**

When given a new instance for classification, the algorithm calculates the posterior probability of each class given the feature values using Bayes' theorem. It selects the class with

the highest posterior probability as the predicted class.

Handling Continuous and Categorical Variables:

Depending on the type of features, different distributions (Gaussian, multinomial, etc.) can be used to calculate likelihood probabilities.

Advantages of Naive Bayes:

Efficiency:

Naive Bayes is computationally efficient, making it suitable for large datasets.

Simple and Easy to Implement:

It's straightforward to understand and implement, making it a good choice for a quick baseline classification model.

Effective with High-Dimensional Data:

It performs well even when the number of features is large compared to the number of instances.

Handles Missing Data:

It can handle missing feature values without requiring imputation techniques.

3.5.5 Support vector machine

Support Vector Machine (SVM) is a powerful supervised learning algorithm used for both classification and regression tasks. It works by finding an optimal hyperplane in a high-dimensional space that best separates classes or predicts continuous values. The Sequential Minimal Optimization (SMO) algorithm is a specific technique used to efficiently train SVMs.

SVM Overview: SVM aims to find a hyperplane that maximizes the margin between classes. In a binary classification scenario, the hyperplane is the decision boundary that best separates the classes. The support vectors are the data points closest to the decision boundary. SVMs are especially effective in cases where the data is not linearly separable by using techniques like the kernel trick to map data into higher-dimensional spaces.

Key Concepts:

Margin:

The margin is the distance between the hyperplane and the nearest data points from

each class. SVM aims to maximize this margin, providing better generalization to unseen data.

Kernel Trick:

SVM can use different kernel functions (e.g., linear, polynomial, radial basis function) to transform data into higher-dimensional spaces where classes may become separable.

Support Vectors:

Support vectors are the data points closest to the decision boundary. They play a crucial role in defining the decision boundary.

Soft Margin vs. Hard Margin:

SVM can be implemented with a soft margin to allow for some misclassification, which is crucial in cases where the data is not perfectly separable.

Advantages of SVM : Effective in High-Dimensional Spaces:

SVM and SMO are well-suited for datasets with a large number of features.

Robust to Overfitting:

SVM's margin maximization helps in generalizing well to unseen data.

Kernel Trick:

The ability to use different kernels allows SVM to handle non-linearly separable data.

Efficient Training:

SMO efficiently solves the optimization problem, making SVM training feasible even on large datasets.

3.6 Evaluation Measures

Building an effective learning model involves many steps, including model evaluation. "Accuracy" will be the most often used classification evaluation metric. One of the often used evaluation measures for classification problems is accuracy, this is determined as the total properly predicted values for a dataset divided by the total anticipated values. When the target class is balanced, accuracy is useful; nevertheless, imbalanced classes do not make for a suitable decision.

It is crucial to evaluate your model using a variety of measures. This is because a model could perform well when utilising one evaluation metric's measurement, but poorly while using a different measurement from the same evaluation metric. In order to make sure that your model is functioning properly and optimally, evaluation metrics are essential.

The performance of the classification model is evaluated using counts of test records that it correctly and incorrectly predicted. A table called a confusion matrix is used to assess how effectively a classification system works. A confusion matrix shows and summarises a classification algorithm's performance. The confusion matrix gives a more insightful picture of a predictive model's performance, showing which classes are predicted correctly and incorrectly as well as the types of errors that are being made. To give an example, the confusion matrix table below clearly shows how the four different combinations of predicted and actual values (TP, FP, FN, and TN) are calculated.

Prediction outcome

Actualvalue		**p**	**n**	**total**
	p′	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

condition positive (P) : the number of real positive cases in the data

condition negative (N) : the number of real negative cases in the data

true positive (TP):

A test result that correctly indicates the presence of a condition or characteristic

true negative (TN) :

A test result that correctly indicates the absence of a condition or characteristic

false positive (FP) :

A test result which wrongly indicates that a particular condition or attribute is present

false negative (FN) :

A test result which wrongly indicates that a particular condition or attribute is absent

The following is a brief illustration of the important evaluation metrics:

3.6.1 Accuracy

Accuracy is defined as the proportion of all the predictions that were correct. The final total number of accurate predictions the classifier made is taken into consideration.

Accuracy is defined by the below equation

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

3.6.2 Precision

The percentage of positive values out of all projected positive cases is known as precision or the positive forecasted value. Alternatively, accuracy is the proportion of accurately identified positive values that are present.

Precision mathematically defined as

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

3.6.3 Recall

The percentage of positive values out of all genuine positive instances, that is the share of actual positive occurrences that are correctly identified—is known as sensitivity, recall, or the TP rate (TPR).

It is the TP rate that is measured by equation

$$Sensitivity = \frac{TP}{TP + FN} \tag{3.3}$$

A higher sensitivity number would indicate a higher value of true positive andlower false negative. Lower sensitivity would translate to lower true positive and larger false negative values. Generally, models with high sensitivity will be desired.

3.6.4 Specificity

Specificity offers the percentage of negative values out of all actual negative cases. In other words, it measures the proportion of accurately identified genuine negative cases. The FP rate is given by equation

$$Specificity = \frac{TN}{TP + FN} \tag{3.4}$$

A higher true negative value and a lower rate of false positives would result from greater specificity. Lower specificity would translate to larger false positive and lower true negative values.

3.6.5 F-measure (F1-score)

The F-measure is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall. The F1-score ranges from 0 to 1, with higher values indicating better model performance.

$$Fmeasure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.5}$$

The harmonic mean gives more weight to lower values. This means that for the F1-score to be high, both precision and recall must be high.

The F-measure is particularly useful when dealing with imbalanced datasets, where one class significantly outnumbers the other. In such cases, a high accuracy can be misleading, as the model may simply be predicting the majority class.

The F-measure provides a more balanced view of a model's performance, taking into account both false positives and false negatives.

3.6.6 Geometric Mean

The G-mean is the square root of the product of sensitivity and specificity. Mathematically, it is expressed as:

$$Gmean = \sqrt{Sensitivity * Specificity} \tag{3.6}$$

The G-mean is useful because it gives equal weight to sensitivity and specificity. This is particularly important in situations where you want to ensure a balance between correctly identifying positive cases (sensitivity) and correctly identifying negative cases (specificity).
When to Use G-Mean Imbalanced Datasets:

In situations where the classes are imbalanced (one class significantly outnumbers the other), accuracy might not be the best metric. The G-mean can provide a more balanced assessment of model performance.

Cost-Sensitive Applications:

In scenarios where the costs associated with false positives and false negatives are different (e.g., medical testing, fraud detection), optimizing for sensitivity or specificity alone might not be sufficient. The G-mean can help find a compromise.

Anomaly Detection:

G-mean is commonly used in anomaly detection tasks where correctly identifying rare events (positives) while avoiding false alarms (negatives) is crucial.

Interpretation of G-Mean: A G-mean of 1 indicates perfect balance between sensitivity and specificity. A G-mean less than 1 indicates an imbalance in favor of one of the metrics. A higher G-mean implies a better balance.

3.6.7 AUC

AUC is one of the evaluation metrics for binary classification problems. AUC is the area under the receiver operator characteristic (ROC) curve. Plotting the TPR versus the FPR at various threshold values, in general, separates the "signal" from the "noise." The Area Under the Curve (AUC), used as a summary of the ROC curve, quantifies a classifier's ability to distinguish between classes.

The greater the AUC, the better the model does at separating the positive and negative groups.

- When AUC = 1, then all Positive and Negative class points are appropriately distinguished between by the classifier in a faultless manner. The classifier would be predicting all Negatives as Positives and all Positives as Negatives, however, if the AUC had been 0.
- When $0.5 < AUC < 1$, The likelihood that the classifier will be able to tell the difference between the positive class values and the negative class values is high. This is the case because the classifier can detect more True positives and True negatives than False positives and False negatives.
- When AUC = 0.5, If this is the case, the classifier cannot tell the difference between Positive and Negative class points. This indicates that the classifier is either predicting a random class or a fixed class for each data point.

3.6.8 Information Gain:

bug prediction for method-level code, attribute selection is a critical step to identify the most important metrics. One widely used technique for attribute selection is Information Gain.

Definition:

Information Gain is a measure of the reduction in uncertainty or entropy achieved by partitioning a dataset based on a specific attribute. In the context of bug prediction, it helps identify which metrics provide the most valuable information for predicting bugs.

Steps to Calculate Information Gain:

Calculate Initial Entropy (H(S)): Measure of disorder or impurity in the dataset before any partitioning.

Partition the Data: Split the dataset based on the values of the attribute being evaluated.

Calculate Weighted Average Entropy for Each Partition: This is the entropy of each partition multiplied by the fraction of data points in that partition.

Calculate Information Gain: It's the initial entropy minus the weighted average entropy after the partitioning. It represents the reduction in uncertainty achieved by considering that attribute.

Repeat for All Attributes: Calculate Information Gain for each available attribute and select the one with the highest gain.

Importance in Bug Prediction:

In the context of bug prediction, attributes can be various metrics related to code quality, complexity, history, etc. The goal is to find which metrics are most informative for predicting bugs.

Attributes with higher Information Gain are considered more important because they contribute the most to reducing the uncertainty in bug prediction. These are the metrics that provide the most discriminating power in distinguishing between buggy and non-buggy code.

Chapter 4

Methodology

4.1 Problem Statement

In earlier studies, there has been a substantial amount of research focused on the prediction of software bugs at various levels of granularity, including file level, package level, and module level. Surprisingly, there has been limited emphasis on predicting bugs at the method level. This project begins by compiling 18 open-source datasets that have conducted bug predictions at the method level using code metrics and historical data.

Then evaluates the effectiveness of different types of prediction models. Additionally, the project conducts in-depth studies on each metric's ability to forecast, using the findings to further refine the prediction models. The study encompasses three main aspects: the Efficiency of Method-Level Predictions, Model Performance, and Metric Variance. To facilitate the analysis, this work makes use of the Weka tool, which offers robust analytical capabilities. The integration of a Java plugin with Weka enables seamless model development and evaluation. Furthermore, the incorporation of the G-mean metric adds an extra dimension to the assessment of prediction model performance.

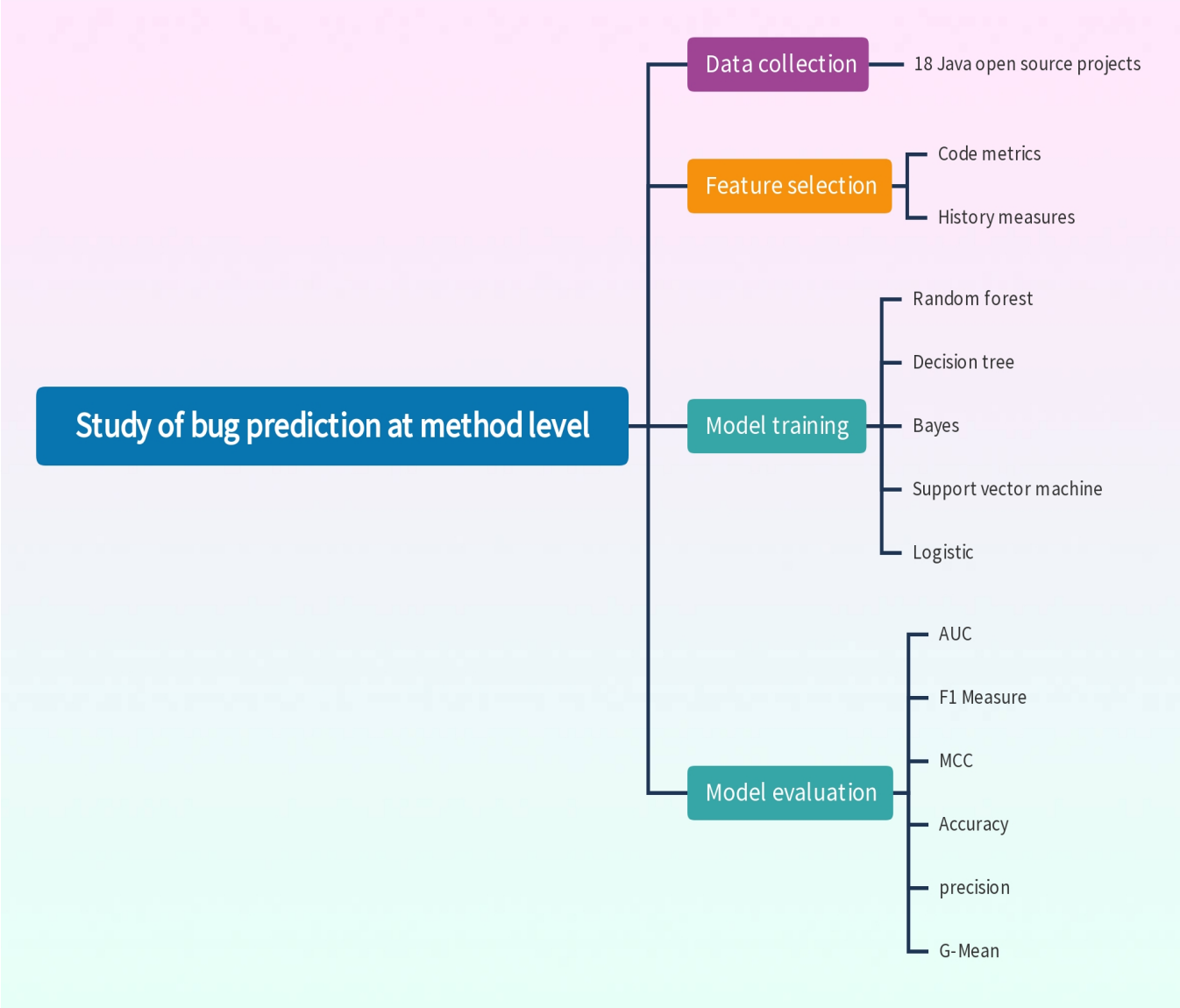


Figure 4.1: Model architecture

4.2 Model architecture

4.3 Integrating weka api with Java

Weka library and Java API for machine learning tasks within Eclipse:
Eclipse is a popular Integrated Development Environment (IDE) for Java, and it offers several advantages when it comes to developing machine learning applications using the Weka library or any other Java-based machine learning framework:
Java Compatibility, Code Assistance, Debugging, Project Organization etc. Here iam using eclipse for this purpopse the step-by-step explanation of how to use Weka and the Java

API in Eclipse.

Step 1: Install Eclipse First install Eclipse IDE for Java Developers from the official website.

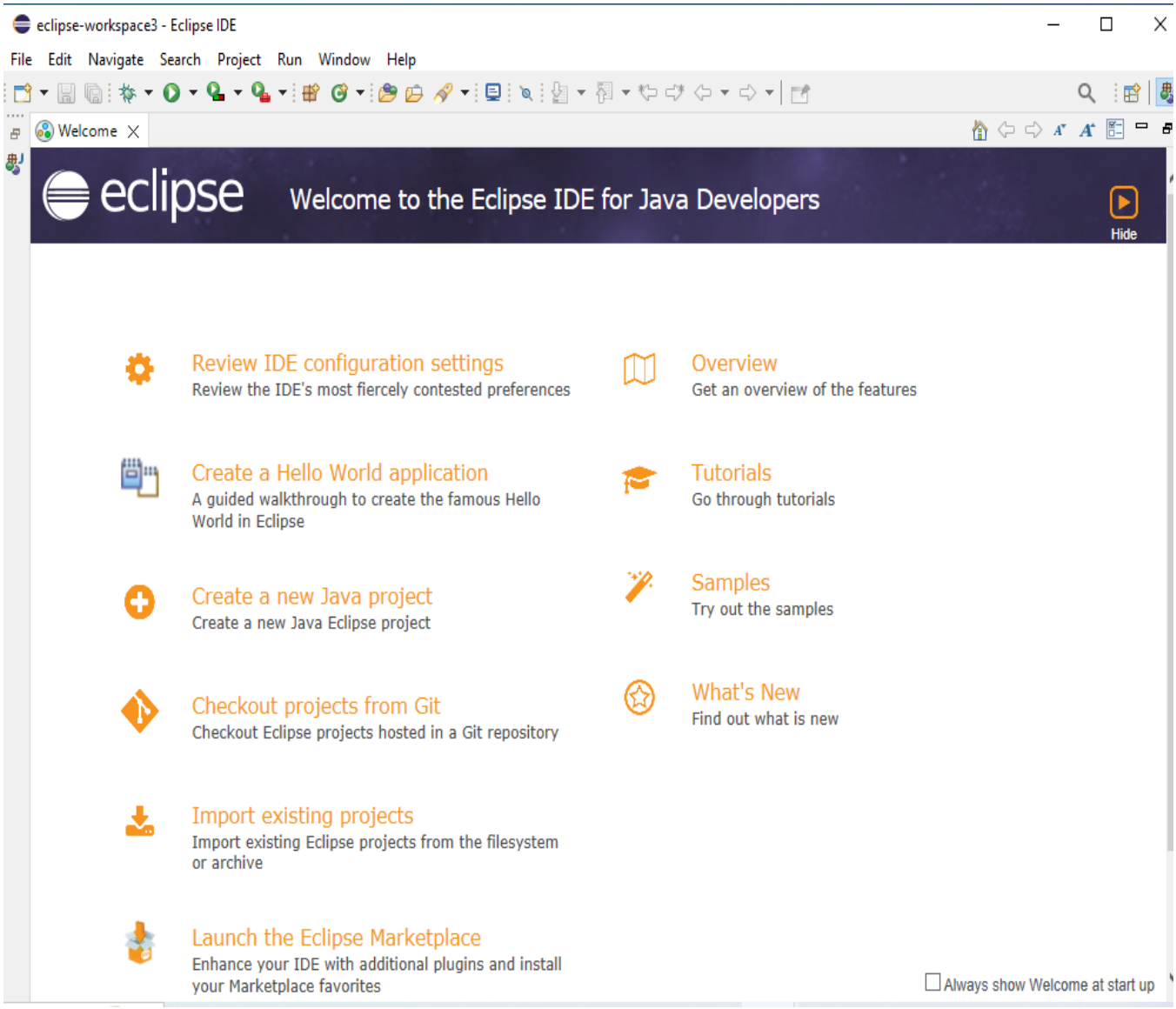


Figure 4.2: Eclipse installation

Step 2 : Create a New Java Project Open Eclipse, go to "File" "New" "Java Project" and give a project name and click "Finish."

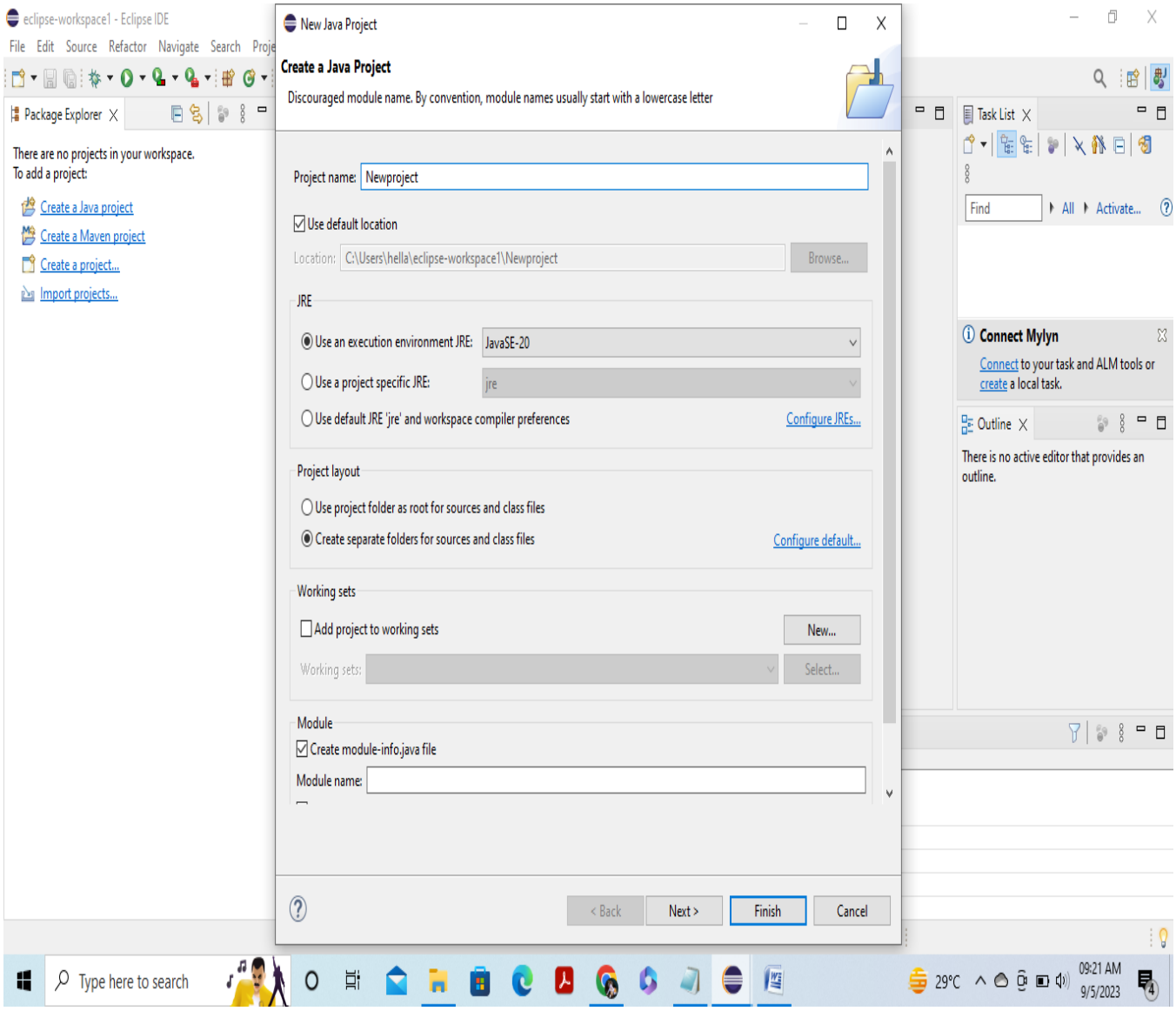


Figure 4.3: New project setup

Step 3: Download Weka Library

Go to the Weka website (https://www.cs.waikato.ac.nz/ml/weka/) and download the latest version of the Weka library. It usually comes in a compressed ZIP file.

Step 4: Add Weka JAR to Your Project

1) Unzip the downloaded Weka ZIP file to a location on your computer.

2) In Eclipse, right-click on your project in the "Package Explorer" panel on the left side.

3) Select "Build Path" ¿ "Configure Build Path."

4) In the "Libraries" tab, click the "Add External JARs" button.

5) Navigate to the location where you extracted the Weka ZIP file and select the Weka

JAR file (usually named something like "weka.jar").

Click "Open" and then "Apply and Close" to add the Weka library to your project.

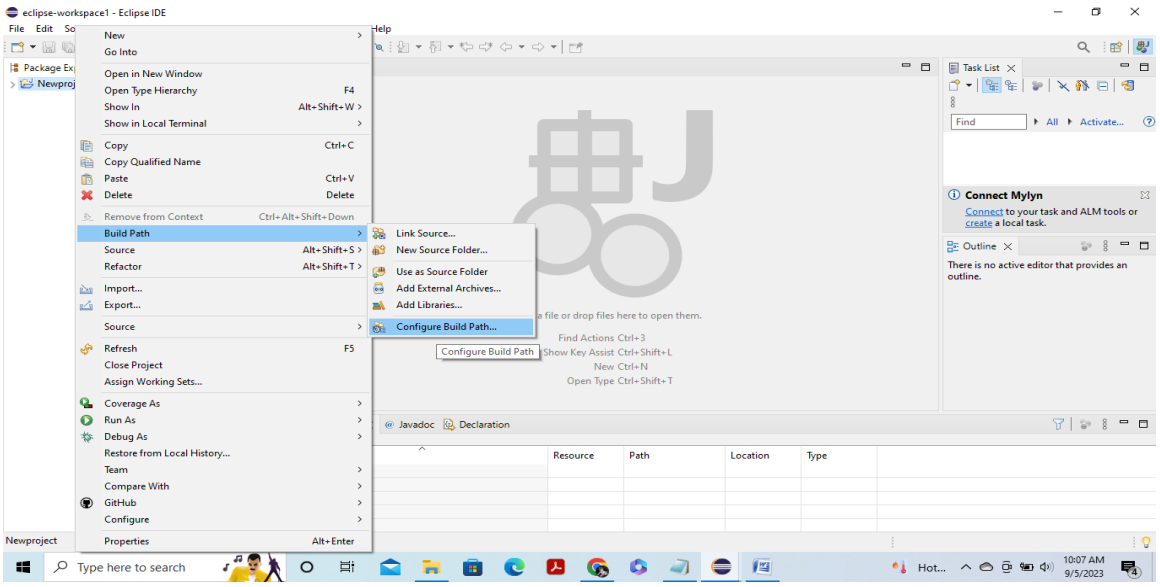
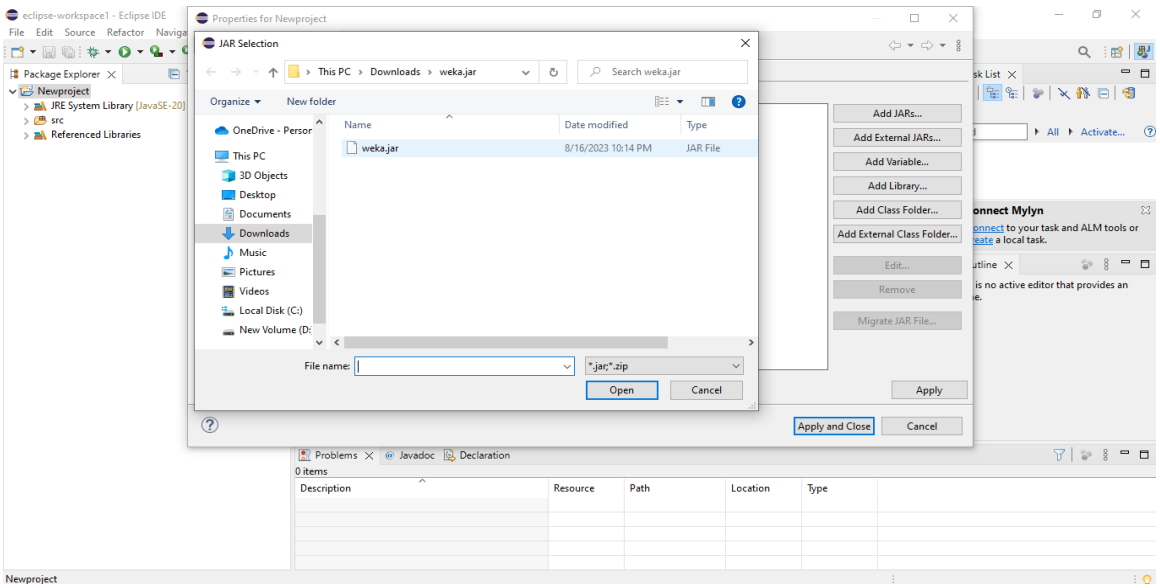


Figure 4.4: Add Weka Jar file



Step 5: Create a New Java Class

1) In the "Package Explorer," right-click on your project and choose "New" , "Class."

2) Click "Finish" to create the class.

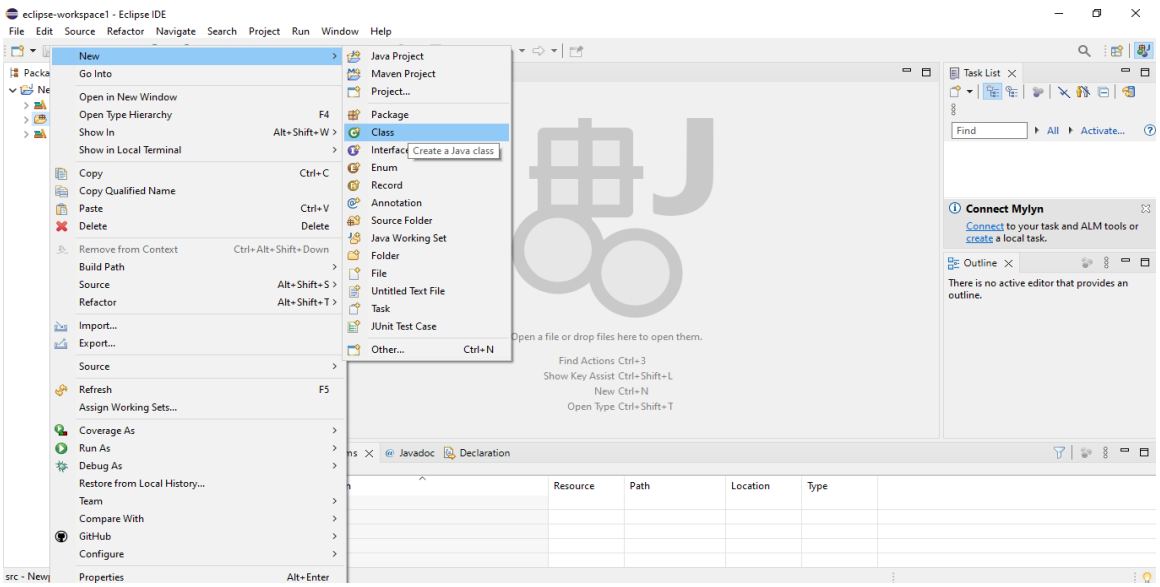


Figure 4.5: creating class

Step 6: Import Weka Classes

At the top of Java class file, we need to import Weka classes. Depending on specific task, we may need different imports, but here iam importing.

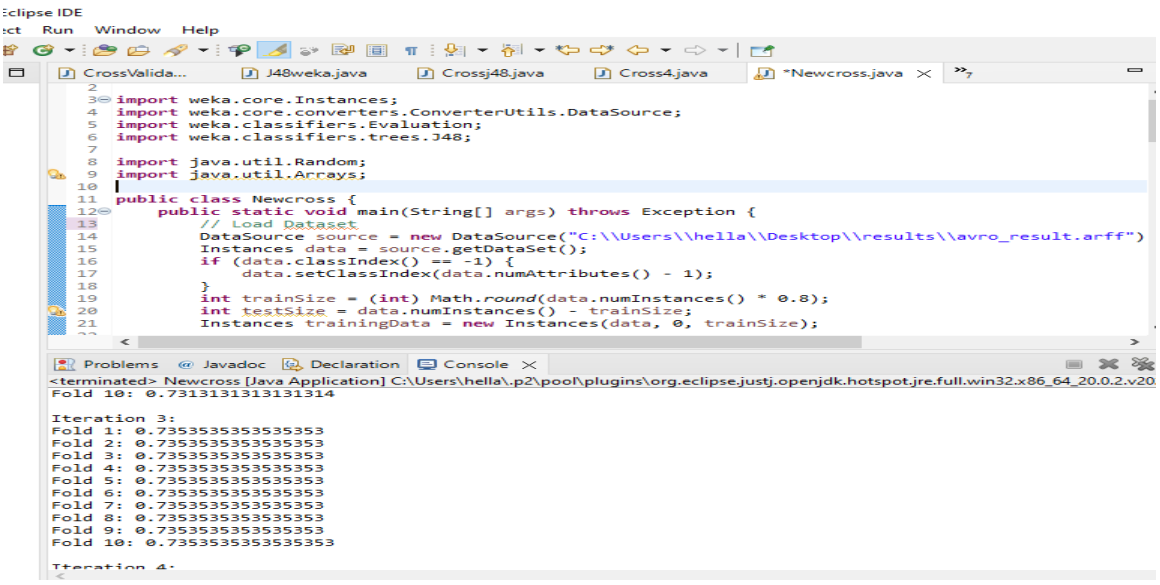


Figure 4.6: Import Weka Classes

Table 4.1: Hyper parameters

Technique	Parameters
Random forest	Max-depth=None , n-estimators=100
Support vector machine	Batchsize=100 ,c=1 .0,v=-1
Logistic regression	Batchsize=100 ,debug=False ,Maxlts=-1
Decision tree	Batchsize=100 ,Minnum=2 ,numfolds=3
Bayes	Batchsize=100 , debug=False ,use kernel estimator=False

4.3.1 Parameter tuning

Random forest :

Max Depth (max_depth=None):
In a decision tree, the depth represents the maximum number of levels it can have. The deeper the tree, the more complex the model can be, potentially allowing it to capture more intricate relationships in the data.

When max_depth=None, it means that there is no restriction on the depth of individual trees in the Random Forest. This can lead to very deep trees if the algorithm has enough data to train on.

Number of Estimators (n_estimators=100):
In a Random Forest, the model consists of multiple decision trees (known as "estimators"). The final prediction is made by aggregating the predictions of these individual trees.

n_estimators=100 indicates that the Random Forest will use 100 decision trees to make its predictions. More trees generally lead to better performance, up to a point, as it reduces overfitting.

Support Vector Machine (SVM) algorithm

Batch Size (batch_size=100): In the context of training a Support Vector Machine, the batch size determines how many training samples are used in each iteration of the optimization algorithm (usually gradient descent).

Specifically, for SVMs with a stochastic gradient descent (SGD) optimization strategy, the batch size is the number of training samples randomly selected for each iteration. A larger batch size may lead to slower convergence but can provide a more stable estimate of the gradient.

C (C=1.0): The C parameter in an SVM controls the trade-off between maximizing the margin (i.e., finding a larger margin hyperplane) and minimizing the classification error on the training data.

A smaller C encourages a larger margin, potentially allowing for a simpler decision boundary that may generalize better to unseen data but could misclassify some training examples. A larger C places more emphasis on classifying all training examples correctly, potentially leading to a more complex decision boundary. **v (v=-1):** The v parameter is typically not a standard parameter in SVMs. It's possible that this parameter might be specific to a particular implementation or variant of the SVM algorithm, which might not be part of the standard SVM formulation.

Logistic

Batch Size (batch_size=100): In the context of iterative algorithms, such as those used in machine learning, the batch size determines how many samples are processed in one iteration or epoch.

Debug Mode (debug=False): This parameter typically controls whether the algorithm runs in a mode optimized for debugging or in a regular operational mode.

When debug=True, it may enable additional logging, extra checks, or provide more detailed information about the algorithm's execution, which can be useful for troubleshooting. When set to False, the algorithm runs without these additional debugging features.

Maximum Iterations (maxIts=-1): This parameter represents the maximum number of iterations or epochs the algorithm will run before stopping. A negative value, like -1,

often indicates no specific limit is set, and the algorithm will continue until a stopping criterion is met.

Decision tree

Batch Size (batch_size=100): In the context of decision trees, the concept of "batch size" is less common compared to iterative optimization algorithms like gradient descent. For decision trees, data is usually not processed in batches; instead, the entire dataset is used at once.

Minimum Number of Instances (min_num=2): This parameter sets the minimum number of data points that must be present in a leaf node of the decision tree. If a node has fewer than this number of instances, it will not be further split, preventing the tree from becoming too complex or overfitting to noisy data.

Number of Folds for Cross-Validation (num_folds=3): This parameter is related to the process of cross-validation, a technique used to evaluate the performance of a machine learning model. In k-fold cross-validation, the dataset is divided into 'k' subsets (or folds). The model is trained 'k' times, each time using a different fold as the test set and the remaining folds as the training set.

Setting num_folds=3 indicates that you're using a 3-fold cross-validation strategy to assess the performance of your decision tree.

Naive Bayes

Batch Size (batch_size=100): Bayes-based algorithms, batch size is not a typical parameter. Batch size is more commonly associated with iterative optimization algorithms like gradient descent.

Debug Mode (debug=False): This parameter typically controls whether the algorithm runs in a mode optimized for debugging or in a regular operational mode.

When debug=True, it may enable additional logging, extra checks, or provide more detailed information about the algorithm's execution, which can be useful for troubleshooting. When set to False, the algorithm runs without these additional debugging features.

Chapter 5

Result and Conclusion

5.1 Results

The evaluation of method-level bug prediction models, utilizing either code metrics, history measures, or a combination of both, demonstrates that the combined approach yields the highest performance. In comparison to predictions solely based on historical data, the integration of code metrics alongside historical measures leads to a notable enhancement in prediction accuracy. For instance, considering the Avro dataset as an illustrative example, when exclusively employing code metrics or history measures, the AUC values for prediction models stand at 0.730 ,0.780 and 0.808 respectively. Moreover, in contrast to the Type1 model, the Type2 model showcases superior results, with further optimization achieved through the combined Type1, Type2, and Type3 model configurations. This underscores the effectiveness of leveraging both code metrics and historical measures in enhancing the predictive capabilities of the models.

- **TYPE1 :** Prediction models based on code metrics;
- **TYPE2 :** Prediction models based on history measures;
- **TYPE3 :** Prediction models based on both code metrics and history measures

Table 5.1: F1,Mcc,Auc values of prediction models using random forest

project	Type1			Type2			Type3		
	F1	MCC	AUC	F1	MCC	AUC	F1	MCC	AUC
ActiveMQ	0.663	0.313	0.718	0.751	0.483	0.845	0.784	0.555	0.876
Avro	0.632	0.243	0.730	0.709	0.420	0.780	0.724	0.432	0.808
Calcite	0.559	0.144	0.599	0.617	0.262	0.688	0.644	0.297	0.710
Camel	0.619	0.235	0.650	0.711	0.429	0.797	0.722	0.446	0.811
Cassandra	0.613	0.224	0.665	0.700	0.394	0.791	0.742	0.471	0.835
CXF	0.752	0.210	0.711	0.824	0.415	0.865	0.836	0.463	0.885
Drill	0.623	0.245	0.702	0.680	0.391	0.769	0.706	0.427	0.802
Flink	0.629	0.068	0.679	0.687	0.235	0.748	0.714	0.298	0.779
Flume	0.626	0.243	0.649	0.661	0.311	0.782	0.693	0.368	0.800
Hbase	0.635	0.284	0.685	0.698	0.401	0.781	0.711	0.418	0.796
Ignite	0.811	0.055	0.708	0.818	0.145	0.875	0.828	0.227	0.881
Kafka	0.649	0.096	0.678	0.677	0.187	0.769	0.710	0.268	0.818
Maven	0.696	0.073	0.721	0.761	0.310	0.868	0.778	0.347	0.886
Nutch	0.590	0.158	0.690	0.664	0.331	0.766	0.712	0.419	0.806
PDFBox	0.683	0.3628	0.773	0.724	0.432	0.786	0.762	0.528	0.858
Struts	0.600	0.255	0.677	0.733	0.458	0.825	0.760	0.502	0.849
Wicket	0.616	0.125	0.698	0.731	0.387	0.852	0.763	0.439	0.869
Zookeeper	0.639	0.276	0.682	0.708	0.429	0.788	0.695	0.389	0.793

Table 5.2: Analysis based on random forest model

project	Accuracy	MCC	precision	AUC	F-mesaure	G-mean
ActiveMQ	0.7862	0.5563	0.7798	08132	0.7219	0.72357
Avro	0.7562	0.5260	0.7564	0.7957	0.7642	0.7610
Calcite	0.7291	0.4274	0.7352	0.7858	0.7143	0.6899
Camel	0.7969	0.4781	0.7689	0.7267	0.7408	0.7805
Cassandra	0.7294	0.4590	0.7295	0.8242	0.7294	0.7295
CXF	0.8698	0.3313	0.8476	0.8555	0.8529	0.5776
Drill	0.7918	0.5847	0.7928	0.8799	0.7916	0.7919
Flink	0.7807	0.3405	0.7737	0.7716	0.7378	0.5852
Flume	0.6626	0.3202	0.66071	0.6868	0.6698	0.6601
Hbase	0.7624	0.4720	0.7621	0.0.8330	0.7660	0.7649
Ignite	0.8750	0.1238	0.8906	0.8830	0.8189	0.6363
Kafka	0.7747	0.4155	0.7632	0.8464	0.7584	0.6825
Maven	0.9012	0.5668	0.8965	0.9340	0.8880	0.6897
Nutch	0.8095	0.4698	0.8016	0.8193	0.7915	0.6836
PDFBox	0.7225	0.4139	0.7138	0.7498	0.7238	0.7130
Struts	0.8630	0.6187	0.8662	0.9342	0.8638	0.8632
Wicket	0.8256	0.4308	0.8143	0.8780	0.8015	0.6602
Zookeeper	0.6971	0.3787	0.6956	0.7700	0.6961	0.6897

Table 5.3: Analysis based on Svm model

project	Accuracy	MCC	precision	AUC	F-mesaure	G-mean
ActiveMQ	0.7245	0.3765	0.7539	0.7189	0.7111	0.6256
Avro	0.7903	0.5783	0.8220	0.7464	0.7743	0.7451
Calcite	0.6435	0.3561	0.6364	0.66641	0.6014	0.6094
Camel	0.7967	0.3382	0.7899	0.7267	0.7808	0.635
Cassandra	0.6710	0.3717	0.7032	0.6694	0.6565	0.6694
CXF	0.7613	0.3127	0.7465	0.7625	0.7439	0.6816
Drill	0.6489	0.3237	0.6478	0.6033	0.6126	0.6162
Flink	0.7543	0.3076	0.72378	0.7086	0.7265	0.6829
Flume	0.6631	0.3202	0.6631	0.6868	0.66329	0.6601
Hbase	0.7493	0.4986	0.7495	0.8234	0.7492	0.7490
Ignite	0.7493	0.4985	0.7494	0.8192	0.7492	0.7491
Kafka	0.7252	0.3559	0.7314	0.7133	0.7280	0.6801
Maven	0.8197	0.3255	0.8238	0.6990	0.8217	0.6484
Nutch	0.7301	0.3407	0.7406	0.7080	0.7346	0.6737
PDFBox	0.6925	0.3869	0.6949	0.7502	0.6922	0.6930
Struts	0.7916	0.5584	0.7899	0.8304	0.7900	0.7747
Wicket	0.8079	0.3426	0.8008	0.6016	0.7643	0.5651
Zookeeper	0.6338	0.2663	0.6413	0.6689	0.6355	0.6345

Table 5.4: Analysis based on Logistic model

project	Accuracy	MCC	precision	AUC	F-mesaure	G-mean
ActiveMQ	0.7334	0.4069	0.7283	0.7917	0.7219	0.6847
Avro	0.7580	0.4899	0.7557	0.7821	0.7550	0.7386
Calcite	0.6735	0.3271	0.6775	0.6861	0.6751	0.6650
Camel	0.8969	0.3781	0.8789	0.8267	0.8808	0.5805
Cassandra	0.6949	0.3938	0.6993	0.7650	0.6929	0.6943
CXF	0.8698	0.3313	0.8476	0.8555	0.8529	0.5776
Drill	0.7459	0.4937	0.7476	0.8033	0.7456	0.7462
Flink	0.7663	0.3246	0.7448	0.7154	0.7456	0.6215
Flume	0.6631	0.3202	0.6631	0.6868	0.66329	0.6601
Hbase	0.7493	0.4986	0.7495	0.8234	0.7492	0.7490
Ignite	0.7493	0.4985	0.7494	0.8192	0.7492	0.7491
Kafka	0.7252	0.3559	0.7314	0.7133	0.7280	0.6801
Maven	0.8197	0.3255	0.8238	0.6990	0.8217	0.6484
Nutch	0.7301	0.3407	0.7406	0.7080	0.7346	0.6737
PDFBox	0.6925	0.3869	0.6949	0.7502	0.6922	0.6930
Struts	0.7916	0.5584	0.7899	0.8304	0.7900	0.7747
Wicket	0.8079	0.3946	0.7912	0.7695	0.7939	0.6558
Zookeeper	0.6338	0.2663	0.6413	0.6689	0.6355	0.6345

Table 5.5: Analysis based on Decision tree model

project	Accuracy	MCC	precision	AUC	F-mesaure	G-mean
ActiveMQ	0.7447	0.4622	0.7489	0.7838	0.7463	0.7339
Avro	0.7741	0.5308	0.7742	0.7686	0.7741	0.7653
Calcite	0.7291	0.4555	0.7391	0.72171	0.7313	0.7314
Camel	0.8857	0.3334	0.8667	0.6899	0.8726	0.5782
Cassandra	0.6843	0.3686	0.6843	0.6991	0.6843	0.6843
CXF	0.8503	0.30000	0.8365	0.7667	0.8424	0.5971
Drill	0.7665	0.5332	0.7666	0.7950	0.7665	0.7661
Flink	0.7520	0.3163	0.7368	0.6742	0.7419	0.6368
Flume	0.6842	0.3691	0.6880	0.7240	0.6849	0.6853
Hbase	0.7253	0.4500	0.7253	0.7370	0.7253	0.7253
Ignite	0.8482	0.2956	0.8436	0.7098	0.8458	0.6097
Kafka	0.7252	0.3559	0.7314	0.7520	0.7280	0.6801
Maven	0.8540	0.4414	0.8540	0.8097	0.8540	0.7083
Nutch	0.7619	0.4070	0.7664	0.6970	0.7639	0.7050
PDFBox	0.7115	0.4231	0.7116	0.7176	0.7115	0.7115
Struts	0.8095	0.6086	0.8141	0.8513	0.8107	0.8083
Wicket	0.8145	0.4374	0.8037	0.6857	0.8071	0.6927
Zookeeper	0.6619	0.3163	0.6651	0.6500	0.6630	0.6591

Table 5.6: Analysis based on Bayes model

project	Accuracy	MCC	precision	AUC	F-mesaure	G-mean
ActiveMQ	0.7198	0.4101	0.7216	0.7966	0.7208	0.7031
Avro	0.7338	0.4672	0.7460	0.7704	0.7248	0.7177
Calcite	0.5883	0.1250	0.5350	0.5148	0.7465	0.5017
Camel	0.6858	0.3034	0.8772	0.8074	0.7421	0.7291
Cassandra	0.7055	0.4120	0.7998	0.6991	0.7050	0.7052
CXF	0.7856	0.3982	0.8681	0.8572	0.8126	0.7592
Drill	0.6745	0.3425	0.6819	0.6776	0.6878	0.6778
Flink	0.7172	0.3314	0.7430	0.7578	0.7267	0.6762
Flume	0.6715	0.3734	0.6846	0.6113	0.6984	0.6845
Hbase	0.6773	0.3125	0.6467	0.66733	0.6841	0.6857
Ignite	0.7031	0.3991	0.7998	0.7582	0.7544	0.7765
Kafka	0.7258	0.3558	0.7432	0.7242	0.6986	0.6742
Maven	0.8667	0.3527	0.8512	0.6034	0.8355	5429
Nutch	0.8015	0.4402	0.7958	0.6694	0.7769	0.6562
PDFBox	0.6874	0.3095	0.6744	0.6809	0.6867	0.6245
Struts	0.7559	0.4876	0.7795	0.7027	0.7345	0.7007
Wicket	0.7615	0.4320	0.8057	0.8067	0.7753	0.7427
Zookeeper	0.6984	0.4253	0.6875	0.6995	0.6754	0.6553

5.1.1 The importance of each metric

To assess each metric's ability to predict. Information Gain (IG) is a measure that expresses how well a characteristic (i.e., metric) distinguishes between different data samples. From 0 (no information) to 1 (the most information), the values range. A higher IG value will be assigned to metrics that contribute more information, while a lower IG value will be assigned to metrics that offer less information. A metric is more significant for the method-level bug prediction if it has a higher IG value, which indicates that it provides more information for predicting bugprone methods.We first determined the information gain values of the suggested metrics for each prediction, and then we ranked the metrics from 1 to N, where N is the total number of metrics.

As a result, the metric with the highest IG value will be given the ranking of 1, while the metre with the lowest IG value will be given the ranking of 40. A metric with a higher rank suggests that it may offer more details, which in turn suggests that it is a more significant attribute in the bug prediction.

The H5 measure (i.e., the number of authors) has the highest average rank, which is 1.5, according to the findings. As a result, we believe that H5 is crucial for identifying bug-prone techniques in our research initiatives. This table also reveals that the history measures have better predictive ability than the code metrics, which is in line with RQ2's findings. The top 8 important traits are all drawn from the set of historical measures, while several historical measures (such as the last four ranking measures, H19, H11, H10, and H9) only have a negligible effect on the forecast.

Table 5.7: Rank of each metrics

Metric	Rank	Metric	Rank	Metric	Rank	Metric	Rank
H5	1.5	C1	15.3	H7	21.4	C13	27.4
H1	2.5	C6	15.9	H18	21.7	C15	28.0
H3	3.5	H2	16.0	C19	24.2	C16	29.0
H4	4.3	C10	16.1	C12	24.2	C2	31.2
H12	5.4	H13	17.0	C11	25.0	C20	32.3
H6	7.2	H16	17.3	H17	25.0	C7	34.1
H15	8.5	C21	17.7	C14	25.4	H19	34.6
H14	10.2	H8	19.7	C4	25.9	H11	35.3
C8	15.0	C17	20.5	C5	26.1	H10	36.8
C3	15.3	C18	21.2	C9	26.6	H9	37.0

The distribution of ranks for each statistic is depicted in the figure. Two numbers are included in each box plot to illustrate the range of each metric's rank. Lines of code, or C1, rates from 8 to 23 across all forecasts, whereas added loc, or H1, ranks from 2 to 3 across all research projects.

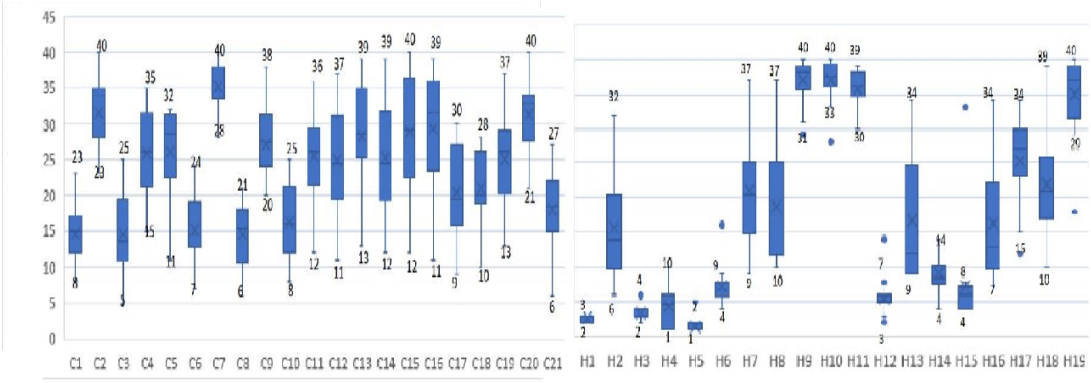


Figure 5.1: Rank distributions for each code metric and history measures

Look into simplifying the prediction models so that a development team can generate predictions similarly without having to rely on a variety of metrics. In other words, the ability to predict method-level bugs using a subset of metrics should help to save time and resources on data collecting and analysis.

We constructed method level bug prediction models using the top 1%, top 5%, top 10%, top 20%, and top 30% rated metrics from the ranks of each measure. The table below displays the results of the predictions made by utilising various collections of chosen metrics.

We can see that by (1) observing the performance of prediction models improves as we use more metrics better;

(2) when using only the top 30% of metrics, all resulting models' AUC values are more than 0.7, indicating that they can accurately identify bug-prone techniques;

(3) The suggested models are capable of accurately predicting errors at the method level even when only employing the top 1-ranked measure.

Table 5.8: Performance of prediction models with selected metrics

Project	TOP 1%	TOP 5%	TOP10%	TOP20%	TOP30%
ActiveMQ	0.786	0.799	0.807	0.822	0.838
Avro	0.673	0.655	0.751	0.773	0.788
Calcite	0.627	0.648	0.661	0.679	0.724
Camel	0.781	0.821	0.812	0.682	0.838
Cassandra	0.734	0.776	0.767	0.760	0.789
CXF	0.817	0.848	0.836	0.845	0.853
Drill	0.638	0.703	0.738	0.759	0.767
Flink	0.691	0.728	0.714	0.723	0.753
Flume	0.703	0.673	0.753	0.765	0.786
Hbase	0.677	0.707	0.753	0.787	0.795
Ignite	0.787	0.818	0.846	0.873	0.875
Kafka	0.730	0.716	0.735	0.757	0.788
Maven	0.815	0.821	0.845	0.861	0.879
Nutch	0.703	0.693	0.705	0.741	0.774
PDFBox	0.624	0.698	0.713	0.731	0.771
Struts	0.765	0.796	0.806	0.807	0.822
Wicket	0.780	0.815	0.823	0.840	0.863
Zookeeper	0.725	0.723	0.744	0.759	0.769

5.2 Analysis of results

5.2.1 Spider Chart (Radar Chart)

A spider chart, also known as a radar chart or a web chart, is a graphical way to display multivariate data in the form of a two-dimensional chart. It consists of a set of axes that radiate from a common center, with each axis representing a different variable or metric. Data points are plotted along each axis, and the resulting shape resembles a spider's web, hence the name.

Spider charts are useful for visualizing and comparing the performance or characteristics of multiple entities (such as classifiers in this case) across various dimensions (metrics in this case). Each line in the spider chart represents one entity, and the length of each line segment indicates the value of the corresponding metric.

A radar chart to facilitate the comparison of various classifiers across different performance metrics. This visual representation allows for a direct assessment of how each classifier fares in terms of metrics like accuracy, precision, F-measure, MCC (Matthews Correlation Coefficient), AUC (Area Under the ROC Curve), and G-mean. Each axis on the chart corresponds to one of these metrics, while each classifier is depicted by a separate curve. The length of each line segment extending from the center to the curve indicates the value of the respective metric. By examining the shape of the curves, one can easily discern the relative strengths and weaknesses of each classifier across all the chosen metrics.

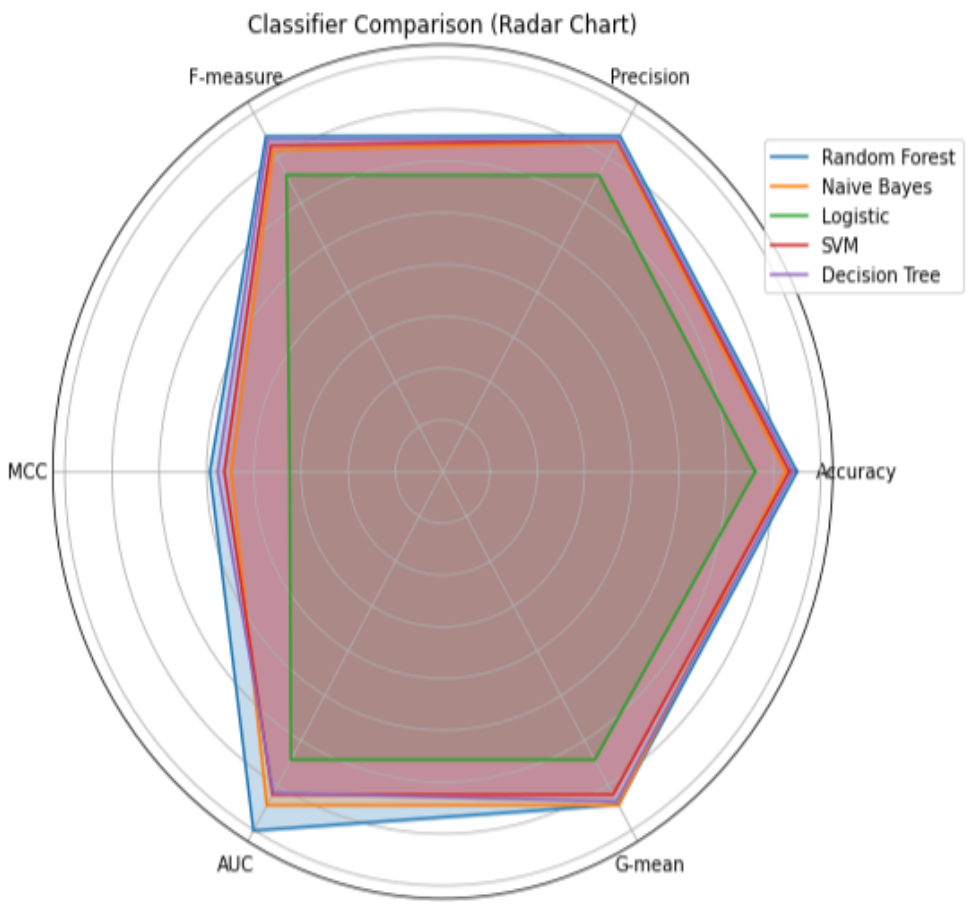


Figure 5.2: Spider graph

5.2.2 Bargraph

A bar graph for comparing the performance of different classifiers across various metrics. Each metric is represented on the x-axis, while the y-axis represents the values or scores achieved by the classifiers for each metric. The different colored bars for each metric correspond to different classifiers, allowing for a direct visual comparison of their performance.

In this specific scenario, the metrics being evaluated include "Accuracy," "Precision," "F-measure," "MCC" (Matthews Correlation Coefficient), "AUC" (Area Under the ROC Curve), and "G-mean." The classifiers being compared are "Random Forest," "Naive Bayes," "Logistic," "SVM," and "Decision Tree."

Each bar's height reflects the performance value of a particular metric for a specific classifier. For instance, the height of the "Accuracy" bar for the "Random Forest" classifier represents its accuracy score. This makes it easy to discern which classifiers excel in specific metrics and where they may have relative strengths or weaknesses.

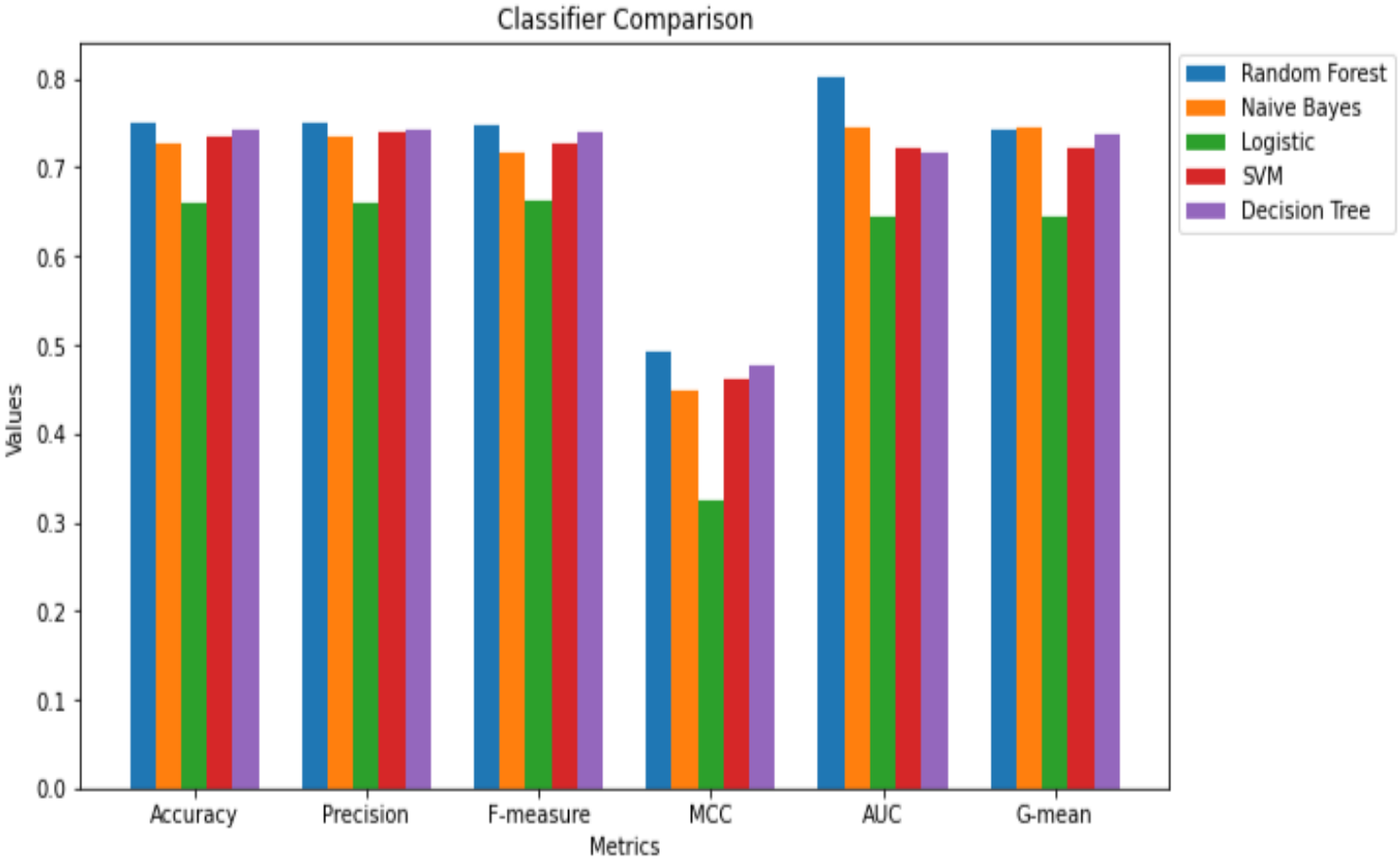


Figure 5.3: Bar graph

5.3 Future work

Future research could delve into a more fine-grained contextual analysis of code and development processes. This might involve considering factors such as code ownership, team dynamics, and developer expertise levels. Understanding how these contextual elements interact with code metrics and historical measures could provide a deeper understanding of bug-prone areas.

Additionally, incorporating natural language processing techniques to analyze developer comments, issue reports, and code reviews could offer valuable contextual information that enhances the precision of bug prediction models. Such an approach would not only refine bug prediction accuracy but also empower developers with targeted insights for proactive bug mitigation.

Additionally, exploring ensemble learning methods that dynamically combine the strengths

of multiple models could further enhance predictive performance. By enabling bug prediction models to adapt in real-time to changing project conditions, this research avenue has the potential to revolutionize how software development teams approach bug prevention and maintenance.

5.4 Conclusion

Our extensive analysis of bug prediction models at the method level has yielded several significant findings.

We have demonstrated that incorporating historical metrics into the model-building process leads to improved performance, with the most robust results achieved when using a combination of both method-level code metrics and historical measures.

Moreover, our comparative evaluation of various machine learning algorithms has provided valuable insights into their effectiveness in bug prediction.

Notably, our study highlights the critical importance of the H5 measure as the foremost attribute for predicting bug-prone methods. Historical measures, in general, exhibit superior predictive power compared to other metrics.

Furthermore, we have shown that prediction models can be simplified without sacrificing accuracy by focusing on a select few of the highest-ranked metrics.

Overall, this research significantly advances our understanding of bug prediction modeling, offering practical insights for the development of streamlined and effective prediction models at the method level.

Chapter 6

References

[1] Ran Moa, Shaozhi Wei, Qiong Feng , Zengyang Li , "An exploratory study of bug prediction at the method level," Information & Software Technology, Apr. 01, 2022. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106794

[2] Yasutaka Kamei , Emad Shihab ,"Defect Prediction: Accomplishments and Future Challenges," IEEE Conference Publication — IEEE Xplore, Mar. 01, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7476771

[3] Tim Menzies; Jeremy Greenwald; Art Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," IEEE Journals & Magazine — IEEE Xplore, Jan. 01, 2007. [Online].
Available: https://ieeexplore.ieee.org/document/4027145

[4] Emanuel Giger ,Marco D'Ambros,Martin Pinzger,Harald C. Gall "Method-level bug prediction Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement," ACM Conferences. [Online].
Available: https://dl.acm.org/doi/10.1145/2372251.2372285

[5] Martin Shepperd ,David Bowes; Tracy Hall,"Researcher Bias: The Use of Machine Learning in Software Defect Prediction," IEEE Journals & Magazine — IEEE Xplore, Jun. 01, 2014. [Online].

Available: https://ieeexplore.ieee.org/document/6824804

[6] N. N. M. R. R. Wa, "Mining metrics to predict component failures — Proceedings of the 28th international conference on Software engineering," ACM Conferences. [Online].
Available: https://dl.acm.org/doi/10.1145/1134285.1134349

[7] Raed Shatnawi"Improving software fault-prediction for imbalanced data," IEEE Conference Publication — IEEE Xplore, Mar. 01, 2012. [Online].
Available: https://ieeexplore.ieee.org/document/6207774

[8] Ran Moa, Shaozhi Wei a, Qiong Feng b, Zengyang Li An exploratory study of bug prediction at the method level. 2021 [Online].
Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584921002330?via%3Dihub

[9] Giulio Concas; Michele Marchesi; Alessandro Murgia; Roberto Tonelli; Ivana Turnu,"On the Distribution of Bugs in the Eclipse System," IEEE Journals & Magazine — IEEE Xplore, Dec. 01, 2011. [Online].
Available: https://ieeexplore.ieee.org/document/5928349

[10] Raul Robu; Paul Arseni-Ailoi; Dan Ungureanu-Anghel,"Using Weka API for creating a custom classification application," IEEE Conference Publication — IEEE Xplore, May 23, 2023. [Online].
Available: https://doi.org/10.1109/SACI58269.2023.10158580

[11] Hideaki Hata; Osamu Mizuno; Tohru Kikuno,"Bug prediction based on fine-grained module histories," IEEE Conference Publication — IEEE Xplore, Jun. 01, 2012. [Online].
Available: https://ieeexplore.ieee.org/document/6227193