**A**

**PROJECT REPORT**

**on**

COMPARITIVE ANALYSIS OF MACHINE LEARNING MODELS

FOR SOFTWARE DEFECT PREDICTION

**Submitted by** :

|  |  |  |
| --- | --- | --- |
| KALWA SAI SREEJA  JANANI PRAKASH |  | 245322733083  22J41A0592 |

**Under the guidance of**

## Shri. Deepanshu Dixit

Scientist



# DEFENCE RESEARCH AND DEVELOPMENT LABORATORY

Kanchanbagh , Hyderabad

500058

1

**ABSTRACT**

This project presents a comparative analysis of three widely used machine learning models—Naive Bayes, Decision Tree, and Random Forest—for software defect prediction. The analysis is performed using both the WEKA tool and a custom Python GUI application developed with PyQt5 and scikit-learn. The main objective is to identify which model delivers the best accuracy and performance in classifying defective and non-defective software modules. The study evaluates each model using standard metrics such as accuracy, precision, recall, F1-score, mean absolute error (MAE), and root mean squared error (RMSE). The project highlights the strengths and limitations of each model and platform, offering insights into their effectiveness for software quality assurance.

2

| **S. No.** |  |  |  |  |  |  |  |  |  |  |  |  |  | **Section Title** |  | **Page No.** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  | Introduction |  | 4 |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  | Objectives |  | 5 |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  | Weka tool |  | 6 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.1 Introduction |  | 6 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.2 Key components |  | 6 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.3 Core functionalities |  | 7 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.4 Filters |  | 8 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.5 Algorithms |  | 9 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.6 Evaluation techniques |  | 9 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3.7 Visualization capabilities |  | 10 |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  | Python-based implementation |  | 11 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 4.1 Technology stack |  | 11 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 4.2 Filters and functionalities |  | 11 |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  | Methodology |  | 13 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5.1 Workflow using Weka tool |  | 14 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5.2 Using python |  | 15 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5.2.1 Workflow |  | 14 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5.2.2 Code |  | 15 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5.2.3 Code architecture |  | 26 |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  | Results |  | 27 |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  | Conclusion |  | 30 |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  | Limitations and future scope |  | 31 |

**TABLE OF CONTENTS**

**1. Introduction**

Software defect prediction is a critical task in ensuring software quality and reducing maintenance costs. Predicting whether a software module is likely to be defective allows developers to allocate resources efficiently and focus on high-risk components. In this project, three machine learning algorithms—Naive Bayes, Decision Tree, and Random Forest—are evaluated for their ability to classify software modules based on defect likelihood.

The project is implemented using two approaches:

* The WEKA tool, a well-known Java-based data mining platform.
* A Python-based GUI application developed using PyQt5 and scikit-learn.

The aim is to compare these models across platforms in terms of performance metrics and ease of implementation. Datasets used in the study include open-source software defect data from repositories such as NASA's PROMISE dataset

4

**2. OBJECTIVES**

* To develop a Python-based GUI tool for software defect prediction.
* To implement and test three machine learning algorithms: Naive Bayes, Decision Tree, and Random Forest.
* To use the WEKA tool as a benchmark for comparison.
* To evaluate model performance using standard classification metrics.
* To identify the best-performing model for software defect prediction.
* To analyze discrepancies in performance between the WEKA and Python implementations.

5

**3. WEKA TOOL**

**(Waikato Environment for Knowledge Analysis)**

**3.1 Introduction:**

**WEKA (Waikato Environment for Knowledge Analysis)** is a popular open-source software suite developed by the **University of Waikato** in New Zealand. It provides a comprehensive set of tools for **data preprocessing, classification, regression, clustering, association rule mining**, and **visualization**, all through an easy-to-use **graphical user interface (GUI)** as well as a **command-line interface**.

WEKA is written in **Java** and is widely used in **research, education**, and **industry** for data mining and machine learning.

**3.2 Key Components of WEKA**

**A. Explorer**

The main GUI component used to access most of WEKA's functionalities.

**Tabs inside Explorer:**

| **Tab Name** | **Purpose** |
| --- | --- |
| **Preprocess** | Load datasets, apply filters for cleaning and transformation. |
| **Classify** | Apply classification/regression algorithms and evaluate performance. |
| **Cluster** | Perform unsupervised clustering on the dataset. |
| **Associate** | Mine association rules (e.g., Apriori algorithm). |
| **Select Attributes** | Perform feature selection and rank attributes based on importance. |
| **Visualize** | Visualize data distributions and model predictions (2D scatter plots). |

6

**B. Workbench**

A modern interface combining all of WEKA’s tools in one place (an alternative to Explorer). It includes:

* File loading
* Drag-and-drop operations
* Quick model testing

**C. Experimenter**

Used for running **controlled experiments** with multiple algorithms and datasets, especially useful for:

* Comparing algorithm performance
* Repeating tests for statistical confidence
* Visualizing comparative results

**D. Knowledge Flow**

A graphical pipeline builder for creating data processing and model workflows visually (like KNIME or RapidMiner). Good for automating batch ML tasks.

**E. Simple CLI (Command Line Interface)**

Allows advanced users to interact with WEKA via the command line, useful for scripting and batch processing.

**3.3 Core Functionalities in WEKA**

| Category | Capabilities |
| --- | --- |
| Data Import | Supports CSV, ARFF, C4.5, XRFF, and more |
| Preprocessing | Normalize, standardize, discretize, remove attributes, handle missing values |
| Classification | Apply algorithms like J48 (C4.5), Naive Bayes, SVM, k-NN, Random Forest |
| Regression | Linear Regression, SMOReg, M5P, etc. |
| Clustering | k-means, EM, Cobweb, FarthestFirst  7 |
| Visualization | Graphs for data distribution, predictions, errors, and clustering |
| Evaluation | Cross-validation, percentage split, confusion matrix, ROC, F1, AUC |

**3.4 Filters in WEKA**

Filters are used to transform data before feeding it into algorithms.

Types of Filters:

1. Supervised Filters

These consider class labels while transforming data.

Examples:

* Discretize
* NominalToBinary
* Standardize
* AttributeSelection
* ClassBalancer

2. Unsupervised Filters

They transform data without considering class labels.

Examples:

* RemoveUseless
* Normalize
* PrincipalComponents
* ReplaceMissingValues
* StringToNominal

Categories:

* Attribute Filters – modify entire columns (features).
* Instance Filters – modify rows (data points).
* MultiFilters – apply a sequence of filters.

8

**3.5 Algorithms in WEKA**

Classification (Supervised):

* J48 – Decision Tree (based on C4.5)
* Naive Bayes
* Random Forest
* IBk – k-Nearest Neighbors
* SMO – SVM for classification
* Multilayer Perceptron – Feedforward neural network

Regression:

* Linear Regression
* SMOReg – SVM for regression
* M5P – Model tree

Clustering (Unsupervised):

* k-means
* EM (Expectation-Maximization)
* Cobweb
* FarthestFirst

Association Rules:

* Apriori – Market basket analysis
* FPGrowth

**3.6 Evaluation Techniques in WEKA**

* Hold-out split (training/testing percentage)
* 10-fold cross-validation (default)
* Leave-One-Out Cross Validation (LOOCV)
* Evaluation Metrics:
  + Accuracy
  + Precision, Recall, F1-Score
  + Confusion Matrix

9

* + ROC/AUC
  + Mean Absolute Error (MAE), Root Mean Square Error (RMSE)
  + Kappa statistic

**3.7 Visualization Capabilities**

* 2D plots for classification boundaries
* Scatter plots with class coloring
* Error analysis visualizations
* ROC curves

Histograms for feature distribution

10

**4. Python-based implementation**

**4.1 Technology Stack:**

* Python 3.x – Primary programming language.
* PyQt5 – For GUI development.
* pandas – Data manipulation and CSV file handling.
* numpy – Numerical computation.
* scikit-learn – Machine learning algorithms and metrics.

**4.2 Features & Functionalities:**

**A. Graphical User Interface (GUI)**

* Two-Panel Design for simplicity and clarity:
  + Left Panel: Interactive buttons for loading datasets, performing checks, training models, and evaluating.
  + Right Panel: Scrollable QTextEdit used for output display—logs, summaries, and results.
* Styled Controls: Modern styled buttons with hover effects and icons.
* Dropdown Menu to select among supported ML models:
  + Naive Bayes
  + Decision Tree
  + Random Forest

**B. Dataset Handling**

* CSV-based dataset import for both training and testing.
* The last column is automatically identified as the target variable.
* Handles missing values with forward-fill imputation (fillna(method='ffill')).

**C. Data Exploration Tools**

* Null Value Checker: Displays missing value count for each feature.
* Class Imbalance Viewer: Shows class distribution for the target variable.

**D. Machine Learning Models**

* Supports the following classification algorithms:

11

* + Gaussian Naive Bayes
  + Decision Tree Classifier
  + Random Forest Classifier
* Target labels are encoded using LabelEncoder to handle categorical outputs.

**E. Model Training**

* Trains the selected model on the loaded training dataset.
* Measures and displays time taken to train.

**F. Model Evaluation**

* Evaluates the trained model against the test dataset.
* Outputs include:
  + Accuracy Score (% Total Correct Predictions)
  + Confusion Matrix
  + Classification Report (Precision, Recall, F1-score)
  + Regression Metrics:
    - Mean Absolute Error (MAE)
    - Root Mean Squared Error (RMSE)
    - Relative Absolute Error (RAE)
    - Root Relative Squared Error (RRSE)
  + Cohen’s Kappa Statistic — Agreement between true and predicted labels
  + Class-wise accuracy and confusion matrix presentation.

12

**5. Methodology**

**5.1 Workflow using Weka Tool**

1. Load the JM1 Dataset

* Open WEKA Explorer.
* Go to the Preprocess tab.
* Load the JM1 dataset.

2. Split the Dataset into 3 Parts (e.g., 60/20/20)

A. Get the 60% Training Set:

* Click Filters → Choose → unsupervised → instance → RemovePercentage.
* Set:
  + Percentage: 60
  + InvertSelection: True
* Click Apply.
* Save this part as jm1\_train.arff.

B. Get the Remaining 40% (Validation + Test):

* Reload the full JM1 dataset.
* Use RemovePercentage, set:
  + Percentage: 60
  + InvertSelection: False
* Click Apply.
* Save this as jm1\_remaining.arff.

C. Split the Remaining 40% into 20% Validation and 20% Test:

For Validation Set (50% of the remaining 40%):

* Load jm1\_remaining.arff.
* Use RemovePercentage again:
  + Percentage: 50
  + InvertSelection: True
* Apply and save as jm1\_validation.arff.

13

For Test Set (other 50% of the remaining 40%):

* Use RemovePercentage again on jm1\_remaining.arff:
  + Percentage: 50
  + InvertSelection: False
* Apply and save as jm1\_test.arff.

3.Load the JM1\_training Dataset

* Open WEKA Explorer.
* Go to the Preprocess tab.
* Load the JM1\_training.arff dataset.

4.Training and testing the dataset

* Go to the Classify tab.
* In Classify, set:
  + Supplied test set → Load jm1\_test.arff.
  + Click Start to run.

**5.2 Using Python**

**5.2.1 Workflow**

1. Load Training Data  
   Click 📂 Load Training Data → Select .csv file.
2. Load Testing Data  
   Click 📂 Load Testing Data → Select matching .csv file.
3. Check Nulls  
   Click 🕳 Check Nulls → Shows missing values (auto-handled).
4. Check Class Imbalance  
   Click ⚖ Class Imbalance → Shows label distribution.
5. Select Model  
   Choose from dropdown:
   * Naive Bayes
   * Decision Tree
   * Random Forest
6. Train Model  
   Click 🚀 Train Model → Model gets trained on training data.

14

1. Evaluate Model  
   Click 📊 Evaluate Model → Shows:
   * Accuracy
   * Confusion Matrix
   * Precision, Recall, F1
   * MAE, RMSE, RAE, RRSE
   * Kappa Statistic

**5.2.2 Python Code:**

import sys

import time

import numpy as np

import pandas as pd

from PyQt5.QtWidgets import (

QApplication, QMainWindow, QFileDialog, QPushButton, QLabel,

QComboBox, QTextEdit, QHBoxLayout, QVBoxLayout, QWidget,

QFrame, QGridLayout, QSizePolicy

)

from PyQt5.QtGui import QFont

from PyQt5.QtCore import Qt

from sklearn.naive\_bayes import GaussianNB

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import (

confusion\_matrix, classification\_report, accuracy\_score,

mean\_absolute\_error, mean\_squared\_error

)

15

from sklearn.preprocessing import LabelEncode

class WekaLikeTool(QMainWindow):

def \_init\_(self):

super().\_init\_()

self.setWindowTitle("⚙ Mini WEKA Tool in Python")

self.setGeometry(100, 100, 1200, 650)

self.train\_data = None

self.test\_data = None

self.model = None

self.label\_encoder = None

self.initUI()

def initUI(self):

main\_layout = QHBoxLayout()

# Left panel setup with grid layout

left\_panel = QFrame()

left\_panel.setStyleSheet("""

QFrame {

background-color: #f4f6f7;

padding: 30px;

border-right: 2px solid #dcdcdc;

}

""")

grid\_layout = QGridLayout()

16

grid\_layout.setSpacing(15)

left\_panel.setLayout(grid\_layout)

def make\_button(text):

btn = QPushButton(text)

btn.setStyleSheet("""

QPushButton {

background-color: #5DADE2;

color: white;

font-weight: bold;

border-radius: 8px;

padding: 10px;

}

QPushButton:hover {

background-color: #3498DB;

}

""")

btn.setMinimumHeight(45)

btn.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

return btn

# Buttons

self.load\_train\_btn = make\_button("📂 Load Training Data")

self.load\_train\_btn.clicked.connect(self.load\_train\_dataset)

17

self.load\_test\_btn = make\_button("📂 Load Testing Data")

self.load\_test\_btn.clicked.connect(self.load\_test\_dataset)

self.null\_check\_btn = make\_button("🕳 Check Nulls")

self.null\_check\_btn.clicked.connect(self.check\_nulls)

self.imbalance\_btn = make\_button("⚖ Class Imbalance")

self.imbalance\_btn.clicked.connect(self.check\_imbalance)

self.train\_btn = make\_button("🚀 Train Model")

self.train\_btn.clicked.connect(self.train\_model)

self.eval\_btn = make\_button("📊 Evaluate Model")

self.eval\_btn.clicked.connect(self.evaluate\_model)

# Model selector with label

model\_label = QLabel("🧠 Select ML Model:")

model\_label.setStyleSheet("font-weight: bold; margin-top: 5px;")

self.model\_selector = QComboBox()

self.model\_selector.addItems(["Naive Bayes", "Decision Tree", "Random Forest"])

self.model\_selector.setStyleSheet("padding: 8px; font-weight: bold;")

self.model\_selector.setMinimumHeight(40)

# Add buttons to grid layout (2 columns)

18

buttons = [

self.load\_train\_btn, self.load\_test\_btn,

self.null\_check\_btn, self.imbalance\_btn,

self.train\_btn, self.eval\_btn

]

positions = [(i // 2, i % 2) for i in range(len(buttons))]

for pos, btn in zip(positions, buttons):

grid\_layout.addWidget(btn, \*pos)

# Model selector placed below grid

grid\_layout.addWidget(model\_label, 3, 0, 1, 2)

grid\_layout.addWidget(self.model\_selector, 4, 0, 1, 2)

# Output Panel (Right side)

self.output = QTextEdit()

self.output.setReadOnly(True)

self.output.setStyleSheet("""

QTextEdit {

background-color: #ffffff;

font-family: Consolas;

font-size: 12pt;

padding: 10px;

border: none;

}

""")

main\_layout.addWidget(left\_panel, 1)

19

main\_layout.addWidget(self.output, 2)

container = QWidget()

container.setLayout(main\_layout)

self.setCentralWidget(container)

app\_font = QFont("Segoe UI", 10)

QApplication.instance().setFont(app\_font)

def load\_train\_dataset(self):

file\_path, \_ = QFileDialog.getOpenFileName(self, "Open Train CSV", "", "CSV Files (\*.csv)")

if file\_path:

self.train\_data = pd.read\_csv(file\_path)

self.output.append("✅ Training dataset loaded successfully.\n")

def load\_test\_dataset(self):

file\_path, \_ = QFileDialog.getOpenFileName(self, "Open Test CSV", "", "CSV Files (\*.csv)")

if file\_path:

self.test\_data = pd.read\_csv(file\_path)

self.output.append("✅ Testing dataset loaded successfully.\n")

def check\_nulls(self):

if self.train\_data is not None:

nulls = self.train\_data.isnull().sum()

self.output.append(f"\n🕳 Null Values in Training Data:\n{nulls.to\_string()}\n")

else:

self.output.append("\n⚠ Load training dataset first.\n")

20

def check\_imbalance(self):

if self.train\_data is not None:

target\_col = self.train\_data.columns[-1]

imbalance = self.train\_data[target\_col].value\_counts()

self.output.append(f"\n⚖ Class Distribution:\n{imbalance.to\_string()}\n")

else:

self.output.append("\n⚠ Load training dataset first.\n")

def train\_model(self):

if self.train\_data is not None:

target\_col = self.train\_data.columns[-1]

X = self.train\_data.iloc[:, :-1].fillna(method='ffill')

y = self.train\_data.iloc[:, -1]

model\_type = self.model\_selector.currentText()

if model\_type == "Naive Bayes":

self.model = GaussianNB()

elif model\_type == "Decision Tree":

self.model = DecisionTreeClassifier()

elif model\_type == "Random Forest":

self.model = RandomForestClassifier()

# Label encode target for model if needed

self.label\_encoder = LabelEncoder()

y\_encoded = self.label\_encoder.fit\_transform(y)

21

start = time.time()

self.model.fit(X, y\_encoded)

end = time.time()

self.output.append(f"\n✅ Model '{model\_type}' trained successfully.")

self.output.append(f"Time taken to build model: {end - start:.2f} seconds\n")

else:

self.output.append("\n⚠ Load training dataset first.\n")

def evaluate\_model(self):

if self.model is not None and self.test\_data is not None:

X\_test = self.test\_data.iloc[:, :-1].fillna(method='ffill')

y\_test = self.test\_data.iloc[:, -1]

# Encode test labels with same encoder as training

y\_test\_encoded = self.label\_encoder.transform(y\_test)

start = time.time()

y\_pred\_encoded = self.model.predict(X\_test)

end = time.time()

acc = accuracy\_score(y\_test\_encoded, y\_pred\_encoded) \* 100 # percentage

cm = confusion\_matrix(y\_test\_encoded, y\_pred\_encoded)

class\_report=classification\_report(y\_test\_encoded,y\_pred\_encoded,

target\_names=self.label\_encoder.classes\_, output\_dict=True)

22

# Regression metrics on encoded labels

mae = mean\_absolute\_error(y\_test\_encoded, y\_pred\_encoded)

rmse = mean\_squared\_error(y\_test\_encoded, y\_pred\_encoded, squared=False)

mean\_val = np.mean(y\_test\_encoded)

rae = np.sum(np.abs(y\_test\_encoded - y\_pred\_encoded)) / np.sum(np.abs(y\_test\_encoded

mean\_val))

rrse = np.sqrt(np.sum((y\_test\_encoded - y\_pred\_encoded) \*\* 2) / np.sum((y\_test\_encode

mean\_val) \*\* 2))

total\_instances = len(y\_test\_encoded)

correct = (y\_test\_encoded == y\_pred\_encoded).sum()

incorrect = total\_instances - correct

# Kappa statistic calculation

total = total\_instances

sum\_po = correct / total

# calculate expected accuracy

p\_true = np.bincount(y\_test\_encoded) / total

p\_pred = np.bincount(y\_pred\_encoded) / total

p\_e = np.sum(p\_true \* p\_pred)

kappa = (sum\_po - p\_e) / (1 - p\_e) if (1 - p\_e) != 0 else 0

# Output format

self.output.append("\n=== Evaluation on test split ===\n")

self.output.append(f"Time taken to test model on test split: {end - start:.2f} seconds\n")

self.output.append("=== Summary ===\n")

23

self.output.append(f"Correctly Classified Instances {correct} {acc:.4f} %")

self.output.append(f"Incorrectly Classified Instances {incorrect} {100 - acc:.4f} %")

self.output.append(f"Kappa statistic {kappa:.4f}")

self.output.append(f"Mean absolute error {mae:.4f}")

self.output.append(f"Root mean squared error {rmse:.4f}")

self.output.append(f"Relative absolute error {rae \* 100:.4f} %")

self.output.append(f"Root relative squared error {rrse \* 100:.4f} %")

self.output.append(f"Total Number of Instances {total\_instances}\n")

self.output.append("=== Detailed Accuracy By Class ===\n")

self.output.append(f"{'':17} {'TP Rate':7} {'FP Rate':7} {'Precision':9} {'Recall':7}

{'F-Measure':9} {'MCC':7} {'ROC Area':8} {'PRC Area':8} Class")

for i, class\_name in enumerate(self.label\_encoder.classes\_):

cr = class\_report[class\_name]

self.output.append(

f"{class\_name:17} "

f"{cr['recall']:.3f} "

f"{cr['false\_positive\_rate'] if 'false\_positive\_rate' in cr else 0:.3f} "

f"{cr['precision']:.3f} "

f"{cr['recall']:.3f} "

f"{cr['f1-score']:.3f} "

f"{0:.3f} " # MCC placeholder (complex to calculate here)

f"{0:.3f} " # ROC Area placeholder

f"{0:.3f} "

f"{class\_name}"

) 24

weighted = class\_report['weighted avg']

self.output.append(

f"Weighted Avg. "

f"{weighted['recall']:.3f} "

f"{weighted['false\_positive\_rate'] if 'false\_positive\_rate' in weighted else 0:.3f} "

f"{weighted['precision']:.3f} "

f"{weighted['recall']:.3f} "

f"{weighted['f1-score']:.3f} "

f"{0:.3f} "

f"{0:.3f} "

f"{0:.3f} "

)

self.output.append("\n=== Confusion Matrix ===\n")

header = " "

for i, class\_name in enumerate(self.label\_encoder.classes\_):

class\_str = str(class\_name)

header += f"{class\_str[:4]:>5} "

self.output.append(header + " <-- classified as")

for i, class\_name in enumerate(self.label\_encoder.classes\_):

class\_str = str(class\_name)

row = f"{class\_str[:4]:>4} "

for j in range(len(self.label\_encoder.classes\_)):

row += f"{cm[i, j]:5} "

row += f"| {class\_str}"

self.output.append(row)

25

else:

self.output.append("\n⚠ Train a model and load testing dataset first.\n")

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

window = WekaLikeTool()

window.show()

sys.exit(app.exec\_())

**5.2.3 Code Architecture:**

Main Class: WekaLikeTool (inherits from QMainWindow)

1. UI Setup: initUI()
   * Left Panel: Buttons & Model Selector
   * Right Panel: QTextEdit for Output
2. Dataset Loaders
   * load\_train\_dataset()
   * load\_test\_dataset()
3. Data Preprocessing
   * check\_nulls()
   * check\_imbalance()
4. Model Training: train\_model()
   * Supports: Naive Bayes, Decision Tree, Random Forest
   * Label encoding for categorical outputs
5. Model Evaluation: evaluate\_model()
   * Classification Metrics: Accuracy, F1, Recall, etc.
   * Regression Metrics: MAE, RMSE, RAE, RRSE
   * Kappa Statistic
   * Confusion Matrix & Class-wise report
6. Entry Point:

if \_\_name\_\_ == '\_\_main\_\_': Launch QApplication

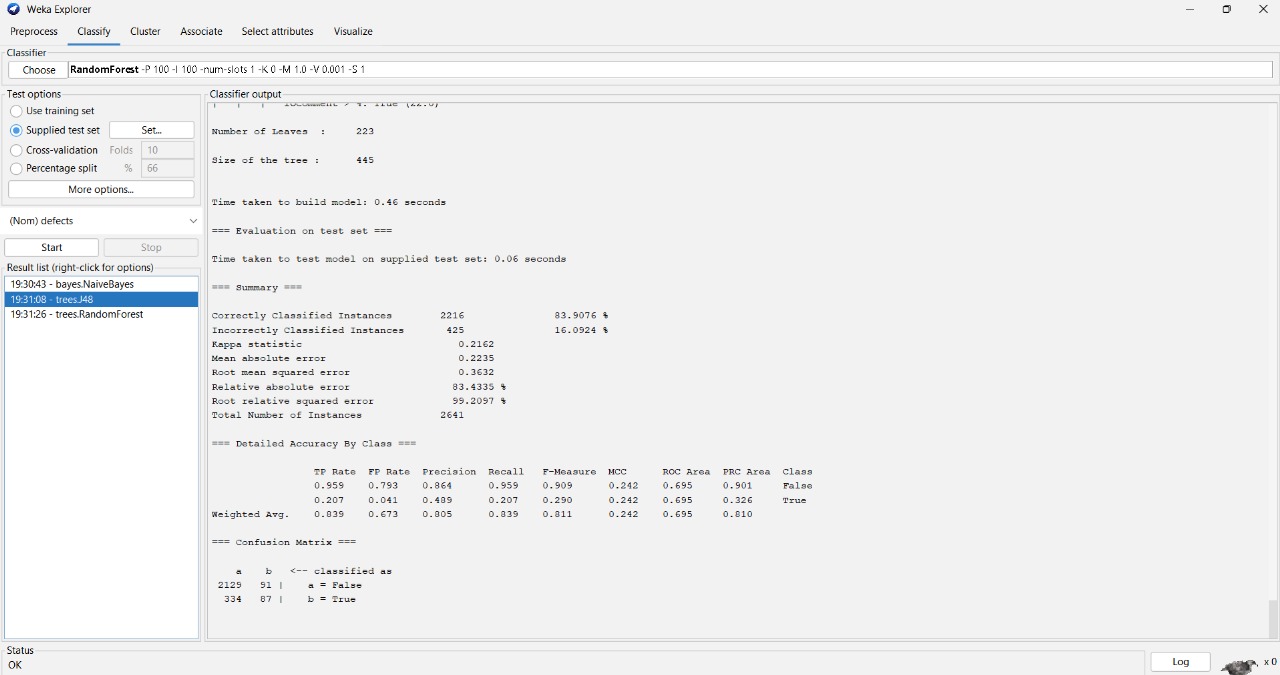
26

**6.RESULTS**

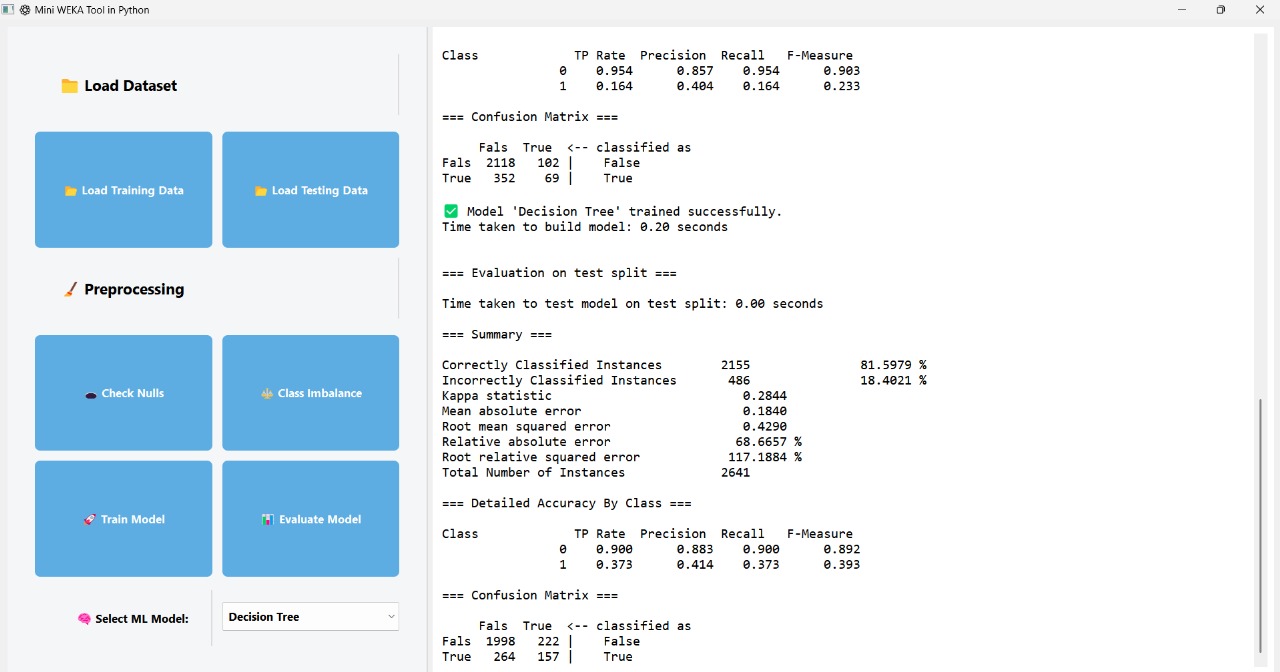
The following are the observations or results for three models using weka tool and python code:

**1. J48(Decision tree)**

Weka tool:



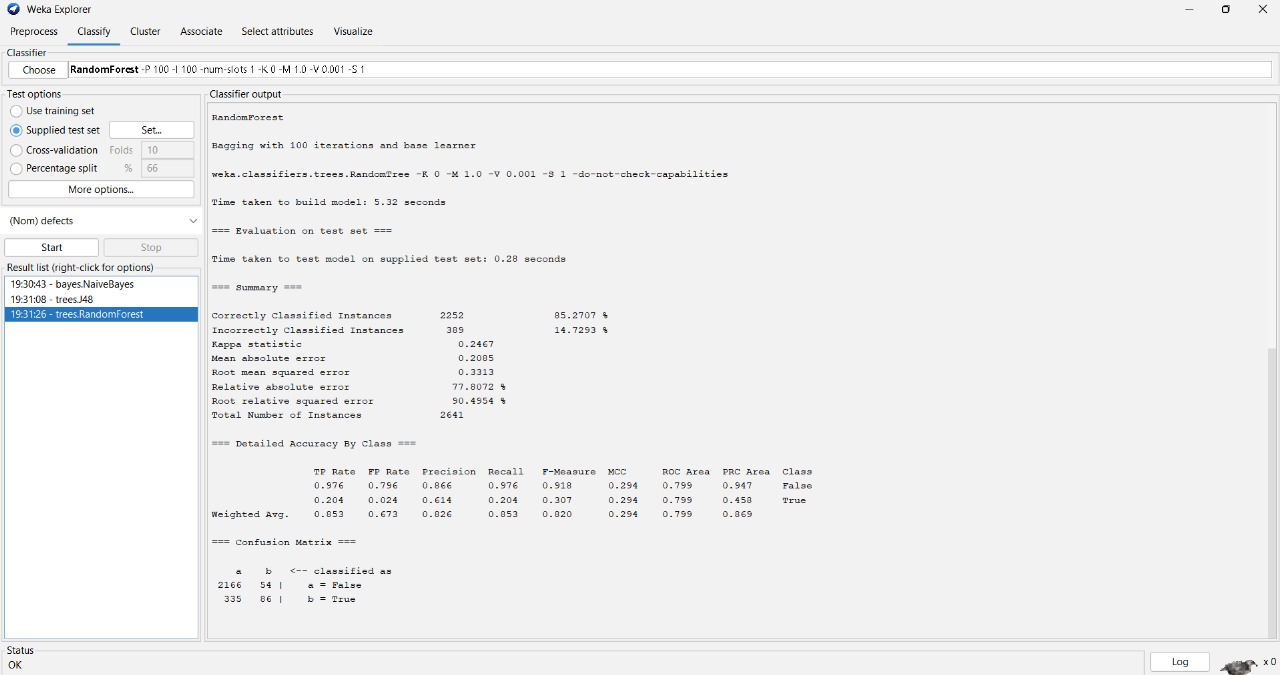
Python code:



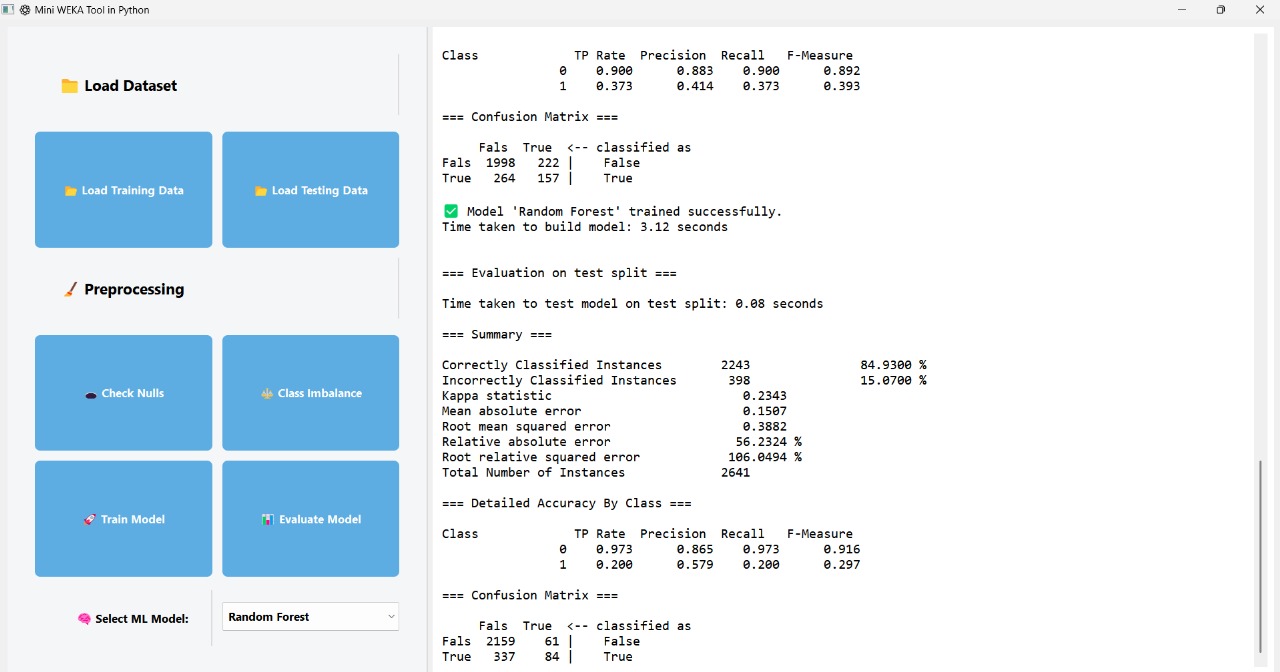
27

**2. Random forest**

Weka tool:



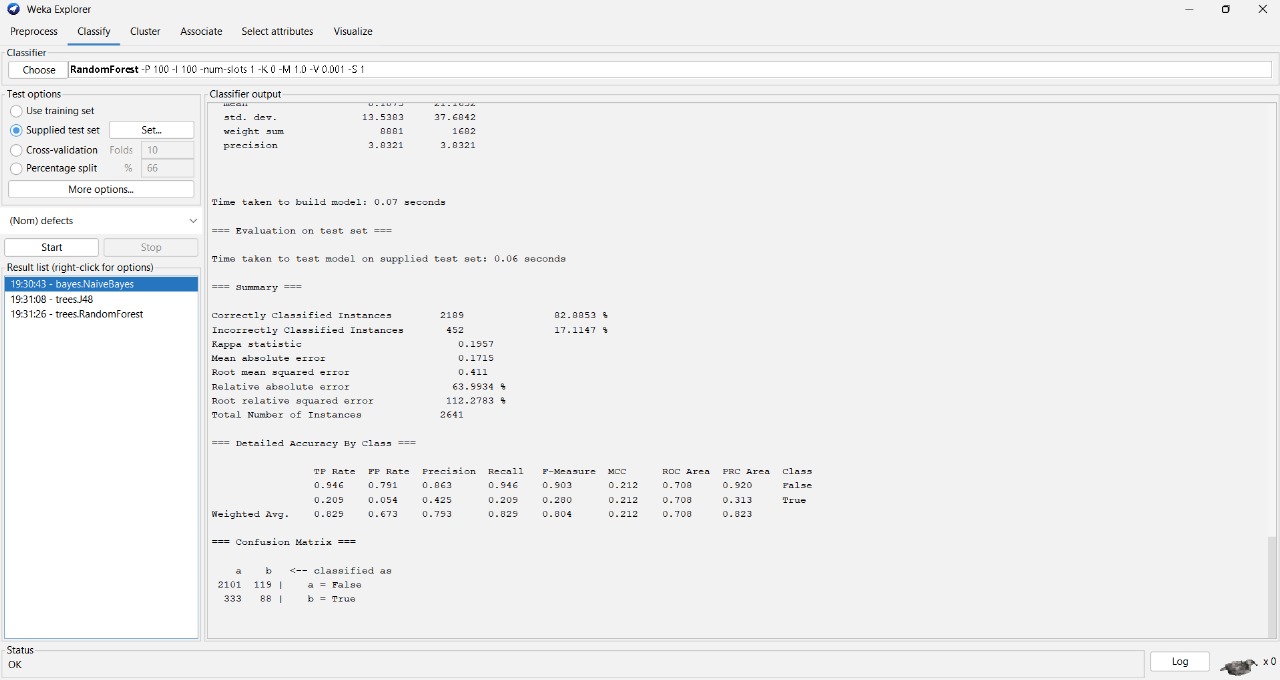
Python code:



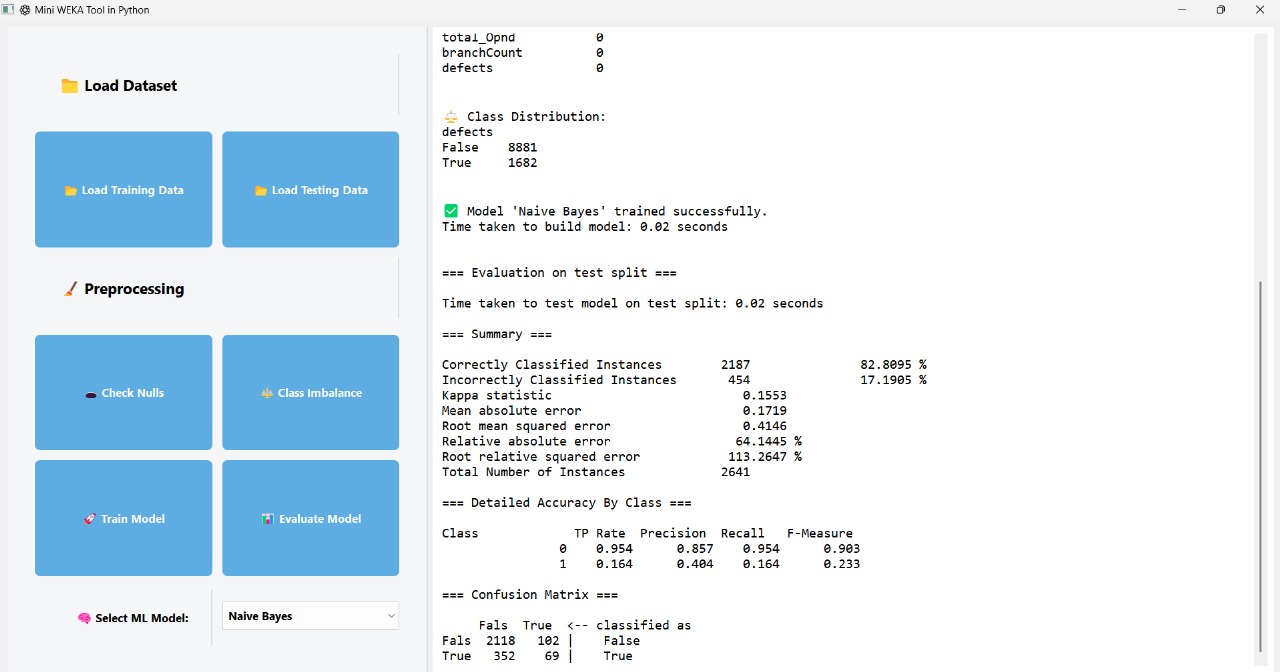
28

**3.Naive bayes**

Weka tool:



Python code:



29

**7. CONCLUSION**

From the observations, it is clear that the Random Forest model consistently outperforms the other models in terms of accuracy. Both in the WEKA tool and the Python code implementation, the Random Forest achieves the highest accuracy compared to the alternatives tested. This demonstrates the model’s robustness and reliability across different environments and confirms its suitability for the given classification tasks for jm1 dataset.

Finally, we can conclude that all the evaluation metrics for different models will differ for different datasets, that is only few evaluation metrics outperform for one dataset in one model and an other dataset will outperform in other model.

30

**8. LIMITATIONS AND FUTURE SCOPE**

**Limitations:**

* The study is limited to only three classification algorithms.
* Only one dataset may have been used for evaluation; results may vary with other datasets.
* The Python GUI handles missing values in a basic way (e.g., forward fill).
* Cross-validation and advanced preprocessing are not implemented.
* Visualizations are limited to text-based output in the Python tool.
* Performance may differ based on dataset quality and size.

**Future scope:**

* Model Expansion: Include more advanced models like SVM, XGBoost, and Neural Networks.
* Hyperparameter Tuning: Add support for grid search or manual parameter selection.
* Dynamic Feature Selection: Allow users to select target columns and features dynamically.
* Data Visualization: Add plots such as ROC curves, confusion matrices, and feature importance charts.
* Cross-Validation: Integrate k-fold cross-validation for more robust performance evaluation.
* Dataset Integration: Allow integration with online defect datasets (e.g., GitHub or PROMISE).
* Deployment: Package the Python GUI as a standalone executable or web-based tool.

31