A Project report on

SOFTWARE BUG PREDICTION USING MACHINE LEARNING

A Dissertation submitted to JNTU Hyderabad in partial fulfillment of the academic requirements for the award of the degree.

Bachelor of Technology

in

Information Technology

Submitted by

S. Balaji Manikanta Sai
(19H51A1227)

V. Koushik
(19H51A1230)

D. Navaneeth Sai
(19H51A1240)

M. Avinash
(19H51A1241)

Under the esteemed guidance of

Dr. K.L.S. Soujanya (B. Tech, M. Tech, Phd)
(Professor & HOD)

Department of Information Technology

CMR COLLEGE OF ENGINEERING & TECHNOLOGY
(UGC Autonomous)
*Approved by AICTE *Affiliated to JNTUH *NAAC Accredited with A^+ Grade
KANDLAKOYA, MEDCHAL ROAD, HYDERABAD - 501401

2019- 2023

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

KANDLAKOYA, MEDCHAL ROAD, HYDERABAD – 501401

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that the Major Project report entitled **"SOFTWARE BUG PREDICTION USING MACHINE LEARNING"** being submitted by S. Balaji Manikanta Sai (19H51A1227), V. Koushik (19H51A1230), D. Navaneeth Sai (19H51A1240), M. Avinash (19H51A1241) in partial fulfillment for the award of **Bachelor of Technology in Information Technology** is a record of bonafide work carried out his/her under my guidance and supervision.

The results embodied in this project report have not been submitted to any other University or Institute for the award of any Degree.

Dr. K L S Soujanya
Professor and HOD
Dept. of IT

Dr. K L S Soujanya (B. Tech, M. Tech, Phd)
Professor and HOD
Dept. of IT

ACKNOWLEDGEMENT

With great pleasure, we want to take this opportunity to express my heartfelt gratitude to all the people who helped in making this project work a grand success.

We would like to thank **Dr. K L S Soujanya,** Professor & Head of the Department of Information Technology, CMR College of Engineering and Technology, for her valuable technical suggestions and guidance during the execution of this project work, who is the major driving force to complete my project work successfully.

We are very grateful to **Dr. Vijaya Kumar Koppula**, Dean-Academic, CMR College of Engineering and Technology, for his constant support and motivation in successfully carrying out the project work.

We are highly indebted to **Dr. V A Narayana,** Principal, CMR College of Engineering and Technology, for giving permission to carry out this project in a successful and fruitful way.

We would like to thank the Teaching & Non- teaching staff of the Department of Information Technology for their cooperation.

We sincerely thank **Mr. Ch. Gopal Reddy**, Secretary, CMR Group of Institutions, for his continuous care.

Finally, we extend thanks to our parents who stood behind us at different stages of this Project. We sincerely acknowledge and thank all those who gave support directly and indirectly in the completion of this project work.

S. Balaji Manikanta Sai	19H51A1227
V. Koushik	19H51A1230
D. Navaneeth Sai	19H51A1240
M. Avinash	19H51A1241

TABLE OF CONTENTS

List of Figures

List of Tables

ABSTRACT

Software Bug Prediction (SBP) is an important process in software development and maintenance, which concerns the overall software success. This is because predicting the software faults in the earlier phase improves the software quality, reliability, and efficiency and reduces the software cost. However, developing a robust bug prediction model is challenging, and many techniques have been proposed in the literature. This paper presents a software bug prediction model based on machine learning (ML) algorithms. Three supervised ML algorithms have been used to predict future software faults based on historical data. These classifiers are Naïve Bayes (NB), Decision Tree (DT), and Artificial Neural Networks (ANNs). The evaluation process showed that ML algorithms can be used effectively with a high accuracy rate. Furthermore, a comparison measure is applied to compare the proposed prediction model with other approaches. The collected results showed that the ML approach has a better performance.

CHAPTER 1
INTRODUCTION

CHAPTER 1

INTRODUCTION

1.1 Problem Statement:

Software Bug Prediction (SBP) is a critical process in software development and maintenance, that is concerned with the overall success of software. This is because predicting software faults earlier in the development process improves software quality, reliability, and efficiency reducing software costs. However, creating a robust bug prediction model is a difficult task, and numerous techniques have been proposed in the literature. This paper presents a machine learning (ML)-based software bug prediction model. Based on historical data, three supervised ML algorithms were used to predict future software faults. These classifiers include Nave Bayes (NB), Decision Trees (DT), and Artificial Neural Networks (ANN) (ANNs). The evaluation process demonstrated that ML algorithms can be used effectively and accurately.

1.2. Research Objective:

Identifying which Machine Learning (ML) algorithms have been used in previous studies of SDP is highly important for this master thesis to make sure that it contributes to the research community. [The purpose of this master thesis is to investigate if the prediction performance of metric sets is increased if test metrics are included in the set. By using algorithms used in other studies, [the effects of incorporating test metrics can be directly studied]. The prediction performance of the produced tool can also be made state of the art as lessons from other studies regarding prediction performance can be incorporated.

It is of major importance to recognize what kind of source code metrics that previously have been studied. This makes sure that the results of the study can be compared to those of other studies, and in turn, makes sure that the thesis contributes to the research community.

1.3 Project Scope and Objectives:

Project Scope:

Their one-phase model uses only previously fixed files as labels in the training process and therefore cannot be used to recommend files that have not been fixed before when being presented with a new bug report. Existing methods require runtime executions.

Objectives:

- To find the Accuracy of Bug Detection by using Machine Learning Algorithms.
- Prediction of Bugs occurring before the development of the Project.

CHAPTER 2
BACKGROUND WORK

CHAPTER 2

BACKGROUND WORK

2.1. Feature Identification: -

2.1.1. Introduction

Feature identification is a well-known technique to identify subsets of a program source code activated when exercising a functionality. Several approaches have been proposed to identify features. We present an approach to feature identification and comparison for large object-oriented multi-threaded programs using both static and dynamic data. We use processor emulation, knowledge filtering, and probabilistic ranking to overcome the difficulties of collecting dynamic data, i.e., imprecision and noise. We use model transformations to compare and visualize identified features. We compare our approach with a naive approach and a concept analysis-based approach using a case study on a real-life large object-oriented multi-threaded program, Mozilla, to show the advantages of our approach. We also use the case study to compare processor emulation with statistical profiling.

2.1.2. Merits, Demerits, and Challenges

Merits:

i. It shows the results based on Features present in the software.

ii. It helps in building Micro-Architectures based on Statistical Data and Dynamic Data.

Demerits:

i. Not accurate and low performance.

Challenges:

i. Performance should be improved.

ii. No accurate results.

2.1.3. Implementation:

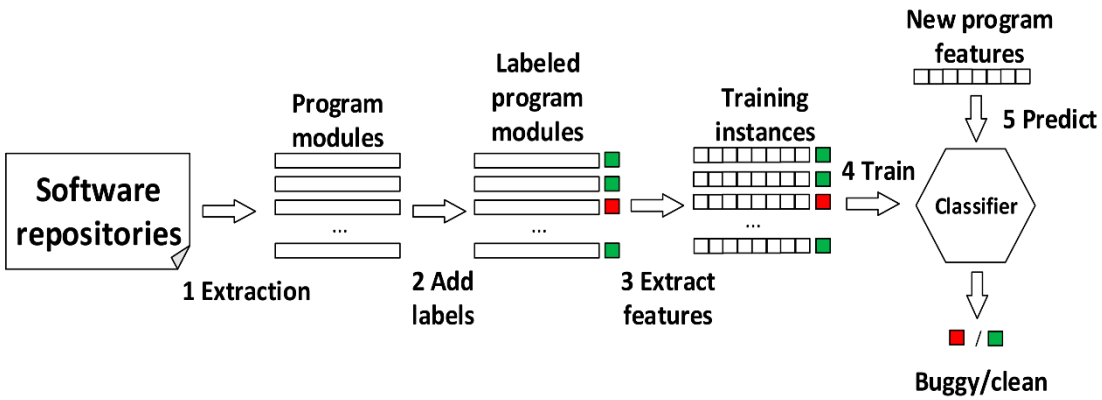

Fig:1 Feature Identification

- Several works propose feature identification techniques. However, few works attempt to compare features with one another.

- In their precursor work, Wilde and Scully propose a technique to identify features by analysing execution traces of test cases.

- They use two sets of test cases to build two execution traces: An execution trace where functionality is exercised; An execution trace where the functionality is not.

- Then, they compare execution traces to identify the feature associated with the functionality in the program.

- In their work, the authors only use dynamic data to identify features, no static analysis of the program is performed. In their work, the authors only use dynamic data to identify features, no static analysis of the program is performed.

	CM1		JM1		KC1		KC3		PC1	
	Accuracy%	ROC	Accuracy%	ROC	Accuracy%	ROC	Accuracy%	ROC	Accuracy%	ROC
FULL TRAINING										
LWL	88.08	0.81	81.66	0.7	84.64	0.8	85	0.78	92.23	0.87
J Star	100	1	95.13	0.98	96.13	0.98	97.5	0.99	99.21	1
IBK	100	1	99.03	0.99	98.52	0.99	99	0.99	99.21	1
Random Tree	100	1	99.03	0.99	98.52	0.99	99	0.99	99.21	1
Random Forest	98.84	0.99	98.23	0.99	98.89	0.98	98	0.99	98.81	0.99
10CV										
LWL	87.5	0.7	81.66	0.65	84.64	0.77	82.5	0.66	91.96	0.79
J Star	81.69	0.71	81.9	0.71	85.97	0.8	80.5	0.7	91.17	0.87
IBK	80.81	0.56	76.42	0.62	82.92	0.69	76.5	0.64	89.46	0.63
Random Tree	81.98	0.59	75.1	0.57	82.39	0.61	78	0.66	88.67	0.62
Random Forest	84.6	0.66	73.97	0.7	84.16	0.78	80.5	0.74	91.17	0.81
Percentage Split (66%)										
LWL	86.32	0.75	80.96	0.69	85.27	0.81	83.82	0.83	91.08	0.76
J Star	82.91	0.71	81.02	0.69	86.82	0.78	80.88	0.73	88.76	0.84
IBK	81.2	0.54	75.26	0.6	82.88	0.7	83.82	0.71	86.43	0.63

Table 1: Feature Identification

- This approach decomposes into a process and a set of tools to support the process.
- The process makes use of processor emulation, knowledge filtering, probabilistic ranking, and model transformations to support the analysis of large multi-threaded object-oriented programs and to deal with imprecision and noise.
- They use data collected from the realization of functionality, under different scenarios, to filter static data modeled as class diagrams, thus relating classes with features and with scenarios, and highlighting differences among features.
- Maintainers can use this approach to build and compare micro-architectures to precisely locate responsibilities and feature differences.

2.2. Debug Advisor:

2.2.1. Introduction

Debugging large software is difficult. Large systems have several tens of millions of lines of code. No single person understands all the code in such systems, and often new hires, who have little or no context about the code, are given the job of debugging failures. Our recent study with Microsoft's Windows Serviceability group revealed that developers and testers spend significant time during diagnosis looking for similar issues that have been resolved in the past. They search through bug databases, articles from the online Microsoft Developer Network (known as MSDN), email threads, and talk to colleagues to find this information. For example, the same bug or a very similar bug may have been encountered and fixed in another code branch, and the programmer would greatly benefit from knowing this information. Consequently, we decided to build a recommender system to improve the Productivity of debugging by automating the search for similar issues from the past. Prior work in mining software repositories such as Hipikat, and Fran provides us with inspiration and very useful ideas. However, there are important challenges in building a recommender system for debugging.

2.2.2. Merits, Demerits, and Challenges

Merits:

i) Our work mines software repositories to help programmers and testers during debugging.

ii) The presence of duplicate bugs in software repositories has received much attention.

Demerits:

i) It is not suitable for a few Software and does not show accurate predictions.

Challenges:

i. The system does not provide a perfect result in the existing work.

2.2.3. Implementation:

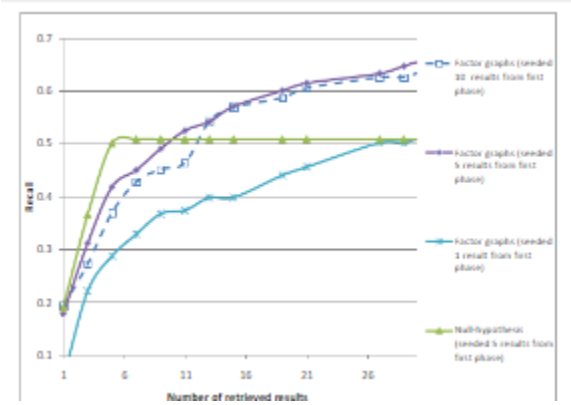

Fig:2 Implementation of Debug Advisor

- Large systems have several tens of millions of lines of code.
- No single person understands all the code in such systems, and often new hires, who have little or no context about the code, are given the job of debugging failures.
- For example, the same bug or a very similar bug may have been encountered and fixed in another code branch, and the programmer would greatly benefit from knowing this information.
- Consequently, we decided to build a recommender system to improve the productivity of debugging by automating the search for similar issues from the past.
- However, there are important challenges in building a recommender system for debugging.
- This Debug Advisor can work only to a particular extent. If we want to get more accurate results, then it is a little bit hard by using this Debug Advisor.

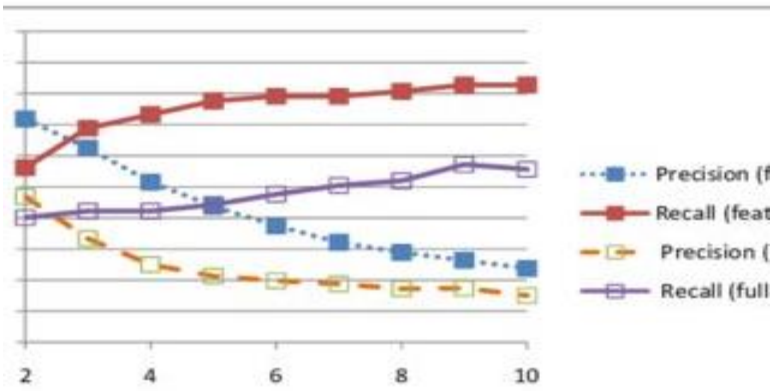

Fig:4 Results of Debug Advisor

There are two kinds of documents that we index for the first phase of Debug Advisor:
(1) bug records from the Windows bug databases (2) debug logs from actual debugging sessions that happen from stress breaks and from debug sessions that are done by developers and testers. For collecting debug logs, we have implemented a wrapper around the kernel debugger, which logs all commands issued by the user and responses from the debugger into a text file. The wrapper allows the developer to record the bug identifier (if it exists) for the current debugging sessions.

CHAPTER 3
PROPOSED SYSTEM

CHAPTER 3
PROPOSED SYSTEM

3.1 Objective of Proposed Model:

The proposed system is a software bug prediction model based on machine learning algorithms. It uses historical data to predict future software faults to improve software quality, reliability, and efficiency, and reduce software costs. The system uses three supervised machine learning classifiers: Naïve Bayes, Decision Tree, Random Forest, and Convolutional Neural Networks.

The evaluation process of the system showed high accuracy rates and effectiveness in predicting software faults. Additionally, a comparison measure was applied to compare the proposed prediction model with other approaches, and the results showed that the machine learning approach outperformed other methods.

Overall, the proposed system is a robust and effective way to predict software bugs and improve the software development and maintenance process. It can be used by software developers and maintenance teams to enhance the quality, reliability, and efficiency of software systems while reducing costs.

ADVANTAGES OF THE PROPOSED SYSTEM:

The proposed system of software bug prediction based on machine learning algorithms has several advantages, including:

- Improved software quality: By predicting software bugs in advance, the proposed system can help software developers to identify and fix the issues before the software is deployed. This can lead to a higher quality software product that meets user requirements and expectations.

- Cost savings: Identifying and fixing software bugs can be time-consuming and expensive. By predicting bugs and addressing them early in the development process, the proposed system can help reduce the overall cost of software development and maintenance.

- Faster development: With the help of the proposed system, software developers can identify and fix bugs earlier in the development cycle, which can help accelerate the development process and reduce time to market.

- Scalability: The proposed system is based on machine learning algorithms, which can learn and improve over time as more data becomes available. This means that the system can be scaled up to handle larger and more complex software projects.

Overall, the proposed system offers a range of advantages that can help software development teams to improve their software quality, reliability, and efficiency while reducing costs and time to market.

3.2 Algorithms Used for Proposed Model:

Naïve Bayes Algorithm: -

- Naive Bayes is a simple and effective machine learning algorithm that is commonly used for classification tasks.

- The algorithm is based on Bayes' theorem, which states that the probability of a hypothesis (in this case, a classification label) given the evidence (input features) is proportional to the probability of the evidence given the hypothesis.

- Naive Bayes assumes that the input features are independent of each other, which means that the algorithm can be trained quickly and with relatively small amounts of data.

- The algorithm works by calculating the probability of each possible classification label for a given set of input features and then choosing the label with the highest probability.

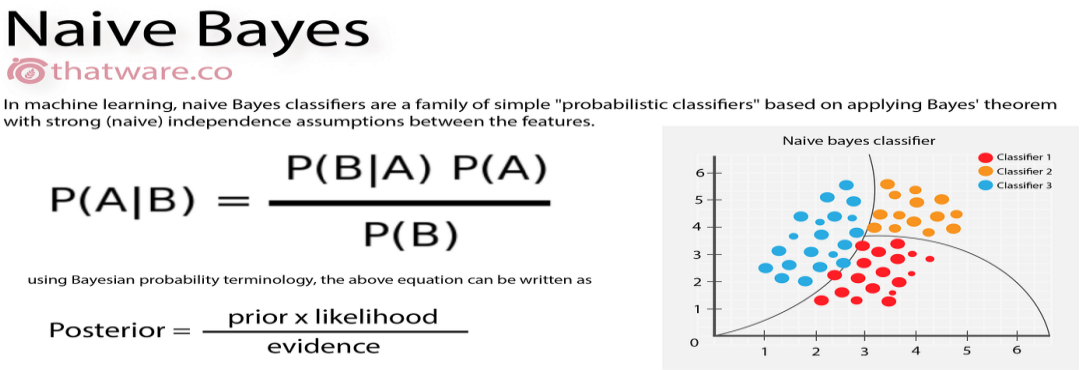

Fig:4 Naïve Bayes Algorithm

How does KNN work?

Step 1 - Import basic libraries.

You can use the below command for importing the basic libraries required.

```
# Importing basic libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Step 2 - Import the dataset.

Using the below code, import the dataset, which is required.

```
# Importing the dataset
dataset = pd.read_csv('Social_Media_Ads.csv')
X = dataset.iloc[:, [3, 4]]
y = dataset.iloc[:, 5]
print("Prediction evidence:\n", X.head())
print("\nFinal Target:\n", y.head())
```

Step 3 - Data pre-processing

The below command will help you with the data preprocessing.

```
# Conversion of variables into arrays
X = X.values
y = y.values

# Dataset splitting into training and test datasets(70:30)
from sklearn.selection_of_model import splitting_of_train_test_dataset
X_train, X_test, y_train, y_test = splitting_of_train_test_dataset(X, y, test_size = 0.30)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.transform_fit(X_train)
X_test = sc.transform(X_test)
```

In this step, you have to split the dataset into a training dataset (70%) and a testing dataset (30%). Next, you have to do some basic feature scaling with the help of a standard scaler. It will transform the dataset in a way where the mean value will be 0, and the standard deviation will be 1.

Step 4 - Training the model.

You should then write the following command for training the model.

```
# Fitting of Naive Bayes Algorithm to the Training Dataset

from sklearn.naive_bayes_algorithm import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

Step 5 - Testing and evaluation of the model

The code for testing and evaluating the model is as below:

```
# Prediction of the test dataset outcomes
y_pred = classifier.predict(X_test)

# Constructing the confusion matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True)
```

A confusion matrix helps to understand the quality of the model. It describes the production of a classification model on a set of test data for which you know the true values. Every row in a confusion matrix portrays an actual class, and every column portrays the predicted class.

Step 6 - Visualizing the model.

Finally, the below code will help in visualizing the model.

```
# Visualizing the test dataset results

from matplotlib.colors import ColormapListed
X_datsetset, y_datasetset = X_test, y_test
X1, X2 = np.meshgrid(np.arrange(start = X_dataset[:,0].min()-1, stop = X_dataset[:. 0].max() + 1, step =
np.arrange(start = X_dataset[:, 1[.min() -1, stop = X_dataset[:, 1].max() +1, step = 0.02))
plt.contourf(X1, X2, Classifier.predict(np.array([X1.ravel(), X2.ravel()].T).rescape(X1.shape),
alpha = 0.3, cmap = ColormapListed((‘yellow’, ‘blue’)))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for u, v in enumerate(np.unique(y_set)):
plt.scatter(X_dataset[y_dataset == v, 0], X_dataset[y_dataset== v, 1],
c = ColormapListed((‘yellow’,’blue’))(i), label = v)
plt.xlabel(‘Current_age’)
plt.ylabel(‘Gross_salary’)
plt.legend()
plt.show()
```

Decision Tree: -

- A decision tree is a popular supervised learning algorithm used for both classification and regression tasks. It works by partitioning the input space into regions based on the values of the input features and assigning a label or value to each region.

- Each node in the decision tree represents a decision based on the value of one of the input features. The branches emanating from each node correspond to the possible values of that feature and lead to child nodes that represent further decisions or final predictions.

- Decision trees can handle both categorical and continuous input features and can handle missing values. They are also easy to interpret and explain, making them useful for understanding the decision-making process of the model.

- Decision trees have been used in a wide range of applications, such as predicting credit risk, diagnosing medical conditions, and detecting fraud. They are often used in conjunction with other machine-learning algorithms to improve overall performance.

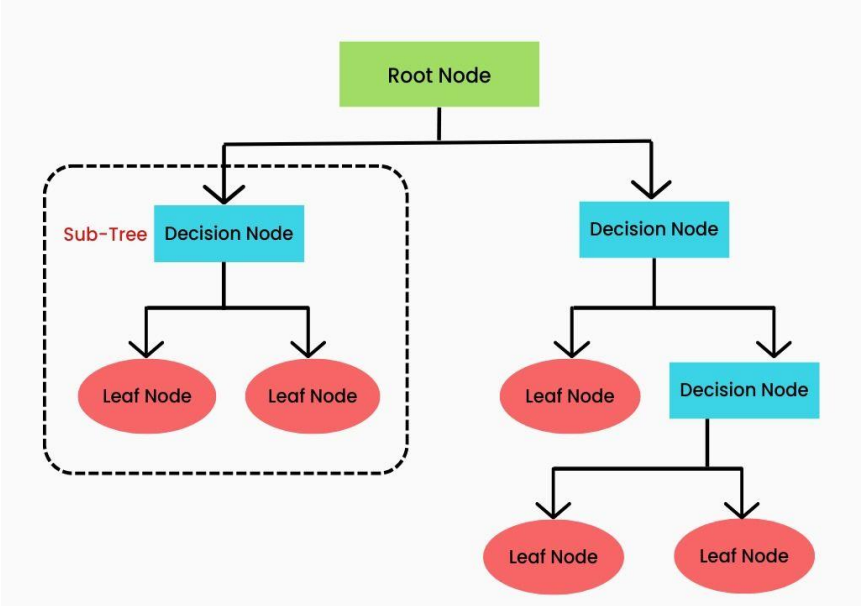

Fig:4 Decision Tree

How Decision Tree Works: -

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of the root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and moves further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm:

- **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.
- **Step-2:** Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
- **Step-3:** Divide the S into subsets that contain possible values for the best attributes.
- **Step-4:** Generate the decision tree node, which contains the best attribute.
- **Step-5:** Recursively makes new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Advantages of Decision Tree: -

- Decision trees are easy to understand and interpret, making them a popular choice for decision-making tasks. The tree structure provides a clear representation of the decision-making process, and the final predictions or classifications can be easily explained.
- Decision trees can handle both categorical and numerical data, making them a versatile tool for a wide range of applications. They can also handle missing values and outliers, making them robust to noisy data.
- Decision trees are computationally efficient, with a time complexity of O (n log n) for constructing the tree and O(log n) for making a prediction, where n is the number of data points. This makes them suitable for large datasets and real-time applications.

Random Forest: -

- Random Forest is a popular machine learning algorithm used for both classification and regression tasks. It is an ensemble method that combines multiple decision trees to improve accuracy and reduce overfitting.

- Random Forest works by training multiple decision trees on different subsets of the data and different subsets of the features. Each tree provides a prediction or classification, and the final output is the mode or means of the individual predictions or classifications.

- Random Forest can handle both numerical and categorical data and can handle missing values and outliers. It is also robust to noise and irrelevant features and can perform well even with high-dimensional data.

- Random Forest provides a way to estimate feature importance, which can help with feature selection and model interpretation. This can be used to identify the most important features for making predictions and can also help with understanding the underlying relationships in the data.

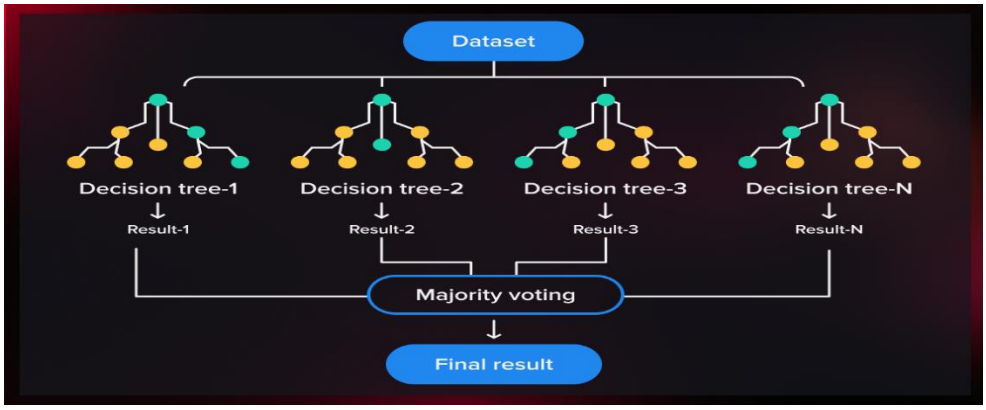

Fig:5 Random Forest

How Random Forest Works: -

Step 1: In the Random Forest model, a subset of data points and a subset of features is selected for constructing each decision tree. Simply put, n random records and m features are taken from the data set having k number of records.

Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on Majority Voting or Averaging for Classification and regression, respectively.

Advantages of Random Forest: -

- Random Forest is an ensemble algorithm that combines multiple decision trees, which reduces the variance and overfitting of individual trees, making it a powerful and accurate algorithm for both classification and regression tasks.

- Random Forest can handle both categorical and numerical data, and it can handle missing values, making it a versatile algorithm that can be applied to a wide range of real-world problems.

- Random forests can estimate feature importance, which can be useful for feature selection and understanding the underlying relationships in the data. This can help to improve the performance of the model and make it more interpretable.

Convolutional Neural Network (CNN): -

- CNNs can be trained on code snippets or other software artifacts to extract features that are indicative of the presence of bugs. These features can be learned automatically by the network during training.

- CNNs can be used to predict whether a given code snippet is likely to contain a bug, based on the learned features. This can be useful for detecting bugs early in the development process before they become more difficult and costly to fix.

- CNNs can be used in combination with other machine learning algorithms, such as Naive Bayes or Decision Trees, to improve the accuracy of bug prediction models.

- CNNs can be used to analyze large volumes of code quickly and efficiently, making them well-suited for bug prediction tasks in large software projects.

- CNNs can be trained on a variety of different software artifacts, including code snippets, commit messages, and bug reports, to improve the accuracy and robustness of bug prediction models.

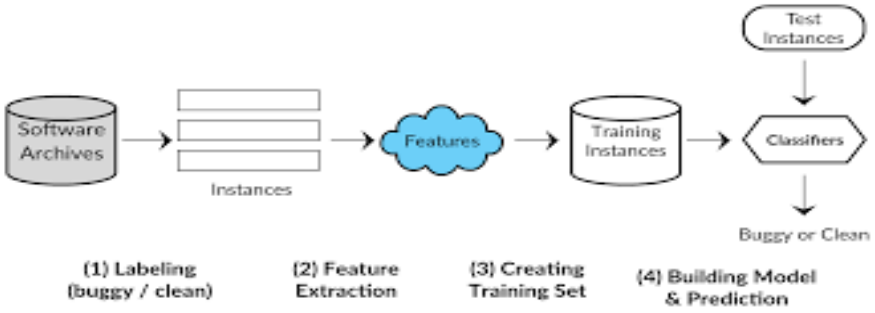

Fig:5 Architecture of CNN

How CNN Works: -

Data Preparation:

- Collect a dataset of software artifacts, such as code snippets, commit messages, and bug reports.
- Label each artifact as containing a bug (positive) or not containing a bug (negative).
- Divide the dataset into training, validation, and testing sets.

Feature Extraction:

- Pre-process the input data (e.g., tokenize and vectorize code snippets).
- Design a CNN architecture with multiple convolutional layers, pooling layers, and fully connected layers to automatically extract features from the input data.
- Train the CNN to optimize the weights and biases of its layers using the training set.
- Use the validation set to tune the hyperparameters of the CNN (e.g., learning rate, number of epochs, dropout rate).

Testing:

- Evaluate the performance of the trained CNN on the testing set to obtain metrics such as accuracy, precision, recall, and F1 score.

Bug Prediction:

- Given a new software artifact, pre-process it in the same way as the training data.
- Feed the pre-processed artifact into the trained CNN and observe the output of the final layer, which corresponds to the predicted label (positive or negative).

3.3 Stepwise Implementation and Code

1. Collect and label a dataset of software artifacts
2. Divide the dataset into training, validation, and testing sets
3. Pre-process the input data
4. Design a CNN architecture with multiple layers
5. Train the CNN using the training set
6. Tune the hyperparameters of the CNN using the validation set
7. Evaluate the performance of the trained CNN on the testing set
8. Given a new software artifact, pre-process it in the same way as the training data
9. Feed the pre-processed artifact into the trained CNN and observe the output of the final layer
10. Return the predicted label (positive or negative)

Note that the specific implementation details of each step may vary depending on the software bug prediction task at hand. Additionally, other machine learning algorithms can be used in combination with CNNs to further improve the accuracy and robustness of bug prediction models.

Code: -

```
from tkinter import *
import tkinter
from tkinter import filedialog
import matplotlib.pyplot as plt
from tkinter.filedialog import askopenfilename
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.preprocessing import normalize
from sklearn.naive_bayes import BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.svm import SVC
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

```
main = tkinter.Tk()
main.title("software Bug Prediction with Machine Learning Methods")
main.geometry("1200x1200")
global filename
global X, Y
global X_train, X_test, y_train, y_test
global dataset
accuracy = []
precision = []
recall = []
fscore = []
global X, Y
global X_train, X_test, y_train, y_test
def plotPerColumnDistribution(df, nGraphShown, nGraphPerRow):
    nunique = df.nunique()
    df = df[[col for col in df if nunique[col] > 1 and nunique[col] < 50]] # For displaying purposes,
pick columns that have between 1 and 50 unique values
    nRow, nCol = df.shape
    columnNames = list(df)
    nGraphRow = (nCol + nGraphPerRow - 1) / nGraphPerRow
    plt.figure(num = None, figsize = (6 * nGraphPerRow, 8 * nGraphRow), dpi = 80, facecolor
= 'w', edgecolor = 'k')
    for i in range(min(nCol, nGraphShown)):
        plt.subplot(nGraphRow, nGraphPerRow, i + 1)
        columnDf = df.iloc[:, i]
        if (not np.issubdtype(type(columnDf.iloc[0]), np.number)):
            valueCounts = columnDf.value_counts()
```

```
            valueCounts.plot.bar()
        else:
            columnDf.hist()
        plt.ylabel('counts')
        plt.xticks(rotation = 90)
        plt.title(f'{columnNames[i]} (column {i})')
    plt.tight_layout(pad = 1.0, w_pad = 1.0, h_pad = 1.0)
    plt.show()
def uploadDataset():
    global filename
    global dataset
    filename = filedialog.askopenfilename(initialdir="Dataset")
    pathlabel.config(text=filename)
    text.delete('1.0', END)
    text.insert(END,filename+" loaded\n\n");
    dataset = pd.read_csv(filename)
    text.insert(END,str(dataset.head))
    plotPerColumnDistribution (dataset, 40, 5)
def preprocess ():
    text.delete('1.0', END)
    global X, Y
    fig, ax = plt.subplots()
    sns.lineplot(data=dataset.isnull().sum())
    fig.autofmt_xdate()
```

```
    dataset.fillna(0, inplace = True)
    cols = ['QualifiedName','Name','Complexity','Coupling','Size','Lack of Cohesion']
    le = LabelEncoder()
    dataset[cols[0]] = pd.Series(le.fit_transform(dataset[cols[0]].astype(str)))
    dataset[cols[1]] = pd.Series(le.fit_transform(dataset[cols[1]].astype(str)))
    dataset[cols[2]] = pd.Series(le.fit_transform(dataset[cols[2]].astype(str)))
    dataset[cols[3]] = pd.Series(le.fit_transform(dataset[cols[3]].astype(str)))
    dataset[cols[4]] = pd.Series(le.fit_transform(dataset[cols[4]].astype(str)))
    dataset[cols[5]] = pd.Series(le.fit_transform(dataset[cols[5]].astype(str)))
Y = dataset.values[:,2]
  dataset.drop(['Complexity'], axis = 1,inplace=True)
  X = dataset.values
  X = normalize(X)
  text.insert(END,str(X)+"\n")
  plt.show()
def featureSelection():
    global X, Y
    text.delete('1.0', END)
    global X_train, X_test, y_train, y_test
    text.insert(END,"Total features found in dataset before applying feature selection algorithm
= "+str(X.shape[1])+"\n")
    pca = PCA(n_components = 30)
    X = pca.fit_transform(X)
    text.insert(END,"Total features found in dataset after applying feature selection algorithm
= "+str(X.shape[1])+"\n")
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
    text.insert(END,"Total records found in dataset are : "+str(X.shape[0])+"\n")
    text.insert(END,"Total records used to train machine learning algorithms are :
"+str(X_train.shape[0])+"\n")
```

```
    text.insert(END,"Total records used to test machine learning algorithms are  :
"+str(X_test.shape[0])+"\n")
    plt.figure(figsize=(75,75))
    sns.heatmap(dataset.corr(), annot = True)
    plt.show()
def runML():
    text.delete('1.0', END)
    global X_train, X_test, y_train, y_test
    accuracy.clear()
    precision.clear()
    fscore.clear()
    recall.clear()
    cls = BernoulliNB(binarize=0.0)
    cls.fit(X_train, y_train)
    predict = cls.predict(X_test)
    p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Bernoulli Naive Bayes Accuracy  : '+str(a)+"\n")
    text.insert(END,'Bernoulli Naive Bayes Precision : '+str(p)+"\n")
    text.insert(END,'Bernoulli Naive Bayes Recall    : '+str(r)+"\n")
    text.insert(END,'Bernoulli Naive Bayes FMeasure  : '+str(f)+"\n\n")
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    fscore.append(f)
    cls = DecisionTreeClassifier()
    cls.fit(X_train, y_train)
```

```
    predict = cls.predict(X_test)
    p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Decision Tree Accuracy  : '+str(a)+"\n")
    text.insert(END,'Decision Tree Precision : '+str(p)+"\n")
    text.insert(END,'Decision Tree Recall    : '+str(r)+"\n")
    text.insert(END,'Decision Tree FMeasure  : '+str(f)+"\n\n")
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    cls = RandomForestClassifier()
    cls.fit(X_train, y_train)
    predict = cls.predict(X_test)
    p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Random Forest Accuracy  : '+str(a)+"\n")
    text.insert(END,'Random Forest Precision : '+str(p)+"\n")
    text.insert(END,'Random Forest Recall    : '+str(r)+"\n")
    text.insert(END,'Random Forest FMeasure  : '+str(f)+"\n\n")
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    fscore.append(f)
cls = LogisticRegression()
    cls.fit(X_train, y_train)
    predict = cls.predict(X_test)
    p = precision_score(y_test, predict,average='macro') * 100
```

```
    r = recall_score(y_test, predict,average='macro') * 100
f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Logistic Regression Accuracy  : '+str(a)+"\n")
    text.insert(END,'Logistic Regression Precision : '+str(p)+"\n")
    text.insert(END,'Logistic Regression Recall    : '+str(r)+"\n")
    text.insert(END,'Logistic Regression FMeasure  : '+str(f)+"\n\n")
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    fscore.append(f)
cls = BaggingClassifier(base_estimator=SVC(), n_estimators=1, random_state=0)
    cls.fit(X_test, y_test)
    predict = cls.predict(X_test)
    p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Bagging Classifier Accuracy  : '+str(a)+"\n")
    text.insert(END,'Bagging Classifier Precision : '+str(p)+"\n")
    text.insert(END,'Bagging Classifier Recall    : '+str(r)+"\n")
    text.insert(END,'Bagging Classifier FMeasure  : '+str(f)+"\n\n")
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    fscore.append(f)
cls = GradientBoostingClassifier()
    cls.fit(X_test, y_test)
    predict = cls.predict(X_test)
```

```
p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    a = accuracy_score(y_test,predict)*100
    text.insert(END,'Gradient Boosting Accuracy  : '+str(a)+"\n")
    text.insert(END,'Gradient Boosting Precision : '+str(p)+"\n")
    text.insert(END,'Gradient Boosting Recall    : '+str(r)+"\n")
    text.insert(END,'Gradient Boosting FMeasure  : '+str(f)+"\n\n")
accuracy.append(a)
  precision.append(p)
  recall.append(r)
  fscore.append(f)
def runCNN():
    global X, Y
    Y1 = to_categorical(Y)
    X_train1, X_test1, y_train1, y_test1 = train_test_split(X, Y1, test_size=0.2)
    cnn_model = Sequential()
    cnn_model.add(Dense(512, input_shape=(X_train.shape[1],)))
    cnn_model.add(Activation('relu'))
    cnn_model.add(Dropout(0.3))
    cnn_model.add(Dense(512))
    cnn_model.add(Activation('relu'))
    cnn_model.add(Dropout(0.3))
cnn_model.add(Dense(6))
    cnn_model.add(Activation('softmax'))
    cnn_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    acc_history = cnn_model.fit(X_test1, y_test1, epochs=10, validation_data=(X_test1,
y_test1))
)
```

```
    print(cnn_model.summary())
  predict = cnn_model.predict(X_test1)
  predict = np.argmax(predict, axis=1)
  testY = np.argmax(y_test1, axis=1)
  acc_history = acc_history.history
  acc_history = acc_history['accuracy']
  acc = acc_history[9] * 100
  p = precision_score(testY, predict,average='macro') * 100
  r = recall_score(testY, predict,average='macro') * 100
  f = f1_score(testY, predict,average='macro') * 100
  text.insert(END,'CNN Accuracy  : '+str(acc)+"\n")
  text.insert(END,'CNN Precision : '+str(p)+"\n")
  text.insert(END,'CNN Recall    : '+str(r)+"\n")
  text.insert(END,'CNN FMeasure  : '+str(f)+"\n\n")
  accuracy.append(acc)
  precision.append(p)
  recall.append(r)
  fscore.append(f)
def graph():
  df = pd.DataFrame([['Naive Bayes','Precision',precision[0]],['Naive
Bayes','Recall',recall[0]],['Naive Bayes','F1 Score',fscore[0]],['Naive
Bayes','Accuracy',accuracy[0]],
                   ['Decision Tree','Precision',precision[1]],['Decision
Tree','Recall',recall[1]],['Decision Tree','F1 Score',fscore[1]],['Decision
Tree','Accuracy',accuracy[1]],
                   ['Random Forest','Precision',precision[2]],['Random
Forest','Recall',recall[2]],['Random Forest','F1 Score',fscore[2]],['Random
Forest','Accuracy',accuracy[2]],
```

```
                ['Logistic Regression','Precision',precision[3]],['Logistic
Regression','Recall',recall[3]],['Logistic Regression','F1 Score',fscore[3]],['Logistic
Regression','Accuracy',accuracy[3]],
                ['Bagging Classifier','Precision',precision[4]],['Bagging
Classifier','Recall',recall[4]],['Bagging Classifier','F1 Score',fscore[4]],['Bagging
Classifier','Accuracy',accuracy[4]],
                ['Gradient Boosting','Precision',precision[5]],['Gradient
Boosting','Recall',recall[5]],['Gradient Boosting','F1 Score',fscore[5]],['Gradient
Boosting','Accuracy',accuracy[5]],
                ['CNN','Precision',precision[6]],['CNN','Recall',recall[6]],['CNN','F1
Score',fscore[6]],['CNN','Accuracy',accuracy[6]],
                ],columns=['Parameters','Algorithms','Value'])
   df.pivot("Parameters", "Algorithms", "Value").plot(kind='bar')
   plt.show()
font = ('times', 15, 'bold')
title = Label(main, text='Software bug Prediction with Machine Learning Methods')
title.config(bg='brown', fg='white')
title.config(font=font)
title.config(height=3, width=120)
title.place(x=5,y=5)
font1 = ('times', 13, 'bold')
uploadButton = Button(main, text="Upload Dataset", command=uploadDataset)
uploadButton.place(x=50,y=100)
uploadButton.config(font=font1)
pathlabel = Label(main)
pathlabel.config(bg='brown', fg='white')
pathlabel.config(font=font1)
pathlabel.place(x=600,y=100)
```

```
 processButton = Button(main, text="Preprocess Dataset", command=preprocess)
processButton.place(x=350,y=100)
processButton.config(font=font1)

fsButton = Button(main, text="Features Selection Algorithms", command=featureSelection)
fsButton.place(x=50,y=150)
fsButton.config(font=font1)

mlButton = Button(main, text="Run Machine Learning Algorithms", command=runML)
mlButton.place(x=350,y=150)
mlButton.config(font=font1)

cnnButton = Button(main, text="Run CNN Algorithm", command=runCNN)
cnnButton.place(x=50,y=200)
cnnButton.config(font=font1)
graphButton = Button(main, text="Comparison Graph", command=graph)
graphButton.place(x=350,y=200)
graphButton.config(font=font1)
font1 = ('times', 12, 'bold')
text=Text(main,height=20,width=90)
scroll=Scrollbar(text)
text.configure(yscrollcommand=scroll.set)
text.place(x=10,y=250)
text.config(font=font1)
main.config(bg='brown')
main.mainloop()
```

CHAPTER 4
RESULTS AND DISCUSSION

CHAPTER 4
RESULTS AND DISCUSSION

4.1. Comparison of Existing Solutions:

There are many studies about software bug prediction using machine learning techniques. For example, the study in [2] proposed a linear Auto-Regression (AR) approach to predict faulty modules. The study predicts the software's future faults depending on the historical data of the software's accumulated faults. The study also evaluated and compared the AR model with the Known power model (POWM) using Root Mean Square Error (RMSE) measure. In addition, the study used three datasets for evaluation and the results were promising. The studies in [1], and [2] analyzed the applicability of various ML methods for fault prediction. The most important previous research about each ML technique and the current trends in software bug prediction using machine learning. This study can be used as ground or step to prepare for future work in software bug prediction. Malhotra Ruchika in [9] presented a good systematic review of software bug prediction techniques, using Machine Learning (ML). The paper included a review of all the studies between the period 1991 and 2013, analyzed the ML techniques for software bug prediction models, assessed their performance, compared ML and statistic techniques, compared different ML techniques, and summarized the strength and weaknesses of the ML techniques.

In [10], the paper provided a benchmark to allow for common and useful comparisons between different bug prediction approaches. The study presented a comprehensive comparison between a well-known bug prediction approach, also introduced a new approach, and evaluated its performance by building a good comparison with other approaches using the presented benchmark.

The study in [9] assessed various object-oriented metrics by used machine learning techniques (decision tree and neural networks) and statistical techniques (logical and linear regression). The results of the study showed that the Coupling Between Object (CBO) metric is the best metric to predict the bugs in the class and the Line of Code (LOC) is fairly well, but the Depth of Inheritance Tree (DIT) and Number of Children (NOC) are untrusted metrics.

Singh and Chug [9] discussed five popular ML algorithms used for software defect prediction i.e, Artificial Neural Networks (ANNs), Particle Swarm Optimization (PSO), Decision Tree (DT), Naïve Bayes (NB) and Linear Classifiers (LC). The study presented important results including that the ANN has lowest error rate followed by DT, but the linear classifier is better than other algorithms in term of defect prediction accuracy, the most popular methods used in software defect prediction are: DT, BL, ANN, SVM, RBL and EA, and the common metrics used in software defect prediction studies are: Line Of Code (LOC) metrics, object oriented metrics such as cohesion, coupling and inheritance, also other metrics called hybrid metrics which used both object oriented and procedural metrics, furthermore the results showed that most software defect prediction studied used NASA dataset and PROMISE dataset.

Moreover, the studies in [08], [9] discussed various ML techniques and provided the ML capabilities in software defect prediction. The studies assisted the developer to use useful software metrics and suitable data mining technique in order to enhance the software quality. The study in determined the most effective metrics which are useful in defect prediction such as Response for class (ROC), Line of code (LOC) and Lack of Coding Quality (LOCQ). Bavisi et al. presented the most popular data mining technique (k-Nearest Neighbors, Naïve Bayes, C-4.5 and Decision trees). The study analyzed and compared four algorithms and discussed the advantages and disadvantages of each algorithm. The results of the study showed that there were different factors affecting the accuracy of each technique; such as the nature of the problem, the used dataset and its performance matrix.

4.2. Data Collection and Performance Metrics

The used datasets in this study are three different datasets, namely DS1, DS2, and DS3. All datasets are consisting of two measures: the number of faults (Fi) and the number of test workers (Ti) for each day (Di) in a part of the software project's lifetime. The DS1 dataset has 46 measurements that are involved in the testing process presented in [1]. DS2, also taken from [1], measured a system fault during 109 successive days of testing the software system that consists of 200 modules with each having a one-kilo line of code of Fortran. DS2 has 111 measurements. DS3 is developed in [1], which contains real measured data for a test/debug program of a real-time control application presented in. Tables I to III present DS1, DS2, and DS3, respectively. The datasets were preprocessed by a proposed clustering technique. The proposed clustering technique marks the data with class labels. These labels are set to classify the number of faults into five different classes; A, B, C, D, and E. Table IV shows the value of each class and the number of instances that belong to it in each dataset. To evaluate the performance of using ML algorithms in software bug prediction, we used a set of well-known measures [10] based on the generated confusion matrixes. The following subsections describe the confusion matrix and the used evaluation measures.

Table: 2 The First Software Faults Dataset

D_i	F_i	T_i	D_i	F_i	T_i
1	2	75	24	2	8
2	0	31	25	1	15
3	30	63	26	7	31
4	13	128	27	0	1
5	13	122	28	22	57
6	3	27	29	2	27
7	17	136	30	5	35
8	2	49	31	12	26
9	2	26	32	14	36
10	20	102	33	5	28
11	13	53	34	2	22
12	3	26	35	0	4
13	3	78	36	7	8
14	4	48	37	3	5
15	4	75	38	0	27
1`6	0	14	39	0	6
17	0	4	40	0	6
18	0	14	41	0	4
19	0	22	42	5	0
20	0	5	43	2	6
21	0	9	44	3	5
22	30	33	45	0	8
23	15	118	46	0	2

A. Confusion Matrix:

The confusion matrix is a specific table that is used to measure the performance of ML algorithms. Table V shows an example of a generic confusion matrix. Each row of the matrix represents the instances in an actual class, while each column represents the instance in a predicted class or vice versa. Confusion matrix summarizes the results of the testing algorithm and provides a report of the number of True Positive (TP), False Positives (FP), True Negatives TN), and False Negatives (FN)

B. Accuracy:

Accuracy (ACC) is the proportion of true results (both TP and TN) among the total number of examined instances. The best accuracy is 1, whereas the worst accuracy is 0. ACC can be computed by using the following formula:

$$ACC = (TP + TN) / (TP + TN+ FP + FN)$$

C. Precision (Positive Predictive Value):

Precision is calculated as the number of correct positive predictions divided by the total number of positive predictions. The best precision is 1, whereas the worst is 0 and it can be calculated as:

$$Precision = TP / (TP + FP)$$

D. Recall (True Positive or Sensitivity):

Recall is calculated as the number of positive predictions divided by the total number of positives. The best recall is 1, whereas the worst is 0. Generally, Recall is calculated by the following formula:

$$Recall = TP / (TP + FN)$$

E. F-Measure:

F-measure is defined as the weighted harmonic mean of precision and recall. Usually, it is used to combine the Recall and Precision measures in one measure in order to compare different ML algorithms with each other. Fmeasure formula is given by:

$$F\text{- measure} = (2 * Recall * Precision) / (Recall + Precision)$$

F. Root Mean Square Error:

RMSE is a measure for evaluating the performance of a prediction model. The idea herein is to measure the difference between the predicted and the actual values. If the actual value is X and the predicted value is XP then RMSE is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n} * \sum_{i=1}^{n} (Xi - XPi)^2}$$

Table 3: Represents the Average of Algorithms

Datasets	NB	DT	CNN
DS1	1	1	1
DS2	0.905	1	0.990
DS3	0.972	1	0.981
Average	**0.959**	**1**	**0.990**

The third evaluation measure is the recall measure. Table VIII shows the recall values for the three classifiers on the three datasets. Also, herein the ML algorithms achieved a good recall value. The best recall value was achieved by DT classifier, which is 100% in all datasets. On the other hand, the average recall values for ANNs and NB algorithms are 99% and 96%, respectively

In order to compare the three classifiers with respect to recall and precision measures, we used the F-measure value. Fig. 1 shows the F-measure values for the used ML algorithms in the three datasets. As shown the figure, DT has the highest F-measure value in all datasets followed by CNN, then NB classifiers. Finally, to evaluate the ML algorithms with other approaches, we calculated the RMSE value. The work in [2] proposed a linear Auto Regression (AR) model to predict the accumulative number of software faults using historical measured faults. They evaluated their approach with the POWM model based on the RMSE measure. The evaluation process was done on the same datasets we are using in this study.

4.3 Screenshots:

Fig: 8. UI of our Project

- The UI of the project is designed in such a way that it's user-friendly and all the functionalities are labeled to perform various operations over a Software project.

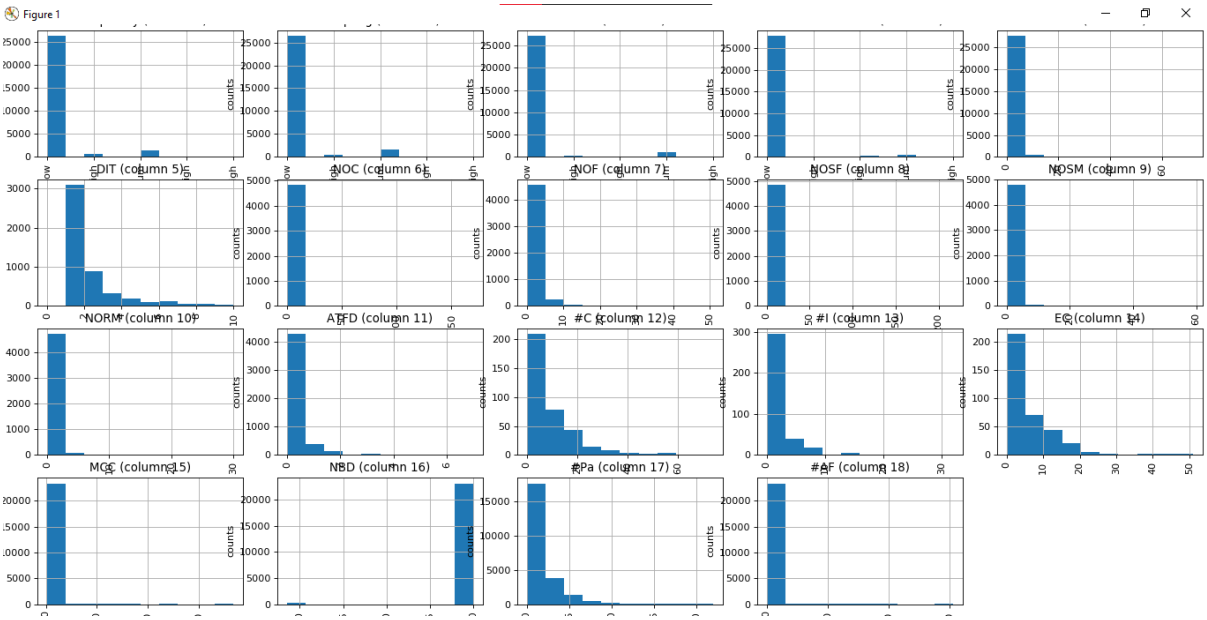

Fig: 9. Evaluation of Dataset based on different attributes

- For the Evaluation process of the dataset we need to consider few attributes that which are provided in the Dataset and it will show us the evaluation of those selected attributes in a dataset in the form pf Bar-Graphs.

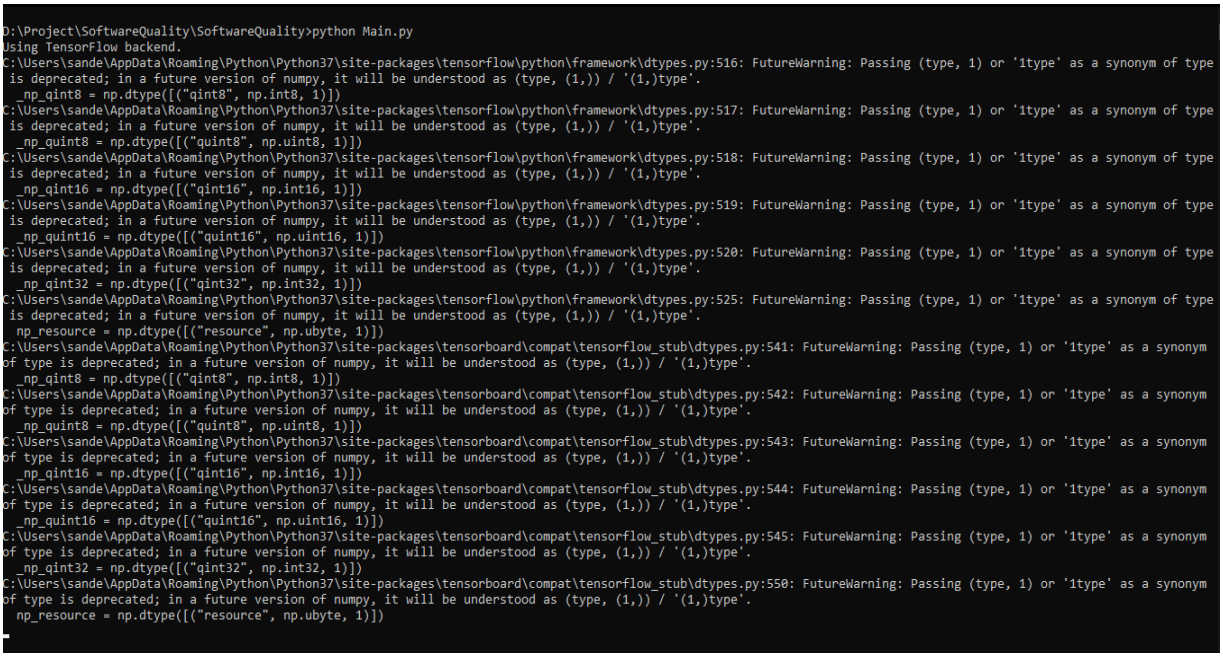

Fig: 10. Show the UI of the Command line

- The Above Figure shows the UI of Command line where we need to run and install all the required packages that makes the UI Functional and will be ready for Execution.

The List of Commands that me need to Run in the Command line:

- pip install matplotlib==3.1.3 –user
- pip install pandas==0.25.3 –user
- pip install seaborn –user
- pip install scikit-learn==0.22.2. post1 –user
- pip install keras==2.3.1 –user
- pip install tensorflow==1.14.0 –user
- pip install protobuf==3.20.0 –user

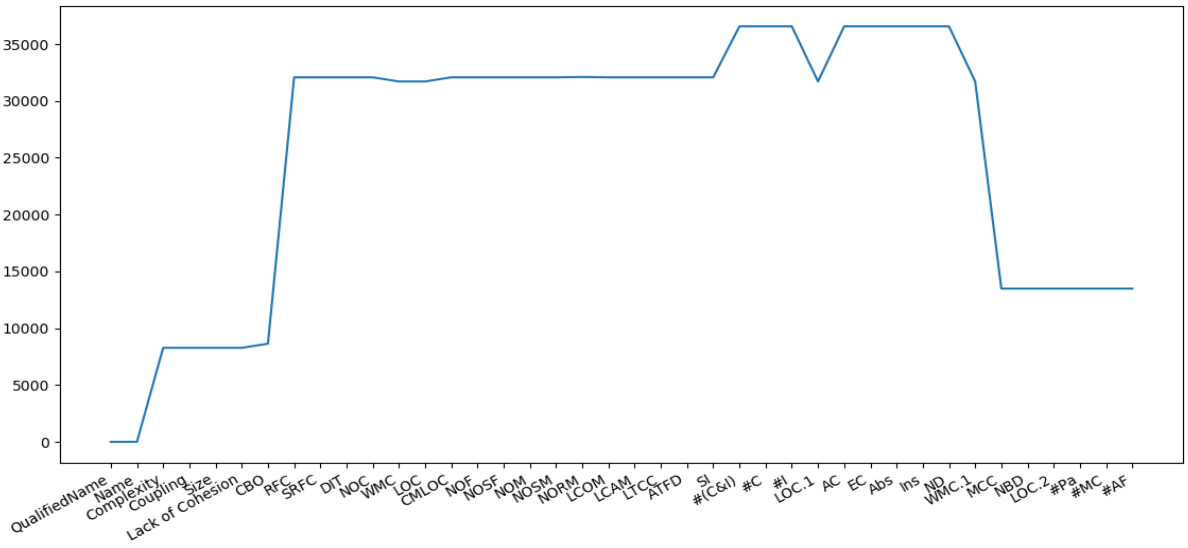

Fig: 11. Describes the Bugs count Based on the selected attributes

- After Pre-processing the data it will generates a curved line based on the selected attributes it will represents the bug count in a Dataset.

```
[[9.99973766e-01 7.24343450e-03 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [9.99965496e-01 8.30556795e-03 6.22139922e-05 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [9.89589580e-01 1.43918165e-01 6.08790883e-05 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [3.60120737e-01 9.32905705e-01 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [3.59564134e-01 9.33120375e-01 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [3.56895244e-01 9.34144413e-01 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]]
```

Fig: 12. Shows the output of the above generated Graph

Fig:13. Show the Removal of unwanted Data from the Dataset

- The Feature Selection Algorithm will help us with the Removal of unwanted Data from the Dataset after Pre-Processing the Dataset and also it will provide a count of Total No. of Records found in the Dataset an also No. of Record that which are used for Training and Testing the Machine Learning algorithms.

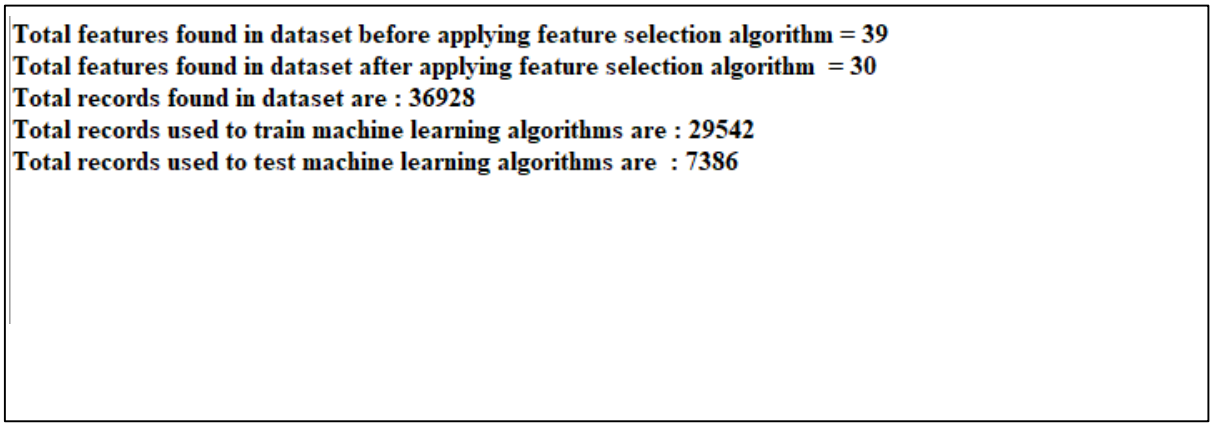

Fig: 14. Shows the Result after Running the Feature Selection Algorithm

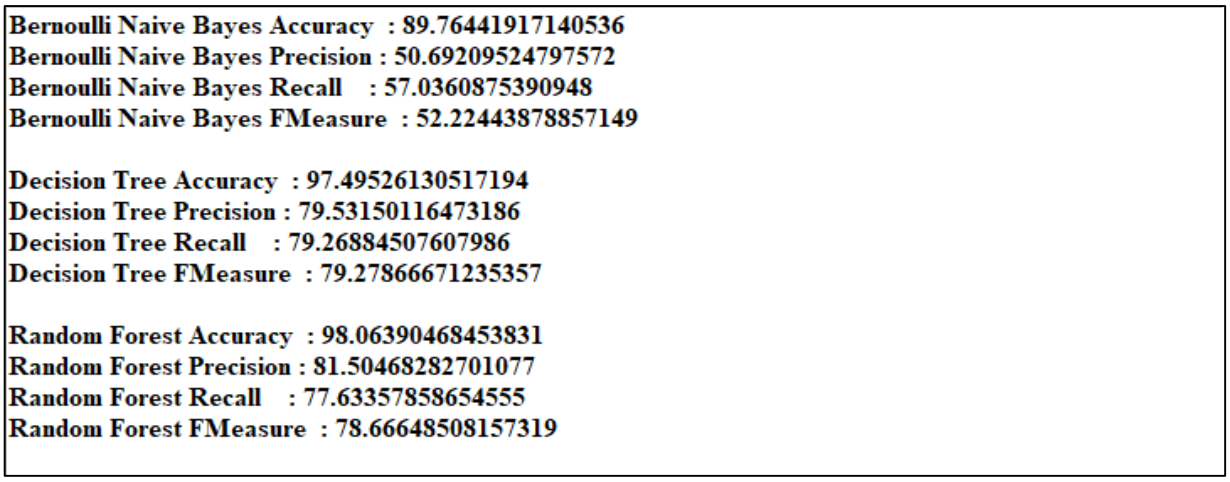

Fig:15. Shows the Results obtained after Running Machine learning algorithms

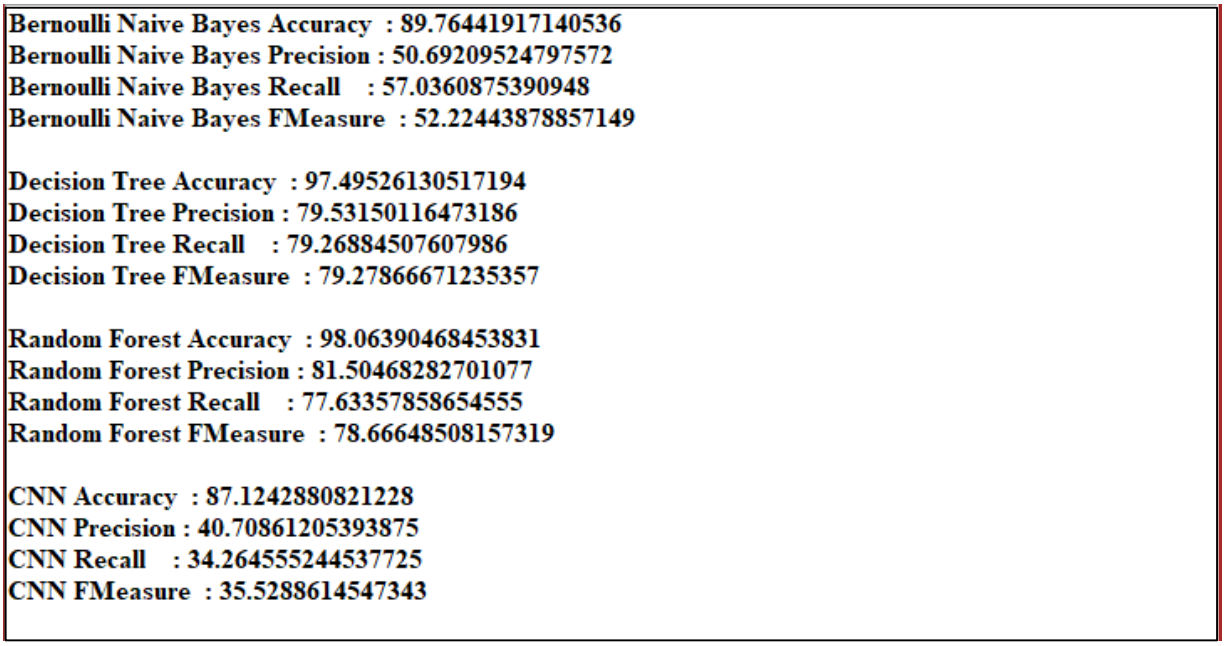

Fig:16. Shows the Result obtained after Running CNN Algorithm

CHAPTER 5
CONCLUSION

CHAPTER 5
CONCLUSION

5.1 Conclusion

To locate a bug, developers use not only the content of the bug report but also domain knowledge relevant to the software project. We introduced a learning-to-rank approach that emulates the bug-finding process employed by developers. The ranking model characterizes useful relationships between a bug report and source code files by leveraging domain knowledge, such as API specifications, the syntactic structure of code, or issue tracking data. Experimental evaluations on six Java projects show that our approach can locate the relevant files within the top 10 recommendations for over 70 percent of the bug reports in Eclipse Platform and Tomcat. Furthermore, the proposed ranking model outperforms three recent state-of-the-art approaches. Feature evaluation experiments employing greedy backward feature elimination demonstrate that all features are useful. When coupled with runtime analysis, the feature evaluation results can be utilized to select a subset of features to achieve a target trade-off between system accuracy and runtime complexity.

5.2 Future Enhancement

In future work, we will leverage additional types of domain knowledge, such as the stack traces submitted with bug reports and the file change history, as well as features previously used in defect prediction systems. We also plan to use the ranking SVM with nonlinear kernels and further evaluate the approach on projects in other programming languages.

CHAPTER 6
REFERENCES

CHAPTER 6

REFERENCES

[1] G. Antoniol and Y.-G. Gueheneuc, “Feature identification: A novel approach and a case study,” in Proc. 21st IEEE Int. Conf. Softw. Maintenance, Washington, DC, USA, 2005, pp. 357–366.

[2] A. Sheta and D. Rine, “Modeling Incremental Faults of Software Testing Process Using AR Models”, the Proceeding of 4th International Multi Conferences on Computer Science and Information Technology (CSIT 2006), Amman, Jordan. Vol. 3. 2006.

[3] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in Proc. Int. Conf. Softw. Eng., Piscataway, NJ, USA, 2013, pp. 712–721.

[4] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports,” in Proc. Int. Conf. Softw. Eng., Piscataway, NJ, USA, 2012 pp. 14–24.

[5] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, “Looking for bugs in all the right places,” in Proc. Int. Symp. Softw. Testing Anal., New York, NY, USA, 2006, pp. 61–72.

[6] D. Sharma and P. Chandra, "Software Fault Prediction Using Machine Learning Techniques," Smart Computing and Informatics. Springer, Singapore, 2018. 541-549.

[7] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” in Proc. 15th Int. Conf. Softw. Eng., Los Alamitos, CA, USA, 1993, pp. 482–498.

[8] D. Binkley and D. Lawrie, “Learning to rank improves IR in SE,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., Washington, DC, USA, 2014, pp. 441–445.

[9] Malhotra, Ruchika. "A systematic review of machine learning techniques for software fault prediction." Applied Soft Computing 27 (2015): 504-518.

[10] D'Ambros, Marco, Michele Lanza, and Romain Robbes. "An extensive comparison of bug prediction approaches." Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010.

GitHub Link

1. RohanDalvik/Software-Bug-Prediction-Using-Machine-Learning (github.com)