Chapter 3

Temporal Bug Pattern Prediction

3.1 Introduction

Predicting the number of bugs in any software application is an important but challenging task. The software manager by modelling the bug numbers, can take timely decisions in deciding the amount of effort investment and also the allocation of resources. The software developers can take effective steps for reducing the number of bugs in the future version of the software application. The end users can make a timely decision on adoption of a particular software application by knowing the trend of the pattern of bugs across versions in advance.

The challenges behind modelling the bug growth patterns are complex processes that cause a bug. A bug in any software may be caused during testing, development or deployment. Causal modelling of bug numbers will be a complex and tedious task as one has to consider many internal characteristics to be modelled. The event of reporting a bug, fixing a bug and a new developer assigned to a project are all uncertain [76]. However in aggregate, all these random like interactions show some statistical patterns, which are not

purely random. Effective time series modelling approach is a potential methodology to model these bug growth patterns across the versions.

Previous studies on temporal bug pattern predictions are based on traditional time series modelling like ARIMA [227] that is used to predict a stationary time series data. In another paper by Hongyu et al.,[227] Polynomial Regression is used to predict the bug growth patterns in Eclipse. In our work, we have used advanced time series modelling for prediction of the increases or decreases in the bug numbers of a software system. Appropriate choice of predicting models becomes an important factor for improving the prediction accuracy.

The organization of the chapter is as follows: Section 2 presents a description of Data Preparation. Section 3 presents a description of the problem. Sections 4, 5 and 6 describes different techniques for predicting the increase and decrease trend in bug numbers of a software system. Section 7 and 8 describes the bug pattern prediction using Markov Model and Hidden Markov Model. A comparison of result with traditional models is given in each section. Section 9 concludes the chapter.

3.2 Dataset Description

The bug information about different software applications is maintained in software repositories [15]. The monthly, weekly and daily bug number information can be extracted from the repositories. These extracted bug number data can be used for time series analysis.

In this chapter, we have analyzed the bug growth pattern of three different software applications. We obtain the bug number data from public bug repository of Debian, Eclipse, and Mozilla respectively. The Debian Bug number data is available in the Debian Bug Repository in the Ultimate Debian Database (UDD)[15]. The Eclipse Bug number data is available in the Eclipse Bug Repository[6]. The Mozilla bug data is available in the

Bugzilla [11].The description of the three software applications is presented in the next section.

3.2.1 Debian

Debian[1] is a desktop as well as server operating system. Debian is a popular and influential Linux distribution. Debian project was started in August 1993, and it has 419M SLOC.

Debian provides more than a pure operating system: it comes with over 51000 packages, precompiled software bundled up in a nice format for easy installation.

Debian has a bug tracking system (BTS) in which details of bugs reported by users and developers are filed. Each bug is assigned a number.

3.2.2 Mozilla Firefox

It is an Open Source -software community started in 1998 by members of Netscape. Mozilla[3] has produced many products such as the Firefox web browser, Thunderbird e-mail client, Firefox OS mobile operating system, Bugzilla bug tracking system, Gecko layout engine, and other projects.

Bugzilla is a web-based general-purpose bugtracker, and testing tool originally developed and used by the Mozilla project, and licensed under the Mozilla Public License. The size is 16M SLOC.

Released as open-source software by Netscape Communications in 1998, it has been adopted by a variety of organizations for use as a bug tracking system for both free and open-source software and proprietary projects and products.

3.2.3 Eclipse

Eclipse[2] is an integrated development environment (IDE) used in computer programming, and is the most widely used Java IDE.[6]

It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java, and its primary use is for developing Java applications.

It may also be used to develop applications in other programming languages via plugins, including Ada, C, C++, COBOL, D, Fortran, JavaScript.

The Eclipse Project uses Eclipse Bugzilla for bug tracking.

The Bug Number data is collected from the bug repository of the three software applications and arranged as a monthly bug number data for time series analysis. The bug growth patterns of the three software applications is presented in Figure 3.1,3.2 and 3.3 respectively . Figure 3.1 and 3.2 shows sporadic peaks at certain releases. The sporadic peaks may be due to introduction of new features leading to increased Lines of Code.

3.3 Problem Description

Let an observed bug sequence be $\{B_1, B_2, B_3, ...B_T\}$ observed over equally spaced time points.

A fairly general model for the time series can be written:

$B_t = f(t) + \varepsilon_t$ Here, we have two components: Systematic Part: $f(t)$ = The Component to be Modelled using Time Series.
Stochastic Part: ε_t, a residual term also called noise, which follows some unknown probabilistic law.

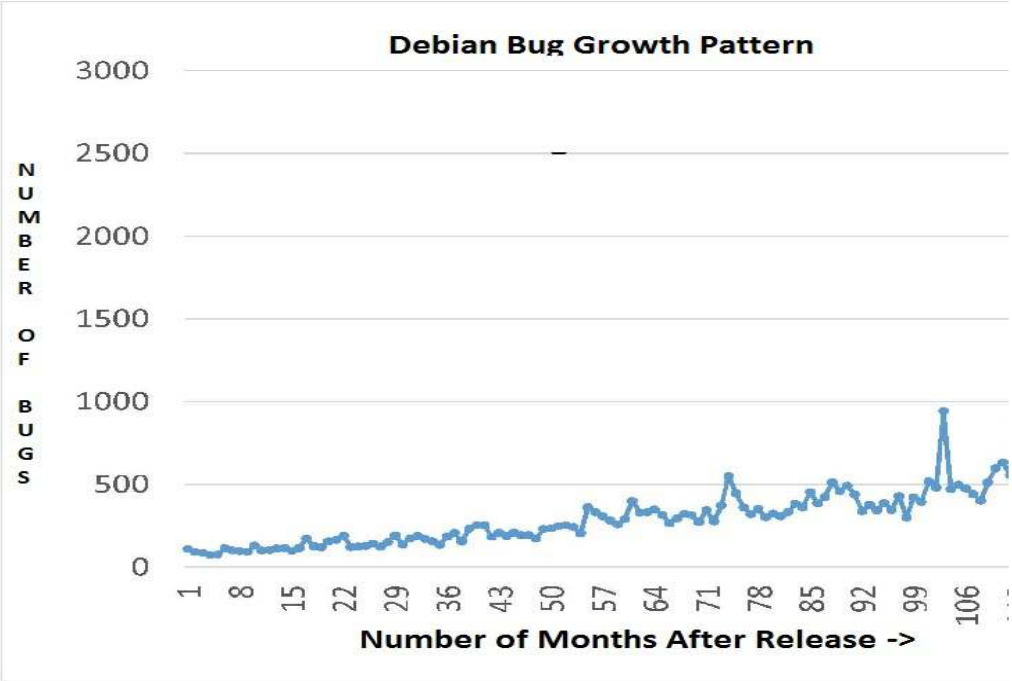


FIGURE 3.1: Debain Bug Growth Pattern

FIGURE 3.2: Eclipse Bug Growth Pattern

The objective is to predict B_t from p past observations.

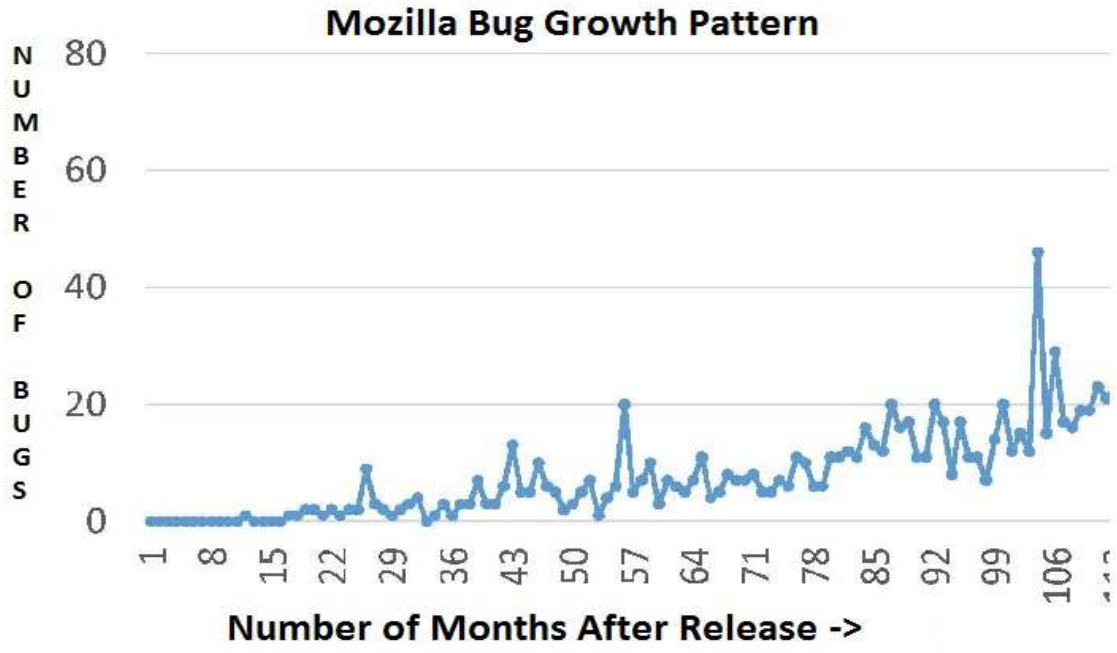


FIGURE 3.3: Mozilla Bug Growth Pattern

$$B_t = f(B_1, B_2, B_3, ...B_{t-p}) + \varepsilon_t$$

The goal is to have a suitable model which can predict the bug numbers accurately with a minimum value of error (ε_t). In the next section, we have discussed various time series models to predict the bug growth patterns.

3.4 Neural Network Approach for Bug Number Prediction

Neural networks are used for classification and prediction task in different domains. Neural networks are also frequently used in time series modelling. They are good approximators for nonlinear patterns in data. As real time bug number data shows nonlinearity, the neural network is a right approach for modelling. In this work [165], we performed

a comparative analysis of different neural network layer configurations and transfer functions on the prediction accuracy. We have also used some heuristic information about bug patterns to predict the bug numbers. The result shows improvement in accuracy after insertion of heuristic information.The model is validated with Debian Bug Number Data.

3.4.1 Neural Network Model

A feed-forward neural network (multilayer) is nonlinear learning model, and it uses supervised training approach. It performs adjustment of the network weights on its inputs and the internal nodes iteratively to minimize the errors between actual and predicted values. It commonly uses simple gradient descent to find the local minima and optimize the network accordingly. The Levenberg - Marquardt algorithm [59, 138] with Bayesian Regularization [171] is the most updated algorithm and commonly used for neural network training.

The data needs to be normalized before being trained by a neural network training algorithm. The normalization is important to obtain good results as well as to speed up the calculations significantly. Sometimes we use special transfer functions which normalize the data before training[8].

3.4.1.1 Transfer Function

There are different transfer functions to catch the linear and nonlinear patterns in the data. As the real bug number data shows nonlinear behavior, we have used only nonlinear transfer functions for the hidden layer. Here are some commonly used transfer functions in the neural network.

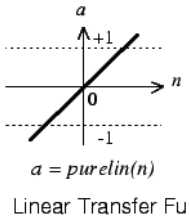


FIGURE 3.4: Linear Transfer Function

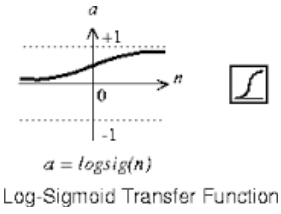


FIGURE 3.5: Log Sigmoid Transfer Function

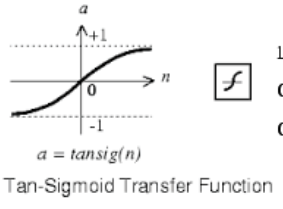


FIGURE 3.6: Tansig Transfer Function

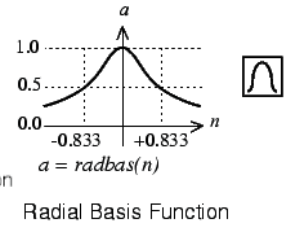


FIGURE 3.7: Radial Basis Function

3.4.1.2 Heuristic Information about Bug Patterns

The neural network is a nonparametric learning method [119] which learns from dataset directly. The more the number of relevant patterns available for training, the more will be the accuracy. So, if we can provide some extra information about bug data, then there is a chance of an increase in accuracy. In this work, we have considered the distribution of bug number data as an external factor for learning. If the bug number data is symmetrically distributed and have outliers, then we can apply K – Median clustering [9]. If the bug number data is not symmetrically distributed, then we can apply K – Means clustering method. We can also apply other clustering methods based on suitability. The clustering process grouped the entire bug number data into different clusters. These clustered series of bug number data can be taken as heuristic information for neural network learning. The sorted bug number data of Debain show symmetric behavior.The bug report as shown in Ultimate Debain Bug Database is presented in Figure 3.8. Figure 3.9 shows the sorted distribution of Debian bug number series.

bug#	package	title	modified
#506805	xml2	mangles UTF-8	2017-08-04
#614497	doc-debian	doc-debian: The OPL-licensed documents taken from Debian website are non-free	2016-10-16
#665334	fontforge	non-DFSG postscript embedded in fontforge (currently August 2014	2017-03-19
#675857	alsa-utils	PulseAudio-related hidden config files & 'olders created in the root directory	2017-05-04
#694320	gsfonts	[gsfonts] Fonts include copyrighted adobe fragment all right reserved	2017-03-19
#694323	lmodern	[gsfonts] Fonts include copyrighted adobe fragment all right reserved	2017-03-19
#694324	tex-gyre	Fonts include copyrighted adobe fragment all right reserved	2017-03-19
#704303	firefox, firefox-esr	iceweasel: MPL license text is missing	2017-04-19
#709198	debconf	debconf: should not use python in maintainer scripts	2017-07-04
#715087	libjack-dev	libjack-dev: broken symlink: /usr/lib/x86_64-linux-gnu/libjackserver.so -> libjackserver.so.0.0.28	2017-07-03
#740893	libjs-jquery-hotkeys	libjs-jquery-hotkeys: jquery.hotkeys changes break python-coverage html reports.	2017-08-03
#741464	grub-pc-bin	grub-pc-bin: freezes after "terminal_input at_keyboard"	2017-04-19
#749991	debian-installer	debian-installer: Wrong kernel in debian-installer package	2017-04-06
#750946	libhtml-html5-parser-perl	libhtml-html5-parser-perl: UTF-8 character breaks parse_file	2017-08-11
#760385	libv8-3.14	nodejs: CVE-2014-5256	2017-04-19
#768843	fetchmail	fetchmail: Improved TLS support	2017-08-12
#773623	libv8-3.14	nodejs: CVE-2014-7192	2017-04-19
#773671	src:libv8-3.14	libv8-3.14: multiple security issues	2017-04-19
#774227	busybox-static	busybox-static: execs applets when chrooting	2017-07-03

FIGURE 3.8: Debian Bug Database (UDD)

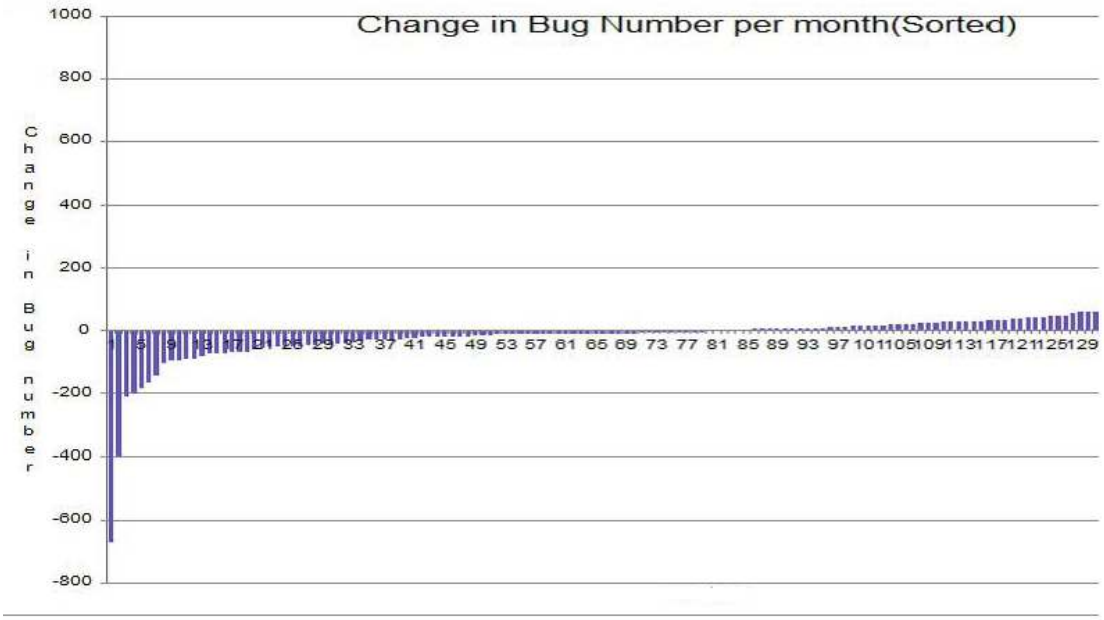


FIGURE 3.9: Sorted Bug Number Data (Debian)

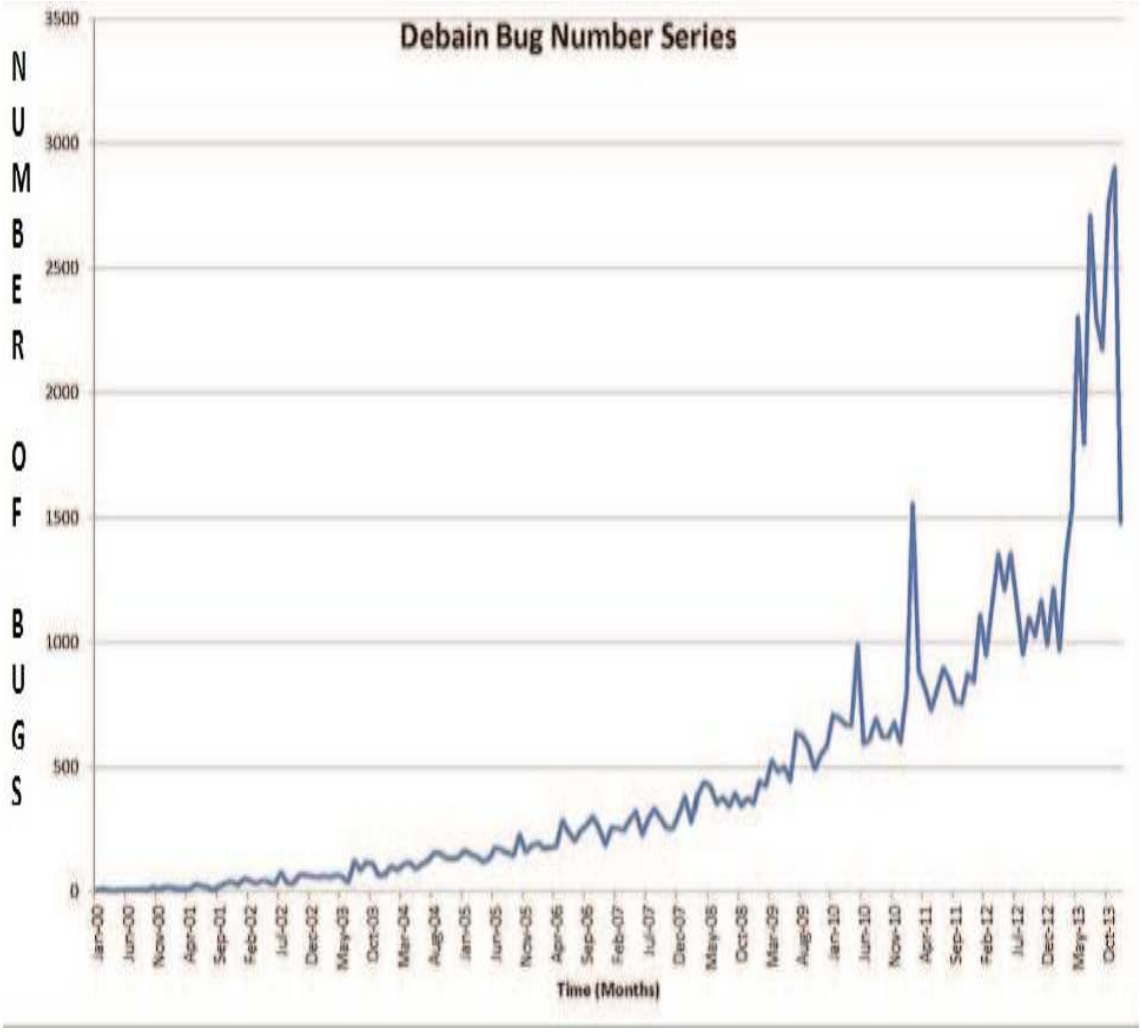


FIGURE 3.10: Debian Bug Number Data

3.4.2 Data Preparation

The detailed bug reports for Debian are available in the Ultimate Debian Database (UDD) [15]. The original data in the repository looks like this: “id, “Last Modified”: 718624; 2013-08 -19. We have considered total bug counts in our analysis. The monthly bug number data from Jan – 2000 to Dec – 2013 is taken for analysis. For Debian, we have 168 monthly bug counts. The bug number series from Jan – 2000 to Dec -2013 shows an increasing trend. The time series of monthly bug counts from Jan – 2000 to Dec – 2013 is given in Figure 3.10.

We have performed ADF (Augmented Dicky – Fuller Test) [223] to test the behavior of Debian bug number series. The ADF test checks the presence of a unit root in a time series. If a unit root exists, the series becomes not–stationary. In this work, we have used [web: reg], an add - into Microsoft excel [4] to perform ADF test. From the ADF test, we observed that the Debian bug number series is non-stationary.

The entire bug number series is partitioned into a training set and a testing set. Partitioning helps in finding how our model fits the data pattern (both trained and unseen data) properly. The model which gives more accuracy on test data is preferably selected.

3.4.3 Proposed System

Figure 3.11 shows the diagrammatic representation of the proposed system. The bug number series is first checked for its stationary behavior. As the Debian bug number series is nonlinear, we have applied a neural network to model the data. The data set then is divided into training set and testing set. Three different neural network transfer functions are used to find out the best fit for Debian bug number series. Then we added cluster number (heuristic information) associated with particular monthly bug counts to our input set. The neural network is trained with Levenberg - Marquardt algorithm with Bayesian regularization [171]. The error values generated by the models with different configurations are compared, and results are statistically verified.

3.4.4 Implementation

Here we have taken a feed forward neural network for predicting the Debian bug number series. The multiple layers of neurons with nonlinear transfer functions allows the network to match the nonlinear patterns in the data. The output layer uses linear transfer function for function fitting problem. The neural network is trained by Levenberg –

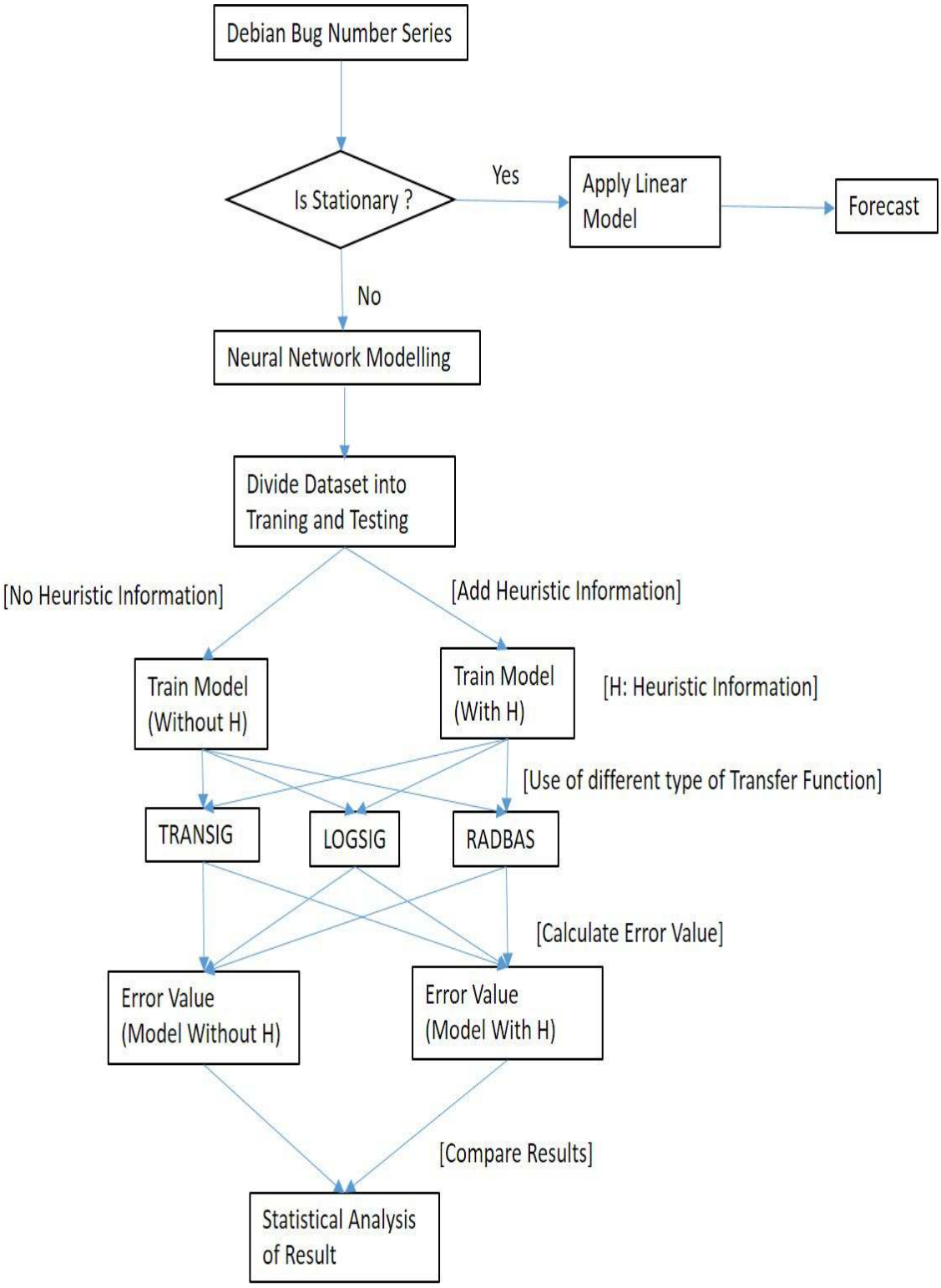


FIGURE 3.11: Proposed System

Marquardt algorithm with Bayesian Regularization (trainbr) [171], an advanced neural network training algorithm. We have taken different nonlinear transfer functions to find the best-fitted function for the Debian bug number series. We have also checked different layered configurations to determine their effect on the prediction accuracy. We have also tested the effect of an additional heuristic factor on the prediction accuracy. All the experiments were conducted in Matlab environment. All the results were verified statistically for their significance.

3.4.5 Evaluation

We have calculated the forecasting Errors (E) by the following formulae:

Let Actual Bug Number Value at a particular Month = A

Let Predicted Bug Number Value at a given month = B

Let Total number of Observations = N

Then for all the observations:

$E = Mean(A - B) = \frac{(A-B)}{N}$

E: Error Values for Training Sample

3.4.6 Results and Discussion

We have used feed forward neural network having 11 input neurons, 15 hidden layer neurons, and single output neuron. The 11 inputs represent previous bug number data of previous 10 months and a bias input of corresponding input vectors. The hidden layer has 15 neurons with a nonlinear transfer function that calculates the output of the layers from its net input. For this model, the training data size is 140 and test data size is 10. Figure 3.12 represents the model of our artificial neural network. In this section, we discuss the effect of different neural network transfer functions, neural network configurations and the effect of using additional heuristic information on the prediction accuracy.

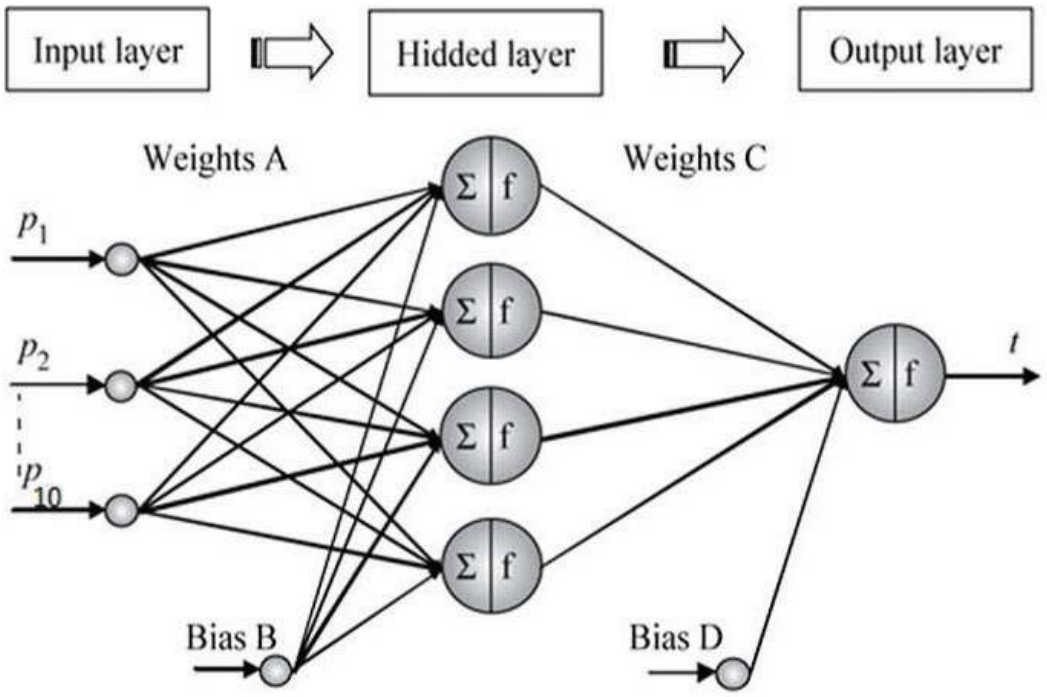


FIGURE 3.12: Neural Network Model for Debian Bug Number Series

3.4.6.1 Effect of Transfer Function

We have tested our model with different transfer functions. As the bug number series is nonlinear, we only apply nonlinear activation functions to match the nonlinear patterns in the data. We have used three different transfer functions like tansig (NN (TANSIG)), logsig (NN (LOGSIG)), and radbasn (NN (RADBAS(n)) to find the function that best fitted to the bug number series. The neural network is trained with trainbr [171] as described in section 4. Here we show a comparative analysis of actual and predicted series with different transfer functions. Figure 3.13, Figure 3.14 and 3.15 represents the comparison between predicted and actual output by the neural network model using the three different transfer functions respectively. The present prediction is based on previous values of the time series. Careful observation of the plot shows effect of decrease in actual data manifests itself after a delay. This effect is more pronounced with **LOGSIG** activation function, decreases with **TANSIG** and almost negligible with **RADBAS** function.

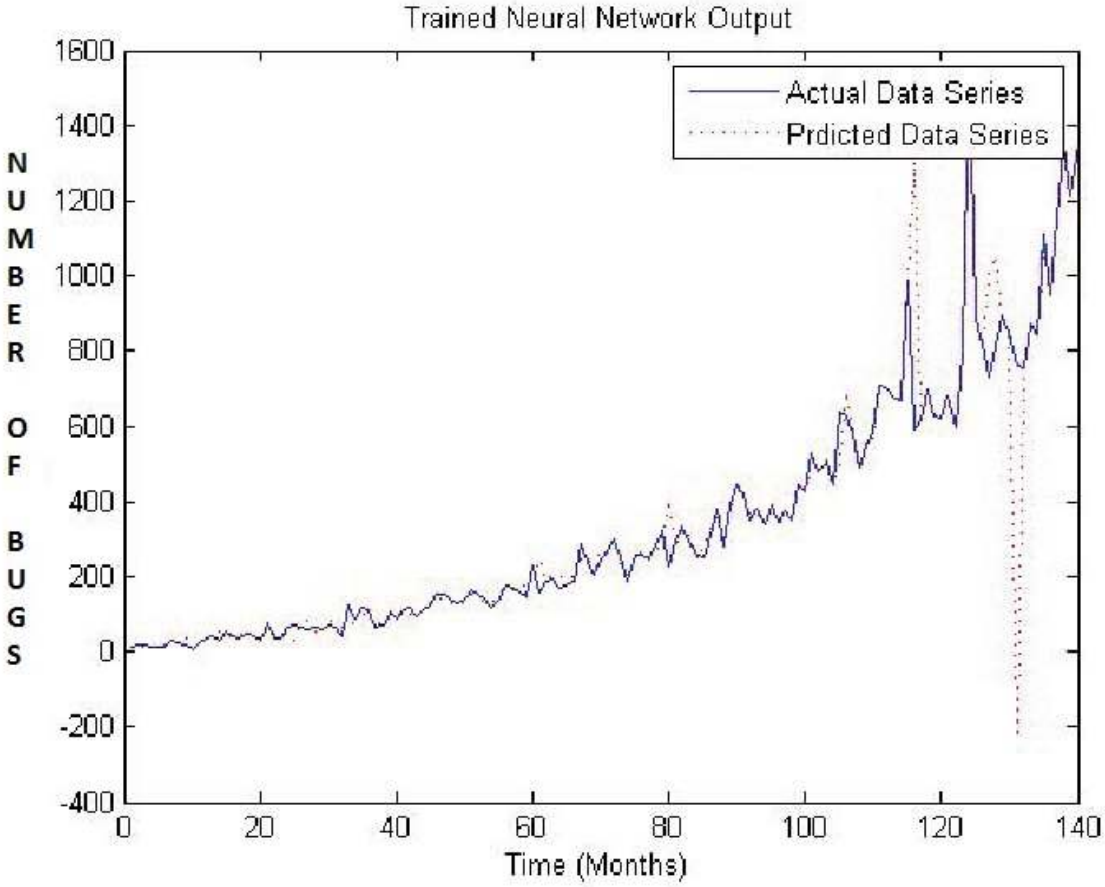


FIGURE 3.13: Comparison of Actual and Predicted output:Logsig

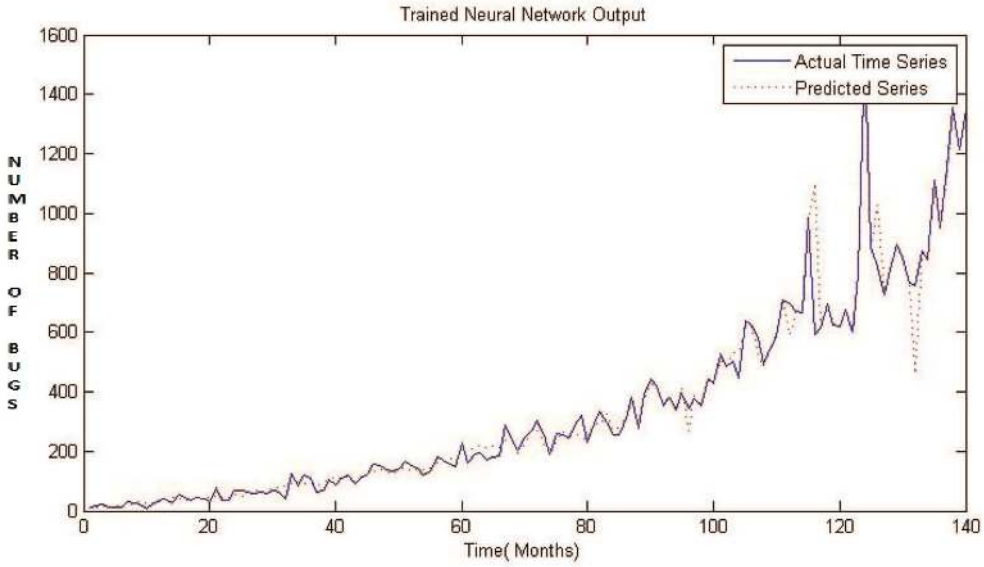


FIGURE 3.14: Comparison of Actual and Predicted output:Tansig

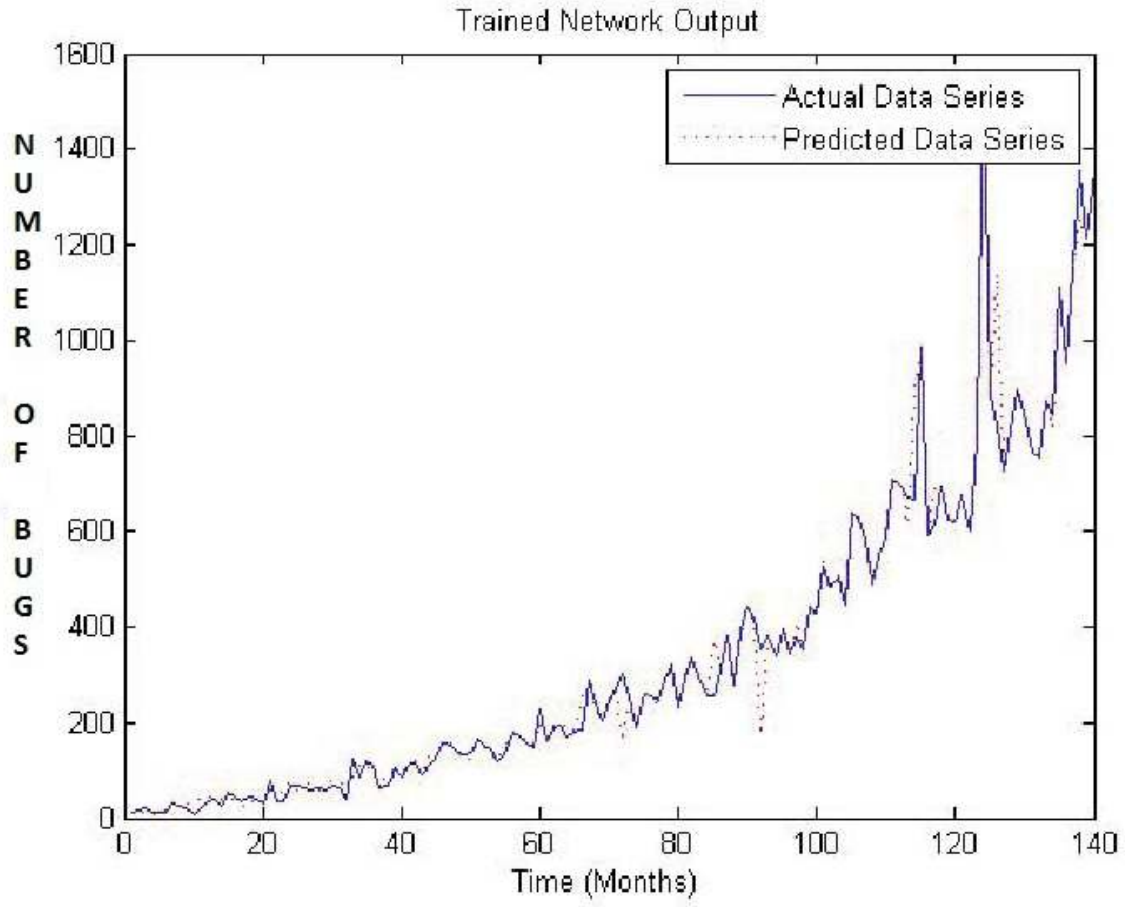


FIGURE 3.15: Comparison of Actual and Predicted output:Untrained NN:RADBSN

To compare the performances of the model using these three transfer functions we have performed a pairwise t-test 4 on the error values as shown in Table 3.1. The null hypothesis is that there is no significant difference in the error values by using these three transfer functions. We have run three kinds of pairwise t-tests: NN (TANSIG) vs. NN (LOGSIG), NN (LOGSIG) vs. NN (RADBSAN) and NN (TANSIG) vs. NN (RADBASN). We have coded the *P–Value* in following ranges.

$\ll$ means that $P - Value$ is lesser than or equal to 0.05 indicating strong evidence that a method generated smaller error value than another.

$\gg$ means that P-Value is bigger than 0.05 stating that there are no significant differences between the methods.

TABLE 3.1: Statistical T- Test Report (Comparison among Transfer Functions)

Compared Models	P- Value	Symbol	Differences (YES/ NO)?
NN (TANSIG) vs. NN (LOGSIG)	0.882951	>>	NO
NN (LOGSIG) vs. NN (RADBSAN)	0.000132	<<	YES
NN (TANSIG) vs. NN (RADBASN)	0.003383	<<	YES

TABLE 3.2: Statistical T- Test Report (Comparison among Number of Hidden Layers)

No of Hidden layers	P- Value	Symbol	Differences (YES/ NO)?
10 vs. 15	0.014097	<<	YES
15 vs. 20	0.080285	>>	NO

Table 3.1 shows the P –values for pairwise t-tests performed between the neural networks with different transfer functions.

From the Table 3.1, we found that NN (RADBASN) as a transfer function generates least error values than other two. Although NN (LOGSIG) has mean error value less than NN (TANSIG), there is no significant difference between these two as shown by t-test result. So, normalized radial basis transfer function is a suitable transfer function for Debian bug number series.

3.4.6.2 Effect of Number of Hidden Layer Neurons

We have also checked the effect of layer configuration on the prediction accuracy. For the model NN (RADBASN), we have changed the number of hidden layers to 10 and 20 respectively. We have also performed a pairwise t-test for the error values generated by NN (RADBASN) with a different number of hidden layer neurons. Table 3.2 shows the t-test results. From the Table 3.2, we observed that there is significant difference in error values for NN (RADBASN) with 15 numbers of hidden neurons and NN (RADBASN) with 10 numbers of hidden neurons. The later is having more error values than the former. So by increasing the number of hidden layers we can increase the accuracy. But there is

TABLE 3.3: Comparison among Number of Input Layers

No of input layer neurons	P- Value	Symbol	Differences (YES/ NO)?
6 vs. 11	0.00013	<<	YES
11 vs. 15	0.052159	>>	NO

no significant difference between NN (RADBASN) with 15 numbers of hidden neurons and NN (RADBASN) with 20 numbers of hidden neurons. It shows that increasing the number of hidden layers beyond some specific range will have no effect on accuracy.

3.4.6.3 Effect of Number of Input Layer Neurons

We have also tested the effect of a change in the input layer neurons on the prediction accuracy. In the previous models, we have considered input layer of size 11 (1 bias input + 10 previous monthly bug counts). Here we have changed the size of the hidden layer to 6 (1 bias input + 5 previous monthly bug counts) and 16 (1 bias input + 15 previous monthly bug counts). A pairwise t-test for the error values generated by NN (RADBASN) with a different number of input layer neurons is also performed. Table 3.3 shows the t-test results. From the Table 3.3, we observe that by taking less number of input neurons the error value increases and also by taking too much history into consideration, there is no significant difference in the error values. So our previous model with 11 number of input layer neurons (1 Serial number + 10 previous monthly bug counts) is the best amongst other models.

3.4.6.4 Effect of Heuristic Information

As the neural network is a nonparametric learner [12], i.e., it learns from dataset directly, the hidden information about the bug number series can also be given as an input set along with bug number information. We have considered the distribution of bug numbers data

TABLE 3.4: Cluster range for Debian Bug Number Series

Cluster Number	Lower Threshold	Upper Threshold
1	-667	-38
2	-37	-1
3	0	15
4	16	70
5	71	756

TABLE 3.5: Statistical Comparison

Compared Models	P- Value	Symbol	Differences (YES/ NO)?
NN (TANSIG) vs. NN (TANSIG) +H	0.001328	<<	YES
NN (LOGSIG) vs. NN (LOGSIG)+H	0.000222	<<	YES
NN (RADBASN) vs. NN (RADBASN) +H	0.005045	<<	YES

for Debian as a heuristic factor. The sorted Debian bug number series show symmetric behavior and also contain some outliers. K –median clustering is suitable for the symmetric dataset and is sensitive to outliers.

We have applied K-Median clustering (with k=5) to Debian bug Number series which clusters the bug number series into 5 clusters. These cluster numbers associated with a particular monthly count acts as an additional input to the neural network. The upper and lower range of different clusters after applying K- Median clustering is given in Table 3.4.

We have calculated the effect of heuristic information (H) on the training error values of the neural network with a different transfer function. Table 3.4 shows the comparison between the training error values for NN (TANSIG), NN (LOGSIG) and NN (RADBASN) models with and without heuristic information (H).

We have also performed a pairwise t-test to show the statistical significance of the effect of heuristic information on the performance of our model. Table 3.5 shows the t-test results. From the Table 3.5, we observed that by adding the heuristic information, there is always a significant decrease in the error value. So the cluster number is an important factor for improving the performance of our model.

In this work, we used a feed forward neural network to predict the trend of the bug numbers in Debian. This work also compares between three nonlinear transfer functions to find the best fit for Debian bug number series. The result confirms NN (RADBASN) as a most appropriate model for Debian bug number series. This work also gives a comparative analysis of the effect of layer configuration on the performance of the model. This work also confirms cluster number as a heuristic factor which can improve the performance of the model.

The contribution of the work consists of two parts: First, it adopts complete neural network approach to predict the trend of the bug numbers in Debian. Second, it uses efficient statistical tests to prove the results obtained by the model. The statistical tests helps to find the best neural network model for predicting the trend of the Debian bug number series.

3.5 A Hybrid Technique for Debian Bug Number Prediction

In this work, we have used a hybrid (ARIMA + neural network)[164] model for predicting the Debian bug number series. There are many hybrid approaches used in time series modelling [95, 96, 58, 24] in areas like stock market predictions, water consumption forecasting, etc. Here, we have used a hybrid approach [241] to predict the Debian bug numbers. We have also performed a comparative analysis of performance hybrid model, ARIMA model and neural network model in predicting the bug numbers of Debian.

3.5.1 The Hybrid Model

ARIMA and ANNs are good at prediction in the linear and nonlinear domains respectively. However ARIMA cannot correctly approximate the complex nonlinear problems.

Similarly, ANN models cannot give more accurate results for linear problems. The real world time series contain both linear and nonlinear terms i.e.

$Y_t = L_t + N_t$

L_t :Linear Component

N_t:Non-Linear Component

First, ARIMA model is applied to predict the linear component, and then the residuals from the linear model contains only the nonlinear relationship. Now the residuals from the nonlinear model can be represented by:

$e_t = Y_t - F(L_t)$

Where $F(L_t)$ is the forecast value for time t for ARIMA.

With n input nodes, the ANN model for the residuals will be:

$e_t = f(e_{t-1}, e_{t-2}, ...e_{t-n}) + \varepsilon_t$

Where f is a nonlinear function determined by neural network and ε_t is the random error. Now let the predicted nonlinear component is $F(N_t)$, the combined forecast will be:

$F(Y_t) = F(L_t) + F(N_t)$

So, the hybrid model work in two steps:

- In the first step, the ARIMA model is used to analyze the linear patterns in the model.
- In the second step, a nonlinear model is used to analyze the residuals.

Figure 3.16 shows the representation of the hybrid system.

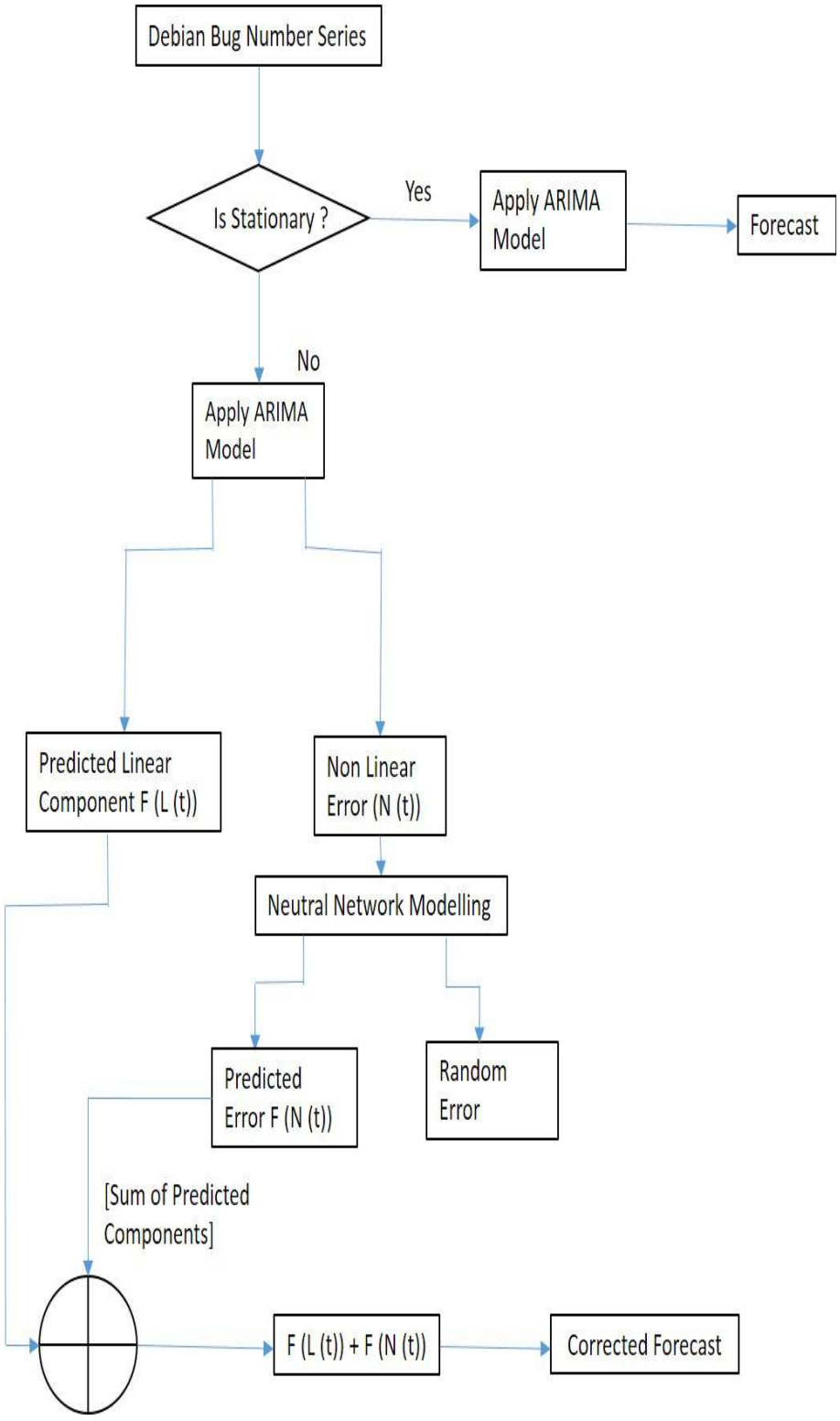


FIGURE 3.16: Proposed Hybrid Model

3.5.2 Data Preparation

We have considered total bug counts in our analysis. The monthly bug number data from Jan – 2000 to Dec – 2013 is taken for analysis. For Debian, we have 168 monthly bug counts. The bug number series from Jan – 2000 to Dec -2013 shows an increasing trend.The entire bug number series is partitioned into a training set and a testing set. Partitioning helps in finding how our model fits the data pattern (both trained and unseen data) properly. The model which gives more accuracy on test data is preferably selected.The size the training dataset and test dataset are taken as 140 and 10 respectively for all the models to have a consistent comparison among the models.

3.5.3 Evaluation

The models are evaluated based on the RMSE (Root Mean Square Error), MAE (Mean Absolute Error). RMSE, MAE are calculated using following formulae.

1. $MAE = \frac{1}{n}\sum_{t=1}^{N}|(A(t) - F(t))|$

2. $RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{N}(A(t) - F(t))^2)}$

3.5.4 Implementation and Result

In this work, we have experimentally verified the predictive performance of three models: ARIMA, ANN and the Hybrid Model (ARIMA + ANN). Here, we have described in details about the Implementation of these models on Debian bug number series.

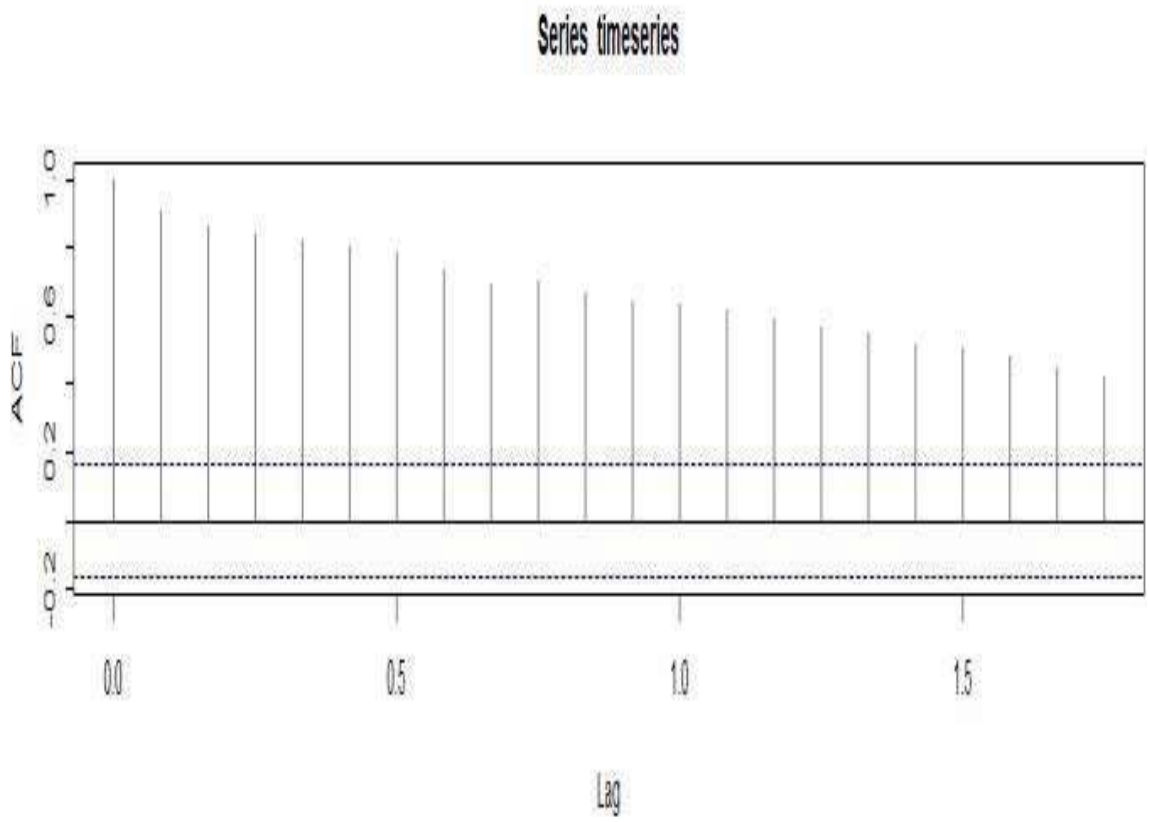


FIGURE 3.17: ACF Plot: Debian Series

3.5.4.1 ARIMA

ARIMA model [227] consists of AR (p), MA (q) and a Differencing (d) operator. To have a good predictive performance the series must be stationary. The stationary behavior can be checked from the Auto Correlation Function (ACF) and Partial Auto Correlation Function (PACF) plots. If the ACF decays slowly, then the series is said to be non-stationary. If the ACF cuts instantly, then the series is said to be stationary. The ACF Plot of Debian Bug Number Series is presented in Figure 3.17. From the figure 3.17, we found the ACF decays slowly.So, the nonstationary behavior of Debian Bug Number Series is confirmed.After differencing the ACF cuts at lag2 instantly and PACF also cuts approximately at lag 2. There are some abnormalities in higher lags which can be ignored. So, the most appropriate model for the Debian bug number series becomes ARIMA (2, 1, 2).

TABLE 3.6: Error Values (ARIMA)

ARIMA Prediction	RMSE	MAE
Training Set	83.5108	38.8269
Test Set	250.8259	197.2487

We have also checked 15 combinations of p, d, q parameters for ARIMA to find the best fit.

After applying the ARIMA (2, 1, 2) we got a predicted series and some error terms. The RMSE and MAE for ARIMA model for Debian bug number series are shown in Table 3.6.

3.5.4.2 Artificial Neural Network (ANN)

Here we have taken a feed forward neural network for predicting the Debian bug number series. The multiple layers of neurons with nonlinear transfer functions allow the network to match the nonlinear patterns in the data. The output layer uses linear transfer function for function fitting problem. The neural network is trained by Levenberg– Marquardt algorithm with Bayesian Regularization (trainbr) [171], an advanced neural network training algorithm. We have taken different nonlinear transfer functions to find the best-fitted function for the Debian bug number series. All the experiments were conducted in Matlab environment.

We have used feed forward neural network having 6 input neurons, 15 hidden layer neurons, and single output neuron. The 11 inputs represent previous bug number data of previous 5 months and a bias input of corresponding input vectors. The hidden layer has 15 neurons with a nonlinear transfer function that calculates output of the layer from its net input.

As the bug number series contains a mixture of linear and non-linear patterns, we apply nonlinear activation functions to match the nonlinear patterns in the data. We have used

TABLE 3.7: Error Values (Neural Network Model)

ANN Model Prediction	RMSE	MAE
Training Set	58.6773	20.458
Test Set	230.847	193.733

radbasn (NN (RADBASN)) as activation function to match the nonlinear patterns in the bug number series.

The RMSE and MAE for neural network model for Debian bug number series for the training set and test set are given in Table 3.7.

3.5.4.3 Hybrid Model (ARIMA + ANN)

The next step is to apply the linear and non-linear combination of models to predict the Debian bug number series. As discussed in section 2, the Debian bug number series is first modelled with ARIMA. The result is predicted linear component $F(L_t)$ and some non-linear error term $F(N_t)$. Now the error contains some non-linear patterns which need non-linear technique to model the error series. The error series is modelled with ANN.

The error series: $e_t = f(e_{t-1}, e_{t-2}, ...e_{t-n})$ is modelled as non-linear autoregressive neural network model of order 5, i.e. five previous values of error are required for predicting the current value. We have used non-linear activation function (radbasn) to match the non-linear patterns in the error series. After applying neural network model to the error component, we get a predicted error component $F(N_t)$ and some random error ε_t . Now the predicted error component $F(N_t)$ and the predicted linear component $F(L_t)$ are added to get the combined forecast value.

The RMSE, MAE and the Error function (E) for neural network model for Debian bug number series for the training set and test set are given in Table 3.8. In this work, we give a comparative analysis of predictive performance of the ARIMA, Neural network and the hybrid (ARIMA + ANN) model. The result confirms the hybrid model as a good

TABLE 3.8: Error Values (ARIMA + ANN)

Hybrid Model Prediction	RMSE	MAE
Training Set	23.224	8.42
Test Set	129.539	49.645

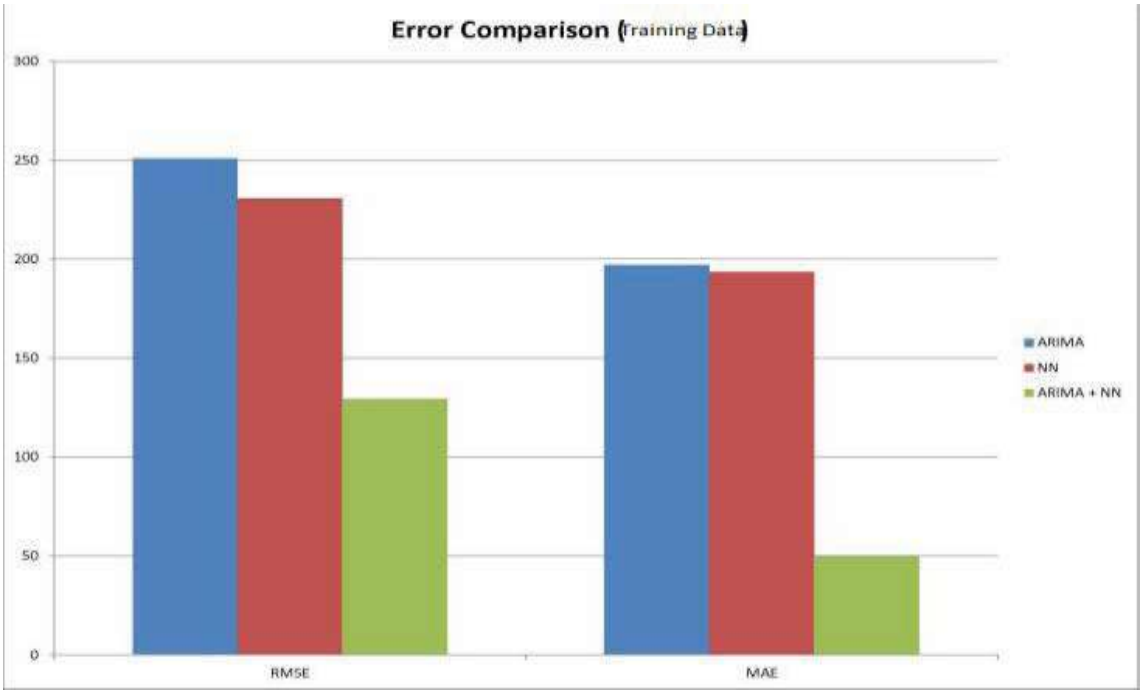


FIGURE 3.18: Comparison of RMSE and MAE value between the three Models (Training Set)

predictor for Debian bug number series. Figure 3.18 and Figure 3.19 show the bar chart representation of comparison between RMSE and MAE for both training and test set for the three models respectively.

3.6 An Ensemble Technique for Bug Number Prediction

In this work, we have used an ensemble technique[168] for predicting the Debian bug number series. We have also performed a comparative analysis of the performance of the Ensemble model, ARIMA model and Neural Network model in predicting the bug numbers of Debian.

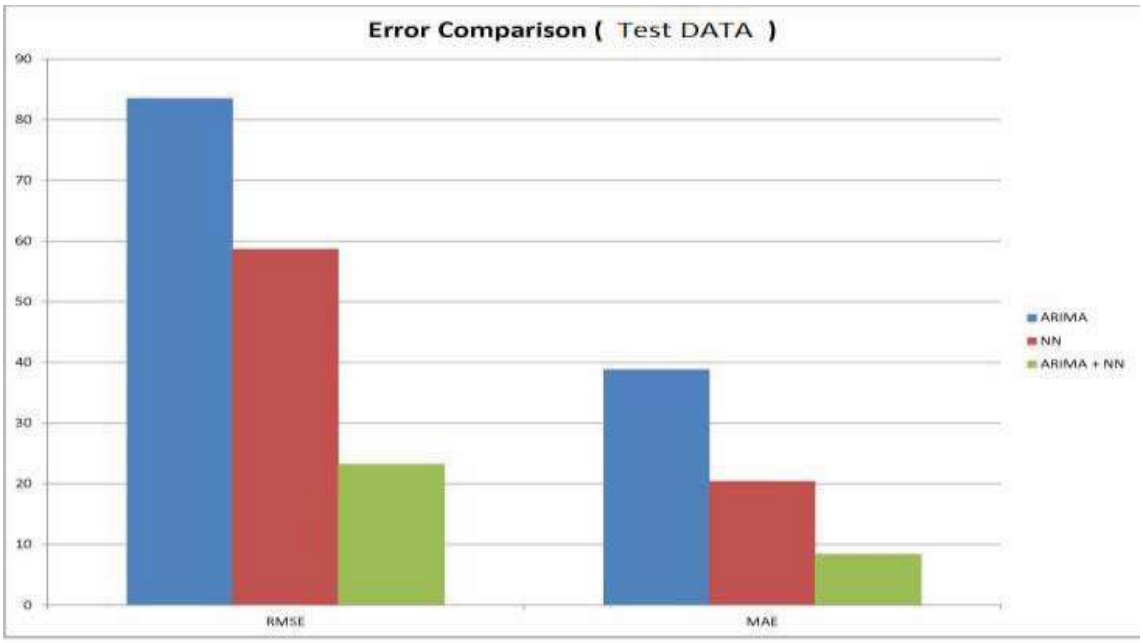


FIGURE 3.19: Comparison of RMSE and MAE value between the three Models (Test Set)

3.6.1 The Ensemble Model

In this work, we have used an advanced hybridization model approach to bug number prediction to yield a more accurate result. Here, we have used a nonlinear integration of autoregressive and moving average terms, i.e., a time series is taken as a nonlinear function of some past observations and some error terms. In ARIMA we have,

$Y_t = F_l(Y_{t-1}, Y_{t-2}, ...Y_{t-m}), (e_{t-1}, e_{t-2}, ...e_{t-n})$

where F_l is a Linear Function.

In the ensemble model, we have used a neural network to have a nonlinear integration of some past observations and some error terms. Here we have:

$Y_t = F_n(Y_{t-1}, Y_{t-2}, ...Y_{t-m}), (e_{t-1}, e_{t-2}, ...e_{t-n})$

Where F_n is a nonlinear function used in the neural network.

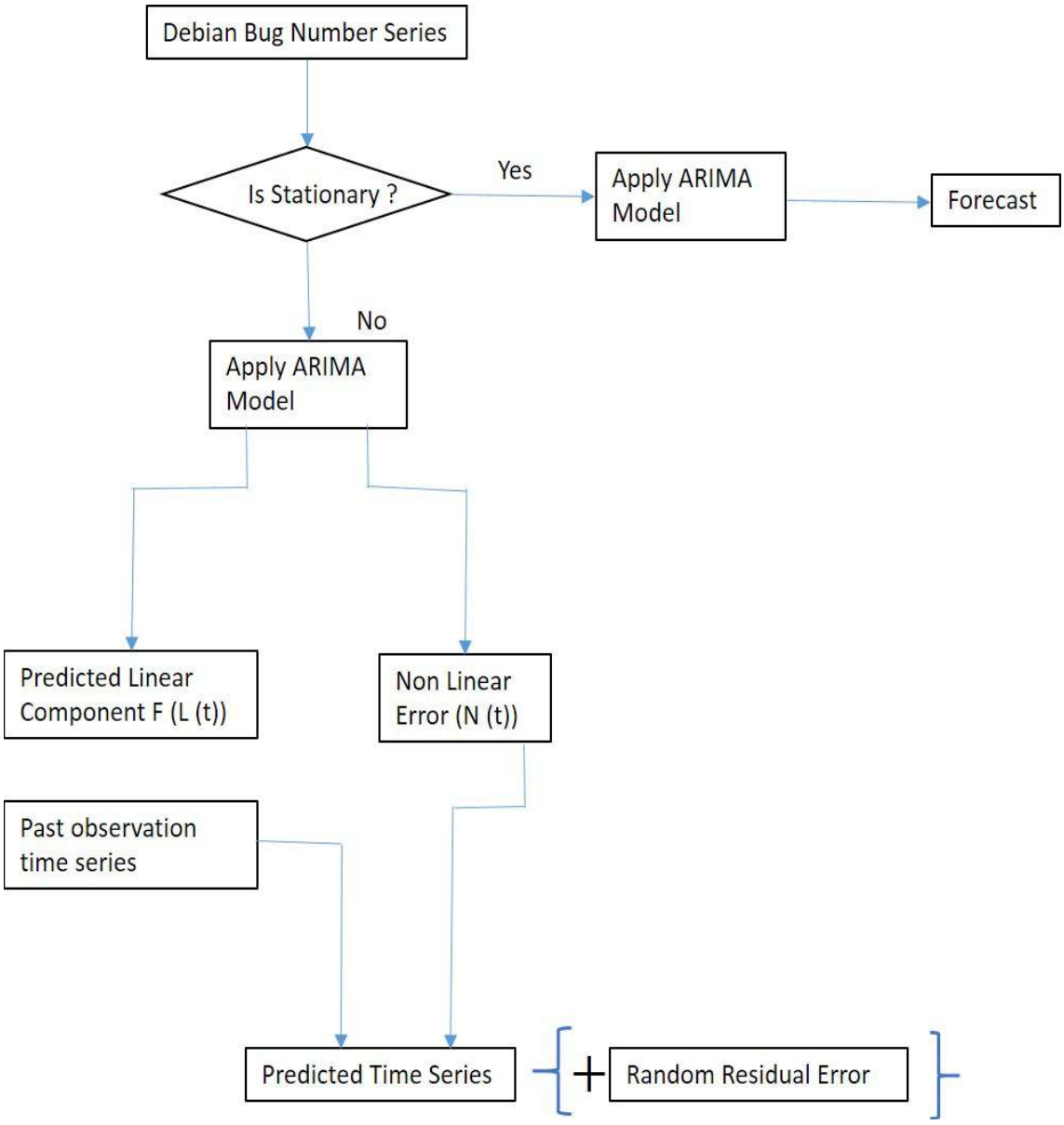


FIGURE 3.20: Ensemble Model

In the first step we have used ARIMA to get the residual terms, and in the second step, we have used a neural network to model the time series as a nonlinear combination of some past observations and some error terms. Figure 3.20 shows the representation of the ensemble system.

3.6.2 Data Preparation

The monthly bug number data from Jan – 2000 to Dec – 2013 is taken for analysis. For Debian, we have 168 monthly bug counts. The bug number series from Jan – 2000 to Dec -2013 shows an increasing trend. The entire bug number series is partitioned into a training set and a testing set. Partitioning helps in finding how our model fits the data pattern (both trained and unseen data) properly. The model which gives more accuracy on test data is preferably selected.

3.6.3 Evaluation

The models are evaluated based on the RMSE (Root Mean Square Error), MAE (Mean Absolute Error). RMSE, MAE are calculated using following formulae.

1. $MAE = \frac{1}{n}\sum_{t=1}^{N}|(A(t) - F(t))|$

2. $RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{N}(A(t) - F(t))^2)}$

3.6.4 Implementation and Result

In this work, we have experimentally verified the predictive performance of three models: ARIMA, ANN and the Ensemble model.

3.6.4.1 ARIMA

It is observed that ARIMA (2, 1, 2) is the best among all the other models as it is having least RMSE and MAE values. After applying the ARIMA (2, 1, 2) we got a predicted

TABLE 3.9: Error Values (ARIMA Model)

ARIMA Prediction	RMSE	MAE
Training Set	83.51	38.82
Test Set	250.825	197.248

TABLE 3.10: Error Values (ANN Model)

ANN Model Prediction	RMSE	MAE
Training Set	58.677	20.458
Test Set	230.847	193.732

series and some error terms. The RMSE and MAE for ARIMA model for Debian bug number series is shown in Table 3.9.

3.6.4.2 ANN Model

We have used feed forward neural network having 6 input neurons, 15 hidden layer neurons, and single output neuron. The 11 inputs represent previous bug number data of previous 5 months and a serial number of corresponding input vectors. The hidden layer has 15 neurons with a nonlinear transfer function that calculates output of the layers from its net input. For this model, the training data size is 140 and test data size is 10. As the bug number series is nonlinear, we only apply nonlinear activation functions to match the non-linear patterns in the data. The RMSE and MAE for neural network model for Debian bug number series for the training set and test set are given in Table 3.10.

3.6.4.3 Ensemble Model

Some past observations and some error series are modelled as non-linear neural network model of order 5, i.e. five previous values of observations and error terms are required for predicting the current value. We have used non-linear activation function (radbasn) to match the non-linear patterns in the error series.

TABLE 3.11: Error Values (Ensemble Model)

Ensemble Model Prediction	RMSE	MAE
Training Set	23.22	8.42
Test Set	129.539	49.647

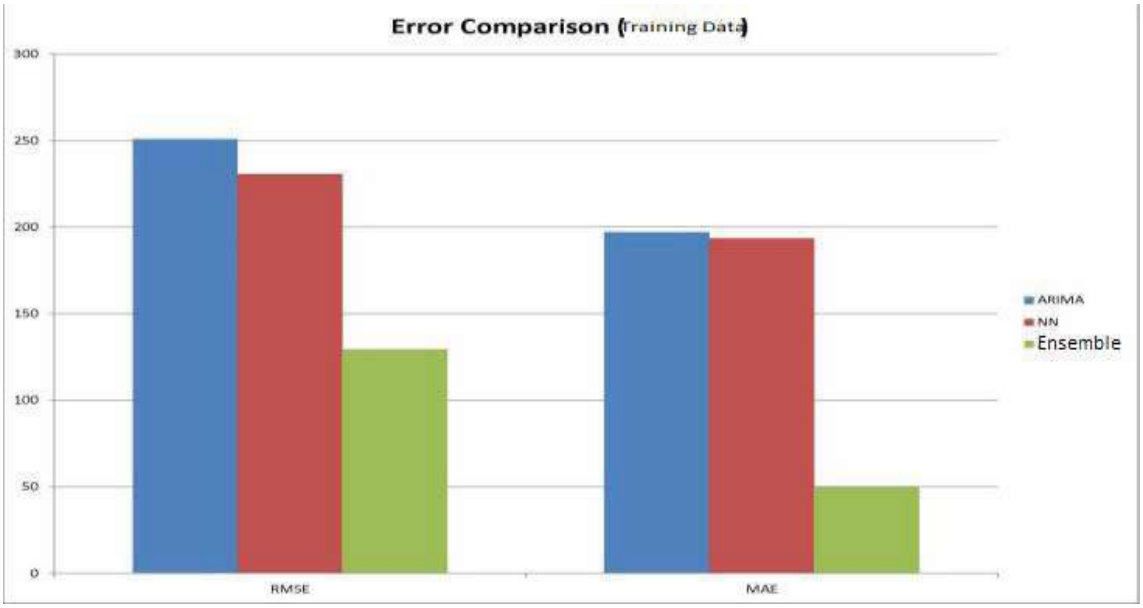


FIGURE 3.21: Comparison of RMSE and MAE value between the three Models (Training Set)

The RMSE, MAE and the Error function (E) for neural network model for Debian bug number series for the training set and test set are given in Table 3.11. In this section we give a comparative analysis of predictive performance of the ARIMA, Neural network and the hybrid (Nonlinear ARIMA) model. The results confirms the ensemble model as a better predictor for Debian bug number series. Figure 3.21 and Figure 3.22 show the bar chart representation of comparison between RMSE and MAE for both training and test set for the three models respectively.

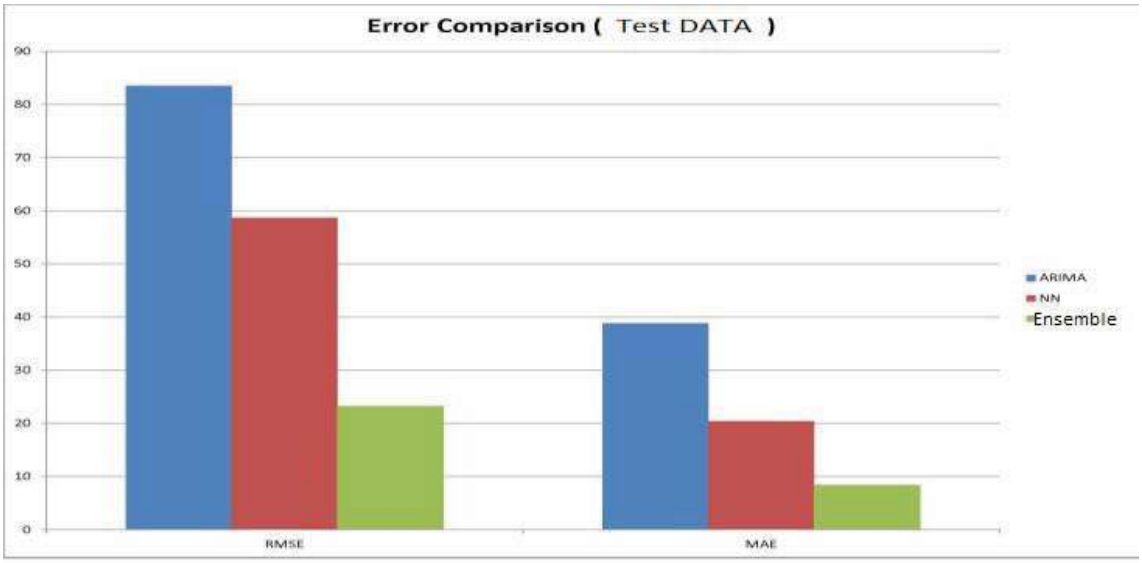


FIGURE 3.22: Comparison of RMSE and MAE value between the three Models(Test Set)

3.7 Prediction of Temporal Bug Patterns of Debian Using Markov Model

This work [166] demonstrates the distribution of temporal bug patterns of an open source software and also the effectiveness of Markov Models in predicting bug growth patterns. As a generalized model, this can also be applicable for bug growth patterns for closed source software. In this work, we have used Markov model for predicting the bug growth patterns in Debian, an open source operating system [15]. We have associated different states to the increasing and decreasing bug patterns to model them as Markov chain. The entire bug number data is clustered using K Median clustering to generate K clusters which represent different states of Markov Chain.

3.7.1 Markov Chain Model

Markov chain [202] is a stochastic process which can be parameterized by empirical estimation of transition probabilities between discrete states in the system under study [130]. In the 1st order Markov chain, the next state only depends on the current state. On higher order Markov chains the next state depends on two or more preceding states.

Let $Z(t)$ be a stochastic process, Containing discrete States Space
$S = 1,2,3....K$
where time series is $t_1 < t_2 < ....t_n$ The conditional probability can be represented as:
$P(x_{t+1} = i_{t+1}|x_0 = 0, x_1 = 1, ...x_t = t) = P(x_{t+1} = i_{t+1}|x_t = i_t)$
This is called one step transition probabilities of Markov chain. These probabilities can be written as:
$P_{ij} = P(x_{t+1} = i|x_t = j)$
The Matrix $P = (P_{ij})_{k\times k}$ is called transition matrix. The elements of P must satisfy two properties: $0 < p_{ij} < 1 \quad \forall i, j \in S$
$\sum_{i=1}^{k} p_{ij} = 1 \quad \forall j \in S$

The 1st order Markov chain model can be represented as $X_{t+1} = PX_t$,
For 3 states, the first order transition matrix P has a size of 3×3. The transition matrix for 2nd order P has a size of 9×3. The 3rd order transition matrix has a form 27×3.

3.7.1.1 Conversion from Higher Order to 1st Order

Higher order Markov chain depends on historical information. For example, a 3rd order Markov chain depend on last 3 states visited. At a given time t, the probability of state s at time $t+1$ is $P(s|x,y,z)$, where x,y,z are current and two preceding states. Our focus is on 1st - order Markov chain because any higher-order Markov chain can be converted

into a 1st order Markov chain [10]. For instance if in a 3rd order Markov chain, the states visited are $...a,b,c,d,a,b,e,a,..$ and if we are interested in predicting probability of next state to d, it depends on its recent history: b,c,d. To construct the equivalent 1st order Markov chain, we have to name the states in such a way that history can be reconstructed from the state name. We have to collect three states worth of history and encode the information into the name of the current state. When the third-order Markov chain is in state d, the equivalent 1st order would be in state and the states visited will be $...??a,?ac,and,cda,dab,abe,bea,ea?a??.....$ Each of the state names gives us the same information that the third-order Markov chain gives at the same moment. So, any information that we get in higher-order representation can be deduced from the longer names in the 1st order representation.

3.7.2 Data Preparation

In this work, we have studied the bug number growth per month, and we have obtained the original data from the public Debian bug data that is available in Ultimate Debian Database[4].We have analyzed the Debian bug data from January 2000 to February 2013. The bug number series has significantly increasing trend from 2000- 2013. The bug growth behavior from 2009- 2013, 2000 - 2008 and 2000-2013 is given in Figure 3.23, Figure 3.24 and Figure 3.25.

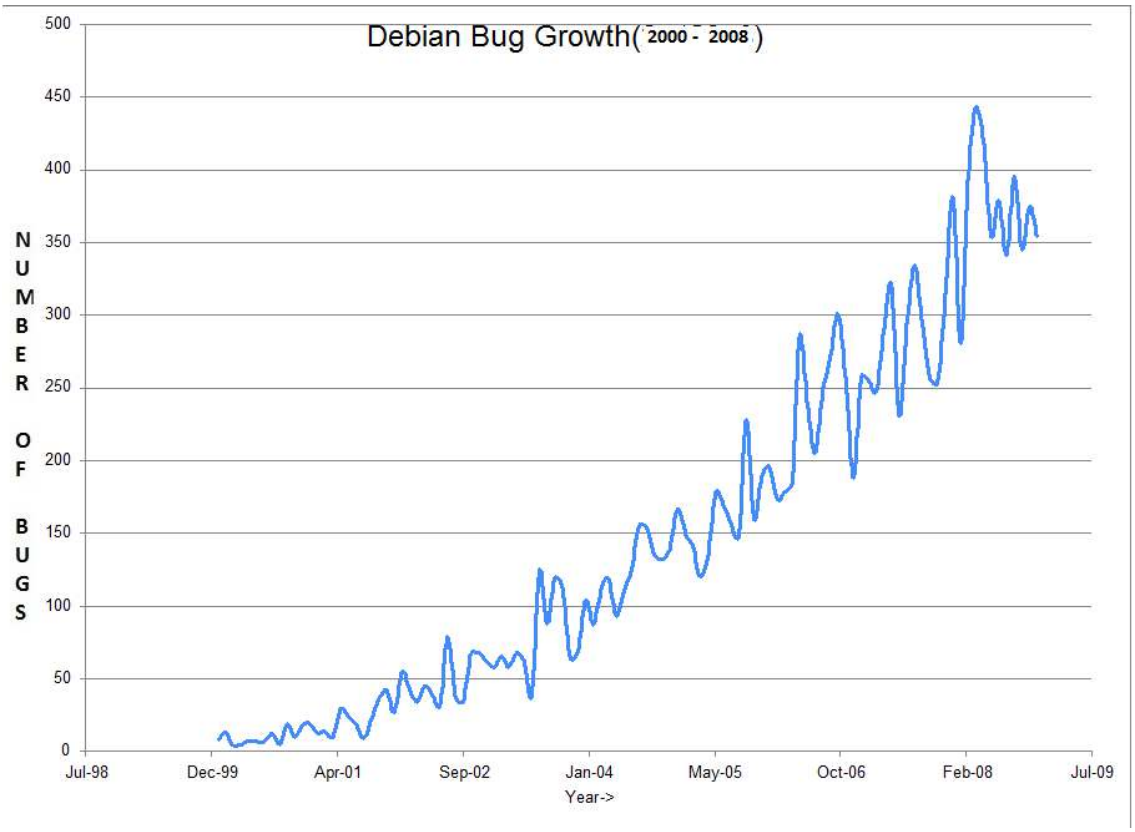


FIGURE 3.23: Debain Bug Growth Pattern:2000-2008

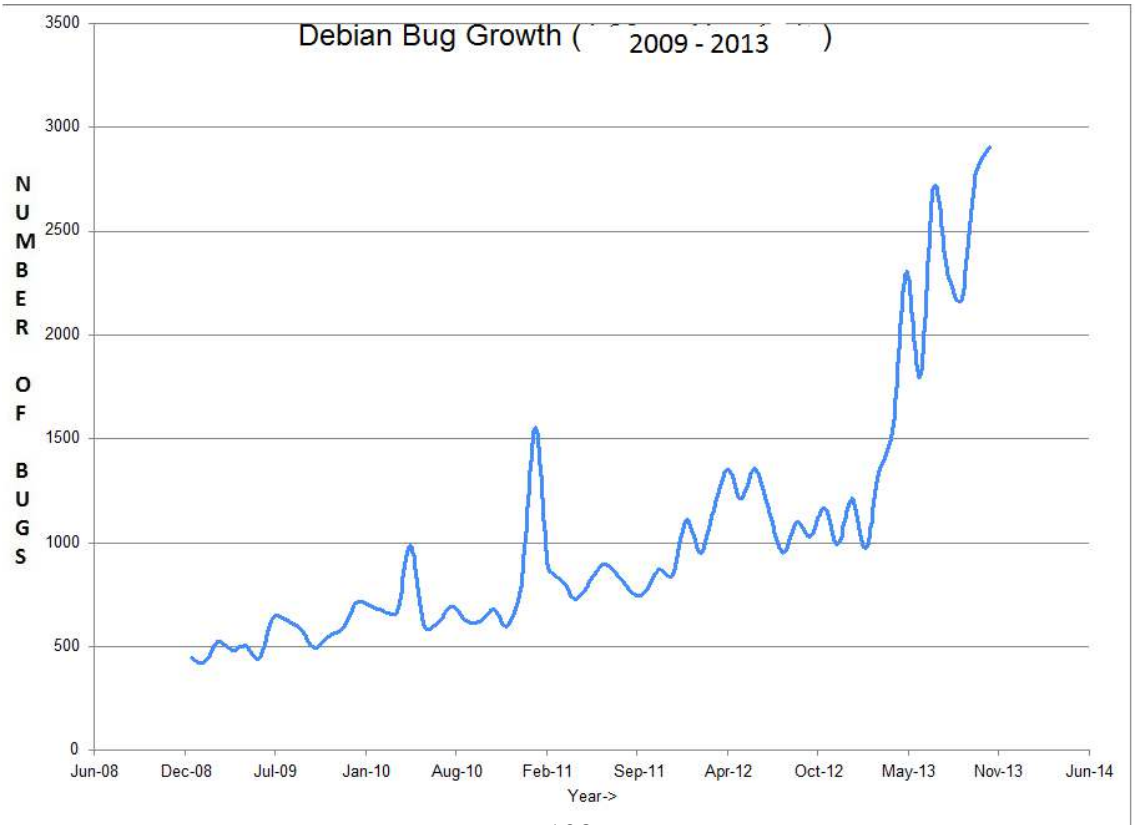


FIGURE 3.24: Debain Bug Growth Pattern:2009-2013

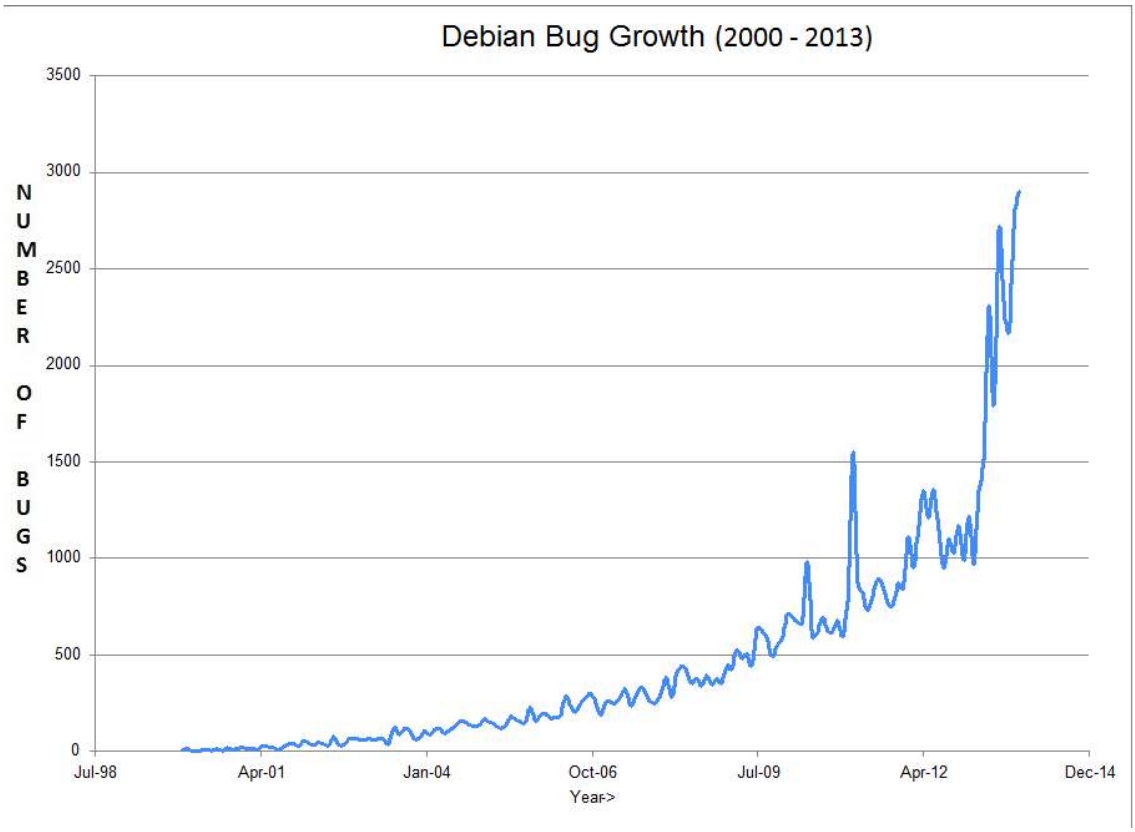


FIGURE 3.25: Debain Bug Growth Pattern:2000-2013

3.7.2.1 Stationarity Test

Here, ADF (Augmented Dickey-Fuller Test)[4] is used to test the behavior of Debian bug number series. The ADF test is used to check the presence of a unit root which leads to violation of assumption of classical linear regression in autoregressive time series models. Presence of a unit root indicates the non stationary behavior of a time series. The standard Dickey Fuller test estimates the equation as given below:

$\Delta y_t = Y_{t-1} + \varepsilon_t$

The Dickey Fuller test is only valid for AR(1) processes. If the time series is correlated at higher lags, the augmented Dickey Fuller test performs a parameter correction for higher order correlation, by adding lag differences of the time series. In this work, the ADF test is performed in the [web:reg] [16] (an Add-In to Excel written by Kurt Annen.). The

unit-root existence is checked in the following formulations:

1. With Constant or intercept $\Delta y_t = \alpha + \gamma Y_{t-1} + \varepsilon_t$

2. With constant+trend $\Delta y_t = \alpha + \gamma Y_{t-1} + \beta \times t + \varepsilon_t$

The ADF test is basically $H_0 : \gamma = 0$
If the test statistic is less (Due to non-symmetrical nature of the test absolute value is not considered) than (a larger negative) the critical value, then the null hypothesis of $H_0 : \gamma = 0$ is rejected and it indicates absence of unit roots.

In our case, we observed that the Debian Bug Number Series is non-stationary as there is the presence of unit roots.

3.7.3 Generation of Markov Model

To model the bug growth pattern as Markov model the requirement is to have different states and transition probability between the states.

3.7.3.1 Formation of Markov States

Here we have considered the frequent increasing and decreasing behavior of bug growth patterns as different states. The entire bug data series contains random number of increasing and decreasing patterns. The change in bug number is calculated by taking the first difference in the bug number time series. Taking the 1st forward difference makes the series stationary as shown by ADF test similar to section 3.1. Therefore Markov chain

model is justified. The increasing and decreasing bug pattern from 2000-2013 is given in Figure 3.27. The sorted order of change in bug number series is also plotted in Figure 3.28, which shows the symmetrical distribution of increasing and decreasing bug patterns. We have used K-Median clustering to cluster the entire increasing and decreasing patterns into K clusters due to the symmetric behavior of data series. Each cluster represents one state of Markov chain. Here, we have taken K=5. Hence we get 5 clusters. Each of them represents one state of Markov chain. After applying clustering to the changing bug growth pattern, we get 5 states:

1. HD (High Decrease)
2. SD (Slow Decrease)
3. I (Insignificant change)
4. SI (Slow Increase)
5. HI (High Increase)

The diagrammatic representation of Markov States is presented in Figure 3.26.

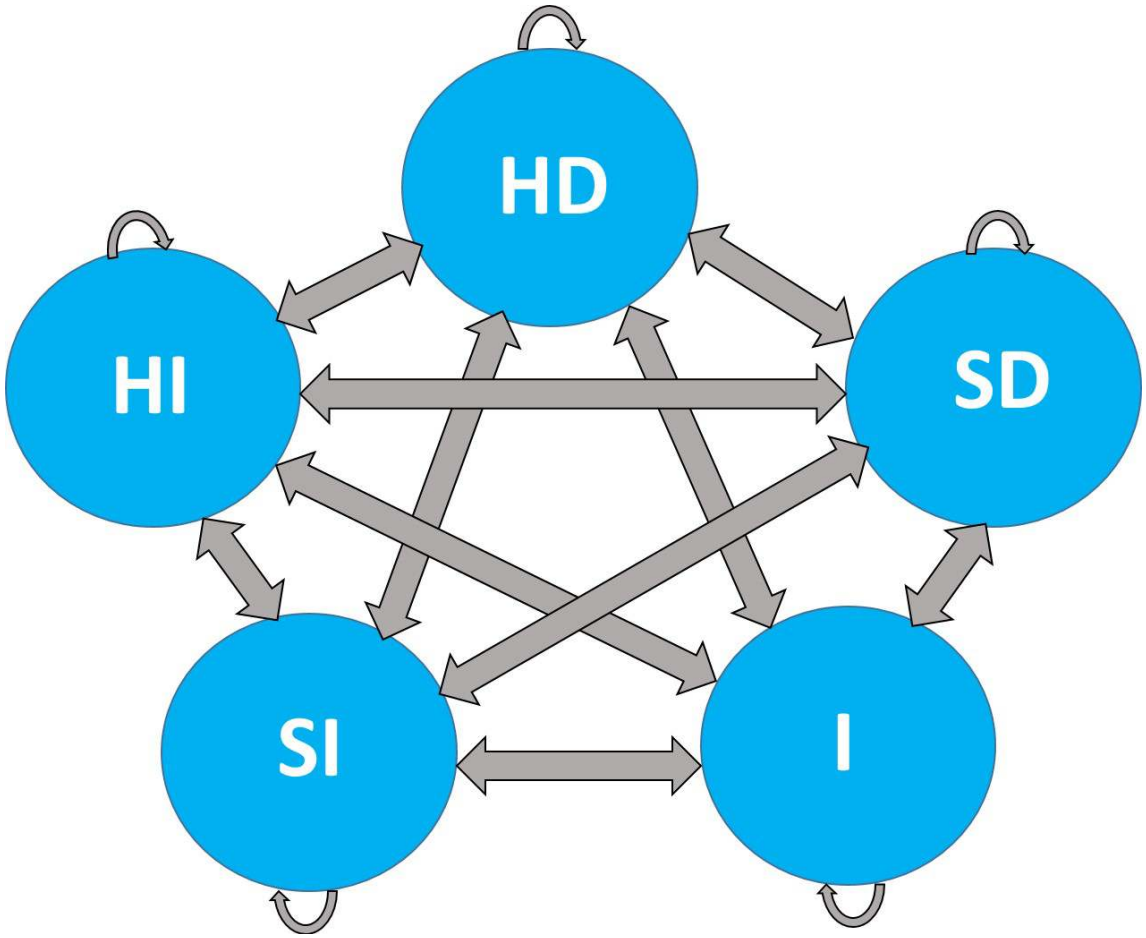


FIGURE 3.26: Markov State Representation

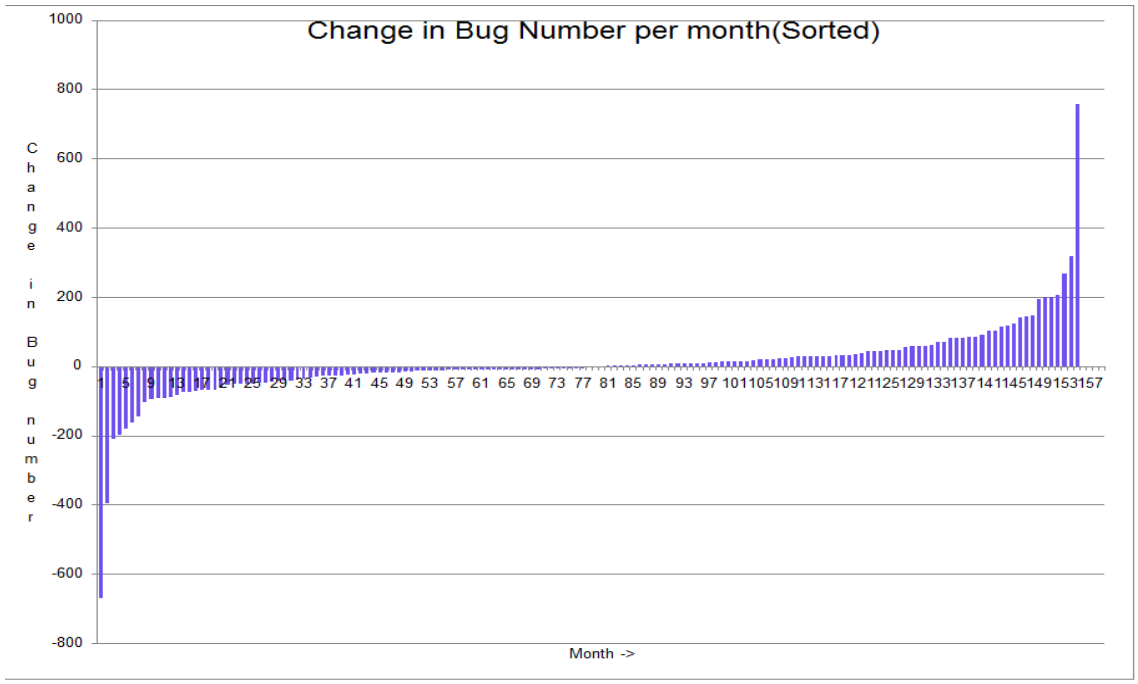


FIGURE 3.28: Plot for Change in Bug Number per Month (Sorted)

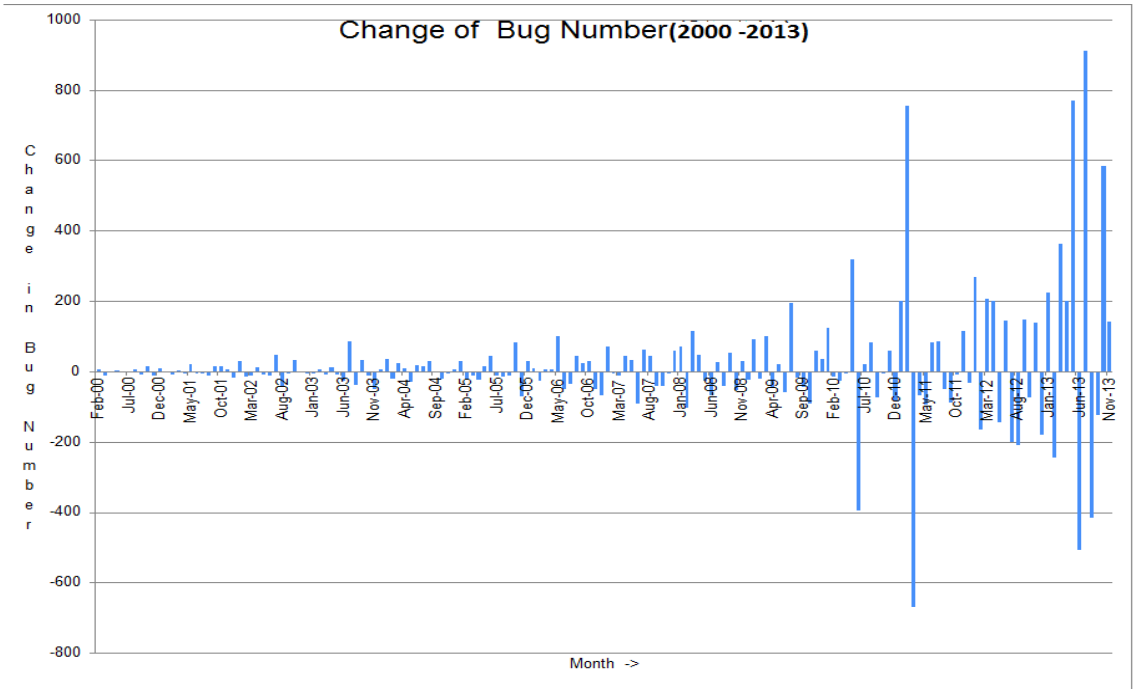


FIGURE 3.27: Plot for Change in Bug Number per Month

TABLE 3.12: Table Showing Threshold Values of Different States

States	Lower Threshold	Upper Threshold
HD	-667	-38
SD	-36	-1
I	0	15
SI	17	70
HI	81	756

The upper threshold and lower threshold values of different states as decided by K-median clustering are given in Table 3.12.

3.7.3.2 K- Median clustering

In data mining, K- median clustering is a variation of K-means clustering where instead of calculating cluster mean to determine its centroid, the median is chosen [9]. The goal of K-median clustering is to create distinct groups based on differences in the data. After clustering, we get K-distinct groups with different characteristics. We have used K-

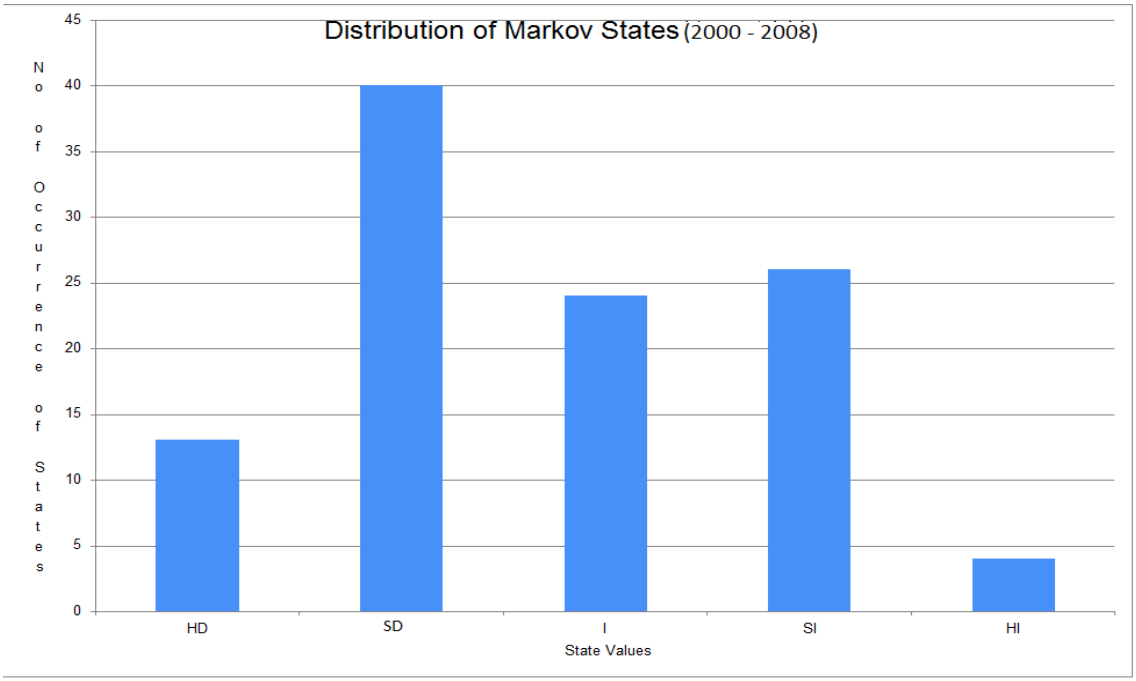


FIGURE 3.29: Distribution of Markov States-1[2000-2008]

median clustering considering the Euclidian distance between data items as our distance metrics. The reason for choosing K-median is that it is not sensitive to outliers. An exploratory Analysis of the Debian data showed the existence of significant outliers. For other data sets we can choose an appropriate clustering algorithm for identifying the number of states.

3.7.4 Distribution of Markov States

The next step is to analyze the distribution of different states in a particular time span. We have divided the entire series into two parts. The first part from 2000-2008, the second part from 2009-2013. Now the distribution of different states in these durations is given in Figure 3.29 and Figure 3.30.

- From Figure 3.29, we found that from 2000-2008, the most frequent state is SD followed by SI and I and least frequent states are HI and HD.

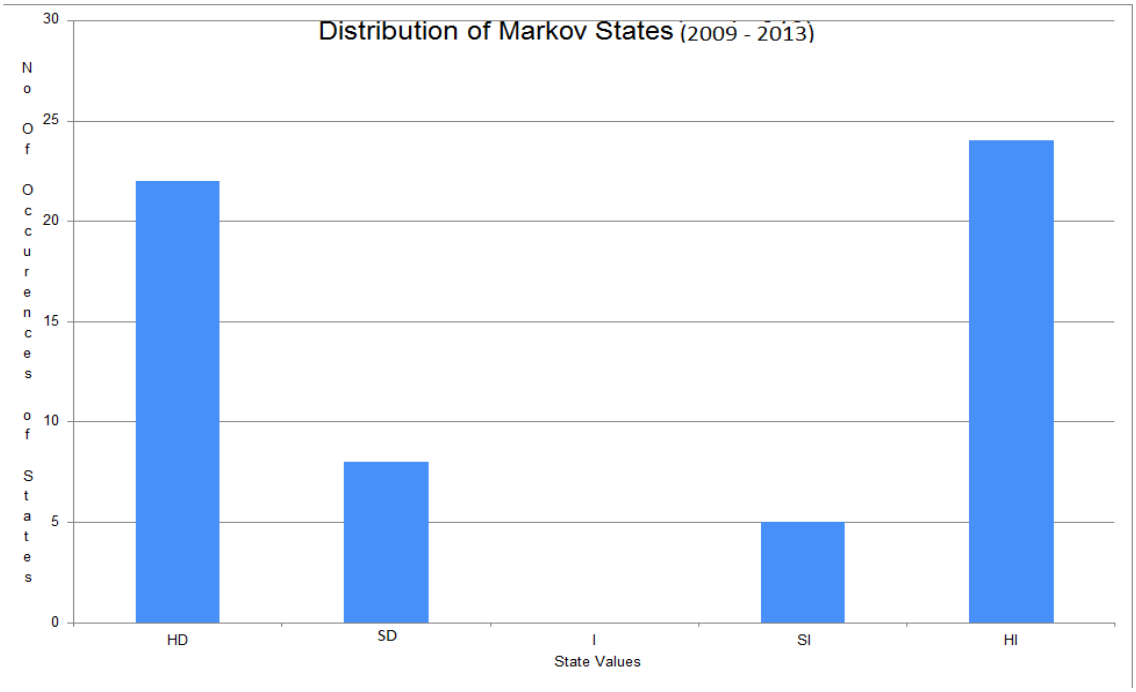


FIGURE 3.30: Distribution of Markov States-1[2009-2013]

- But as shown in Figure 3.30, from 2009-2013 the most frequent states are HI followed by HD which shows large values of increasing and decreasing bug patterns.
- An open source software is initially launched with a small number of people working with fewer components and less number of users using the application.This leads to small and insignificant changes in bug numbers.
- However, gradually the number of contributors and users increases which in turn increases the complexity of software and also the number of added components. The numbers of change requests and commits also increase which leads to large change in bug numbers.

3.7.5 Formation of Transition Matrices

After getting the corresponding state values for bug number data, the next step is to design them as Markov chain. We have arranged the data as first order and second order Markov

TABLE 3.13: Transition Probability for 1st order Markov Model (Training Period: 2000 - 2012)[X: Transition Absent]

	Transition Probability (1st Order)	HD	SD	I	SI	HI
1	HD	0.26666	0.13333	0	0.066667	0.4
2	SD	0	0.4	0	0.2	0.4
3	I	X	X	X	X	X
4	SI	0.5	0	0	0	0.5
5	HI	0.692	0.076	0	0	0.23

chain. In first order Markov chain, the next state is only dependent on the current state, and no history is required. The second order Markov chain requires history for predicting the next state.

Our next step is to calculate the transition probabilities for all types of Markov chains. We estimate the transition probability matrix P_i by the following method.

Given the data series, the transition frequency is counted from the states in the sequence at time $t = m - i + 1$ to the states in the jth sequence at time $t = m + 1$ for $1 \leq i \leq n$.

Thus we construct the transition frequency matrix for the data sequences.

For 2nd order Markov model the data sets are first converted into 1st order Markov model by the technique given in the previous section, and then the transition frequencies are calculated. In our case, as we have 5 states we have transition probability matrix of size 5×5 for first order Markov chain model has been shown in Table 3.13. The second order transition probability matrix is of size 25×5, which is shown in Table 3.14.

From the transition probability matrix we can get the most probable next state values from a given a state values. The state which has highest transition probability $P_{Max}\{State1 \rightarrow State2\}$ in a row is most likely to occur as next state. Table 3.15,3.16 shows the most probable state from a given state for a Markov model trained for the aforementioned period.

TABLE 3.14: Transition Probability for 2nd order Markov Model (Training Period: 2000 - 2012)[X: Transition Absent]

	Transition Probability (2nd Order)	HD	SD	I	SI	HI
1	HD HD	0.25	0.25	0	0	0.5
2	HD SD	0	0	0	0.5	0.5
3	HD I	X	X	X	X	X
4	HD SI	0	0	0	0	1
5	HD HI	0.5	0	0	0	0.5
6	SD HD	X	X	X	X	X
7	SD SD	0	0.5	0	0	0.5
8	SD I	X	X	X	X	X
9	SD SI	1	0	0	0	0
10	SD HI	0.66667	0.3333	0	0	0
11	I HD	X	X	X	X	X
12	I SD	X	X	X	X	X
13	I I	X	X	X	X	X
14	I SI	X	X	X	X	X
15	I HI	X	X	X	X	X
16	SI HD	0	0	0	0	1
17	SI SD	X	X	X	X	X
18	SI I	X	X	X	X	X
19	SI SI	X	X	X	X	X
20	SI HI	1	0	0	0	0
21	HI HD	0.375	0.125	0	0.125	0.375
22	HI SD	0	0.5	0	0	0.5
23	HI I	X	X	X	X	X
24	HI SI	X	X	X	X	X
	HI HI	1	0	0	0	0

TABLE 3.15: Most Probable Next State value 1st order Markov Model (Training Period: 2010 - 2012)[X: Transition Absent]

Current State	Most Probable State
HD	HI
SD	SD HI
I	X
SI	HD HI
HI	HD

TABLE 3.16: Most Probable Next State value 2nd order Markov Model (Training Period: 2010 - 2012)[X: Transition Absent]

Current State	Most Probable State
HD HD	HI
HD SD	SI HI
HD I	X
HD SI	HI
HD HI	HD HI
SD HD	X
SD SD	SD HI
SD I	X
SD SI	HD
SD HI	HD
I HD	X
I SD	X
I I	X
I SI	X
I HI	X
SI HD	HI
SI SD	X
SI I	X
SI SI	X
SI HI	HD
HI HD	HD HI
HI SD	SD HI
HI I	X
HI SI	X
HI HI	HD

Interpretation and Analysis of Transition Matrices

The transition matrix we got from the Markov model can be interpreted to give some valuable information which is in conformance with practical open source software development.

As shown in Table 3.15 we get from the 1st order transition probability matrix that every state except HI, has the maximum probability to go to HI. Because as the open source software grows with time, the number of components are being added and number of

contributors also increases rapidly. The number of commits also increases. This leads to huge increase in bug numbers. The interesting thing is that as we reach HI, the most probable next state is HD. This is due to the fact that as the bug number increases above some peak value, huge bug fixing is done to reduce it to decrease the bug numbers. This process continues until software achieves mature phase. As we have trained the data from 2010 to 2013, we observed that absence of I (Insignificant) states. They only occur in the initial phase.

As shown in Table 3.16 in the 2nd order Markov model we also found that out of 12 feasible transitions 8 transitions gives most probable next state as HI. This is also due to huge increase in commits and no of contributors. When bug increases it is also followed by huge bug fixing which in turn decreases the bug numbers.

3.7.6 Evaluation

To evaluate the performance of the markov model, we have used following formulae. The prediction accuracy r is given by:

$r = \frac{1}{(N-n)} \times \sum_{t=n+1}^{N} \times 100$

Here N is the length of Data Sequence.

$$where \quad a_t = \begin{cases} 1, & if \quad x_t = \hat{x_t} \\ 0, & otherwise \end{cases}$$

x_t: represents the expected next state from the model.

$\hat{x_t}$: represents the observed next state from the model.

$'N'$: represents the total number of instances for a particular period and $'n'$ represents the total number of training instances.

$'N-n'$: represents the number of test instances.

TABLE 3.17: Table Showing Training set, Test set and Accuracy Information(1st Order Markov Chain)

Sl No:	Training Set	Test Set	Accuracy
1	2011, 2012	2013(Up to Nov)	60%
2	2010, 2011, 2012	2013(Up to Nov)	60%
3	2009, 2010, 2011,2012	2013(Up to Nov)	60%
4	2008, 2009, 2010, 2011, 2012	2013(Up to Nov)	60%
5	2007, 2008, 2009, 2010, 2011, 2012	2013(Up to Nov)	40%

During testing, if the predicted state and observed state are same then it will be counted as a True(T), and we increase the counter by one.

Otherwise, It will be counted as False(F) and counter remains same. The performance is the ratio of a number of successful transitions to the total number of transition.

3.7.7 Results and Interpretation

To evaluate the performance of our model we tested it upon different number of training instances. We have also evaluated the performance of higher order Markov chain on the same training instances. Then we have also evaluated the performance based on the size of the prediction window. The prediction window size may be 12(Yearly) or 24(Bi-yearly).

3.7.7.1 Evaluation based on number of Training Instances

We have trained the Markov model for different training window size like 24,36,48,60, 72. We found that for small size of training window accuracy remain same. Also we observed that increasing the training window size too much (more than 5 years) also reduces the accuracy. It is due to the fact that there is change in the trend of bug patterns with time which in turn changes the distribution of states. This results in reduction of prediction accuracy. Table 3.15 shows the most probable state from a given state for the Markov

TABLE 3.18: Prediction Result(1st order Markov Model)

Current State	Next State	Prediction Result (T/F)
HI	HD	T
HD	HI	T
HI	HI	F
HI	HI	F
HI	HD	T
HD	HI	T
HI	HD	T
HD	HD	F
HD	HI	T
HI	HI	F

TABLE 3.19: Table Showing Training set, Test set and Accuracy Information(2nd Order Markov Chain)

Sl No:	Training Set	Test Set	Accuracy
1	2011,2012	2013(Up to Nov)	88%
2	2010,2011,2012	2013 (Up to Nov)	88%
3	2009,2010,2011,2012	2013 (Up to Nov)	88%
4	2008,2009,2010,2011,2012	2013 (Up to Nov)	88%
5	2007,2008,2009,2010,2011,2012	2013 (Up to Nov)	55%

model trained on training set sl no: 2. The Table 3.17 shows accuracy information for 1st order Markov chain based upon different training window.

3.7.7.2 Evaluation based on Higher Order Markov Chain

We have also trained the 2nd order Markov model for different training window size like 24,36,48,60, 72. For second order we get better accuracy than 1st order Markov chain. Table 3.16 shows the most probable state from a given state for the Markov model trained on training set sl no: 2. Table 3.20 show the prediction results for test set for 2nd order Markov Model. As discussed earlier the 2nd order Markov model is converted to 1st order and then the transition probabilities are calculated. As shown in the Table 3.20, $P(HDHD \rightarrow HDHI)$ is equivalent to $P(HDHD \rightarrow HI)$.

TABLE 3.20: Prediction Result(2nd order Markov Model)

Current State	Next State	Equivalent State	Prediction Result (T/F)
HI HD	HD HI	HI	T
HD HI	HI HI	HI	T
HI HI	HI HI	HI	F
HI HI	HI HD	HD	T
HI HD	HD HI	HI	T
HD HI	HI HD	HD	T
HI HD	HD HD	HD	T
HD HD	HD HI	HI	T
HD HI	HI HI	HI	T

TABLE 3.21: Table Showing Training set, Test set and Accuracy Information for different prediction window.A: 1st Order Markov Chain B: 2nd Order Markov Chain

Prediction Window Size	Training Set	Test Set	Accuracy: A	Accuracy: B
Yearly	2010, 2011, 2012	2013(up to Nov)	60%	88%
Bi-yearly	2009,2010, 2011	2012,2013(up to Nov)	48%	68%

3.7.7.3 Evaluation Based upon Size of Prediction Window

We have also tested the performance based on size of the prediction window (Yearly and Bi-yearly). The result of the prediction using 1st order Markov chain is given in Table 3.21. We have also plotted the comparison of predictive performance of the 1st order and second order Markov model on different size of prediction window in Figure 3.31. The figure shows that the both models give better result on prediction window of smaller size. We also observed that 2nd order gives better prediction accuracy than 1st order Markov model.

This work adopts Markov chain modelling to predict the software bug growth patterns. The bug data values are represented as 1st order and higher order Markov chain for monthly data values. Our results show the Markov chain as a better predictor of software bug growth patterns.

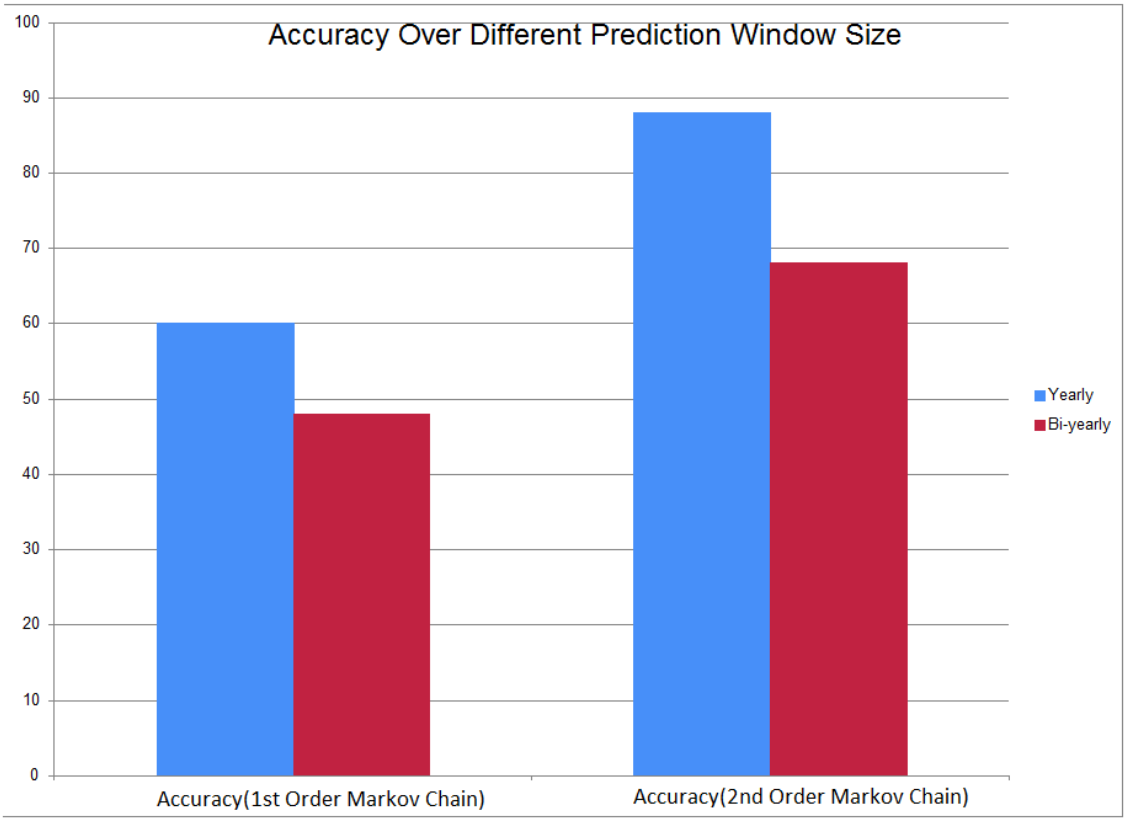


FIGURE 3.31: Prediction Result for Prediction window of different size [Markov Model(1st Order vs 2nd Order)]

3.8 Analysis of Temporal Bug Patterns in Open Source Software Using Hidden Markov Model[169]

In this work, we have used Hidden Markov Models [202] to analyze the temporal bug patterns of two open source software applications: These are Mozilla Firefox Browser and Debian Operating System. Here we have considered the increase and decrease in bug numbers with respect to time as observation sequences and define them as temporal bug patterns. We have used the number of contributors and the number of commits as the hidden states affecting the observed temporal bug patterns. The main objective of the work is modelling of the temporal bug patterns for the open source software applications. Three types of Hidden Markov Models are built using the number of contributors, the

number of commits and the simultaneous effect of both on the temporal bug patterns. We have also shown the effectiveness of these models in predicting short and long temporal bug patterns.

3.8.1 Hidden Markov Model

Hidden Markov Models[202] are popularly used in temporal classification and areas like speech, handwriting and gesture recognition. A conventional finite state machine [45]

emits a deterministic symbol in a particular state and it also deterministically changes to another state. Hidden Markov Models do neither deterministically; rather both the transitions and emissions happen probabilistically [154]. To summarize and formalize, an HMM has the following elements:

1. N:Number of states in the model;$S = S_1, S_2, S_3, ...S_N$
2. M: Number of Distinct Observation Symbols; $V = v_1, v_2, ...v_N$
3. State transition probabilities: $A = [a_{ij}] where a_{ij} = P(q_{t+1} = S_j | q_t = S_i)$
4. Observation Probabilities: $B = [b_j(m) where P(O_t = v_m | q_t = s_j)]$
5. Initial State Probabilities: Π =$\Pi_i$where $\Pi_i = P(q_t = S_i)$

Here,$\lambda = (A, B, \Pi)$ is taken as the parameter set of an HMM.

Given λ, the model can be used to generate an arbitrary number of observation sequences of arbitrary length. We can also estimate the parameters of the model given a training set of sequences.

3.8.1.1 The Basic Problems of HMM

In giving a sequence of observations, three problems can be formulated using HMM [154]:

1. Given a model λ, we would like to evaluate the probability of any given observation sequence, $O = O_1, O_2, O_3, ...O_T$ i.e. $P(O|\lambda)$

2. Given a model λ and an observation sequence O, we would like to find out the state sequence: $Q = q_1, q_2, ...q_T$, which has the highest probability of generating O; i.e. we want to find Q^* that maximizes $P(Q|O, \lambda))$

3. Given a training set of observation sequences X, we would like to learn the model parameters that maximizes the probability of generating X; we want to find λ^*, that maximizes $P(X|\lambda)$.

3.8.2 Data Preparation

In this work, we have analyzed the temporal bug patterns of open source software applications. The original Debian bug number data was obtained from Ultimate Debian Database (UDD) [15]. The Mozilla Firefox bug number data was obtained from Bugzilla [11]. Here, we analyzed 82 monthly counts for Debian OSS and 82 monthly counts for Mozilla Firefox data respectively. The temporal bug patterns behavior for Debian and Mozilla Firefox are visualized in Figure 3.32 and Figure 3.33 respectively.

We also collected the contributor data and committed data for Mozilla Firefox from the open source project website [13]. The growth patterns for the number of commits and the number of contributors is shown in Figure 3.34 and Figure 3.35 respectively.

3.8.3 Model Building

After the bug number data was collected the next phase is to arrange them in the form of increasing and decreasing bug number sequences. The increasing and decreasing bug

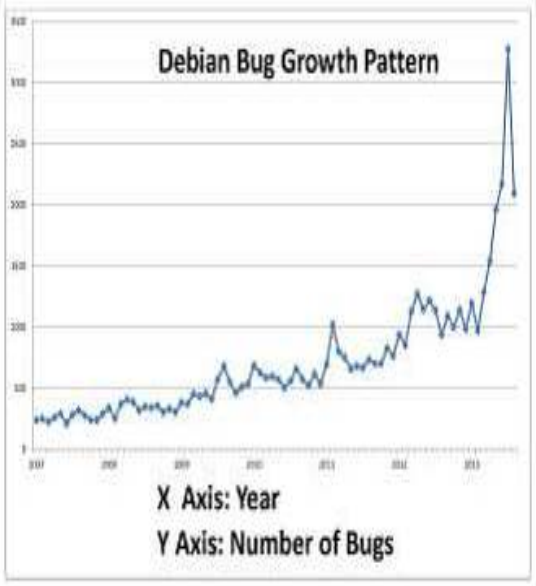


FIGURE 3.32: Debian Bug Growth Pattern

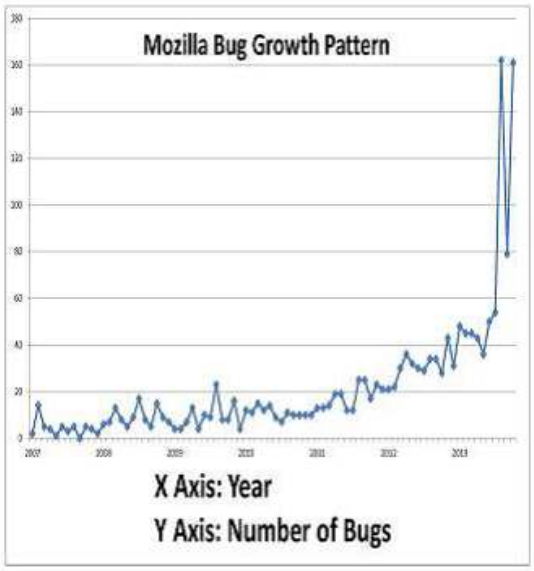


FIGURE 3.33: Mozilla Bug Growth Pattern

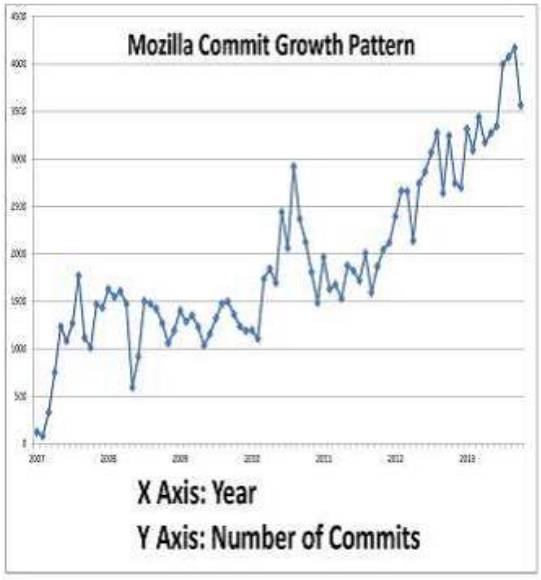


FIGURE 3.34: Mozilla Commit Growth Pattern

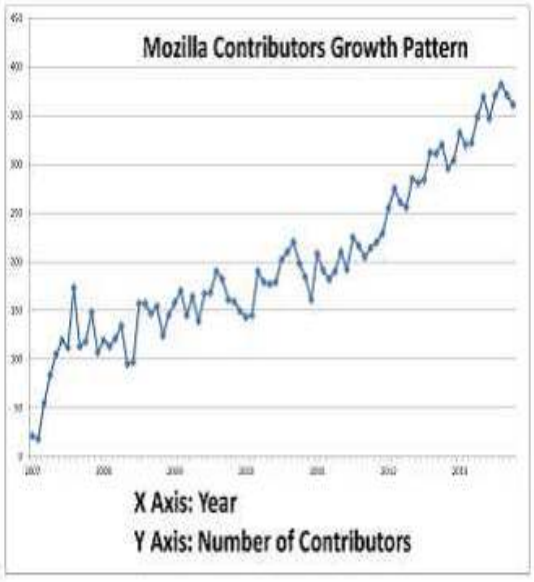


FIGURE 3.35: Mozilla Contributors Growth Pattern

numbers are represented as Observation State- I (Increasing), Observation State-D (Decreasing). Both Debian and Mozilla bug data are stored in the form of increasing and decreasing states. The Increasing and Decreasing contributors' information for Mozilla Firefox is represented as

TABLE 3.22: State Information for Hybrid Model

Commit State	Contributor State	Hybrid State
Com-D	Con-D	LOW
Com-I	Con-I	HIGH
Com-I	Con-D	MID1
Com-D	Con-I	MID2

1. Con-I (Increasing Contributors)

2. Con-D (Decreasing Contributors)

Similarly, the Commits are represented as:

1. Com-I (Increasing Commits)

2. Com-D (Decreasing Commits)

We can also have a Hybrid model which takes both commits and contributors information simultaneously. Now for the hybrid model the state can be represented as given in Table 3.22. The number of contributors and commits is considered as hidden states for the increasing and decreasing bug growth patterns. We have built a Hidden Markov Model having observation sequences and hidden state sequences. The observation sequence and state sequence as discussed above are represented below in Figure 3.36.

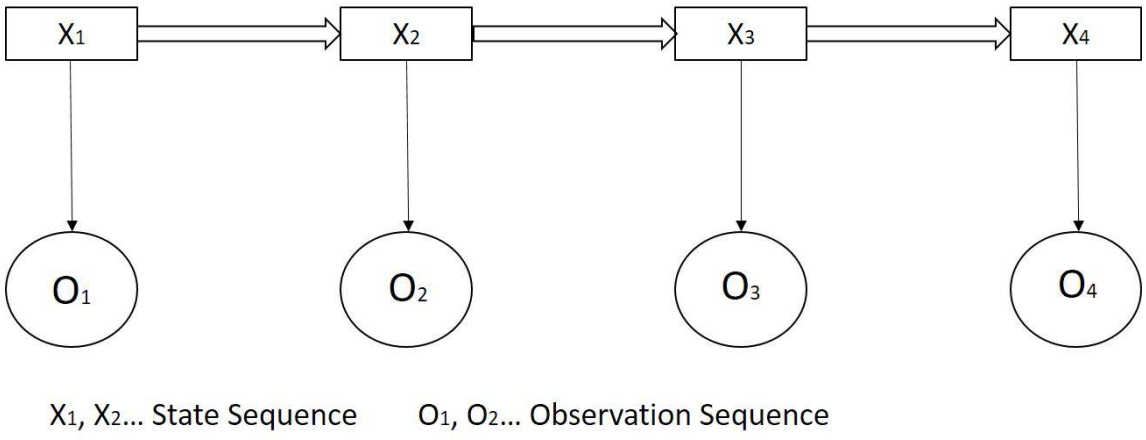


FIGURE 3.36: HMM Model for Temporal Bug Patterns

In Figure 3.36, X_1, X_2.... are the hidden state sequence (Con-I or Con-D or Com-I or Com-D or combination of any two) and O_1, O_2.... are observed temporal bug patterns $(IorD)$.

The objective is to analyze the observation sequence based upon the given state sequence and to build an HMM that can predict bug patterns for test observation sequence, i.e., data other than the training set with the desired accuracy.

3.8.4 Model Implementation

After getting observation sequence and the corresponding state sequences, we can create different HMM models to analyze the temporal patterns associated with the bugs in open source software applications. We constructed three different types of HMM models using the hidden states sequences and temporal bug sequences.

3.8.4.1 HMM Model 1

- Observation Sequence: Sequence of I (Increasing Bug Number) and D (Decreasing Bug Number).

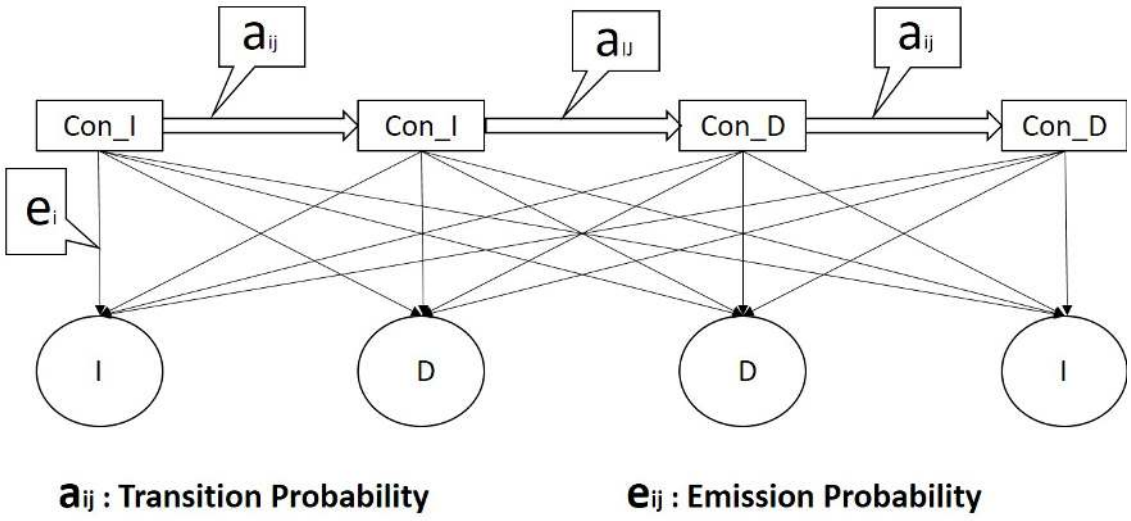


FIGURE 3.37: HMM Mode-l for Temporal Bug Patterns

- State Sequence: Sequence of Con-I (Increasing Contributors) and Con-D (Decreasing Contributors).

After implementation, the transition and emission probabilities are estimated from the available data sequences. Figure 3.37 represents the HMM model 1.

3.8.4.2 HMM Model 2

- Observation Sequence: Sequence of I (Increasing Bug Number) and D (Decreasing Bug Number).
- State Sequence: Sequence of Com-I (Increasing Commits) and Com-D (Decreasing Commits). Figure 3.38 represents the HMM model 2.

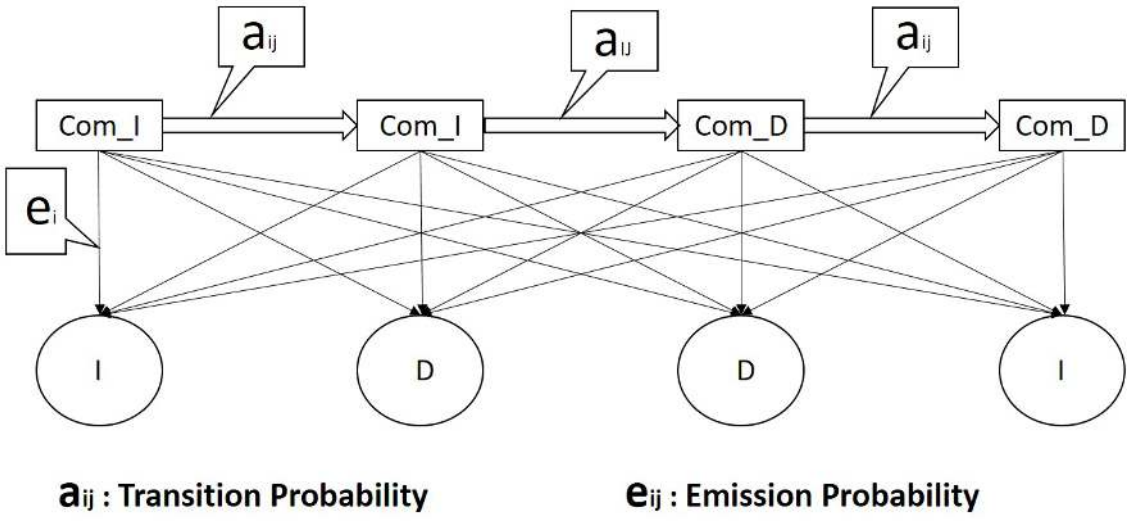


FIGURE 3.38: HMM Model-2 for Temporal Bug Patterns

3.8.4.3 HMM Model 3

- Observation Sequence: Sequence of I (Increasing Bug Number) and D (Decreasing Bug Number).
- State Sequence: Sequence of LOW (Decreasing Contributors and Commits), HIGH (Increasing Contributors and Commits), MID1 (Increase in Commits and Decrease in Contributors), MID2 (Decrease in Commits and Increase in Contributors).

  Figure 3.39 represents the HMM model 3.

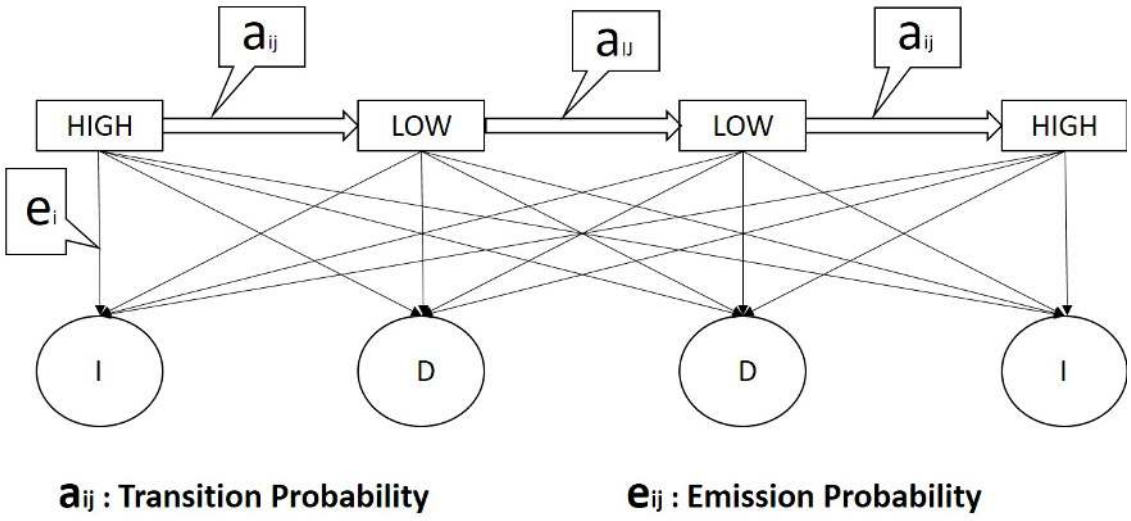


FIGURE 3.39: HMM Model-3 for Temporal Bug Patterns

3.8.5 Results and Interpretation

After HMM model is built, the next step is to evaluate the model on different observation and state sequences.

3.8.5.1 Most probable state sequence

The effect of State on Observation Sequences (Based on Contributors and Commits) is studied. We have sampled 10 observation sequences of size 10 each from the observed Mozilla Firefox bug number sequences. Then we apply HMM to find the state sequence that has the maximum probability of generating the observation sequence.

1. First, we have taken Contributors information as the state sequences. The result is tabulated in Table 3.23.

2. Second, we have taken Commits information as the state sequences. The result is tabulated in Table 3.24.

TABLE 3.23: State Information HMM-1 [O: Observation Sequence, S: State Sequence, ConI: Increasing Contributor, ConD: Decreasing Contributor]

1	O	I	D	I	D	D	D	D	D	D	D
	S	ConI	ConD	ConI	ConD	ConD	ConD	ConD	ConD	ConD	ConD
2	O	D	D	D	D	D	D	I	D	I	D
	S	ConD	ConD	ConD	ConD	ConD	ConD	ConI	ConD	ConI	ConD
3	O	D	D	D	I	I	D	D	D	D	D
	S	ConD	ConD	ConD	ConI	ConI	ConD	ConD	ConD	ConD	ConD
4	O	D	I	I	D	D	D	I	D	I	I
	S	ConD	ConI	ConI	ConD	ConD	ConD	ConI	ConD	ConI	ConI
5	O	D	D	D	D	I	I	D	I	D	D
	S	ConD	ConD	ConD	ConD	ConI	ConI	ConD	ConI	ConD	ConD
6	O	D	I	I	D	I	D	I	D	I	D
	S	ConD	ConI	ConI	ConD	ConI	ConD	ConI	ConD	ConI	ConD
7	O	D	D	D	I	I	I	D	I	D	D
	S	ConD	ConD	ConD	ConI	ConI	ConI	ConD	ConI	ConD	ConD
8	O	I	D	I	I	I	D	D	D	D	D
	S	ConI	ConD	ConI	ConI	ConI	ConD	ConD	ConD	ConD	ConD
9	O	D	I	I	I	D	D	I	D	I	D
	S	ConD	ConI	ConI	ConI	ConD	ConD	ConI	ConD	ConI	ConD
10	O	I	I	I	D	I	D	D	D	D	I
	S	ConI	ConI	ConI	ConD	ConI	ConD	ConD	ConD	ConD	ConI

3. Third, we studied the effect of Hybrid State (Based on Contributors & Commits) on Observation Sequences. The result is tabulated in Table 3.25.

Interpretation

- From the 1st model result, we interpret that for all cases when the number of the contributor is increasing the bug also increases and when the number of the contributor is decreasing the bug also decreases. This is obvious for the open source software application as each contributor adds or deletes some portion of code to the database which in turn increases bugs. The inexperienced contributors may also add some new bug due to coding errors. It is noteworthy that although Linus's law[219]

TABLE 3.24: State Information HMM-2 [O: Observation Sequence, S: State Sequence, ComI: Increasing Commits, ComD:Decreasing Commits]

1	O	D	D	I	I	I	D	I	D	D	I
	S	ComD	ComD	ComI	ComI	ComI	ComD	ComI	ComD	ComD	ComI
2	O	I	D	D	I	I	I	D	D	I	D
	S	ComI	ComD	ComD	ComI	ComI	ComI	ComD	ComD	ComI	ComD
3	O	I	D	D	I	D	D	D	I	I	I
	S	ComI	ComD	ComD	ComI	ComD	ComD	ComD	ComI	ComI	ComI
4	O	I	D	D	I	D	D	D	I	I	I
	S	ComI	ComD	ComD	ComI	ComD	ComD	ComD	ComI	ComI	ComI
5	O	I	D	I	D	D	I	D	D	D	I
	S	ComI	ComD	ComI	ComD	ComD	ComI	ComD	ComD	ComD	ComI
6	O	I	D	I	I	D	D	I	D	I	D
	S	ComI	ComD	ComI	ComI	ComD	ComD	ComI	ComD	ComI	ComD
7	O	I	D	D	D	D	D	D	I	D	I
	S	ComI	ComD	ComD	ComD	ComD	ComD	ComD	ComI	ComD	ComI
8	O	D	I	D	I	I	I	I	I	I	I
	S	ComD	ComI	ComD	ComI	ComI	ComI	ComI	ComI	ComI	ComI
9	O	I	D	D	D	D	I	I	I	D	I
	S	ComI	ComD	ComD	ComD	ComD	ComI	ComI	ComI	ComD	ComI
10	O	D	I	D	I	D	D	D	D	I	D
	S	ComD	ComI	ComD	ComI	ComD	ComD	ComD	ComD	ComI	ComD

claims that ”Given enough eyeballs, all bugs are shallow.” there is a lack of supporting evidence. Apparently, there is a maximum number of useful reviewers, and additional reviewers uncover bugs at a much lower rate [219].

- From the second model result, we interpreted that for all the cases the number of commits also have a comparable relationship with bugs as the number of contributors. As we know commits are actions taken by developers to enhance the open source software applications; any commit is always associated with a chance of bugs. Increase commits may lead to more number of bugs.

- From the 3rd model, we have an important interpretation which shows that even if the number of contributors decreases, if there is an increase in commits there is always an increase in bugs. This is also possible in open source software applications

TABLE 3.25: State Information HMM-2 [O: Observation Sequence, S: State Sequence, (HIGH, LOW, MID1, MID2): Hybrid States]

1	O	D	I	D	D	I	D	D	D	D	I
	S	LOW	HIGH	MID2	LOW	HIGH	MID2	MID2	MID2	LOW	HIGH
2	O	I	I	D	D	D	I	I	D	I	D
	S	HIGH	HIGH	MID2	MID2	LOW	HIGH	HIGH	LOW	HIGH	MID2
3	O	D	D	D	D	D	D	D	I	D	I
	S	MID2	MID2	MID2	MID2	MID2	MID2	LOW	HIGH	LOW	HIGH
4	O	D	D	D	D	D	D	I	I	D	D
	S	MID2	MID2	MID2	MID2	MID2	LOW	HIGH	HIGH	MID2	LOW
5	O	D	I	I	I	D	D	I	D	D	D
	S	LOW	HIGH	MID1	HIGH	MID2	LOW	HIGH	MID2	MID2	LOW
6	O	D	D	D	I	D	I	D	I	D	I
	S	MID2	MID2	LOW	HIGH	LOW	HIGH	LOW	HIGH	LOW	HIGH
7	O	D	I	I	D	D	D	D	D	I	I
	S	LOW	HIGH	HIGH	MID2	MID2	MID2	MID2	LOW	HIGH	HIGH
8	O	I	I	D	D	D	D	D	D	D	I
	S	HIGH	HIGH	MID2	MID2	MID2	MID2	MID2	MID2	LOW	HIGH
9	O	D	D	D	D	D	D	D	D	I	D
	S	MID2	MID2	MID2	MID2	MID2	MID2	MID2	LOW	HIGH	MID2
10	O	D	I	D	I	D	D	D	I	I	I
	S	LOW	HIGH	LOW	HIGH	MID2	MID2	LOW	HIGH	MID1	HIGH

as there are some active contributors and some inactive contributors. The active contributors supplement removal of inactive contributors.

3.8.5.2 Most frequent observation sequence

For all the models we have taken all combination of observation sequences up to a sequence of size 3. We used HMM model for predicting the sequences. The results are given in Table 3.26.

Interpretation:

For HMM models, we also predicted the most frequent observation sequence (bug growth pattern) in the Mozilla Firefox bug number data. This shows that up to a sequence of size

TABLE 3.26: Log(Probability) Values for Observation Sequence

Observation Sequence	log(Probability)
P{I}	-0.6539
P{D}	-0.734
P {I,I}	-1.9067
P {I,D}	-0.9904
P {D,I}	-1.3879
P {D,D}	-1.4679
P {I,I,I}	-3.1594
P {D,D,D}	-2.2019
P {I,I,D}	-2.2431
P {D,D,I}	-2.1219
P {I,D,D}	-1.7244
P {D,I,I}	-2.6407
P {I,D,I}	-1.6443
P {D,I,D}	-1.7224

3, the increasing pattern has the maximum probability than the others in the Mozilla bug number data. i.e. $\{I\}$ is the most probable one among all single observation sequences. Here, $\{I,D\}$ is the most probable one among all observation sequences of size 2. Here, $\{I,D,I\}$ is the most probable one among all observation sequences of size 3.

In this work, we implemented the HMM to predict the temporal bug patterns of two open source projects. The result shows the relationship between the number of contributors and number of commits with the temporal bug patterns. Our hybrid model also shows the combined effect of both the number of contributors and the number of commits on the temporal bug patterns.

3.9 Conclusion

In this chapter, we have predicted the temporal bug numbers and patterns across different versions of a software application. We have applied advanced time series modelling techniques to model the temporal patterns in the bug numbers. Advance knowledge about

bug numbers will help the software managers to decide on resource allocation and effort investments. The developers will be aware of the number of bugs in advance and can take effective steps to reduce the number of bugs in the new version. The end user can decide on adopting a particular software application among a variety of applications by knowing the bug growth patterns of the particular software application.

Clone Evolution Prediction

3.10 Introduction

Software evolution is an important aspect in the field of software development. Software evolves due to changes in requirements, to adopt to changes in environments and also to improve software quality. During software evolution, there is always a tendency of code fragment being copied and changed slightly in the subsequent version. These recurring code fragments are called clones. If we can detect these recurring code fragments and model them, it can be immensely helpful in software maintenance activities.

Reuse is a controlled and documented process. However, cloning is an undocumented and uncontrolled process. The cloned components are considered as bad smell [94] for software as they always increase the maintenance effort. Cloning also enhances the probability of increasing number of bugs in the software application due to the faulty code fragment being repeatedly used in different places. These cloned code fragments have a bad effect on software maintainability and quality of software. It also leads to the poor understandability of the software system as it takes more time to understand the code. Code clones are classified into different types based on semantics and syntax.

Our focus is on the identification of the different type of clone components and prediction of cloned components in subsequent versions of the software application. The first part of our study is to identify the cloned components using various clone detection techniques.

The second part is to predict the cloned components using advanced time series modeling. A large number of papers have identified clones within the same version of software. The primary aim of this work is to model the clones across different versions of the software and to predict the clone evolution pattern.

The organization of the chapter is as follows: Section 2 describes a Hybrid Modeling Approach for Software Clone Evolution Prediction. Section 3 describes Machine Learning Strategies for Temporal Analysis of Software Clone Evolution using Software Metrics. Section 4 presents a Comparison among ARIMA, BP-NN, and MOGA-NN for Software Clone Evolution Prediction Section 5 describes the application of Clone Evolution Prediction. Section 6 concludes the paper and give direction for future work.

3.11 Software Clone Types

Two types of similarity exist between clone fragments [94].

- Fragments can be similar based on the similarity of their program text.
- Fragments can be similar based on their functionality (independent of their text).

The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we list the types of clones based on both the textual (Types 1 to 3) and functional (Type 4) similarities:

- Type-1: These are identical clone fragments ignoring white spaces, comments, and layout (Exact Match Clone Sets)
- Type-2: Same syntax clones with variations in literals and another type 1 parameters.(Near Miss Clone sets)

- Type-3: Semantically equivalent or almost similar clones with variations in statements.(Near Miss Clone sets)
- Type-4: Code fragments performing same computation but are implemented by different syntactic variants.

3.12 Dataset Description

The models for clone evolution prediction are validated against 31 versions of ArgoUML[5] ranging from version 0.9.5 to the latest stable release 0.34.0. This spans a duration of over 6 years from 2006 to 2011. For the subsequent versions of the software, the clone content was successfully predicted using the time series modeling techniques.

ArgoUML is a UML diagramming application written in Java and released under the open-source Eclipse Public License. By virtue of being a Java application, it is available on any platform supported by Java.

It was decided to use this software as our code base, because it was part of MSR data challenge, 2006.

3.13 Clone Detection Process

A clone detector is supposed to find pieces of code of high similarity in software's source text. The main problem is that there is no beforehand knowledge about which code fragments may be repeated. As such, the detector has to compare every possible fragment with every other possible fragment. Such a comparison may become intractable from a computational viewpoint, and thus, several measures are used for reduction of the domain of comparison before performing the actual comparisons. Even after identifying

potentially cloned fragments, further analysis and tool support are required to identify the actual clones. In this section, we provide an overall summary of the basic steps in a clone detection process. This generic overall picture allows us to compare and evaluate clone detection tools[187] with respect to their underlying mechanisms . A typical clone detection process involves following steps.

3.13.1 Pre-Processing

Pre-processing partitions the source code and determines the domain of comparison. Three main objectives exist in this phase [187, 181, 139].

1. Filtering the code: This involves eliminating code elements that are not required in comparison. This can become necessary in situations when the comparison tool is not language independent. E.g., SQL statements embedded in JAVA code. Also, generated sources, such as table initialization can be removed during this phase.

2. Determine Source Units: The source code is partitioned into disjoint units which are directly involved in clone relations with granularity levels varying from Statements, Blocks, and Functions, etc. to Files.

3. Determine Comparison granularity: This involves setting up the minimum units for comparison or to fix the granularity parameter for comparison. This parameter may or may not affect the running time of the analysis. For metrics based techniques, however, this has little importance.

3.13.2 Transformation

Transformation converts the given pre-processed code into an intermediate representation for the subsequent clone extraction process. Normalization, which additionally removes

whitespaces and comments as well as does some normalizing transformation, is also supported by some tools [187].

3.13.3 Clone Extraction

Source code transformation is also referred as extraction. It involves one or more of the following steps:

1. Tokenization: In token-based approach, using lexical analysis, the source code is divided into tokens. These tokens form sequences which are compared. All the whitespaces and comments are removed [187].

2. Parsing: In syntactic approaches, an abstract syntax tree [AST], possibly annotated, is built from the source code. Metric based approaches may also employ comparing metrics of the sub-trees of the AST [187].

3.13.4 Normalization

The normalization eliminates the superficial differences including whitespaces, comments, formatting or identifier names. Some metric based approaches may use formatting and layout as part of the comparison. Identifier normalization is performed for detection of type 2 clones. Structural transformations may be done for dealing with minor syntactic variations [187].

3.13.5 Match detection

At the final stage, the comparison algorithm takes the transformed units to produce clone sets. Adjacent comparison units may be aggregated to form larger units. The granularity

for comparison can be fixed or set as a parameter. The output is a set of clone pairs with the location of the clone fragments in the source code given as coordinates. Techniques used include suffix trees, dynamic pattern matching, and hash-value comparison [187].

3.13.6 Clone Detection Tools Used

There have been a large number of studies on software clone detection tools[27]. Following tool is used in this chapter for clone detection process.

3.13.6.1 CloneDr

CloneDr is a clone detection tool, based on detection using abstract syntax tree for the code. It accepts multiple parameters to set a threshold of similarity between clones. It outputs multiple parameters.

For clone estimation, some input parameters were used. Similarity threshold is a real value between 0 and 1 which represents the ratio of the volume of identical code in a clone, to the actual total code in the clone. Volume is computed in terms of Abstract Syntax Tree (AST) nodes [27] but can be effectively understood in terms of Source Lines of Code (SLOC). This threshold controls how similar two blocks of code must be to be considered clones [199]. Max Clone Parameter helps the tool to determine a limit on how many deltas are allowed between near-miss clones, thereby limiting the number of parameters [216]. Minimum Clone Size helps control how big a piece of code must be to be considered a clone. This value c is used to determine an AST node count threshold, computed as c*A/L where A is the total number of AST nodes and L is the number of physical lines in the entire software [216]. Remaining parameter was taken as default.

After processing, exact-match clone sets and near-miss clone sets were obtained. Exact-match Clone Sets are the count of cases in which some code block has been cloned without

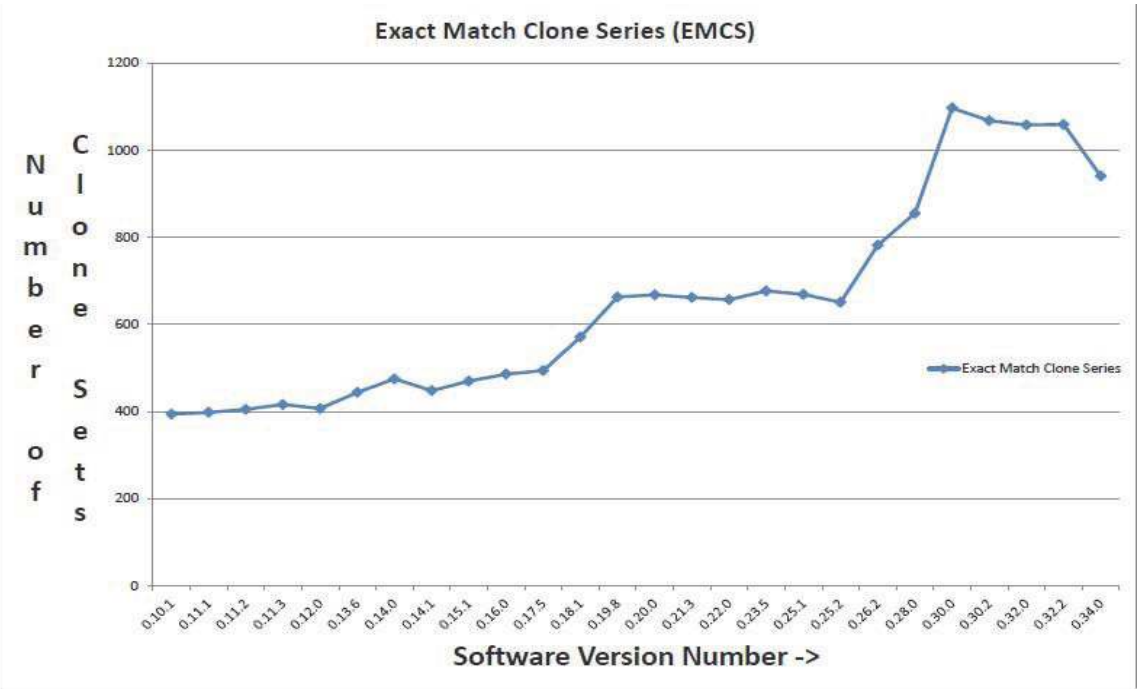


FIGURE 3.40: Time Series Representation for Exact Match Clone Sets for ArgoUML

any changes. Near-miss Clone Sets is the count of abstractions with parameters for which there are multiple clone instances. Figure 3.40 and 3.41 shows the time series representation of Exact-match Clone and Near-miss clone series.

3.14 Problem Description

Let an observed clone sequence be $\{C_1, C_2, C_3, ...C_T\}$ observed over equally spaced time points.

A fairly general model for the time series can be written:

$C_t = g(t) + \varepsilon_t$

Here, we have two components:

Systematic Part: $g(t)$ = The Component to be Modelled using Time Series.

Stochastic Part: ε_t, a residual term also called noise, which follows a probabilistic law.

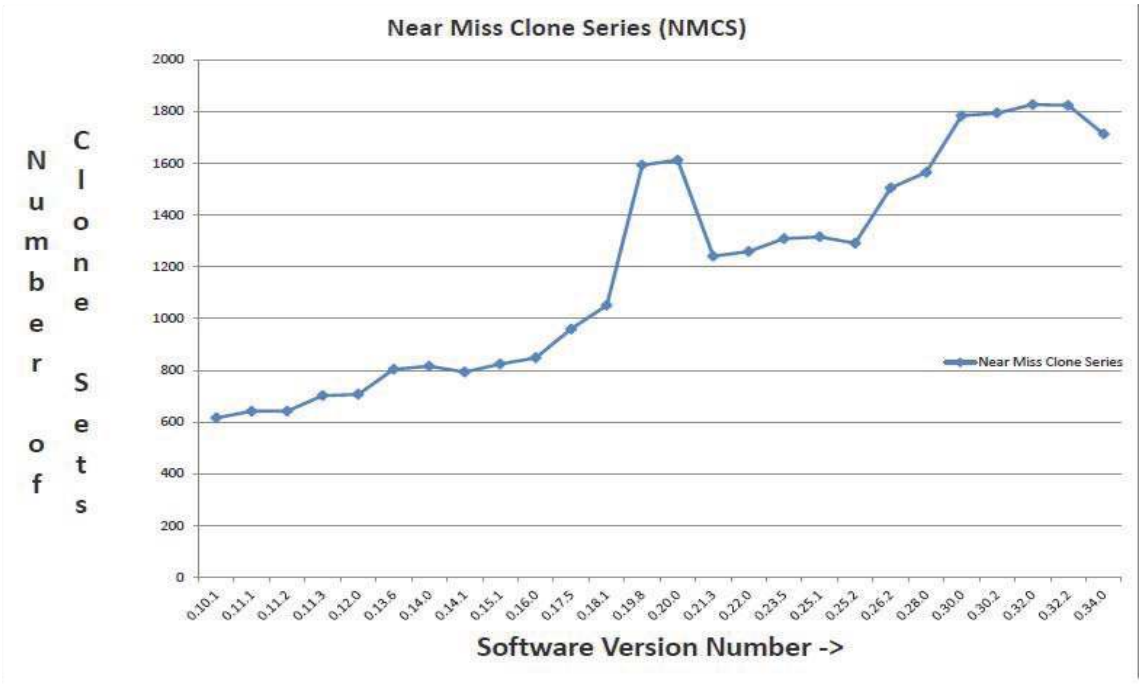


FIGURE 3.41: Time Series Representation for Near Miss Clone Sets for ArgoUML

The objective is to predict C_t from p past observations.

$$C_t = f(C_1, C_2, C_3, ...C_{t-p}) + \varepsilon_t$$

The goal is to have a suitable model which can predict the clone evolution accurately with a minimum value of error (ε_t). In the next section, we have discussed various time series models to predict the evolving clone components.

3.15 Hybrid Modeling Approach for Software Clone Evolution Prediction

A hybrid model [167] is proposed for the prediction of different types of the clone number series. The proposed method is validated on 31 versions of ArgoUML, an open-source

software application for version 0.9.5 to the latest stable release 0.34.0. We observe that the hybrid model is a better predictor than ARIMA model for clone number series.

3.15.1 The Hybrid Model

ARIMA and ANNs are good at prediction in the linear and nonlinear domains respectively. However, ARIMA cannot correctly approximate the complex nonlinear effects. Similarly, ANN models cannot give more accurate results for linear problems and have tendency to overfit. The real world time series contain both linear and nonlinear terms i.e.

$Y_t = L_t + N_t$

L_t :Linear Component

N_t:Non-Linear Component

First, ARIMA model is applied to predict the linear component, and then the residuals from the linear model contains only the nonlinear relationship. Now the residuals from the nonlinear model can be represented by:

$e_t = Y_t - F(L_t)$

Where $F(L_t)$ is the forecast value for time t from the estimated relationship. With 'n' input nodes, the ANN model for the residuals will be:

$e_t = f(e_{t-1}, e_{t-2}, ...e_{t-n}) + \varepsilon_t$

Where f is a nonlinear function determined by neural network and ε_t is the random error. Now let the predicted nonlinear component be $F(N_t)$, the combined forecast will be:

$F(Y_t = F(L_t) + F(N_t)$

So, the hybrid model work in two steps:

- In the first step, the ARIMA model is used to analyze the linear patterns in the model.
- In the second step, a nonlinear model is used to analyze the residuals.

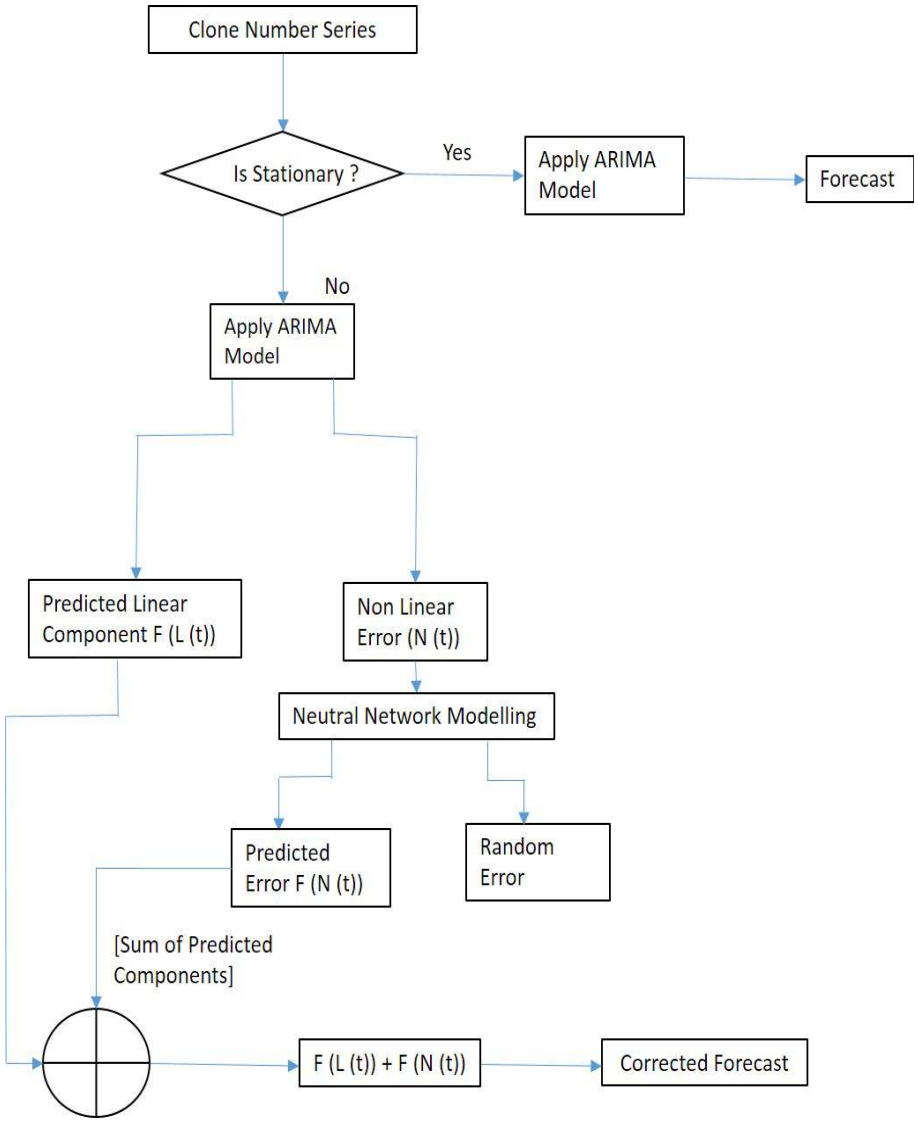


FIGURE 3.42: Proposed Hybrid Model

Figure 3.42 shows the representation of the hybrid system.

3.15.2 Evaluation

The models are evaluated based on the RMSE (Root Mean Square Error), MAE (Mean Absolute Error). RMSE, MAE are calculated using following formulae.

1. $MAE = \frac{1}{n}\sum_{t=1}^{N}|(A(t) - F(t))|$

2. $RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{N}(A(t) - F(t))^2)}$

3.15.3 Data Preparation

Abstract syntax tree based clone detection technique was used in this paper with the help of CloneDr[27] (a clone detection tool). After processing, Exact-match clone sets and Near-miss clone sets were obtained.

1. Exact-match Clone Sets (EMCS): It is the count of cases in which some code block has been cloned without any changes.

2. Near-miss Clone Sets (NMCS): It is the count of abstractions with parameters for which there are multiple clone instances.

The entire clone number series is partitioned into a training set and a testing set. Partitioning is useful in finding the fitness of the model with the data pattern for both training and test data. The model which is more accurate on test data is preferably selected.

3.15.4 Implementation and Result

In this work, we have experimentally verified the predictive performance of two models: ARIMA and the Hybrid Model (ARIMA + ANN).

TABLE 3.27: ARIMA model and RMSE Value

Clone Number Series	ARIMA Best Fit Model	RMSE (Training)	RMSE (Testing)
Exact Match Clones	ARIMA (2, 1, 2)	36.24632405	158.0847138
Near Miss Clones	ARIMA (2, 1, 2)	141.5553064	200.4682236

3.15.4.1 ARIMA

For all the types of clone number series the ACF and PACF plots are drawn to find the value of p, q, and d. There are some abnormalities in higher lags which can be ignored. It is observed that the appropriate model for the exact and near miss clone number series becomes ARIMA (2, 1, 2). The RMSE, for ARIMA model for all the clone number series, is given in Table 3.27.

3.15.4.2 Hybrid Model

The clone number series are first modeled with ARIMA. The result is predicted linear component $F(L_t)$ and some non-linear error term ε_t. Now the error contains some non-linear patterns which need non-linear technique to model the error series. The error series $\varepsilon_t, \varepsilon_{t-1}, \varepsilon_{t-1}$.... is modeled with ANN. The error series is now modeled using a non-linear autoregressive neural network. We have used non-linear activation function tansig to match the non-linear patterns in the error series.

After applying neural network model to the error component, we get a predicted error component $F(N_t)$ and some random error ε_t now the predicted error component $F(N_t)$ and the predicted linear component $F(L_t)$ are added to get the combined forecast value.

The RMSE, for a Hybrid model for both Exact Match Clone Series and Near Miss Clone Series, is given in Table 3.28. Figure 3.43 and Figure 3.44 represents the plot of the actual data series and predicted series by ARIMA and Hybrid model for both Exact Match Clone Series and Near Miss Clone Series.

TABLE 3.28: Hybrid model and RMSE Value

Clone Number Series	ARIMA Best Fit Model	RMSE (Training)	RMSE (Testing)
Exact Match Clones	ARIMA (2, 1, 2)	17.8363159	23.95475456
Near Miss Clones	ARIMA (2, 1, 2)	103.5304816	87.3672884

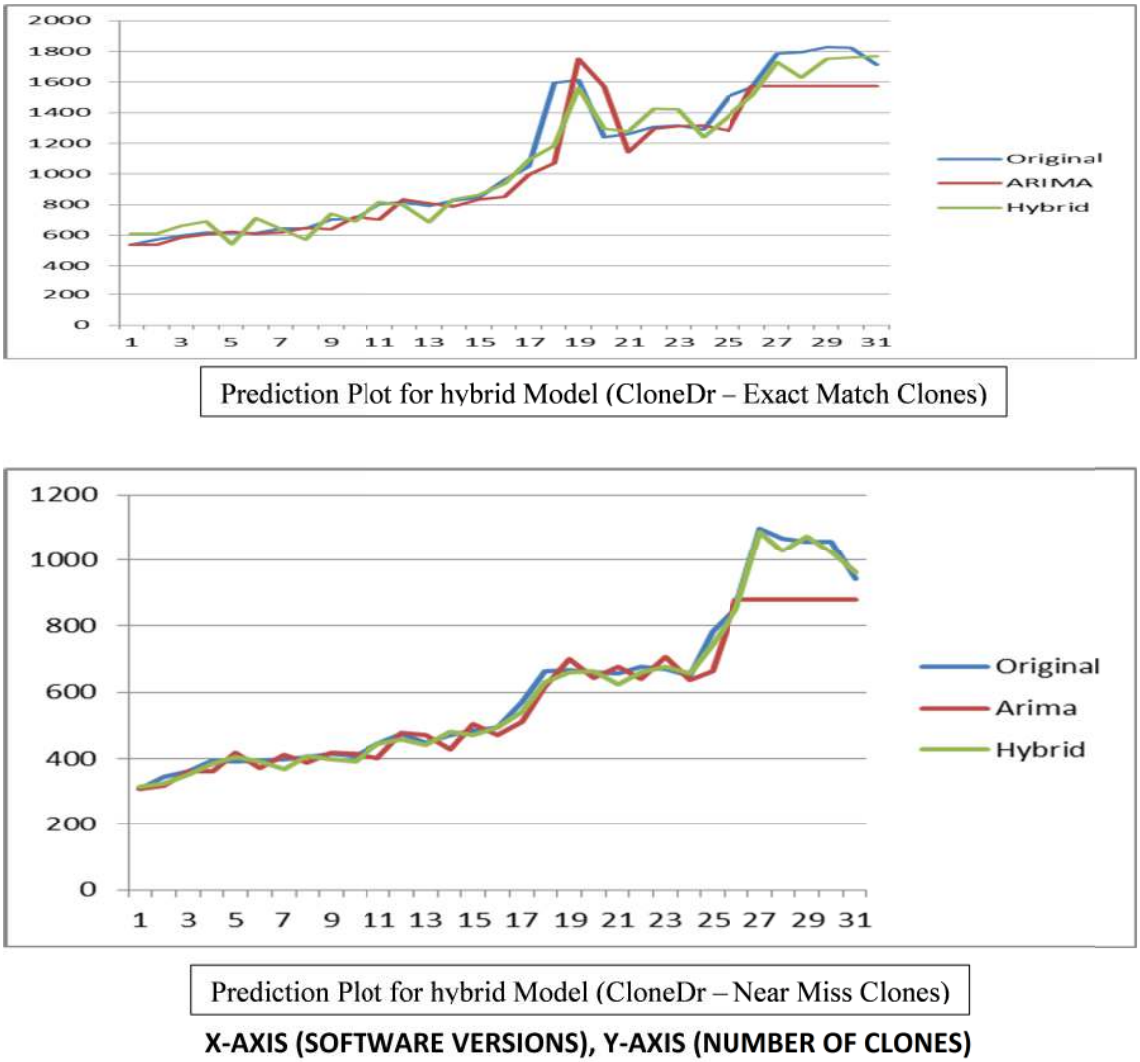
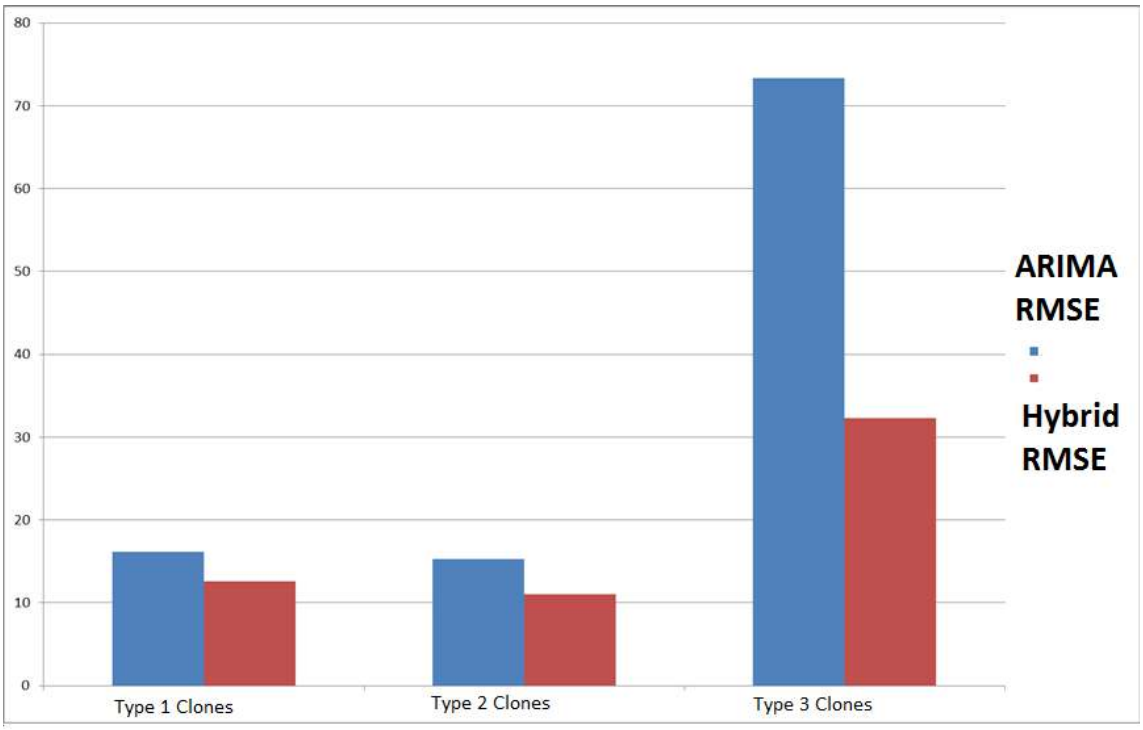


FIGURE 3.43: Hybrid and ARIMA Predicted Series vs. Original Series

Figure 3.44 and Figure 3.45 show the bar chart representation of comparison between RMSE and MAE for both training and test set for the two models respectively.



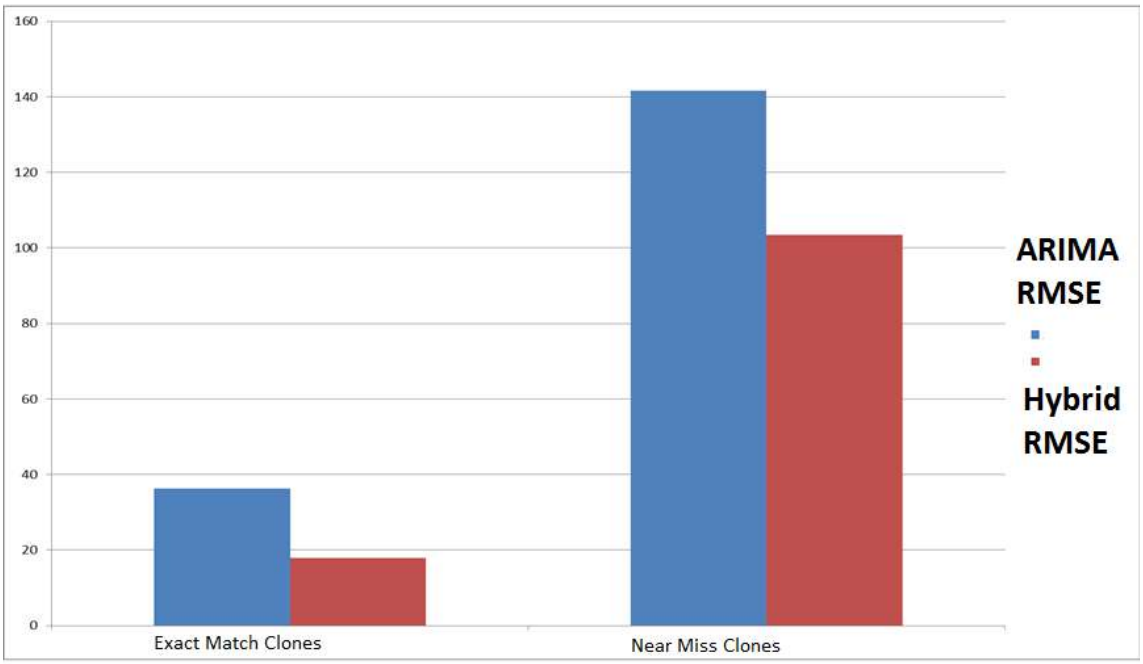
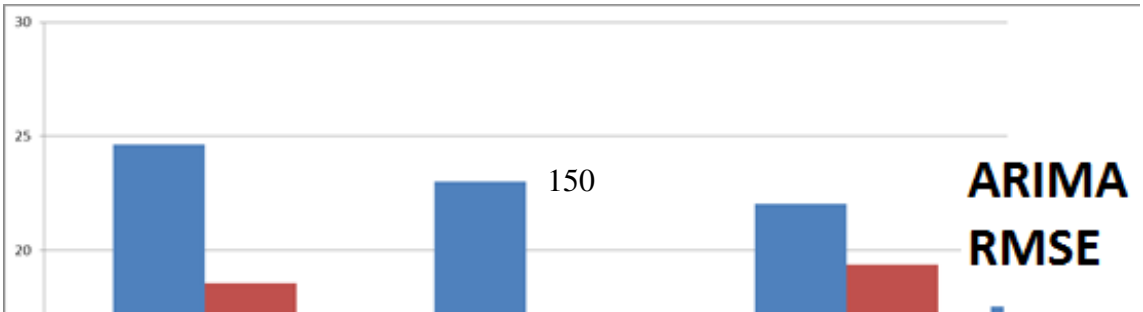


FIGURE 3.44: RMSE Comparison between two Models (Train Data)



This chapter uses a hybrid modeling approach to predict the trend of the clone numbers in Agro UML. This chapter also gives a comparative analysis of the predictive performance between ARIMA and the Hybrid model (ARIMA + ANN). The result confirms that hybrid model is a better predicting model for all types of clone number series.

3.16 Temporal Analysis of Software Clone Evolution using Software Metrics[163]

The primary focus of the work is modelling of the evolution of clones in a software application. Detection of clones in a large software system is challenging as it depends on the internal design of software modules and methods. Object-oriented metrics like DIT, NOC, WMC, LCOM, and Cyclomatic complexity are good indicators of clone contents.We demonstrate a correlation between clones and various metrics of the source. Evolving cloned components are modelled through advanced time series techniques using machine learning approaches.

3.16.1 Software Metrics and Software Clones

Software metric is a standard way of measuring a software system or process. The metrics help to find the degree to which a software system possesses some property. As metrics are a unique way of measuring software systems, there exists a correlation between software metrics and cloned content. Several metrics for object-oriented systems have been proposed [44], and new metrics are still being introduced. The metrics used in time series analysis of software clone evolution are given in Table 3.29. The detailed description of software metrics is presented in the paper [238].

TABLE 3.29: Software Metrics used for Software Clone Evolution Prediction

Metrics	Description
WMC	Weighted Methods per Class
NSC	Number of Children
NORM	Number of Overridden Methods
NOF	Number of Attributes
NSF	Number of Static Attributes
NOM	Number of Methods
NSM	Number of static methods
NOC	Number of Classes
NOI	Number of Interfaces
NOP	Number of Packages
MLOC	Method Line of Codes

3.16.1.1 Software Metrics Extraction

ArgoUML [5], an open-source software application is used for conducting our experiments. We downloaded the source-code for each of the releases of the ArgoUML software. All the experiments are carried out in the Eclipse environment. We have used the eclipse metrics plugin [7] to extract the various software metrics for each release.

3.16.2 Design of Experiments

In this section, the detailed experimental procedure for time series modelling of clone evolution is discussed. We have used both univariate and multivariate time series modelling techniques for prediction of software clone evolution using software metrics.

We used the lag length parameter to transform the time series data into available machine learning data. If we have a time series y(t) and the lag parameter is k, then it treats the first k lagged values of y(t), i.e., y(t-1),y(t-2)..., y(t-k) as explanatory variables. We also add an extra feature named artificial time index to predict the response variable y(t) [28]. If initially, we had n instances or observations in dataset then after this transformation we remove the first k instances with unknown lag-values [186]

After transforming the time series data using lagged value concept, we used the linear regression and Multi-Layer Perceptron to model the multivariate time series jointly. The algorithms use nonlinear hypothesis for analysis and are more powerful and flexible than conventional methods like ARMA or ARIMA [164].

We have used following machine learning algorithms in modelling the clone evolution series.

3.16.2.1 Linear regression

It gives a linear model with the help of training examples to predict for a given test data. If there are m features against a target variable, and
X be the feature-vector of order $m \times n$, it fits the model with a weight vector of order $m \times 1$, $W = (w_1, w_2, ..., w_m)$ moreover, a bias term b is used to minimize the sum of the square of errors between the actual and predicted values of the target variable.

Let $Y(ActualValue) = (Y_1, Y_2, ..., Y_n)$ be the vector of actual values of the target,

$\hat{Y}(PredictedValue) = (\hat{Y}_1, \hat{Y}_2, ...\hat{Y}_n)$ be the vector of predicted values,

$\hat{Y} = b + W^T \cdot X$

That minimizes: $\left|\left|Y - \hat{Y}\right|\right|^2$.

3.16.2.2 Multi-Layer Perceptron

Multilayer perceptron (MLP) is an artificial neural network that has one or more hidden layers of neurons between the input layer and output layer [188].
Let X, be the input vector comprising of the features of a particular training example.
Let the output vector of the hidden layers be

$Y_1, Y_2, Y_3, \ldots.., Y_m$

Where, m=number of hidden layers.

The final output Y is obtained through the layer-by-layer transfer of the output.

If the transfer function used is the logarithmic sigmoid function

$f(x) = \frac{1}{1+e^{-x}}$

Then We have,

$$Y_1 = f(b_1 + w_1^T X)$$
$$Y_2 = f(b_2 + w_2^T Y_1)$$
$$\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots$$
$$Y_m = f(b_m + w_m^T Y_m)$$

Where w_i are the weight vectors of the neurons of i-th layer and b_i is the vector of the bias terms for each neuron of i-th layer. The Diagrammatic representation of MLP is given in Figure 3.46.

3.16.3 Implementation and Result

We have used an AST-based approach for finding the exact match clones and near-miss clones over a time series of length 19. The next step is to use time series analysis to predict the evolution of software clones. We have transformed the time series data using lagged value transformation [186] and applied advanced machine learning techniques for

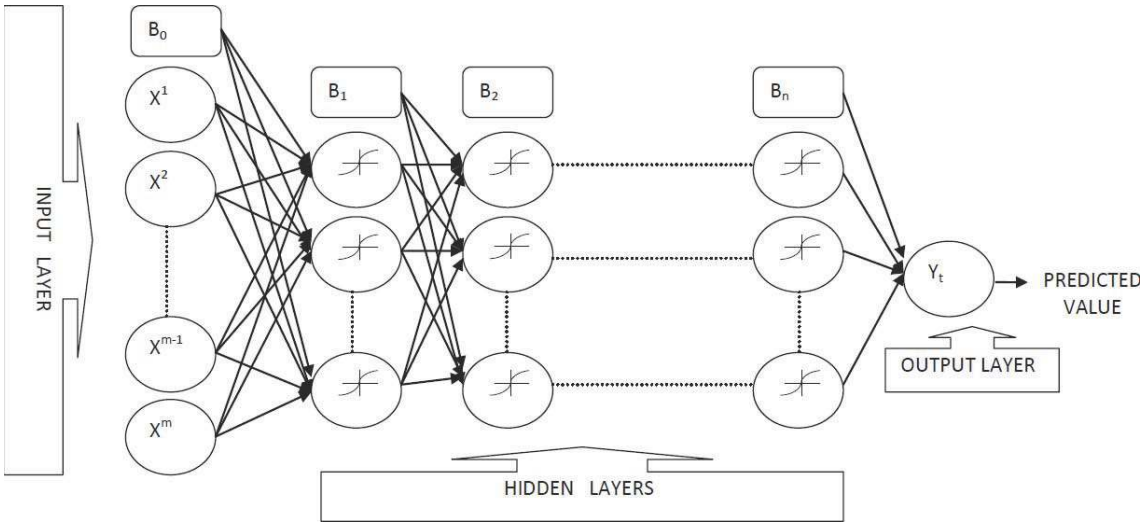


FIGURE 3.46: Multi-Layer Perceptron Model

modelling them [188]. All the machine learning algorithms as discussed above are implemented in Java. We have modelled the Exact Match Clone Series (EMCS) and Near Miss Clone Series (NMCS) separately.

3.16.3.1 First Model

Here we have modelled the clone sets individually using linear regression. If a clone-set time series is denoted by Y_t then, the model comprises of $Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4}$ feature vectors for predicting Y_t (EMCS or NMCS)

Where $Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4}$ = lagged variables

Here only lagged values are used for predicting the clone series.Figure 3.47 represents Diagrammatic Representation of Model 1.

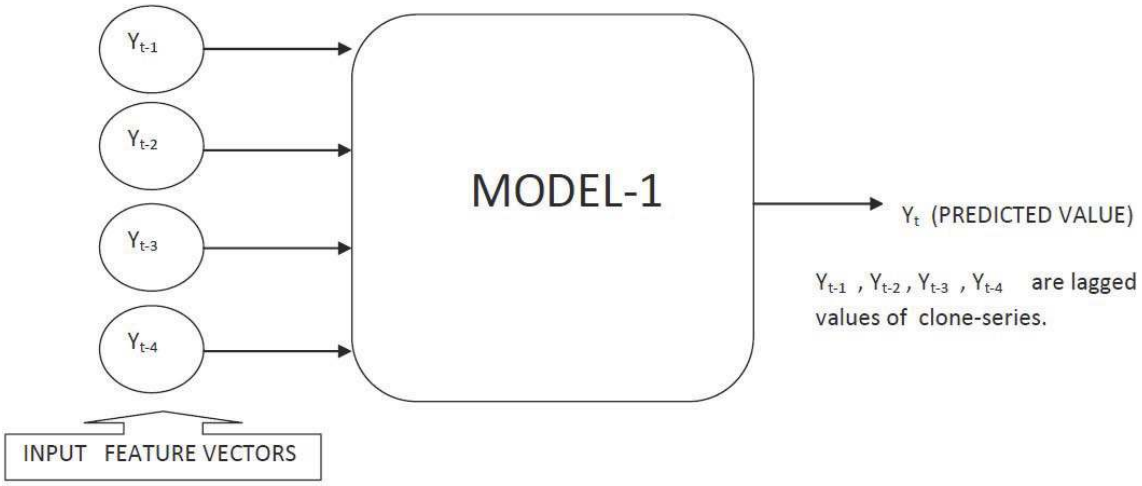


FIGURE 3.47: Diagrammatic Representation: Model 1

3.16.3.2 Second Model

Here we build our predictive model by jointly modelling the relationship between the pair (p, q) of a clone-set and each software metrics using linear regression.

$p \in Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4}$ = Lagged variables

$q \in x1|x2|x3.\ldots.|x11$ = Numerical values for 11 Software Metrics extracted.

Here lagged value and each software metrics value individually is used for joint modelling of clone series. In this model, we also got the correlation between each software metrics and the clone content. Figure 3.48 represents Diagrammatic Representation of Model 2.

3.16.3.3 Third Model

In this model, we include all the software metrics together along with lagged value for modelling of clone series. For all the models, we used lagged feature vectors up to size 4 as it gives a more accurate prediction and minimum error. The model for predicting the response variable, i.e., Clone series (EMCS or NMCS) includes 4 lags of each software metric and lagged feature vectors. The diagrammatic representation of model 3 is given in the Figure 3.49.

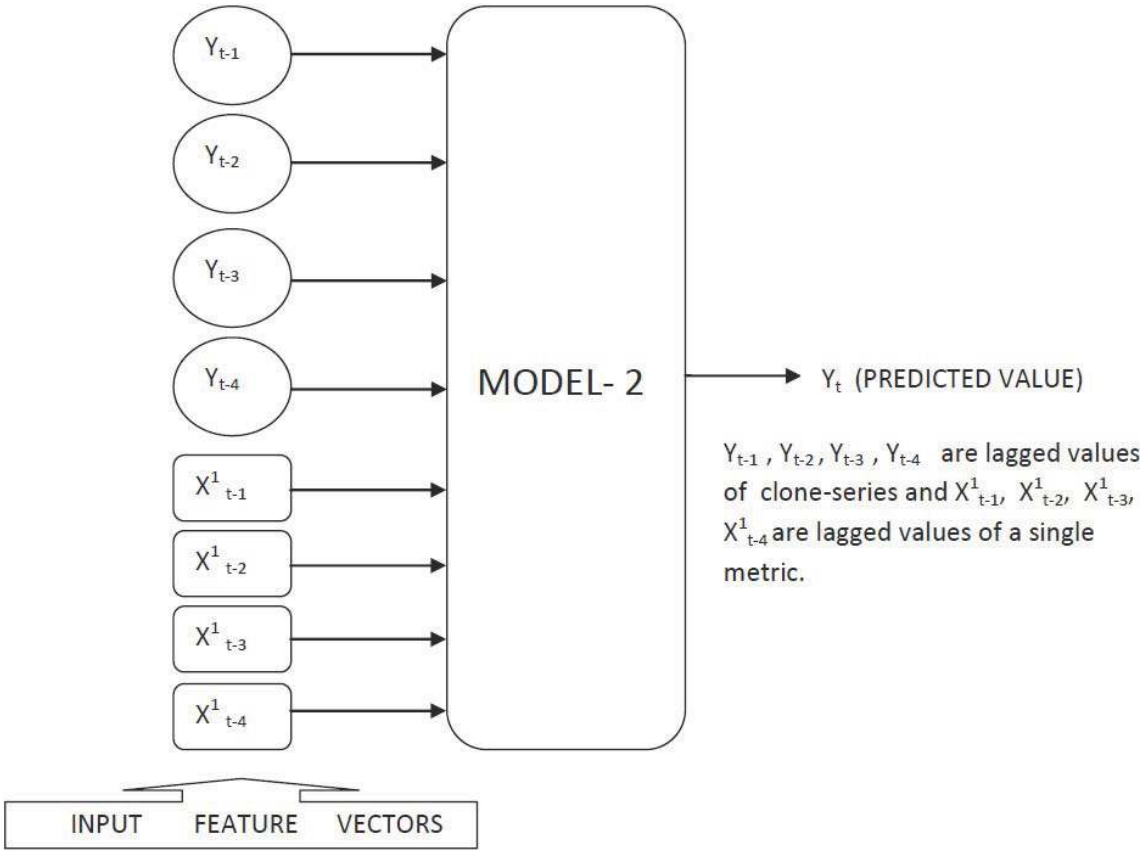


FIGURE 3.48: Diagrammatic Representation: Model 2

3.16.4 Evaluation and Interpretation

The models are evaluated based on the RMSE (Root Mean Square Error), MAE (Mean Absolute Error) and MAPE (Mean Absolute Percentage Error). RMSE, MAE, MAPE are calculated using following formulae.

1. $MAE = \frac{1}{n}\sum_{t=1}^{N}|(A(t) - F(t))|$

2. $RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{N}(A(t) - F(t))^2)}$

3. $MAPE = \frac{1}{n}\sum_{t=1}^{N}\left|\frac{(A(t)-F(t)}{A(t)})\right| * 100$

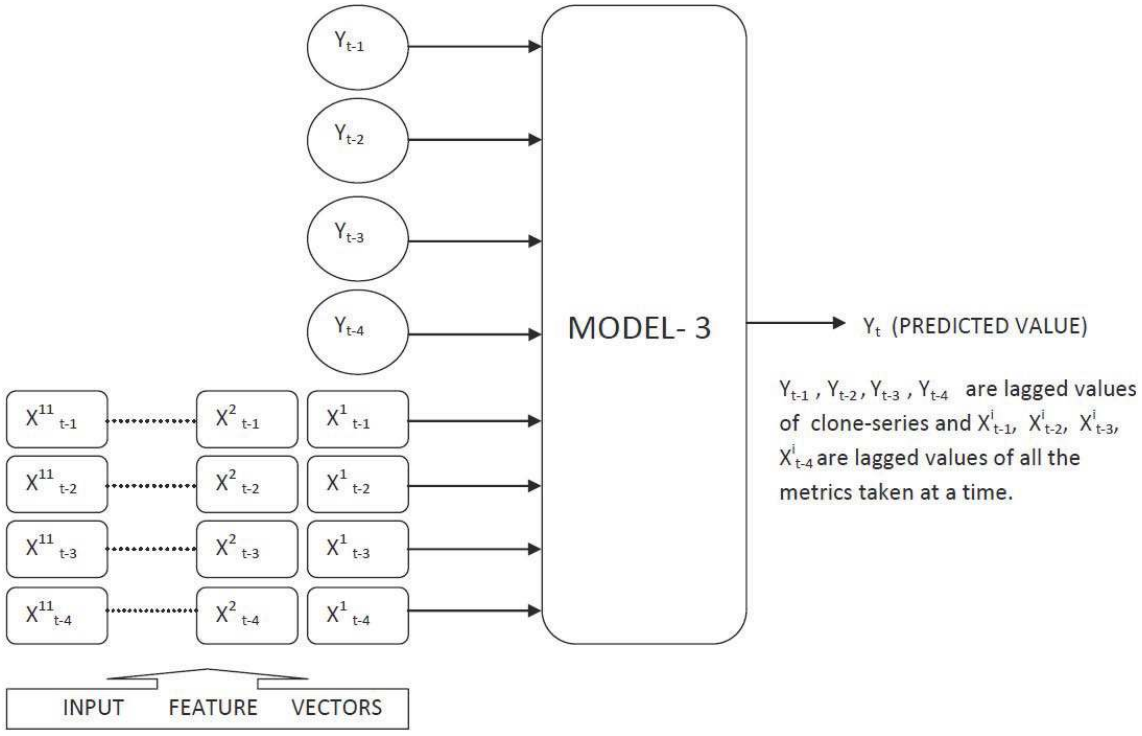


FIGURE 3.49: Diagrammatic Representation: Model 2

Here: A (t): Actual Value, F (t): Predicted Value, N: Number of Terms.

The models are evaluated both on the Training data and Test Data. The Model which gives a minimum error for Test Data is to be selected.

3.16.4.1 Evaluation of First Model

In this model only lagged value of clone series are used for prediction. The results after applying Linear Regression are given in Table 3.30 and Table 3.31.

3.16.4.2 Evaluation of Second Model

Here along with the lagged values of the clone series the individual metric values are also considered for modelling. This Model also determines the relationship between software metrics and Clone sets. Here, we have 11 individual models one for each metrics along

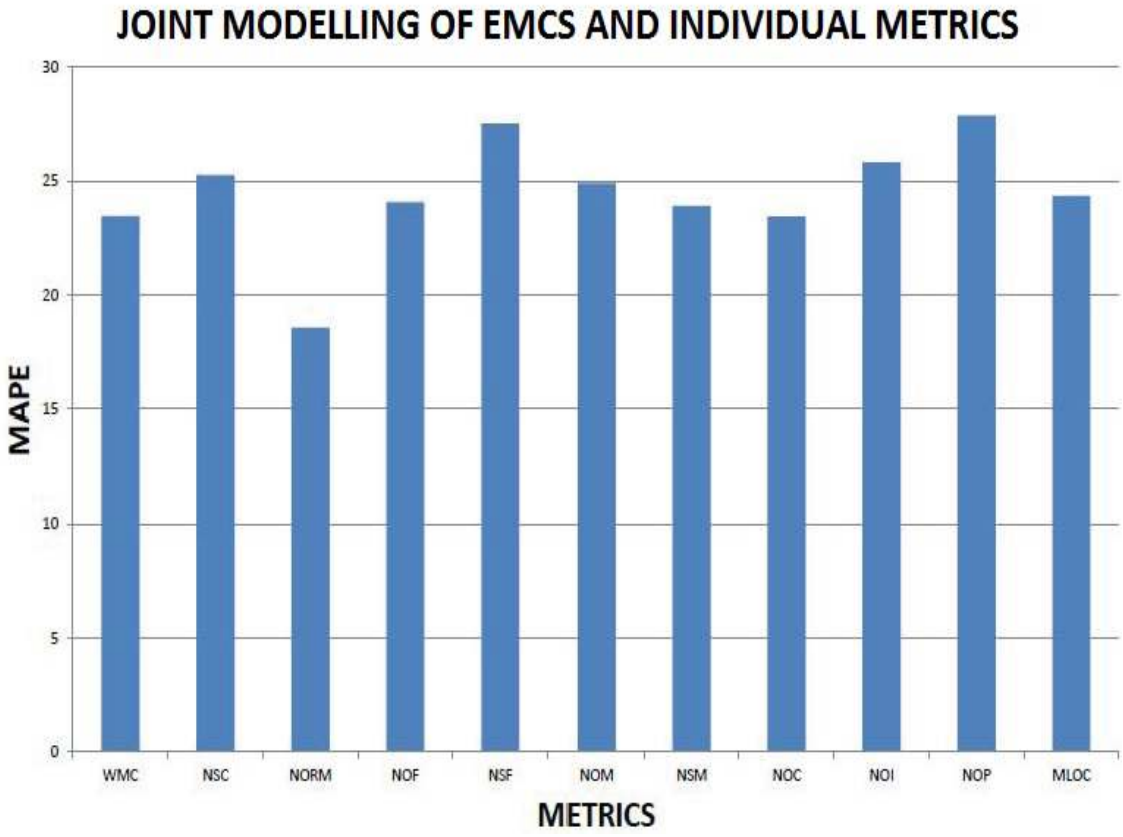


FIGURE 3.50: MAPE for Models using the Individual Metric Values (EMCS)

with lagged values. Figure 3.50 and Figure 3.51 represent the Bar Chart representation of all MAPE for all the models for EMCS and NMCS respectively. The result shown is for test data. Here we have used linear regression for modelling.

3.16.4.3 Evaluation of Third Model

In this model, we consider all the metrics at once along with lagged values of clone series. Here we have applied both Linear Regression and Multilayer Perceptron for modelling. The result after joint modelling of clone number series with linear regression is given in Table 3.32 (for Test Data only). The result after joint modelling with multilayer perceptron is shown in Table 3.33 (for Test Data only).

TABLE 3.30: Evaluation of Training Data (Model 1: Linear Regression)

	MAE	RMSE	MAPE
EMCS	0.0489	0.0673	7.496
NMCS	0.077	0.1047	11.594

TABLE 3.31: Evaluation for Test Data (Model 1: Linear Regression)

	MAE	RMSE	MAPE
EMCS	0.2366	0.2633	29.027
NMCS	0.2405	0.2529	25.7269

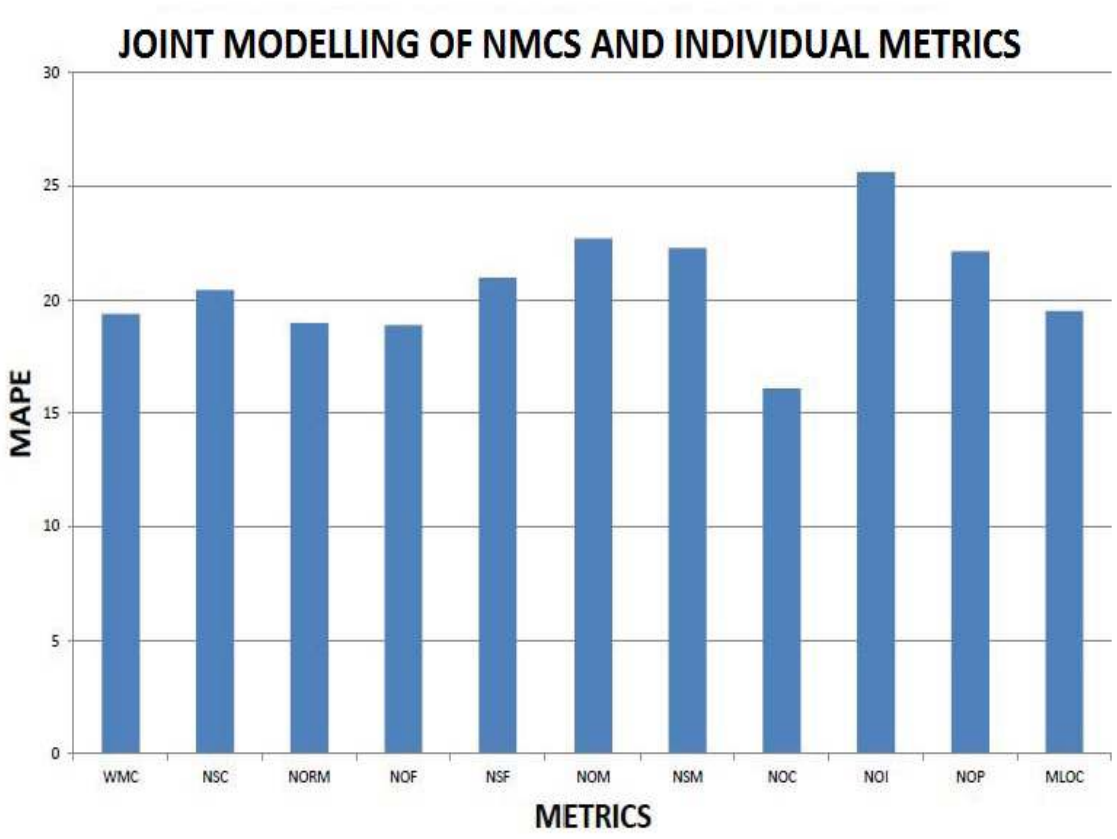


FIGURE 3.51: MAPE for Model using the Individual Metric Values (NMCS)

TABLE 3.32: Evaluation of Test Data (Model 3: Linear Regression)

	MAE	RMSE	MAPE
EMCS	0.1448	0.1798	18.1711
NMCS	0.1157	0.1361	12.5487

TABLE 3.33: Evaluation for Test Data (Model 3: Multilayer Perceptron)

	MAE	RMSE	MAPE
EMCS	0.0911	0.1215	11.59
NMCS	0.0364	0.0364	3.825

3.16.5 Interpretation of the Models

Here we give a comparative analysis of the predictive performance of all the models. We found that after including the software metric values, the errors are reduced significantly. We also found that after applying multilayer perceptron to our combined model 3 the error further reduces. So the inclusion of software metrics and nonlinear modelling is the most appropriate approach for prediction of software clone evolution. From the second model, we found that after including certain software metrics(e.g. NOC), prediction performance of the model improves. Figure 3.39 shows the comparison between Model 1 (Univariate modelling) and Model 3 (Multivariate Modelling) using Linear Regression. Figure 3.40 shows a comparison of a nonlinear and linear model for Model 3 (Multivariate Model).

The work includes software metrics along with the lagged values for modelling the clone evolution. From the result, we found that software metrics are a good indicator of the existence of software clones in any software application. The paper also gives a comparative analysis of modelling software clone evolution. We found Multivariate Nonlinear Model as the most suitable model for predicting the clone evolution.

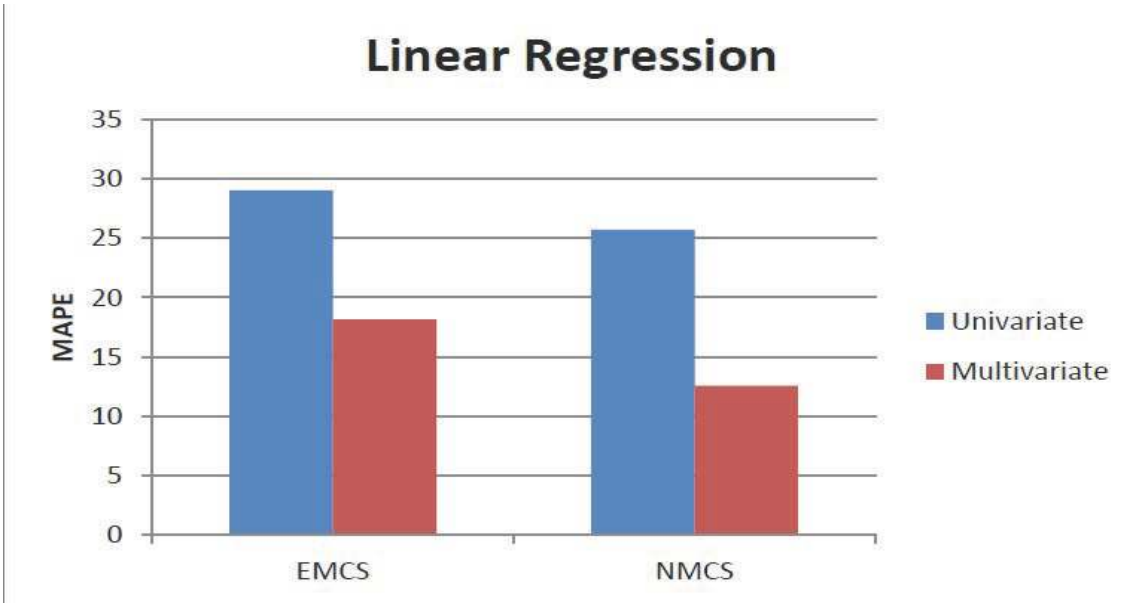


FIGURE 3.52: Comparison among Univariate and Multivariate Modelling (Linear Regression)

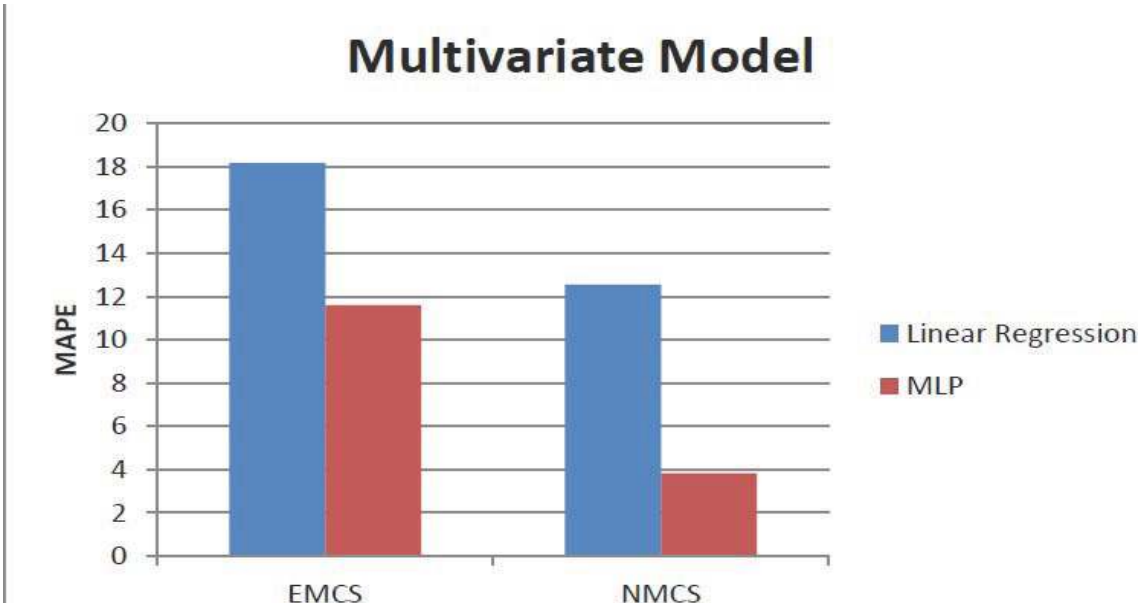


FIGURE 3.53: Comparison among Linear and Nonlinear Modelling (Multivariate Modelling

3.17 A Comparison among ARIMA, BP-NN, and MOGA-NN for Software Clone Evolution Prediction[162]

This work examines the evolution of clone components using advanced time series analysis. In the first phase, software clone components are extracted from the source repository of the software application using Abstract Syntax Tree approach. Then the evolution of software clone components is analyzed. In this paper, three models ARIMA, Back Propagation Neural Network (BP-NN) and Multi-Objective Genetic Algorithm based Neural Network (MOGA-NN)[63] have been compared for prediction of the evolution of software clone components. Evaluation is performed on the large open-source software application, ArgoUML. The ability to predict the clones helps the software developer to reduce the effort during software maintenance activities.

3.17.1 MOGA-NN for Software Clone Evolution Prediction

Neural networks training is performed to minimize the training error corresponding to the given training data. The optimization technique used is Back-propagation. Genetic Algorithm is used to minimize the cost function and to improve the accuracy.

In our context, as we have two objective functions as a cost function, we have used Multi-Objective Genetic Algorithm to optimize them.

A genetic algorithm[50] is one of the most efficient methods of both constrained and non-constrained optimizations. It mimics the natural process of selection based on the fitness of the population. The genetic algorithm has following steps:

1. Generate Initial Population: A random population of n chromosomes is generated. (Each chromosome represents suitable solution for the problem)

2. Evaluation of fitness function: The fitness f(x) of each chromosome x in the population is evaluated.

3. Generate a New Population: Create a new population by eliminating the weak population (Population which does not satisfy fitness criteria).

4. Selection of Parents: Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

5. Perform Crossover Operation: With a particular crossover probability, the crossover operation is applied to generate new offspring.

6. Perform Mutation Operation: With a particular mutation probability, the mutation operation is conducted to make changes at different locations of new offspring. Place new offspring in the new population

7. Replace the old Population by a new one: Now the Genetic algorithm has to be applied to the new population.

8. Stopping Criteria: If the stopping condition is satisfied, stop the algorithm. The best solution from the current population is returned, else

9. Loop: Go to step 2

The Multi-Objective Genetic Algorithm [63] is the application of the Genetic Algorithm to optimize multiple objectives. It gives a set of optimal values $x = x_1, x_2, ...x_l \in R$ which minimizes a set of given objective functions subject to a given set of constraints (if any). Basically, instead of point prediction through ANNs, we are here going to have prediction intervals for a sample as they can handle the uncertainty in the model parameters and also noise in the input data[194].

3.17.1.1 Prediction Intervals

Prediction Interval (PI) [107] is nothing but the upper bound and the lower bound of the prediction. For a given dataset $X = \{(x_i, y_i) \mid i = 1, 2, ...n\}$,
Let the prediction interval of (x_i, y_i) *be* $[L(x_i), U(x_i)]$. We define the Mean Prediction Interval Width (MPIW) and Prediction Interval Coverage Probability (PICP) for X as:

$$MPIW = \frac{1}{n}\sum_{i=1}^{n}(U(x_i) - L(x_i))$$

$$PICP = \frac{1}{n}\sum_{i=1}^{n} c_i,$$

$$where \quad c_i = \begin{cases} 1, & if \quad y_i \in [L(x_i), U(x_i)] \\ 0, & otherwise \end{cases}$$

For a better interval prediction, we have optimized the parameters PICP [205, 106, 194] to be maximum and MPIW [205, 106] to be minimum. We also see that as MPIW decreases, PICP increases. Hence we have a competitive multi-objective optimization problem:

$$Minimize \quad NMPIW(X) \quad and \quad 1 - PICP(X) \quad Simultaneously \quad Such\ that,$$
$$NMPIW \geq 0 \quad and \quad 0 \leq PICP(X) \leq 1$$

We have taken normalized $MPIW(NMPIW)$
instead of $MPIW$, *where* $NMPIW = \frac{MPIW}{y_{max} - y_{min}} and$
We choose to minimize $(1 - PICP)$ instead of maximizing $PICP$ because both are equivalent.

The optimal set x is known as a Pareto-optimal set. It is a set of solutions, satisfying the objective functions at accepted levels without being dominated by any other solution.
In other words, if we have to minimize N objective functions, $f_1(x), f_2(x), ...f_N(x)$, subject to some given constraints,

Let X be the solution space of feasible solutions, then $x_1 \in X$ is said to dominate $x_2 \in X$

$iff \quad f_i(x_1) \leq f_i(x_2)$

$for\ all\ \ i \in [1,N]\ \ and$

$f_j(x_1) < f_j(x_2)\ \ for\ some\ \ j \in [1,N].$

Hence, for x being a Pareto optimal set, it should not be dominated by any other solution in the solution space.

3.17.1.2 NSGA-II [50]

In this algorithm, after each generation, on the new population non-dominance sorting algorithm [50] is applied to sort the chromosomes in the order of dominance. The chromosomes are selected by crowding distance by using binary tournament selection. The crossover and mutation operation is performed to produce the next generation, which also becomes a part of the population. As soon as the maximum number of generations is reached, the first Pareto front of the sorted population is returned, and the corresponding set is the Pareto optimal set [107].

Here we have presented a brief description of the implementation of MOGA in neural networks:

1. Initialize a feed forward neural network $N \quad where \quad N(x_i)$ represents the predicted value of the training sample $x_i \in X$.

2. Train the neural network with MOGA as the optimization function for minimizing the cost functions, i.e. $NMPIW(X) \qquad and \qquad 1 - PICP(X)$.

3. The Pareto optimal set of weights and biases, P (Pareto Set) obtained after the neural network training, is used to find the most optimal solution by setting each set of

weights $(w_i, b_i) \in P$ It is also used for calculating the NRMSE for the training dataset X.

4. The solution yielding the minimum training NRMSE is the most optimal set of weights, which can be now used to train the neural network again and apply on the test dataset.

The Flow Diagram for MOGA-NN implementation is given in Figure 3.54.

3.17.2 Design of Experiments

In the first phase, we got the cloned components by AST based clone detector, the CloneDr. For the ArgoUML software application, the result of the clone detection phases is the exact match and near miss clone components. We got time series of size 31 for each type of cloned components. The next step is to model them using suitable time series modeling. We have compared ARIMA, Back Propagation and MOGA-NN for predicting clone evolution.

3.17.2.1 ARIMA Modelling

As discussed in the previous section; The ARIMA model consists of AR (p), MA (q) and a differentiating (d) operator. ARIMA is a good predictor of stationary time series. ACF and PACF plots are used to for determination of stationary behavior of the series. If the ACF decays slowly, the behavior of the series is non-stationary and if ACF decays instantly, the behavior of the time series is stationary. For the EMCS and NMCS, we plot the ACF and PACF to find out the p, d and q parameters.The most appropriate ARIMA model that gives the best fit for the clone evolution series is ARIMA (2, 1, 2). Figure 3.57 presents diagram of ARIMA model.

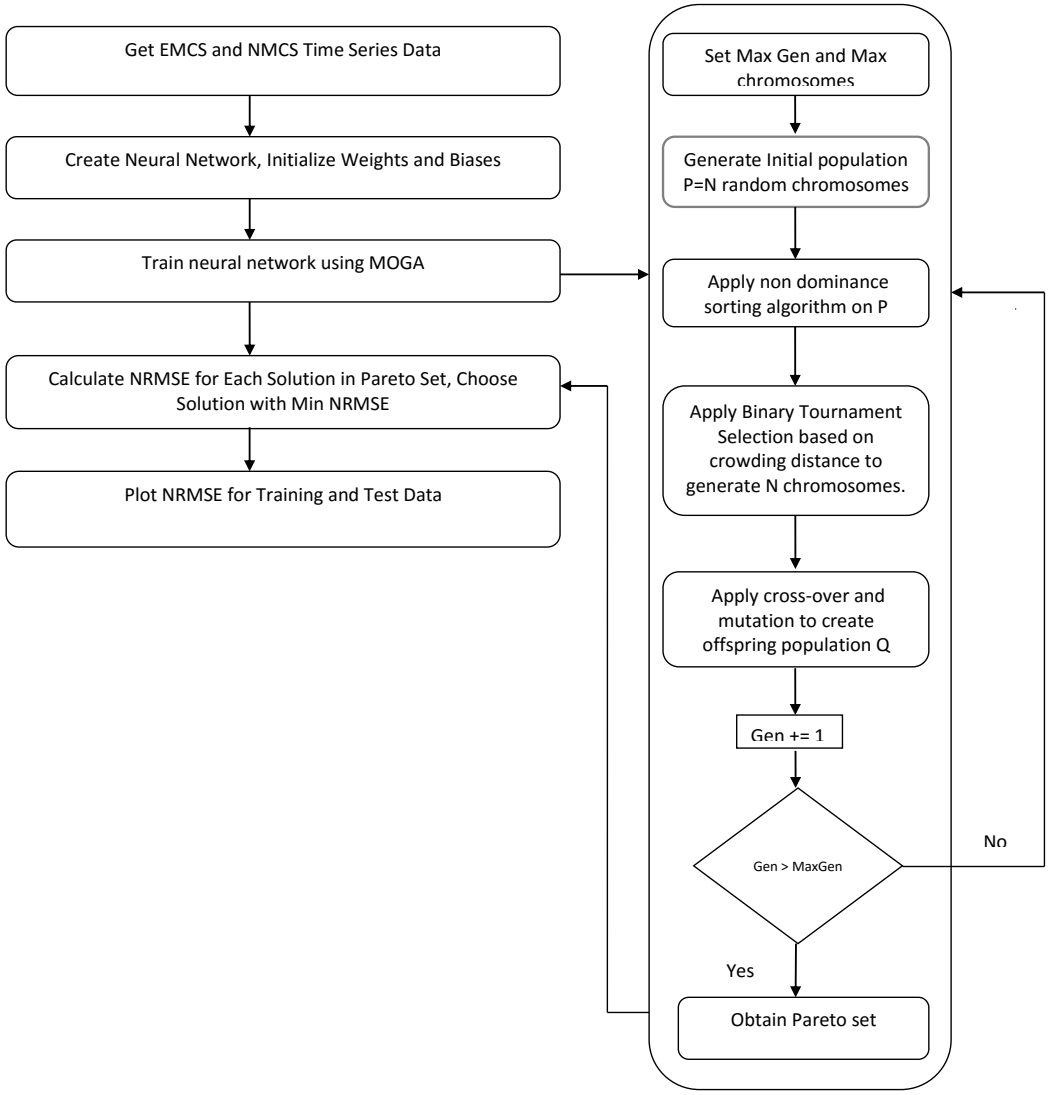


FIGURE 3.54: Flow Diagram Representation for MOGA-NN

3.17.2.2 Back Propagation based Learning

In this work, we have also implemented a multilayer neural network having one input layer, one hidden layer, and one output layer. From the ARIMA model we obtained the autoregressive parameter(p) for both EMCS and NMCS value as two; i.e., the current value is dependent upon 2 past values of the series. Hence the input layer contains two neurons for the two lags as features, the hidden layer can include a custom number of neurons which we have optimally chosen by error-and-trial as 10, and finally, the output layer includes single neuron denoting the NRMSE value. Figure 3.58 presents diagram of Bp-NN model.

3.17.2.3 Multi-Objective Genetic Algorithm Based Neural Network Learning

Neural networks training is performed to minimize the error corresponding to the given training data. The optimization technique used is "Back-Propagation" using gradient descent. Genetic Algorithm is used to minimize the cost function and to improve the accuracy. In our context, as we have two objective functions as a cost function. We have used Multi-Objective Genetic Algorithm to optimize the error. Here is a brief description of the implementation of MOGA in neural networks. Figure 3.59 presents diagram of MOGA-NN model.

Steps for Implementation MOGA in Neural Network:

1. The ArgoUML [5] dataset for 31 versions was divided into training dataset with 25 samples and test dataset with remaining 6 samples.

2. The training data was fed into an artificial neural network consisting of an input layer, a hidden layer, and output layer composed of two neurons each.

3. The neural network was trained to optimize the prediction intervals for the training data, through minimizing NMPIW and maximizing PICP. The Pareto Front for

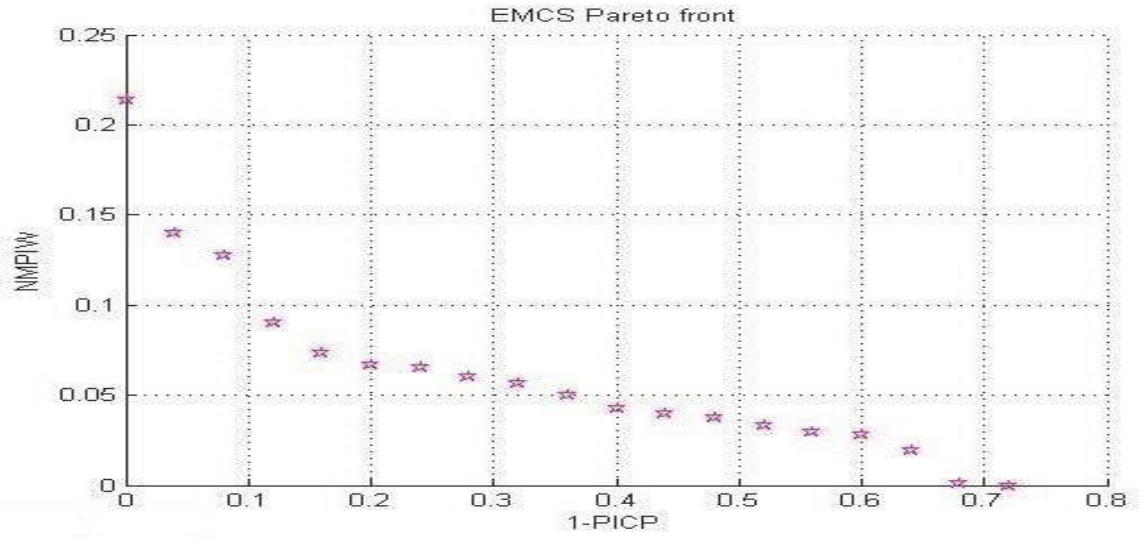


FIGURE 3.55: Pareto Plot Between X-AXIS:(1-PICP) and Y-AXIS: NMPIW(EMCS)

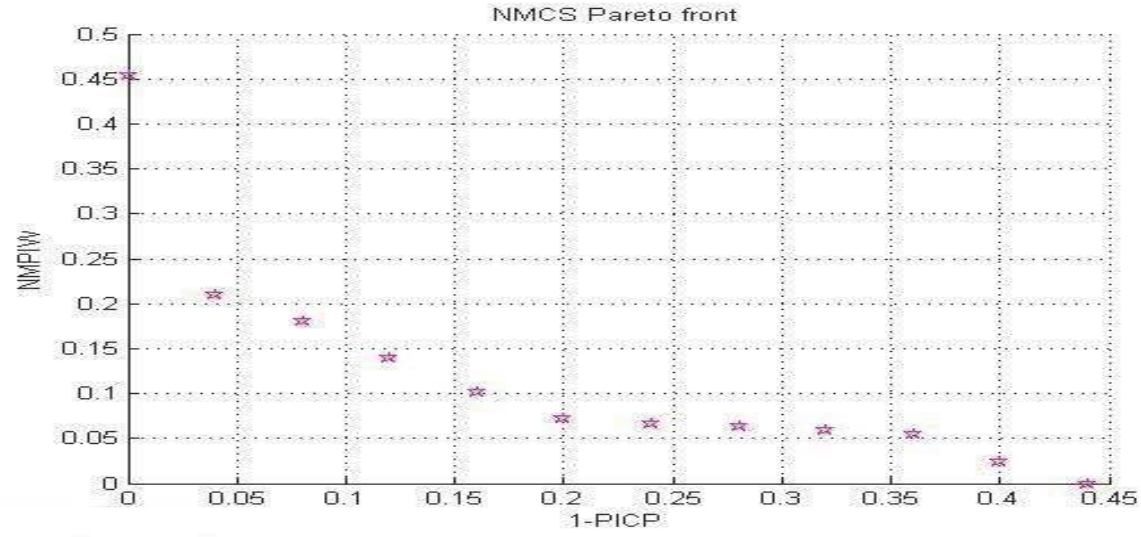


FIGURE 3.56: Pareto Plot Between X-AXIS:(1-PICP) and Y-AXIS: NMPIW(NMCS)

EMCS and NMCS after applying MOGA-NN is presented in Figure 3.55 and 3.56 respectively.

4. The optimal set of weights and bias terms, also known as a Pareto-optimal set, is used to find the most optimal solution by setting them one by one and getting the most optimal weights and bias which gives minimum NRMSE.

TABLE 3.34: Parameters Description

ARIMA	
p	Autoregressive Parameter
d	Differencing Parameter
q	Moving Average Parameter
H	Forecast Horizon
NN(BackPropagation	
I	Number of Input Neuron
O	Number of Output Neuron
T_r	Transfer Function
T_n	Training Function
Reg.P	Regularization Parameter
λ	Learning Rate
NN(MOGA)	
I	Number of Input Neuron
O	Number of Output Neuron
T_r	Transfer Function
C_p	Crossover Probability
M_p	Mutation Probability
Max_G	Maximum No. of generation
n_p	Number of Chromosome

5. The trained model is applied to the test data to get the test accuracy.

The description of parameters for all the three models is given in Table 3.34 and the value of all the parameters is given in Table 3.35, 3.36 and 3.37.

TABLE 3.35: Value of Parameter(ARIMA Model)

ARIMA Model			
Parameters	p	d	q
EMCS	2	1	2
NMCS	2	1	2

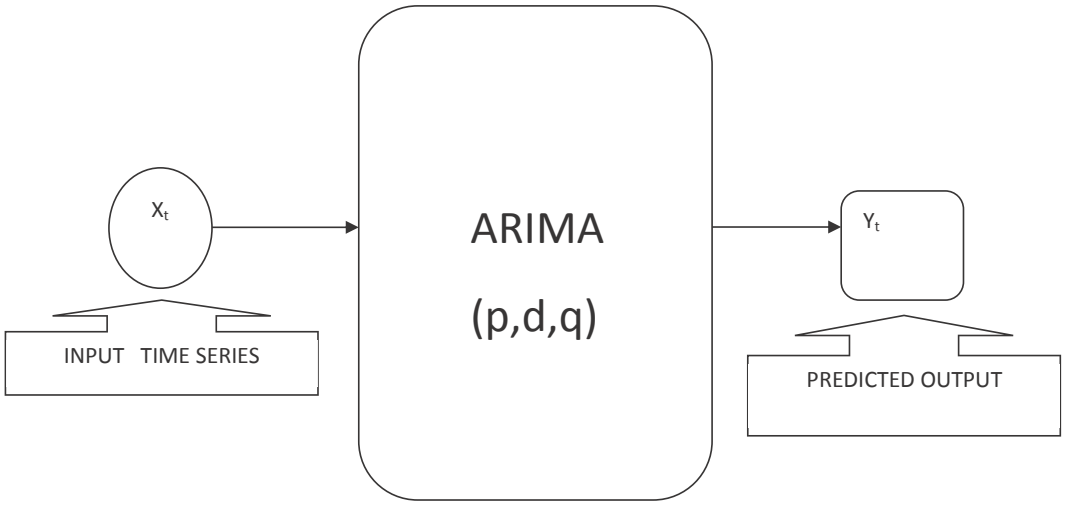


FIGURE 3.57: ARIMA Model Representation

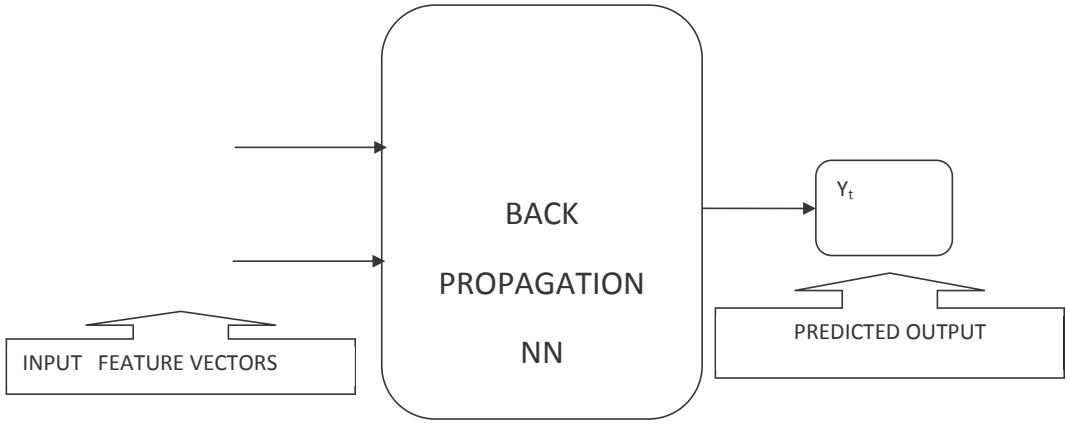


FIGURE 3.58: Back-Propagation based Neural Network

TABLE 3.36: Value of Parameter(NN (Back Propagation))

NN (BACK Propagation)						
H	I	O	T_r	T_n	Reg.P	λ
6	2	1	Tansig	BFGS . QN	0.01	0.15
6	2	1	Tansig	BFGS . QN	0.01	0.15

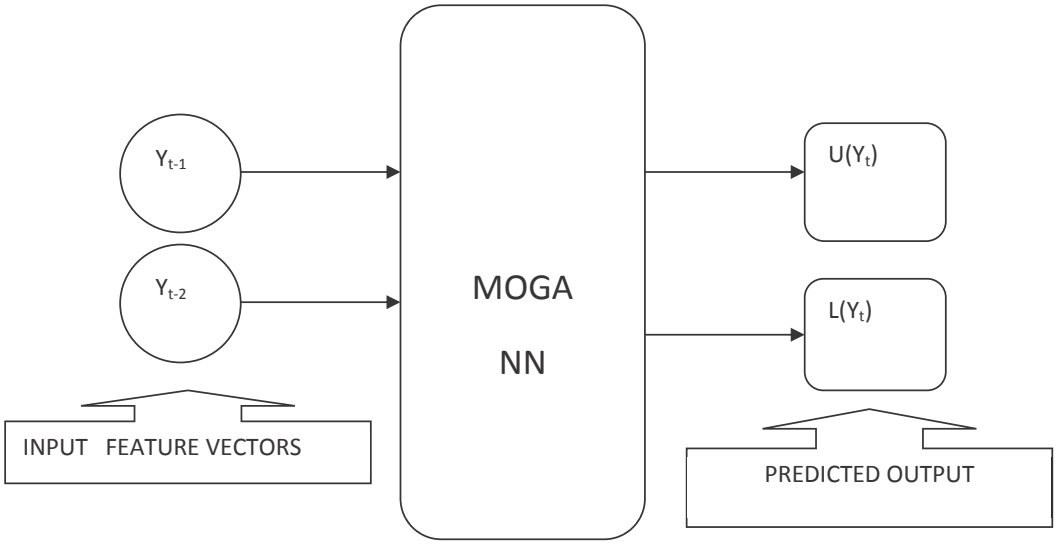


FIGURE 3.59: MOGA based Neural Network Modelling for EMCS and NMCS

TABLE 3.37: Value of Parameter(MOGA-NN)

NN (MOGA)						
I	O	T_r	C_p	M_p	Max_G	N_p
2	2	Tansig	0.8	0.06	100	25
2	2	Tansig	0.8	0.06	100	25

3.17.3 Evaluation and Interpretation

To evaluate the Prediction result, we have standard assessment methodology; the normalized root mean square error (NRMSE). The NRMSE can be defined as:

$$NRMSE = \frac{1}{y_{max} - y_{min}} \left(\sqrt{\frac{\left(\sum_{t=1}^{n_p} (y_t - \hat{y}_t) \right)}{n_p}} \right)$$

For ARIMA and Back Propagation based Neural Network modeling NRMSE is directly calculated from the point estimates for training and test data. For MOGA - NN approach, the bounds of PIs, are computed first. NRMSE is calculated by taking the mean of the upper bound and lower bound of each training sample. The models are evaluated both on the trained and test data. The model with minimum NRMSE on test data is selected.
In this section, we give a detailed presentation of NRMSE value by all the three models.

3.17.4 ARIMA Model Evaluation

As described in the section 4.6.4, the most appropriate ARIMA model that gives the best fit for the clone evolution series is ARIMA (2, 1, 2). The model is applied to both EMCS, and NMCS clone evolution series and the forecast value is also predicted for the next six values in the series. The Training error and Prediction error for both EMCS and NMCS are presented in Table 3.39. Figure 3.60 and Figure 3.63 represent the diagrammatic representation of original vs. ARIMA predicted series for EMCS and NMCS respectively.

From the Table 3.39, we interpreted that ARIMA model is a poor predictor for the test data for both EMCS and NMCS series respectively. As we see from the graph, there is a significant deviation from the original series from 25 to 31 which are the forecasted values by ARIMA model. This result shows the poor predictive capacity of Linear ARIMA model for clone evolution data which contain a mixture of linear and nonlinear components.

3.17.5 Back Propagation based NN Evaluation

As discussed in the previous section, we have a multi-layer perceptron having a single input, hidden and an output layer. We have also mentioned the number of neurons in each layer in the previous section. The model is trained with BFGS quasi-Newton method, and

the NRMSE is calculated for the train and test data. The model is validated against both EMCS and NMCS respectively. The NRMSE as given by the model is presented in Table 3.40. Figure 3.61 and Figure 3.64 represent the diagrammatic representation of original vs. MLP predicted series for EMCS and NMCS respectively.

From the Table 3.40, we found an improvement of prediction accuracy for MLP modeling. We also see that the NRMSE for the test data for both EMCS and NMCS is lower than the ARIMA modeling. However, there is still a deviation from the original series in the region 25 -31 which needs to be minimized to make this model more reliable. Though Back propagation is a good predictor of nonlinear data, simultaneously it also suffers from local optima problem. We need to further optimize the error (NRMSE) for more reliable and robust model for clone evolution prediction.

3.17.6 MOGA-NN Evaluation

The detailed procedure for implementation of MOGA-NN is already described in the previous section. Here the bounds of PIs, are calculated first. NRMSE is calculated by taking the mean of upper bound $U(X)$ and lower bound $L(X)$ of each training sample x.

The NRMSE as given by the MOGA-NN model is presented in Table 3.41. Figure 3.62 and Figure 3.65 represents the diagrammatic representation of original vs. MOGA NNpredicted series for EMCS and NMCS respectively.

From the Table 3.41, we see that MOGA-NN improves not only the accuracy but also a most reliable predictor of the clone evolution. We found from Figure 3.62 and 3.65 respectively that interval estimates are robust method of predicting the series as it completely captures the series within the intervals except for some points. Point estimates sometimes are affected by the uncertainties of the model parameters. We also see that the intervals give a clear picture of the prediction. From the Table 3.41, we also found that the

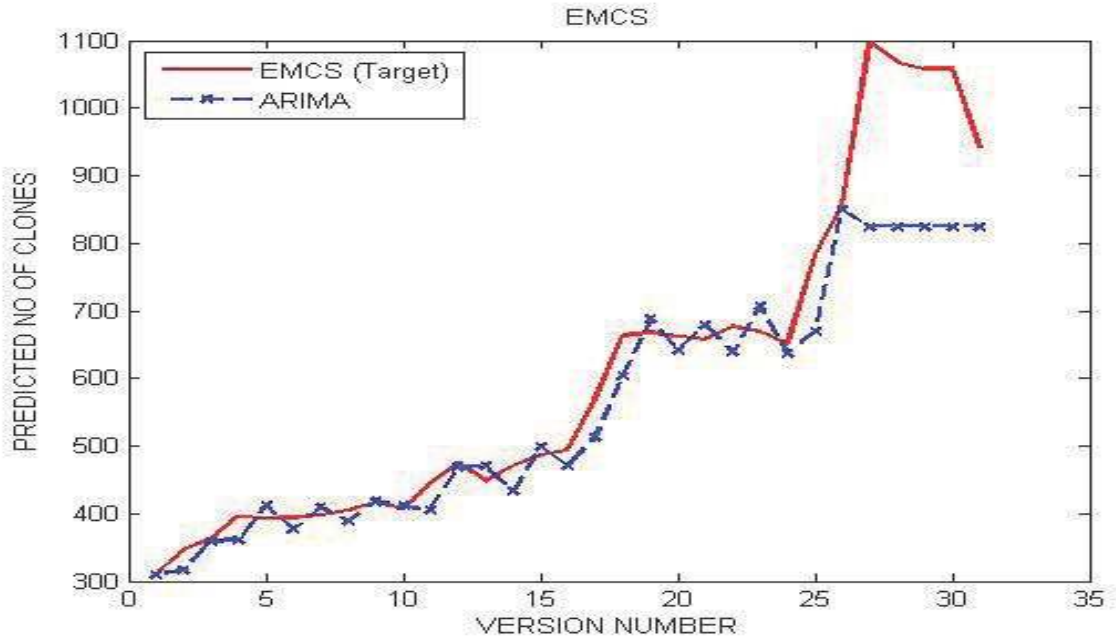


FIGURE 3.60: Actual vs. ARIMA Predicted Series (EMCS)[ArgoUML]

NRMSE value for the MOGA-NN model outperforms both ARIMA and Backpropagation model for the test data for both EMCS and NMCS. Figure 3.66 and Figure 3.67 present a bar chart representation of NRMSE for both EMCS and NMCS for test data only.

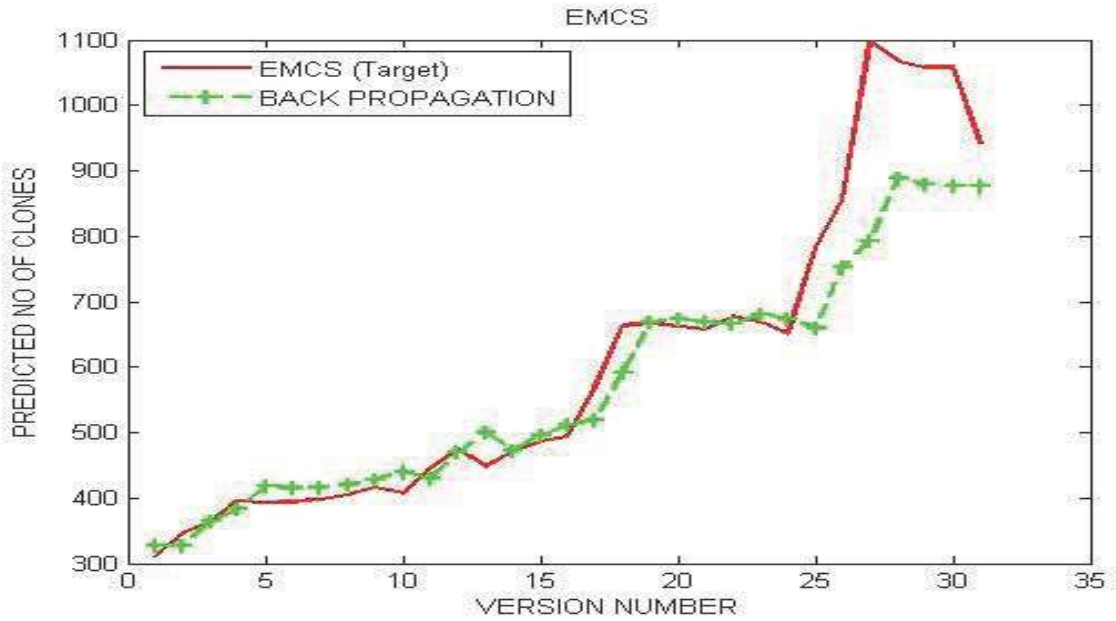


FIGURE 3.61: Actual vs. Back Propagation based NN Predicted Series (EMCS)[ArgoUML]

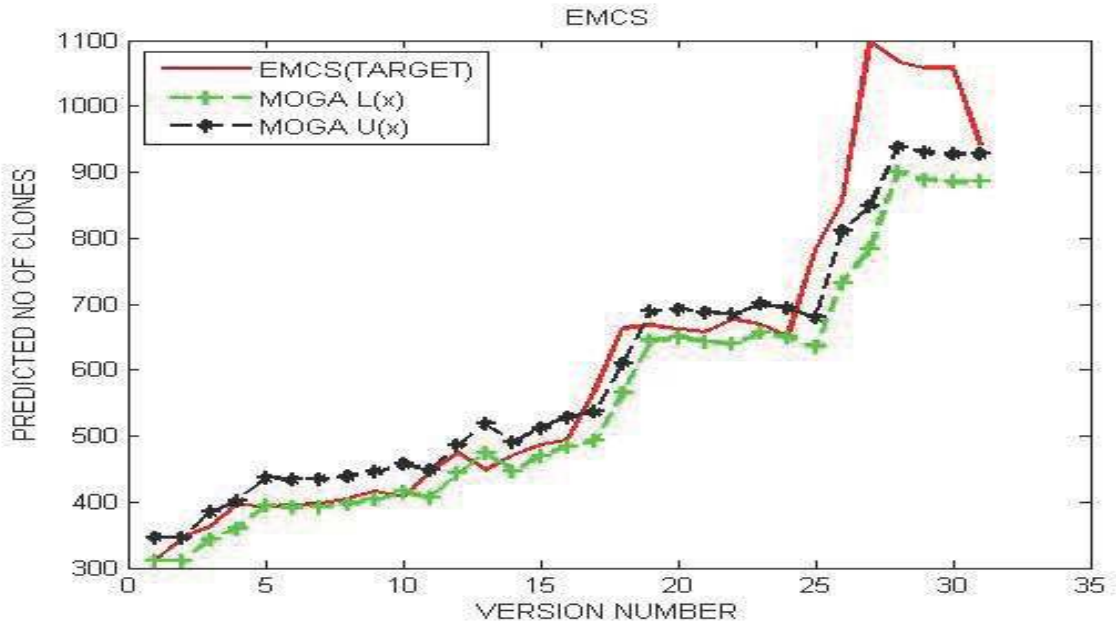


FIGURE 3.62: Actual vs. MOGA based NN Predicted Series (EMCS)[ArgoUML]

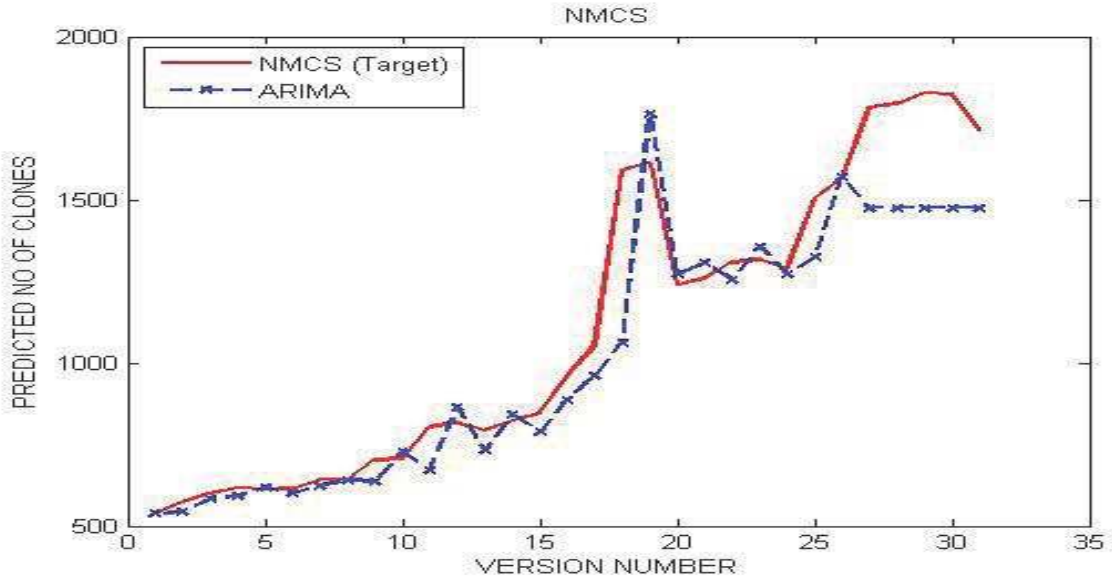


FIGURE 3.63: Actual vs. ARIMA Predicted Series (NMCS)[ArgoUML]

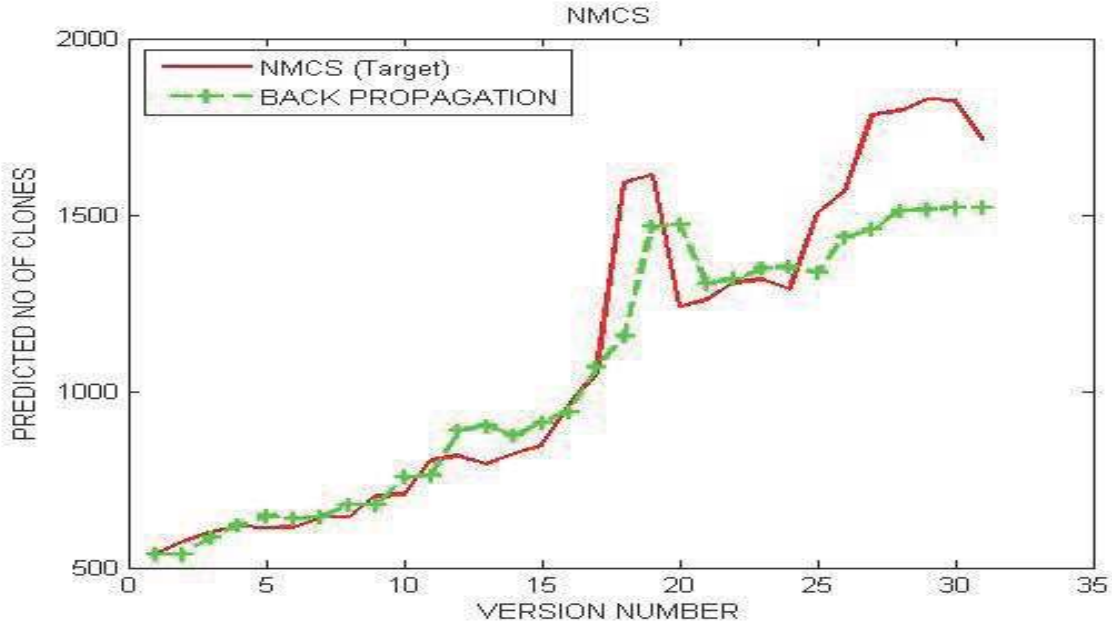


FIGURE 3.64: Actual vs. Back Propagation based NN Predicted Series (NMCS)[ArgoUML]

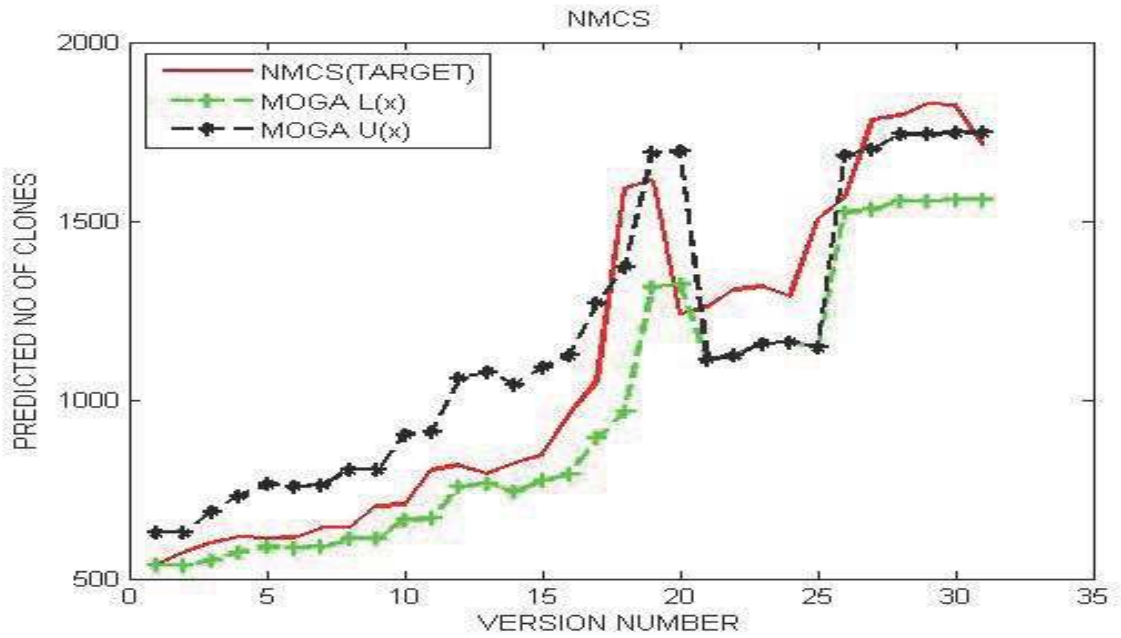


FIGURE 3.65: Actual vs. MOGA based NN Predicted Series (NMCS)[ArgoUML]

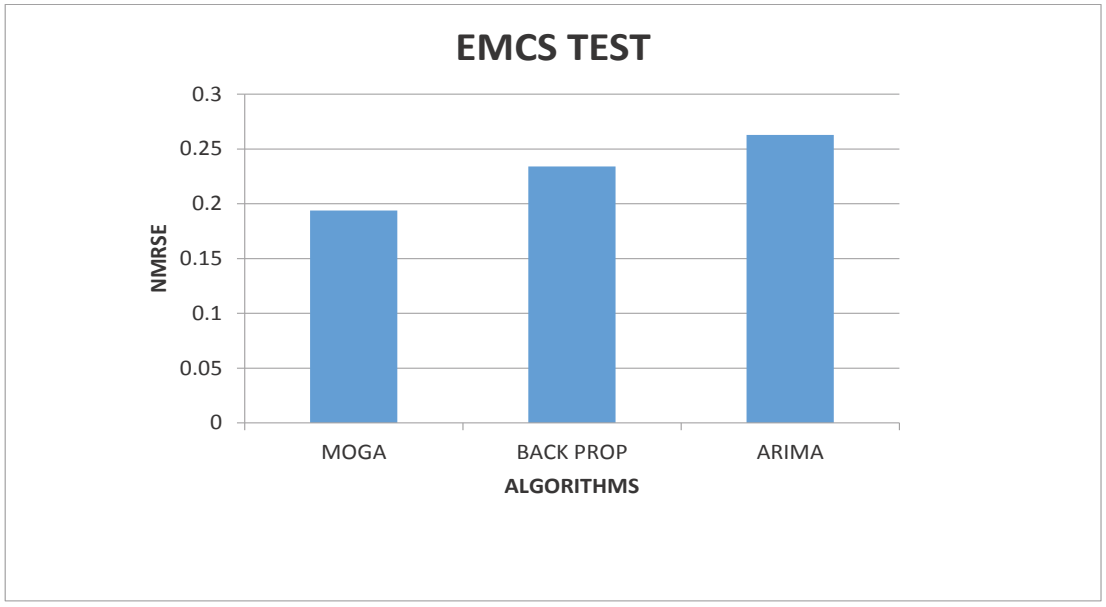


FIGURE 3.66: EMCS NRMSE Test Data[ArgoUML]

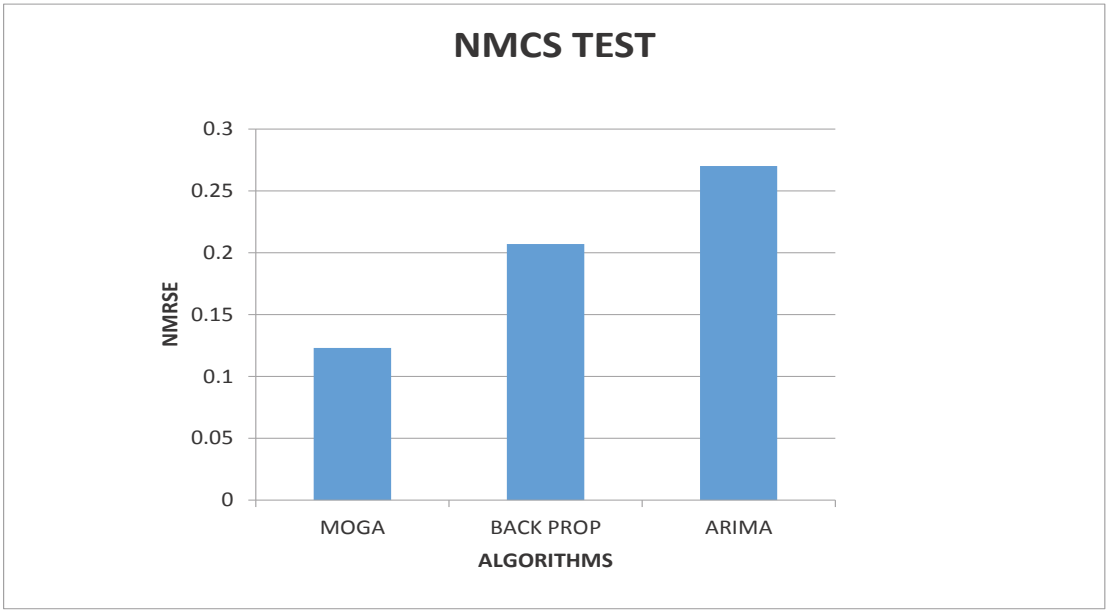


FIGURE 3.67: EMCS NRMSE Test Data[ArgoUML]

As discussed in the previous section the model which gives more accuracy (Minimum NRMSE) has to be selected for prediction of clone evolution. From the Result, we interpreted that MOGA - NN is a most suitable model for clone evolution prediction as it is more accurate, robust and reliable.

Advantages of MOGA-NN over Other Models

- It also gives us a broad scope of decision-making by providing a Pareto set of optimal solutions which provide valuable information about the underlying problem.
- Also, NSGA-II does not require derivative calculations or calculation of Jacobian or Hessian matrices.
- MOGA does not get stuck at local optimal solutions like the gradient descent techniques.

TABLE 3.38: ARIMA Result[NRMSE]

	ARIMA-TRAIN	ARIMA-Test
EMCS	0.045	0.263
NMCS	0.097	0.27

TABLE 3.39: NN- BackPropagation Result[NRMSE]

	NN-BackPropagation-TRAIN	NN-BackPropagation-Test
EMCS	0.0445	0.234
NMCS	0.0894	0.207

TABLE 3.40: NN- MOGA Result[NRMSE]

	NN-MOGA-TRAIN	NN-MOGA-Test
EMCS	0.043	0.194
NMCS	0.094	0.123

In Table 3.41 we have compared the three different techniques based on seven criteria in details.

In this work, we presented a comparative analysis of the predictive performance between ARIMA and two different types of Neural Network Modeling. The result confirms the MOGA-NN model as a best predicting model for both types of clone number series. This paper also confirms the poor performance of ARIMA model in predicting the nonlinear patterns in data.

TABLE 3.41: Comparison of Different Models

Comparison of Properties of Different Models		ARIMA	NN-BackPropagation	NN-MOGA
1	Generalization Ability	Poor	Good	Optimal
2	Algorithm Description	Simple Auto-Regression and Moving Average ensemble	Simple BackPropagarion-Trained Neural Network	MOGA-Trained Neural Network
3	Parameter Setting	Parameters setting done by ACF and PACF plots	Parameter Tuning required for Learning rate, Regularization Parameter and Training Function	Parameters tuning required for number of chromosomes, Max generations and crossover and mutation functions
4	Training Function Characteristic	Auto Regression, Moving Average	BFGS quasi-Newton method	Multiobjective Genetic Algorithm NSGA-II
5	Computation Load	Negligible	Low for both Training and Test data	High for the Training data; Low for Test data
6	Prediction Accuracy	Poor	Average	Optimal
7	Suitability (For Clone Evolution Prediction)	Not suitable	Medium Suitability	Most Suitable

3.18 Conclusion

In this chapter, we have modelled the evolving clones in the software using time series modelling. It is observed that MOGA-NN based approach most promising and efficient in predicting clone evolution. Clone detection is useful for reducing the Corrective Maintenance[203] and Preventive Maintenance [200] which involves modification of code content to solve and prevent problems in the software respectively. Because if we can identify and detect the cloned areas, the defect in all the similar code fragments can be resolved at once. The software clone evolution prediction is immensely helpful in Perfective Maintenance[200], [23] and Adaptive Maintenance [200], [23] because the effort required to evolve a software is also dependent on the amounts of cloned contents in the software.

Software Reliability Prediction

3.19 Introduction

Assurance of software quality is an important issue in the development of software applications. Software quality consists of many quality attributes like software reliability, availability, maintainability, robustness, performance safety, etc. [144]. Software reliability is the most important factor that affects the quality of the software [144]. Software reliability takes into account all kinds of faults and failures that may cause serious hazards in any types of software applications. The success rate of any software system depends upon its reliable performance throughout the operation of the software. Reliability is an important factor in both commercial software as well as in end-user software. The unreliable software can cause significant loss to the client, and it also creates disturbances in usual operations in any organization. This problem requires an effective modeling of software reliability. Modeling the software reliability is a challenging issue as it depends on many internal parameters.

The reliability prediction based on the internal parameters is always not accurate due to the complex relationship between the parameters. There are also many statistical models used for prediction of software reliability, but they are not so accurate. There exist a good deal of software reliability growth models (SRGMs), but they also suffer from unrealistic assumptions and dependency on the particular environment. Some authors have also used

ARIMA [230, 20] models for prediction of software reliability but the Linear ARIMA model is not able to catch the nonlinear temporal patterns in the data.

In this chapter, we have used advanced time series techniques for software reliability prediction. Software reliability is usually measured in terms of Time Between Failure (TBF) [148]. In the first phase, we have collected the Time Between Failure (TBF) series for three different types of the software applications. The next step is temporal modelling of TBF series for accurate prediction of software reliability. The models are validated on three different types of software systems:

1. Real Time System
2. Military System
3. Word processing System

The failure interval data for these systems is available on CSIAC Software Reliability Dataset given by J Musa [147].

The organization of the chapter is as follows: Section 2 describes Dataset Description. Section 3 describes a Hybrid Technique for Software Reliability Prediction. Section 4 describes a Prediction Interval Based Estimation Approach for Software Reliability Modelling Using MOGA-NN and ELM. Section 5 describes the application of Software Reliability Prediction. Section 6 concludes the paper and gives direction for future work.

3.20 Dataset Description

We have predicted the software reliability based on the failure interval(TBF) data as given by J Musa [147] in CSIAC software reliability dataset. There are 16 failure datasets on different domains. We have validated our model against one Real-Time application, one

TABLE 3.42: Dataset Details

Dataset Name	Number of Lines of Code	Total Number of Failures
Real Time System (System 5)	2,445,000	831
Military System (System 40)	1,80,000	101
Word Processing (System SS3)	1,00,000	278

Military Application, and one Word Processing system applications. The dataset contains the failure interval and the day of failure for each failure that occurs in the software application. We have considered the failure interval for the prediction of software reliability. The number of lines of code and the number of failures for each dataset is given in Table 3.42. The entire failure interval series is partitioned into a training set and a testing set. Partitioning helps in finding how our model fits the data pattern (both trained and unseen data) properly. The model which gives more accuracy on test data is preferably selected. To have a consistent comparison, we have distributed the training data and test data uniformly for all the three datasets.

3.21 Problem Description

Let an observed sequence of failure intervals be $\{F_1, F_2, F_3, ...F_T\}$ observed over equally spaced time points.

A fairly general model for the time series can be written:

$F_t = g(t) + \varepsilon_t$ Here, we have two components:
Systematic Part: $g(t)$ = The Component to be Modelled using Time Series.
Stochastic Part: ε_t, a residual term also called noise, which follows some unknown probabilistic law.

The objective is to predict F_t from p past observations.

$$F_t = f(F_1, F_2, F_3, ...F_{t-p}) + \varepsilon_t$$

The goal is to have a suiTable model which can predict the software reliability from Time Between Failures(TBF) accurately with a minimum value of error (ε_t). In the next section, we have discussed various time series model to predict the software reliability from Time Between Failures(TBF).

3.22 A Hybrid Technique for Software Reliability Prediction

In this work, we have applied a hybrid approach[170] (ARIMA + Neural Network Model) to predict the software reliability. We have tested the performance of our model on failure data of three different software applications. The advantage of the paper is that it uses a complete data-oriented approach and it uses an ensemble technique which gives better prediction accuracy than a single model. We have also performed a comparative analysis of performance hybrid model and ARIMA model in predicting the software reliability.

3.22.1 The Hybrid Model

ARIMA and ANNs are good at prediction in the linear and nonlinear domains respectively. However, ARIMA cannot correctly approximate the complex nonlinear problems. Similarly, ANN models cannot give more accurate results for linear problems. The real-world time series contain both linear and nonlinear terms i.e. The real world time series contain both linear and nonlinear terms i.e.

$Y_t = L_t + N_t$

L_t :Linear Component

N_t:Non-Linear Component

First, ARIMA model is applied to predict the linear component, and then the residuals from the linear model will contain only the nonlinear relationship. Now the residuals from the nonlinear model can be represented by:

$e_t = Y_t - F(L_t)$

Where $F(L_t)$ is the forecast value for time t from the estimated relationship (2). With 'n' input nodes, the ANN model for the residuals will be:

$e_t = f(e_{t-1}, e_{t-2}, ...e_{t-n}) + \varepsilon_t$

Where f is a nonlinear function determined by neural network and ε_t is the random error. Now let the predicted nonlinear component is $F(N_t)$, the combined forecast will be:

$F(Y_t = F(L_t) + F(N_t)$

So, the hybrid model work in two steps:

- In the first step, the ARIMA model is used to analyze the linear patterns in the model.
- In the second step, a nonlinear model is used to analyze the residuals. Figure 1 shows the representation of the hybrid system.

Figure 3.68 shows the representation of the hybrid system.

3.22.2 Evaluation

The models are evaluated based on the RMSE (Root Mean Square Error), MAE (Mean Absolute Error). RMSE, MAE are calculated using following formulae.

1. $MAE = \frac{1}{n}\sum_{t=1}^{N} |(A(t) - F(t))|$

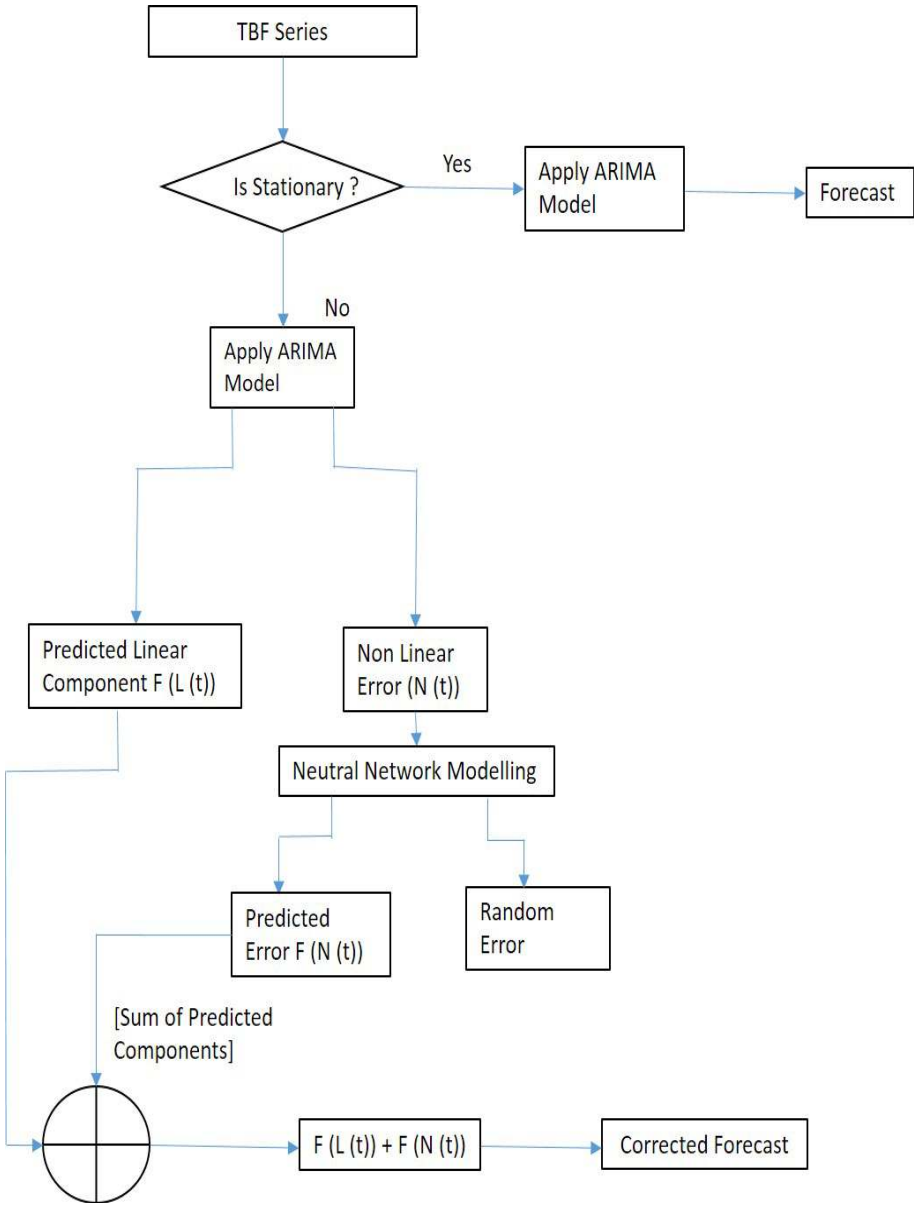


FIGURE 3.68: Proposed Hybrid Model

2. $RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{N}(A(t) - F(t))^2)}$

3.22.3 Implementation and Result

In this paper we have experimentally verified the predictive performance of two models: ARIMA and the Hybrid Model (ARIMA + ANN). In this paper we have considered three types of systems applications as discussed earlier:

- Real-Time System Application (System 5)
- Military System Application (System 40)
- Word processing System Application (System SS3)

3.22.3.1 ARIMA

The ACF cuts at lag 2 instantly, and PACF also cuts approximately at lag 1. There are some abnormalities in higher lags which can be ignored. So, the most appropriate ARIMA model for the above three software systems becomes $ARIMA(1,0,2)$, $ARIMA(2,1,2)$ and $ARIMA(1,0,1)$ respectively. We have also checked 15 combinations of p,d,q parameters for ARIMA to find the best fit. After calculation, it is also observed that these $ARIMA(p,d,q)$ are the best among all the other models for the three software systems respectively.

3.22.3.2 Hybrid Model

The error series is modeled as nonlinear autoregressive neural network model of order 3, i.e., three previous values of error are required for predicting the current value. We have used nonlinear activation function (radbasn) to match the non-linear patterns in the error series.

TABLE 3.43: RMSE (ARIMA and Hybrid Model)

RMSE Values	Train Data (ARIMA)	Train Data (Hybrid)	Test Data (ARIMA)	Test Data (Hybrid)
System 5	0.776519	0.48	0.595195	0.425
System SS3	0.71	0.348742	0.925	0.529109
System 40	0.706303	0.35	0.924692	0.53

TABLE 3.44: MAE (ARIMA and Hybrid Model)

MAE Values	Train Data (ARIMA)	Train Data (Hybrid)	Test Data (ARIMA)	Test Data (Hybrid)
System 5	0.608363	0.378251	0.440586	0.370147
System SS3	0.560832	0.348742	0.667709	0.39
System 40	0.560832	0.349	0.667709	0.388228

After applying neural network model to the error component, we get a predicted error component $F(N_t)$ and some random error ε_t now the predicted error component $F(N_t)$ and the predicted linear component $F(N_t)$ are added to get the combined forecast value.

The RMSE, MAE values as given by the ARIMA and Hybrid Model for the three software system is presented in Table 3.43 and 3.44.

From the result, we observed that the hybrid model outperforms in all experiments than the ARIMA model and also the hybrid modes fits the data pattern more accurately. In this section, we give a bar chart representation of error values for the test set in all the experiments. Figure 3.69 and Figure 3.70 show the bar chart representation of a comparison between RMSE and MAE for the different types of system applications respectively.

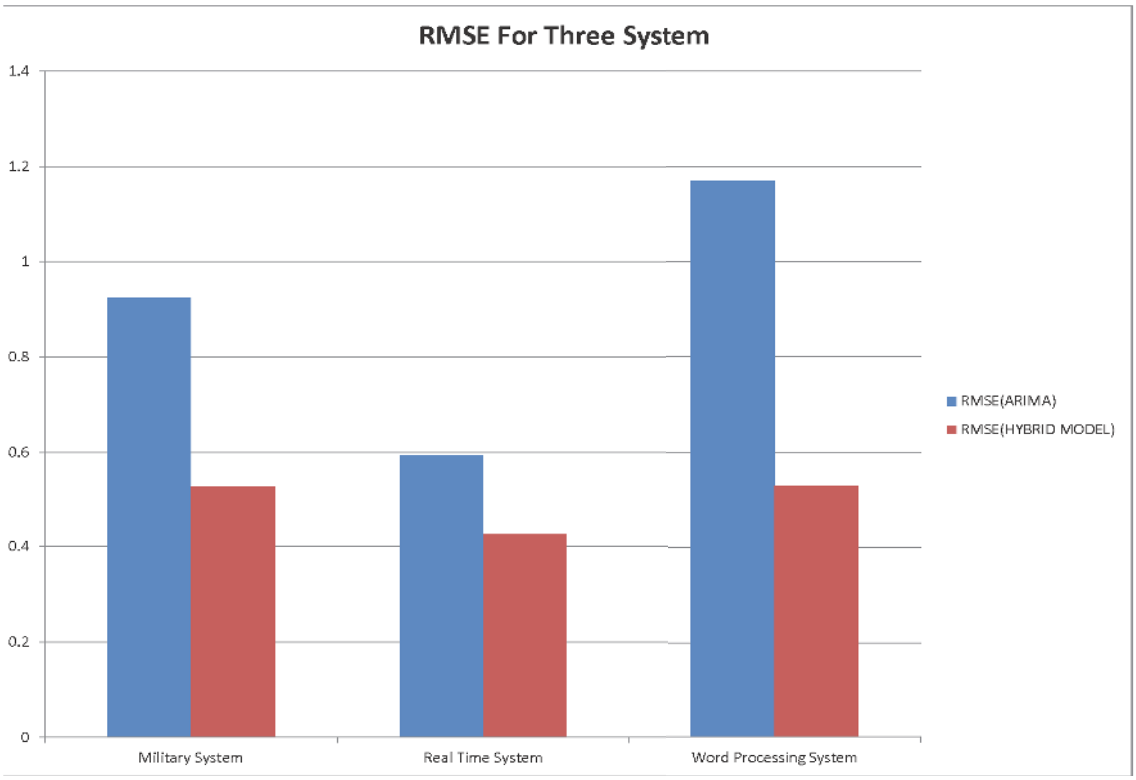


FIGURE 3.69: RMSE Comparison for Test Set

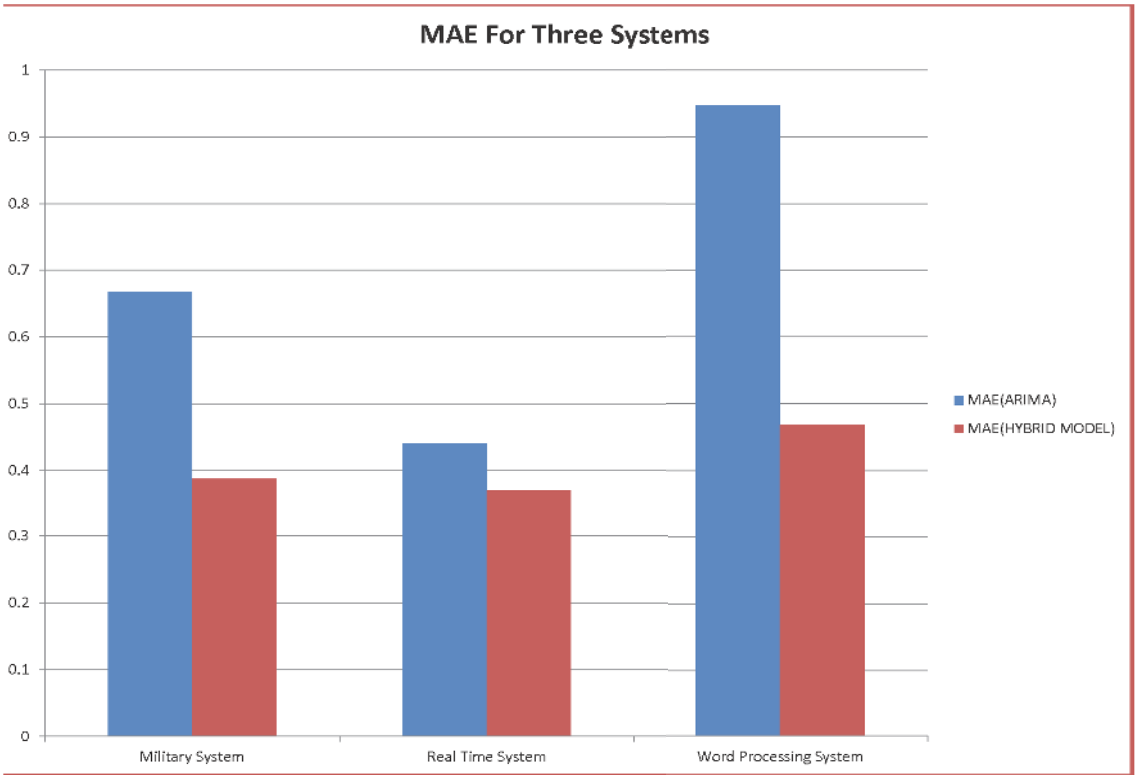


FIGURE 3.70: MAE Comparison for Test Set

From the figures, it is observed that the hybrid models always give less error than ARIMA for all the experiments. The result confirms hybrid model (ARIMA +ANN) as a most appropriate model for software reliability prediction.

3.23 A Prediction Interval Based Estimation Approach for Software Reliability Modelling

In this work, we have used two advanced machine learning approaches for predicting software reliability using the failure interval data. We have used MOGA-NN and ELM+KNN method for modeling the failure interval (Time Between Failure) patterns. In this work, we have prediction intervals for a sample instead of point prediction through artificial neural networks, as they can handle the uncertainty in the model parameters and also noise in the input data [194]. We have also given a comparative analysis of these two types of modeling techniques based on PICP (Prediction Interval Coverage Probability) and NMPIW (Normalized Mean Prediction Interval Width) values.

3.23.1 MOGA-NN Modelling

The most commonly used artificial neural network is Multi Layer Perceptron(MLP). The MLP is frequently used in pattern detection and classification problems. The commonly used training function in MLP is Back Propagation[65]. The problem with Back Propagation is that it suffers from local optima problem. In our context, we have used Multi-Objective Genetic Algorithm[246] as the training function of the neural network. Genetic Algorithm [50] is an efficient optimization algorithm. It mimics the natural process of selection based on the fitness function. A generalized genetic algorithm has following steps:

- Generate Initial Population: A random population of n chromosomes is generated. (Each chromosome represents suiTable solution for the problem)
- Evaluation of fitness function: The fitness f(x) of each chromosome x in the population is evaluated.
- Create a new population: By eliminating the weak population (Population which does not satisfy fitness criteria).
- Selection of Parents: Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
- Perform Crossover Operation: With a specific crossover probability, the crossover operation is applied to generate new offspring.
- Perform Mutation Operation: With a particular mutation probability, the mutation operation is conducted to do changes at different locations of new offspring. Place new offspring in the new population.
- Replace the old Population by a new one: Now the Genetic algorithm has to be applied to the new population.
- Stopping Criteria: If the stopping condition is satisfied, stop the algorithm. The best solution from the current population is returned, else
- Loop: Go to step 2

The Multi-Objective Genetic Algorithm [63] is the application of the Genetic Algorithm to optimize multiple objectives. It gives a set of optimal values $x = x_1, x_2, ...x_l \in R$, where 'l' is the size of Pareto-optimal set, which minimizes a set of given objective functions subject to a given set of constraints (if any). Basically, instead of point prediction through ANNs, we are here going to have prediction intervals for a sample as they can handle the uncertainty in the model parameters and also noise in the input data.

3.23.1.1 Prediction Intervals

Prediction Interval (PI) [194] is nothing but the upper bound and the lower bound of the prediction. For a given dataset $X = \{(x_i, y_i) \mid i = 1, 2, ...n\}$,
Let the prediction interval of (x_i, y_i) *be* $[L(x_i), U(x_i)]$. We define the Mean Prediction Interval Width (MPIW) and Prediction Interval Coverage Probability (PICP) for X as:

$$MPIW = \frac{1}{n}\sum_{i=1}^{n}(U(x_i) - L(x_i))$$

$$PICP = \frac{1}{n}\sum_{i=1}^{n} c_i,$$

$$where \quad c_i = \begin{cases} 1, & if \quad y_i \in [L(x_i), U(x_i)] \\ 0, & otherwise \end{cases}$$

For a better interval prediction, we have optimized the parameters PICP[107, 205, 106] to be maximum and MPIW [107, 205] to be minimum. We also see that as MPIW decreases, PICP increases. Hence we will have a competitive multi-objective optimization problem:

$$Minimize \quad NMPIW(X) \quad and \quad 1 - PICP(X) \quad Simultaneously \quad Such\ that,$$
$$NMPIW \geq 0 \quad and \quad 0 \leq PICP(X) \leq 1$$

We have taken normalized $MPIW(NMPIW)$
instead of $MPIW$, *where* $NMPIW = \frac{MPIW}{y_{max} - y_{min}} and$
We choose to minimize $(1 - PICP)$ instead of maximizing $PICP$ because both are equivalent.

The optimal set x is known as a Pareto-optimal set. It is a set of solutions, satisfying the objective functions at accepted levels without being dominated by any other solution.
In other words, if we have to minimize N objective functions, $f_1(x), f_2(x), ...f_N(x)$, subject to some given constraints,

Let X be the solution space of feasible solutions, then $x_1 \in X$ is said to dominate $x_2 \in X$

$iff \quad f_i(x_1) \leq f_i(x_2)$

$for\ all\ \ i \in [1,N]\ \ and$

$f_j(x_1) < f_j(x_2)\ \ for\ some\ \ j \in [1,N].$

Hence, for x being a Pareto optimal set, it should not be dominated by any other solution in the solution space.

3.23.1.2 NSGA-II[50]

In this algorithm, after each generation, on the new population non-dominance sorting algorithm[50] is applied to sort the chromosomes in the order of dominance. The chromosomes are selected by crowding distance by using binary tournament selection. The crossover and mutation operation is performed to produce the next generation, which also becomes a part of the population. As soon as the maximum generation is reached, the first Pareto front of the sorted population is returned, and the corresponding set is the Pareto optimal set. Here we have presented a brief description of the implementation of MOGA in neural networks:

1. Initialize a feed forward neural network N where $N(x_i)$ represents the predicted value for the training sample $x_i \in X$.

2. Train the neural network with MOGA as the optimization function for minimizing the cost functions, i.e. $NMPIW(X)$ *and* $1 - PICP(X)$.

3. The Pareto optimal set of weights and biases, P (Pareto Set) obtained after the neural network training, is used to find the most optimal solution by setting each set of weights $(w_i, b_i) \in P$ for the training dataset X.

4. Choose four best solutions from the Pareto set which perform best on the test data.

5. Evaluate the performance of the prediction intervals by finding the PICP and MPIW of the intervals.

6. For the chosen solution obtain lower and upper bounds for input x and plot it.

The Flow Diagram for MOGA-NN implementation is given in Figure 3.71.

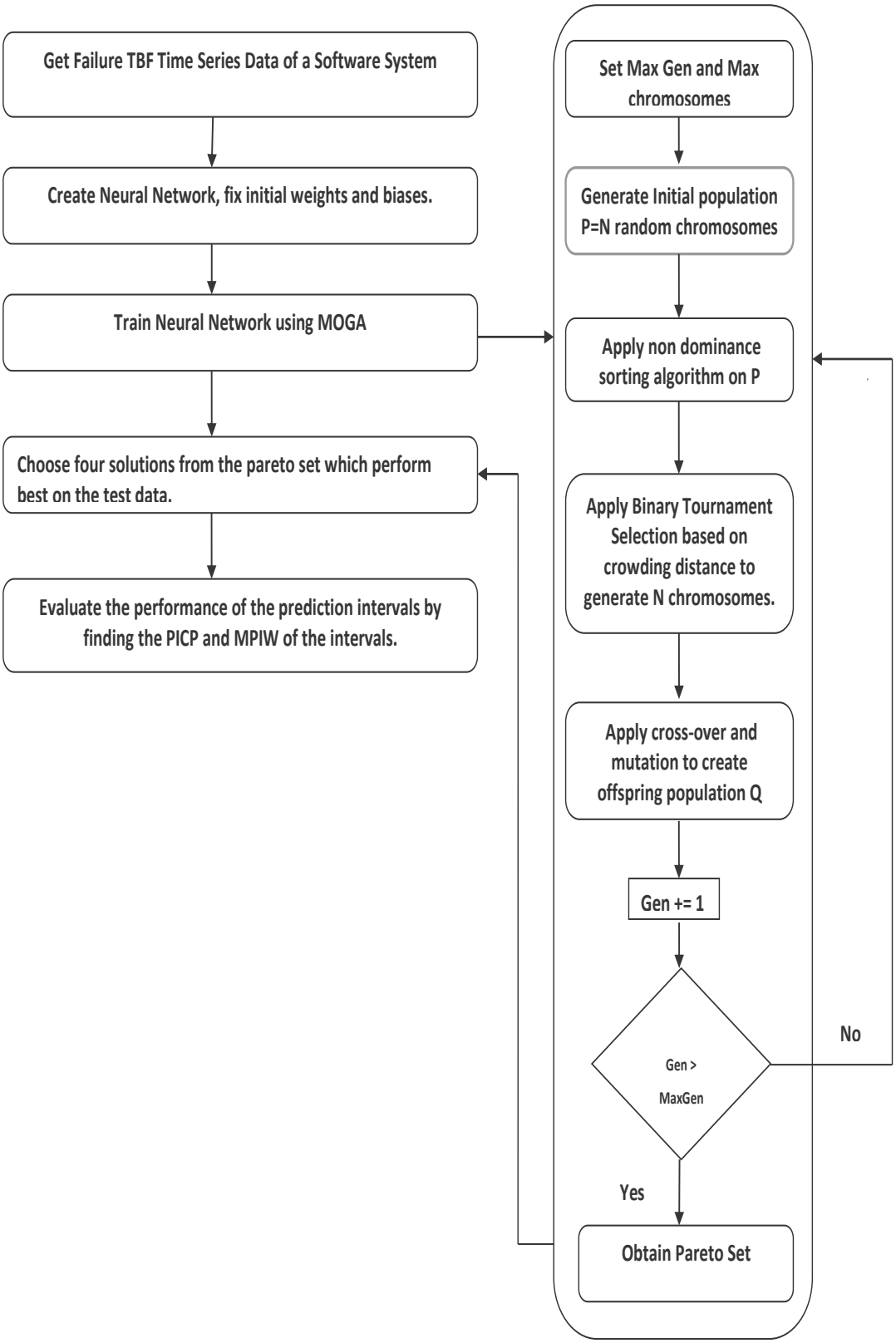


FIGURE 3.71: Flow Diagram Representation for MOGA-NN

3.23.2 ELM+KNN Modelling

The approach proposed for quantifying PIs combines a regression performed by ELMs with the nearest neighbors approach. First, the ELM regression algorithm is trained to provide point estimates and, then, the nearest neighbors approach(KNN)[30] is applied to quantify the PIs, as proposed in. Both the steps of point estimate regression and PIs quantification can be applied independently.

Extreme Learning Machines (ELM)[80, 79] is an extremely fast learning algorithm for the single hidden-layer feed-forward networks with $\tilde{\mathrm{N}}$ hidden neurons,
where $\tilde{\mathrm{N}} \leqslant$ number of training samples.

For N arbitrary distinct samples x_i, t_i , where $x_i \in R_n$ and $t_i \in R_m$, trained in a single hidden-layer feed-forward networks with $\tilde{\mathrm{N}}$ hidden neurons and activation function g(x),
Let $\sum_{i=1}^{\tilde{\mathrm{N}}} \beta_i g_i(w_i \cdot x_j + b_i) = o_j, j = 1,2,3....,N$
where β_i = weight vector connecting the i-th hidden neuron and the output neurons.
w_i = weight vector connecting the i-th hidden neuron and the input neurons
b_i = bias of the i-th hidden neuron. The above equations can be written in the matrix form as:

$$H\beta = O$$

where, $H(w_1, ..., w_{\tilde{\mathrm{N}}}, b_1, ...b_{\tilde{\mathrm{N}}}, x_1....x_N)$

$$H = \begin{bmatrix} g(w_1 \cdot x_1 + b_1) \cdots g(w_{\tilde{\mathrm{N}}} \cdot x_1 + b_{\tilde{\mathrm{N}}}) \\ \vdots \cdots \vdots \\ g(w_1 \cdot x_N + b_1) \cdots g(w_{\tilde{\mathrm{N}}} \cdot x_N + b_{\tilde{\mathrm{N}}}) \end{bmatrix}_{N \times \tilde{\mathrm{N}}}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_{\tilde{\mathrm{N}}}^T \end{bmatrix}_{\tilde{\mathrm{N}} \times m} \quad \text{and}$$

$$O = \begin{bmatrix} o_1^T \\ \vdots \\ o_N^T \end{bmatrix}_{N \times m}$$

We have to minimize the value of $||O - T||$, that is equivalent to find $\beta_i, w_i, b_i (i = 1, 2, 3, ..., \tilde{\mathrm{N}})$ such that :

$$\left|\left| H(w_1, ..., w_{\tilde{\mathrm{N}}}, b_1, ...b_{\tilde{\mathrm{N}}})\beta - T \right|\right| =$$

$$\left|\left| min(w_i, b_i, \beta) H(w_1, ..., w_{\tilde{\mathrm{N}}}, b_1, ...b_{\tilde{\mathrm{N}}})\beta - T \right|\right|$$

The smallest norm least-square solution of the above linear system is :

$\hat{\beta} = H \dagger T$

Where $H\dagger$ = Moore-Penrose generalized inverse of H [37] i.e.

$\left|\left| H\hat{\beta} - T \right|\right| = min_\beta ||H\beta - T||$ and, $\left|\left| \hat{\beta} \right|\right| \leq ||\beta||$

for all $\beta \in \beta : ||H\beta - T|| \leq ||HZ - T||$

$forall \quad z \in R\tilde{\mathrm{N}} * N$

The basic algorithm of ELM is given below.

ELM Algorithm

Given a training set X =$(x_i, t_i) | x_i \in R_n, t_i \in R_m, i = 1, ..., N$

Activation function g(x), and number of hidden neurons = N

- Assign arbitrary input weight w_i and bias $b_i, i = 1, ..., N$

- Calculate the hidden layer output matrix H
- Calculate the output weight $\beta = H \dagger T$

The Flow Diagram for ELM + Nearest Neighborhood implementation is given in Figure 3.72.

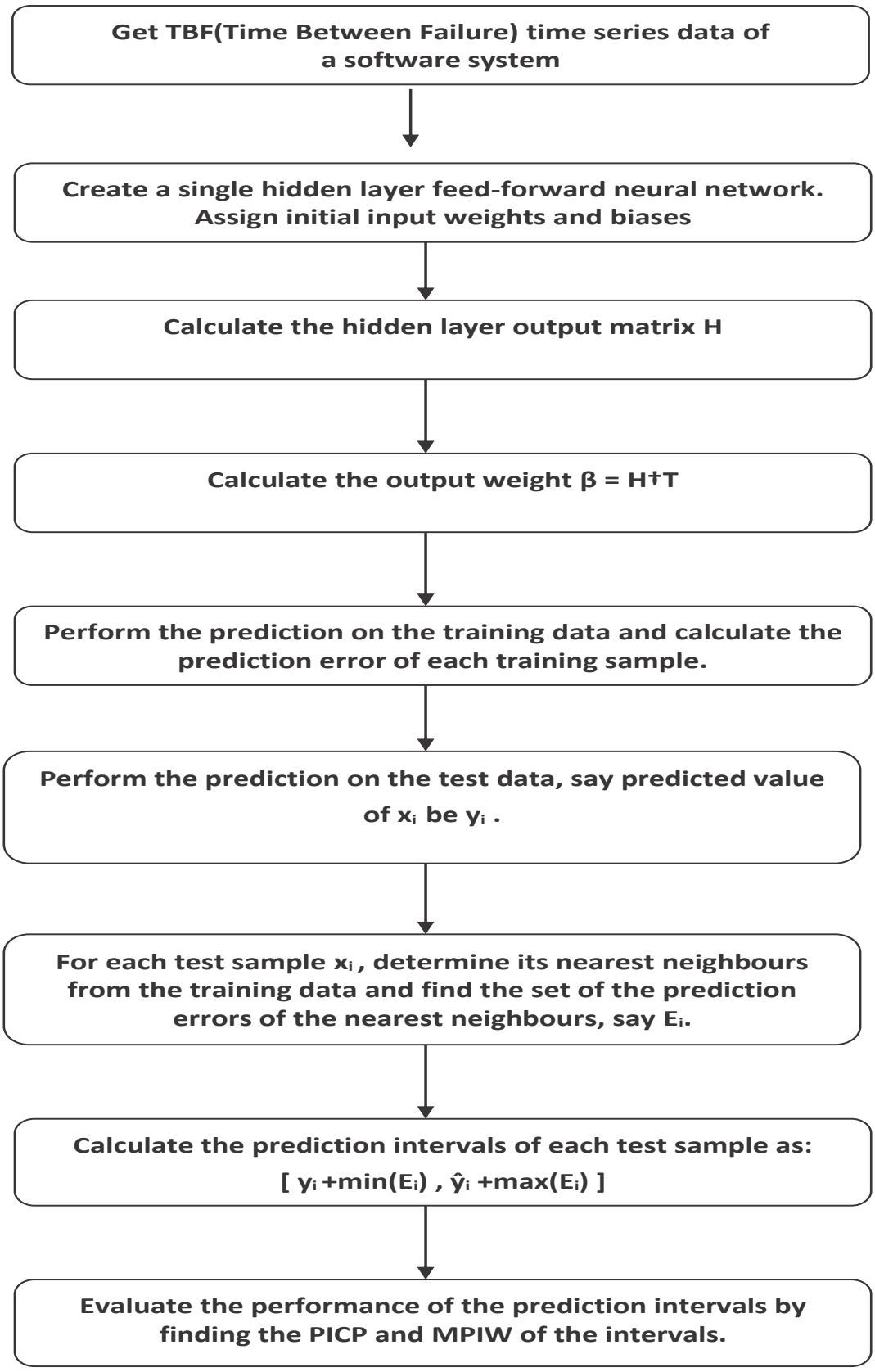


FIGURE 3.72: Flow Diagram Representation for ELM

3.24 Implementation and Result

In the first phase, we have collected the Time Between Failure (TBF) series for three different type of software applications. The next step is temporal modelling of TBF series for accurate prediction of software reliability. As discussed in the previous section, we have used an interval based prediction approach in this paper to handle uncertainty in the internal parameters and noise in the data.

3.24.0.1 MOGA-NN Modelling

Neural networks are trained to minimize the error corresponding to the given training data. The optimization technique frequently used is "Back-Propagation" using gradient descent. To have improved accuracy, "Genetic Algorithm" is used to minimize the cost function. In our context, as we have two objective functions as cost functions. We have used Multi-Objective Genetic Algorithm to optimize the MPIW and PICP. The diagrammatic representation of an implementation of MOGA Based Neural Network model is given in Figure 1. Here is a brief description of the implementation of MOGA in neural networks.

Steps for Implementation MOGA in Neural Network:

- The Reliability dataset for each Software system was divided into training data with 90% samples and test data with remaining 10% samples.
- The training data was fed into an artificial neural network consisting of an input layer, a hidden layer, and output layer composed of two neurons.
- The neural network was trained to optimize the prediction intervals for the training data, by minimizing NMPIW and maximizing PICP. The Pareto Front for the three systems after applying MOGA-NN is presented in Figure 3.73, 3.74 and 3.75.

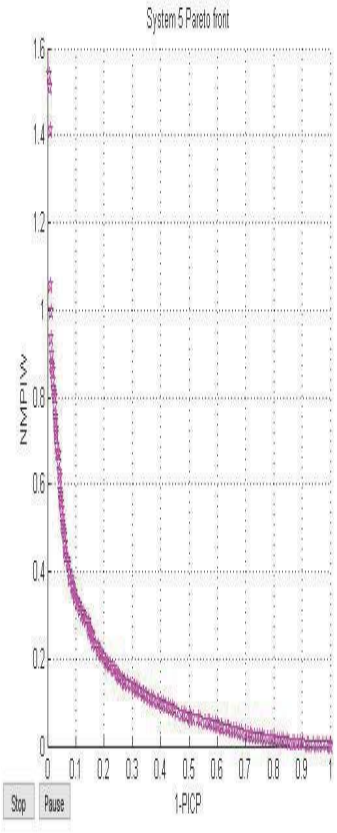


FIGURE 3.73: Pareto Plot Between X-AXIS:(1-PICP) and Y-AXIS: NMPIW (System 5)

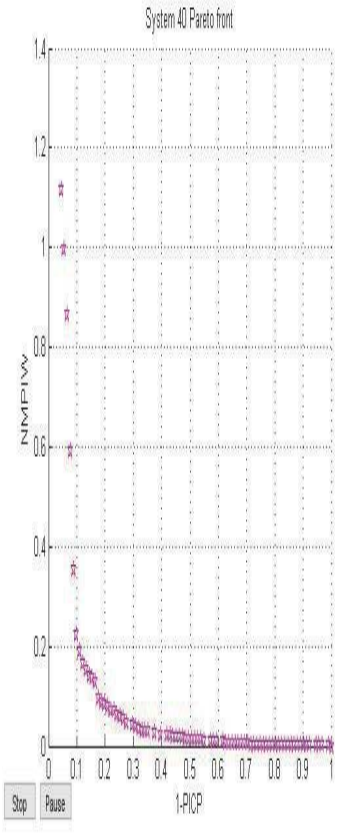


FIGURE 3.74: Pareto Plot Between X-AXIS:(1-PICP) and Y-AXIS: NMPIW (System 40)

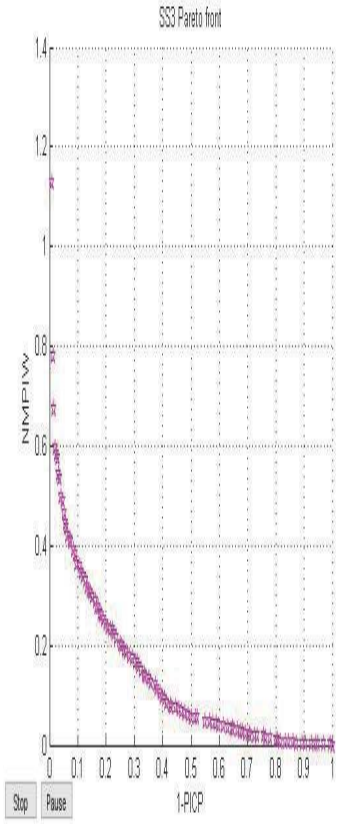


FIGURE 3.75: Pareto Plot Between X-AXIS:(1-PICP) and Y-AXIS: NMPIW (System SS3)

- The optimal set of weights and bias terms, also known as a Pareto-optimal set, is used to find the most optimal solution.
- The trained model is applied to the test data to get the prediction intervals of the test data. Four best solutions are chosen from the Pareto set which performs best on the test data.
- The PICP and NMPIW values are calculated after getting the prediction intervals for the test data.

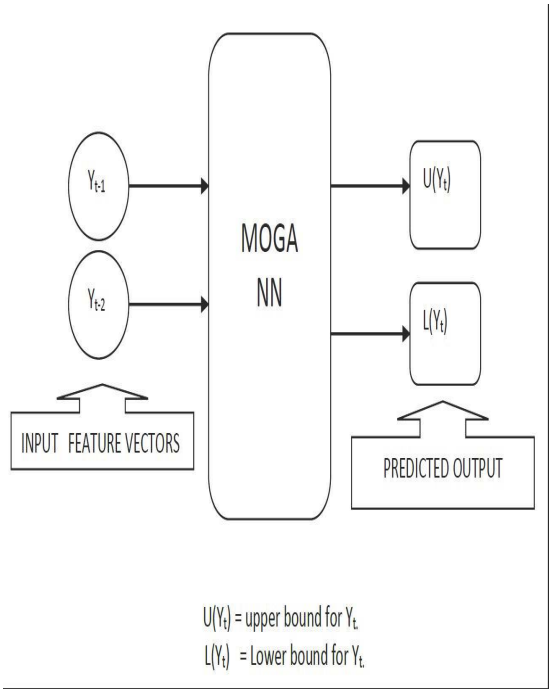


FIGURE 3.76: MOGA-NN Modelling Diagram

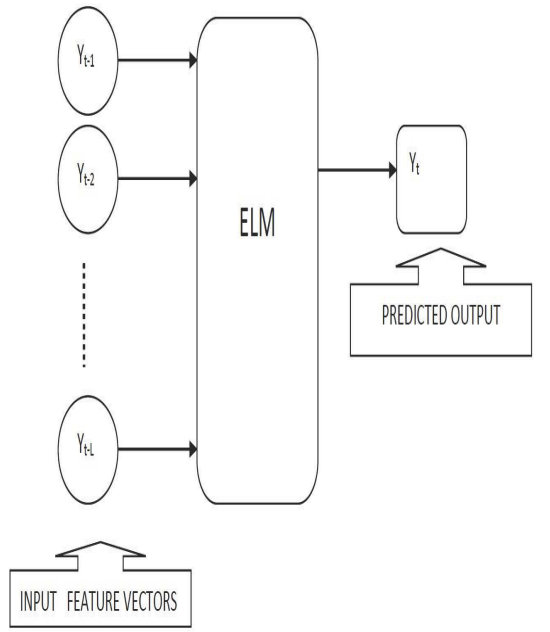


FIGURE 3.77: ELM Modelling Diagram

The number of nodes in the input layer is decided by the autoregressive parameter(p). We obtained the p-value for three TBF series by Partial Auto-Correlation(PACF) plots. We found p-value as 3 for System SS3 and System 5 and 2 for System 40, i.e., the current value is dependent upon two past values of the series. So, the number of neurons in the input layer is decided by the p-value. The number of neurons in the output layer is also two:

- Upper Bound of Interval
- Lower Bound of Interval

Where each neuron represents a bound for the sample

The diagrammatic representation of an implementation of MOGA Based Neural Network model is given in Figure 3.76 The transfer function for both hidden layer and output layer

TABLE 3.45: Parameters Description

ELM	
I	Number of Input Neurons
O	Number of Output Neurons
H	Number of Hidden Neurons
T_o	Output Layer Transfer Function
T_h	Hidden Layer Transfer Function
Reg.P	Regularization Parameter

NN(MOGA)	
I	Number of Input Neurons
O	Number of Output Neurons
H	Number of Hidden Neurons
T_o	Output Layer Transfer Function
T_h	Hidden Layer Transfer Function
C_p	Crossover Probability
M_p	Mutation Probability
Max_G	Maximum No. of generation
N_p	Number of Chromosome
$Reg.P$	Regularization Parameter

TABLE 3.46: Value of Parameters MOGA-NN

	MOGA-NN Parameters									
System Name	I	O	H	C_p	M_p	N_p	Max_G	$Reg.$	T_h	T_o
System 5	3	2	5	0.8	0.06	25	100	0.01	tansig	tansig
System 40	2	2	3	0.8	0.06	25	100	0.01	tansig	tansig
System SS3	3	2	3	0.8	0.06	25	100	0.01	tansig	tansig

is taken as tangent sigmoid (tansig). The description of all the parameters for the MOGA-NN for the three TBF series is given in Table 3.45 and the value of all the parameters are given in Table 3.46.

3.24.0.2 ELM+KNN Modelling

In this section, we have applied ELM modelling technique to TBF series for all the three software systems. ELM is a powerful learning technique and has good generalization capability and more accurate prediction.

Here we have presented a brief description of implementation of ELM : Steps for Implementation of ELM[79, 80]+ KNN [120]:

- The Reliability dataset for each Software system was divided into training data with 90% samples and test data with remaining 10% samples.
- Create a single hidden-layer feed forward network. Assign arbitrary input weights and biases.
- Calculate the hidden layer output matrix H.
- Calculate the output weight $\beta = H \dagger T$
- Perform the prediction on the training data and calculate the prediction error of each training sample.
- Perform the prediction on the test data, say the predicted value of x_i be $\hat{y}_i$.
- For each test sample x_i, determine its nearest neighbors from the training data and find the set of prediction errors of the nearest neighbors, say E_i
- Calculate the prediction intervals of each test sample as: $[\hat{y}_i + min(E_i), \hat{y}_i + max(E_i)]$
- Evaluate the performance of the prediction intervals by finding the PICP and MPIW of the intervals.

The diagrammatic representation of an implementation of ELM + KNN is given in Figure 3.77. For ELM, the p value is obtained for the three TBF series by Partial Auto-Correlation(PACF) plots. We got the lag value as two for System 5; i.e. the current value is dependent upon past two values of the series.Similarly, we got the lag value as three for System 5 and System SS3. So, the input layer contains number of neurons according to the p-value. The number of neurons in the output layer is one in the case of ELM.

TABLE 3.47: Value of Parameters for ELM +KNN

	ELM Modelling Parameters					
System Name	I	O	H	T_r	T_H	k
System 5	3	1	5	tansig	tansig	11
System 40	2	1	3	tansig	tansig	11
System SS3	3	1	3	tansig	tansig	11

The transfer function for both hidden layer and output layer is taken as tangent sigmoid (tansig). The description of all the parameters for the ELM + KNN for the three TBF series is given in Table3.45 and the value of all the parameters are given in Table 3.47.

3.24.1 Evaluation and Interpretation

As discussed in the previous section, we have applied an interval based prediction approach for software reliability modelling. The MOGA-NN produced a pareto set containing two optimized solutions. The first one is the Prediction Interval Coverage Probability (PICP) and the second one is the Normalized Mean Prediction Interval Width(NMPIW). The first one assures the probability of the predicted values, to be within the intervals while the second one minimizes the interval width. The objective is to create a robust model for software reliability prediction. So, we have evaluated the two models based on PICP and NMPIW. The definition for PICP and NMPIW is given below. In case of ELM, K values is used to create prediction intervals.

$$1 : NMPIW = \frac{1}{n}\sum_{i=1}^{n}(U(x_i) - L(x_i))$$

$$2 : PICP = \frac{1}{n}\sum_{i=1}^{n} c_i,$$

$$where \quad c_i = \begin{cases} 1, & if \quad y_i \in [L(x_i), U(x_i)] \\ 0, & otherwise \end{cases}$$

TABLE 3.48: Four Best Solutions and Their PICP and NMPIW Values Obtained by MOGA-NN and ELM +KNN Model:
System 5

System 5	MOGA-NN	ELM+KNN
Selected Solutions	(PICP, NMPIW)	(PICP, NMPIW)
1	(0.8554, 0.4177)	(0.8795, 0.3843)
2	(0.8434, 0.4298)	(0.8554, 0.379)
3	(0.8193, 0.3921)	(0.8313, 0.1745)
4	(0.7831, 0.3900)	(0.7952, 0.3245)

3.24.1.1 MOGA - NN Evaluation

We have evaluated the model based on PICP, NMPIW. We have already defined them in this section. We have plotted the graphs for showing the Target series, Lower bound L(X) and Upper bound U(X). Plots of Target series for the trained data for the three software systems is shown in Figure 3.78,3.79,3.80. The plots of Target series for the test data for the three software systems is shown in Figure 3.81,3.82,3.83.

From the Figures, it is observed that MOGA-NN is a good predictor of software reliability. The intervals U(X) and L(X) completely captures the target series except for some cases. We also able get a satisfactory value for PICP and NMPIW in the Multi-Objective modelling. The model is validated for both train data and test data. Usually, the model which gives a better result for test data is preferred. The PICP and NMPIW as calculated by the MOGA-NN model, for all the three software systems is presented in Table 3.48, 3.49 and 3.50 (For Test Data). We have presented four best solutions as given by MOGA-NN Model.

3.24.1.2 ELM + KNN Evaluation

In this case, for quantifying the PIs, we used regression performed by ELM followed by the nearest neighborhood approach. ELM modelling generates point estimates. After getting the point estimates, we got the intervals by the nearest neighborhood approach. As

TABLE 3.49: Four Best Solutions and Their PICP and NMPIW Values Obtained by MOGA-NN and ELM +KNN Model:
System 40

System 40	MOGA-NN	ELM+KNN
Selected Solutions	(PICP, NMPIW)	(PICP, NMPIW)
1	(0.9091, 0.8425)	(0.7273, 0.5972)
2	(0.8182, 0.72)	(0.6364, 0.4708)
3	(0.7273, 0.2712)	(0.6364, 0.4628
4	(0.7273, 0.2387)	(0.6364, 0.4557)

TABLE 3.50: Four Best Solutions and their PICP and NMPIW Values Obtained by MOGA-NN and ELM +KNN Model:
System SS3

System SS3	MOGA-NN	ELM+KNN
Selected Solutions	(PICP, NMPIW)	(PICP, NMPIW)
1	(0.8214, 0.5678)	(0.8571, 0.4855)
2	(0.7857, 0.5694)	(0.8214, 0.494)
3	(0.75, 0.5594)	(0.7857, 0.4586)
4	(0.7143, 0.5193)	(0.75, 0.4441)

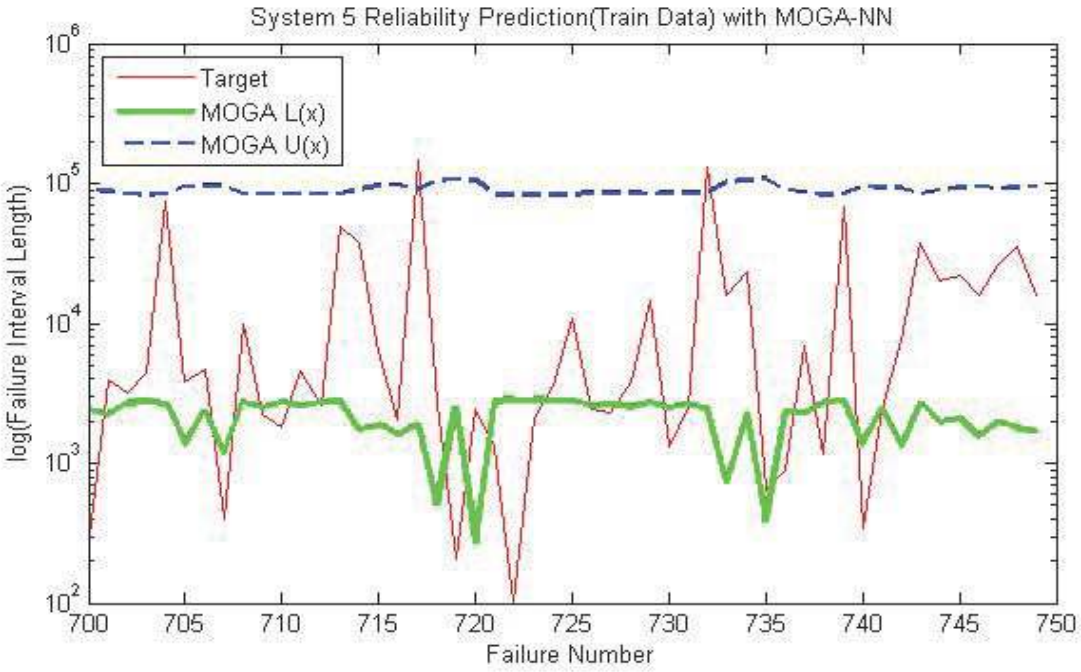


FIGURE 3.78: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 5)[MOGA-NN (Train Data)]

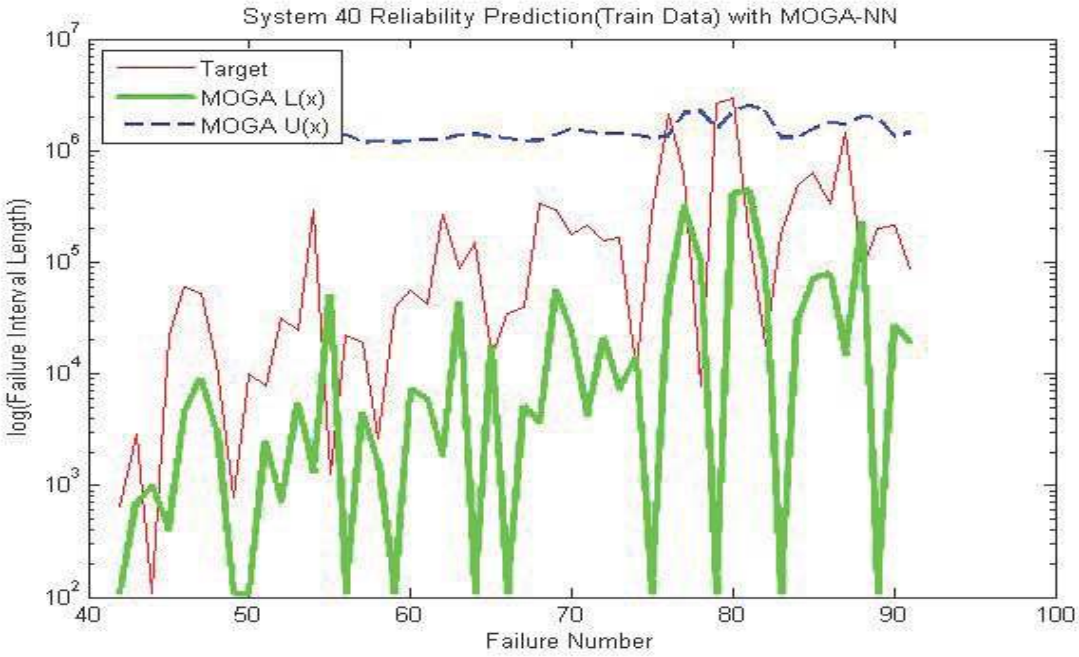


FIGURE 3.79: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 40)[MOGA-NN (Train Data)

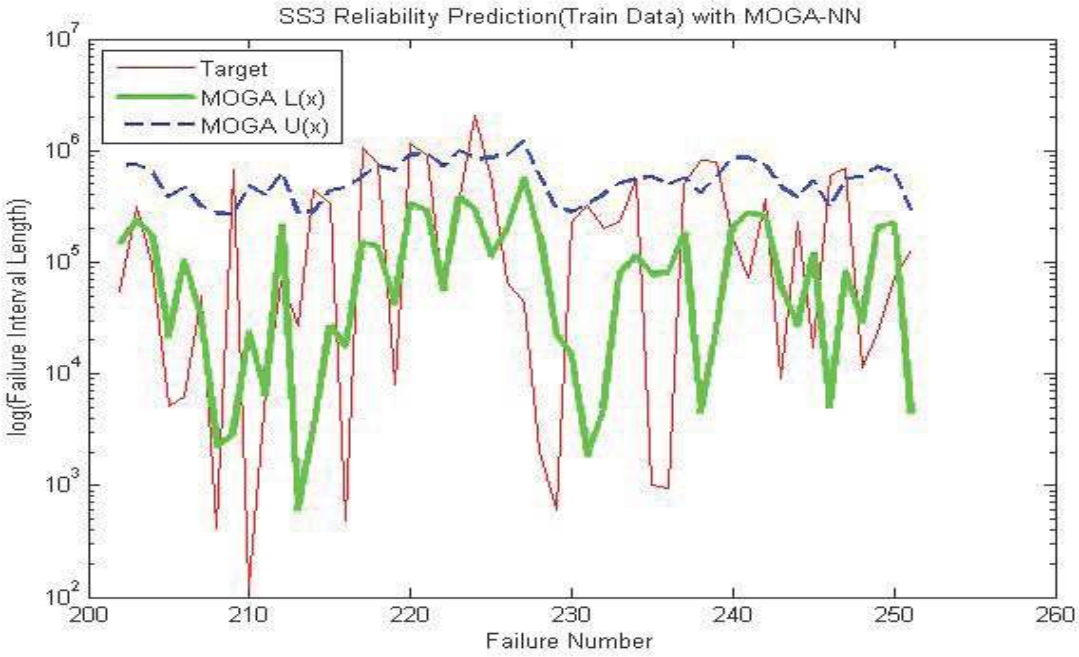


FIGURE 3.80: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System SS3)[MOGA-NN (Train Data)

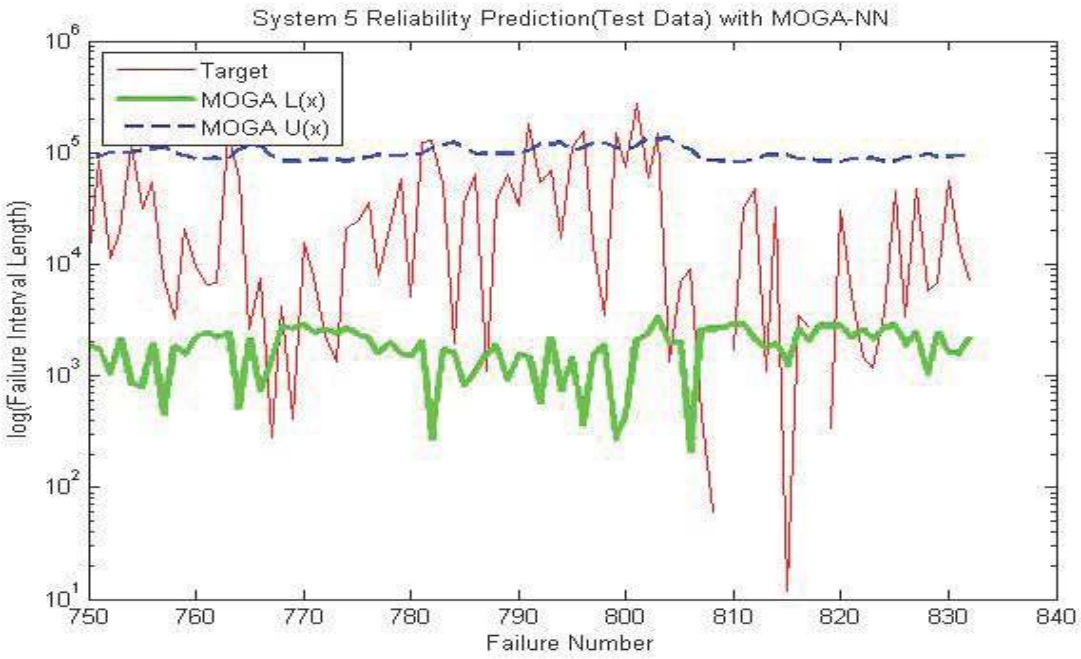


FIGURE 3.81: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 5)[MOGA-NN (Test Data)]

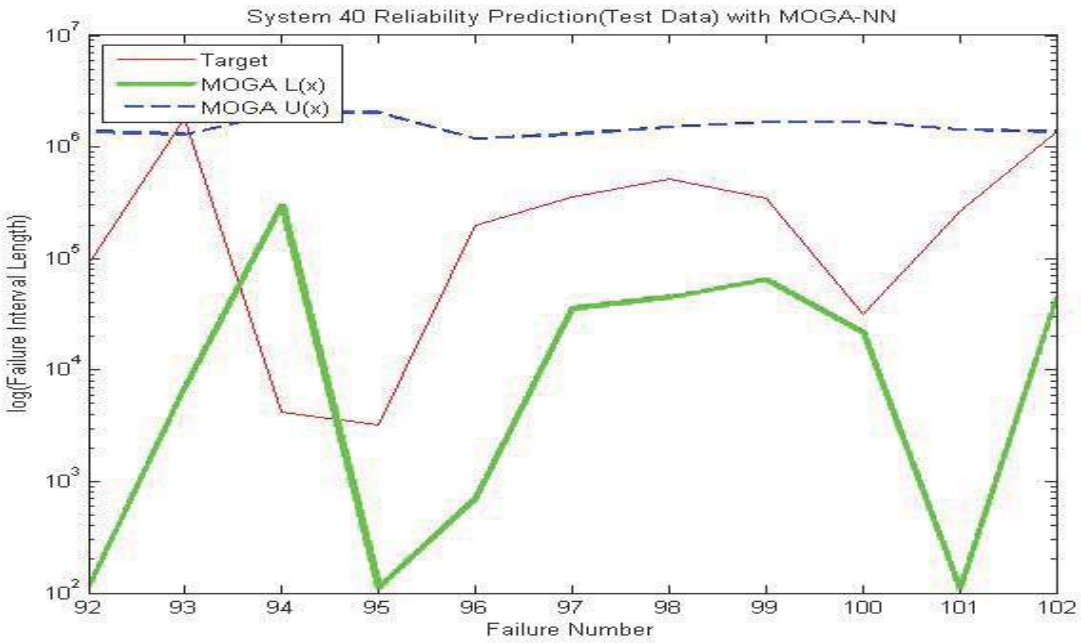


FIGURE 3.82: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 40)[MOGA-NN (Test Data)]

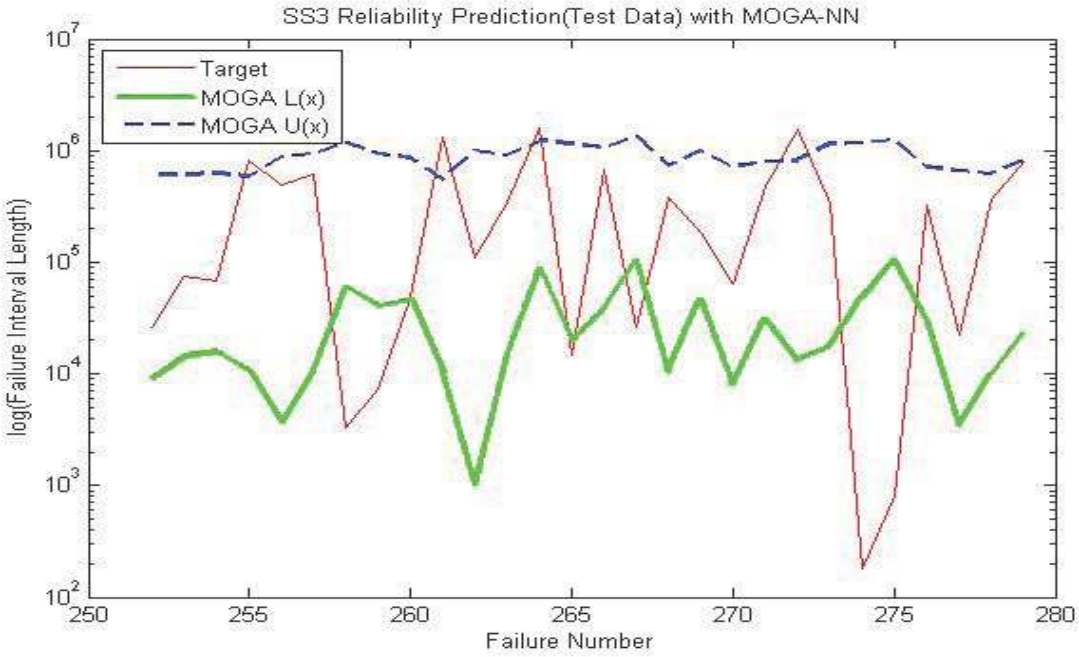


FIGURE 3.83: Plot of Target series, Lower bound L(X) and Upper bound U(X).
(System SS3)[MOGA-NN (Test Data)]

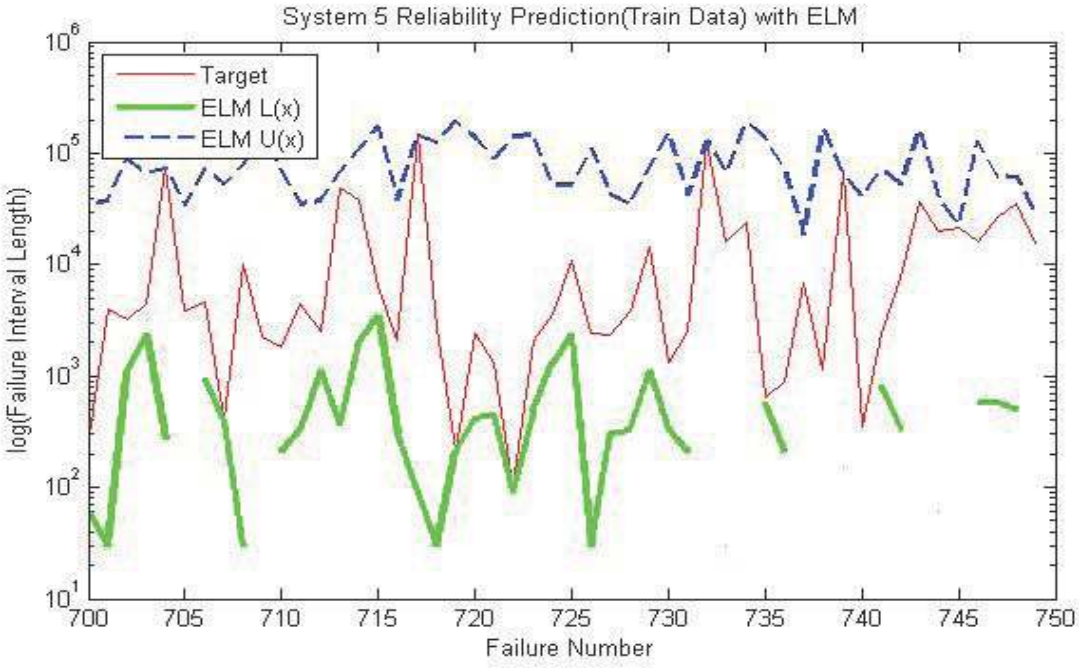


FIGURE 3.84: Plot of Target series, Lower bound L(X) and Upper bound U(X).
(System 5)[ELM (Train Data)]

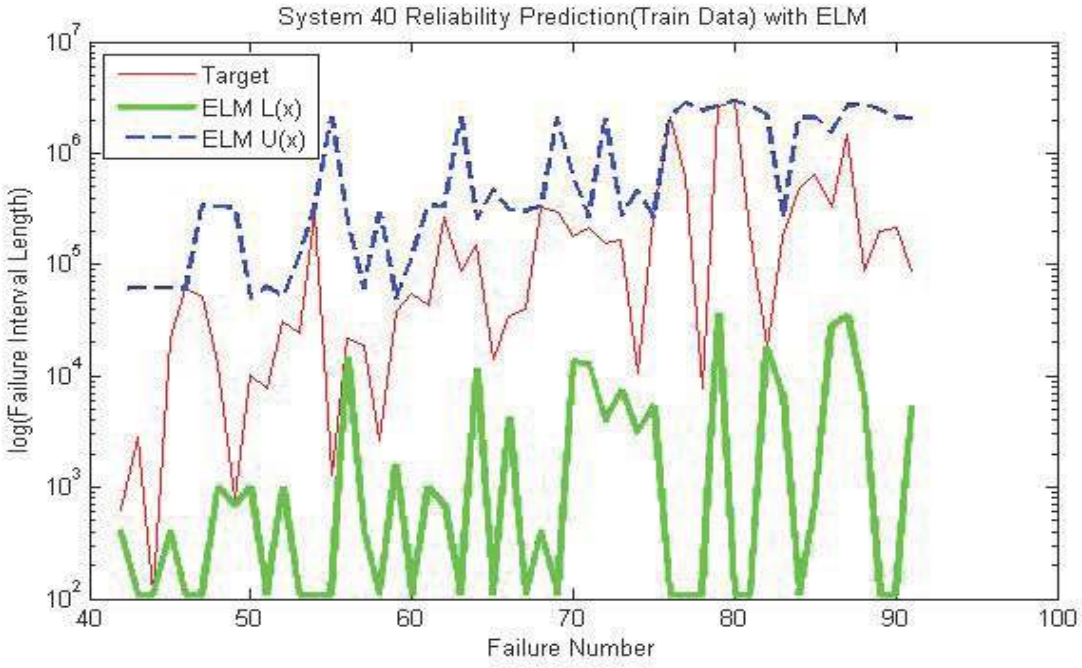


FIGURE 3.85: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 40)[ELM (Train Data)]

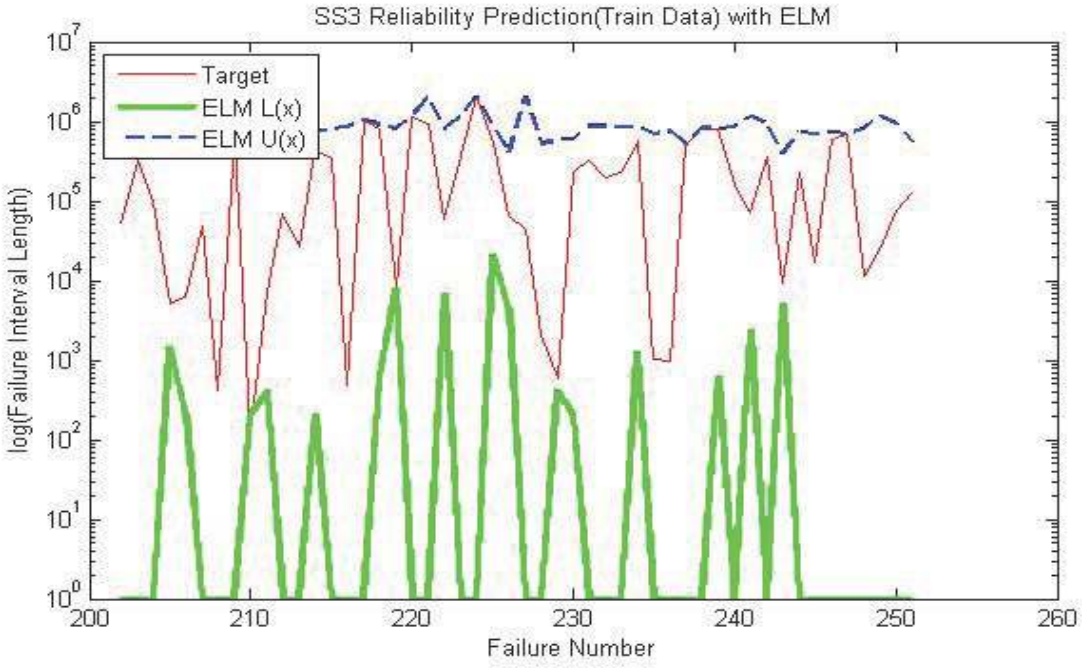


FIGURE 3.86: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System SS3)[ELM (Train Data)]

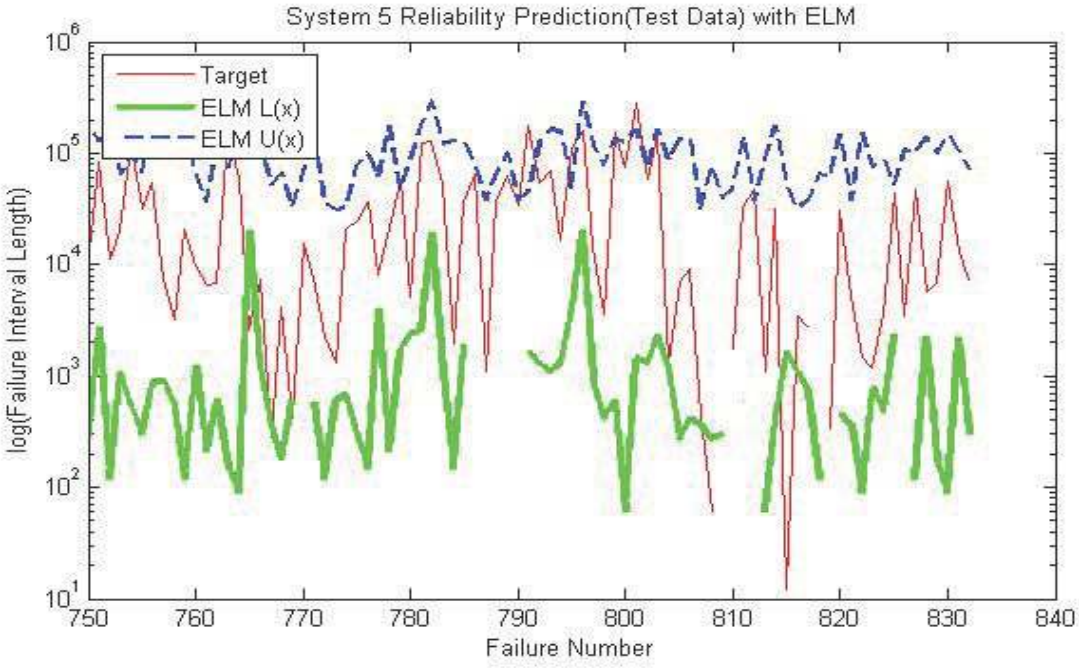


FIGURE 3.87: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 5)[ELM (Test Data)]

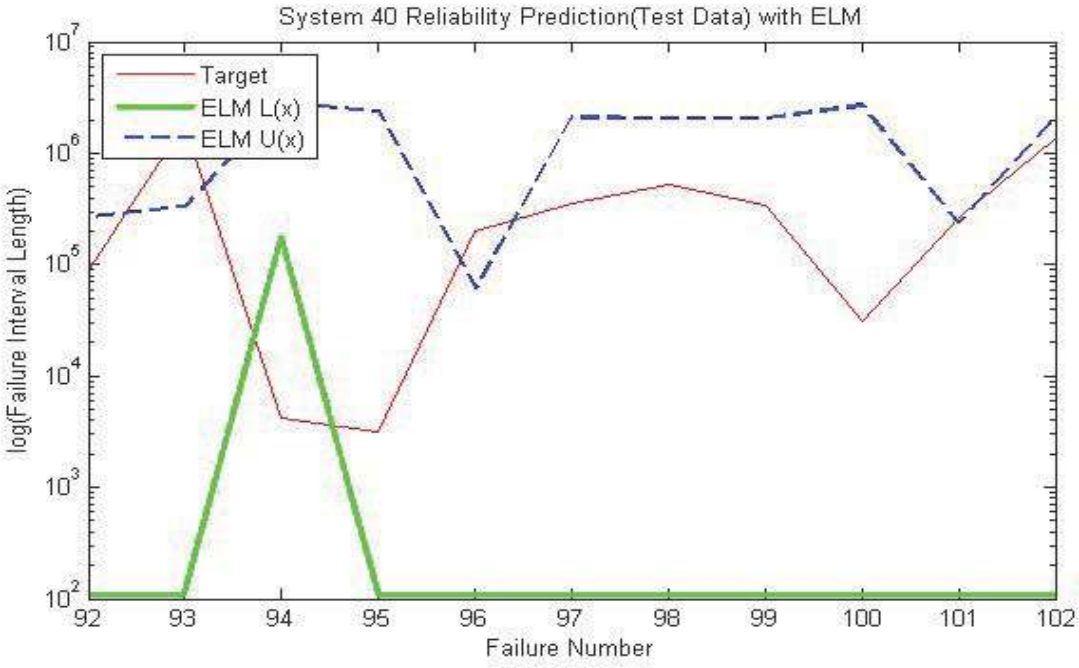


FIGURE 3.88: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System 40)[ELM (Test Data)]

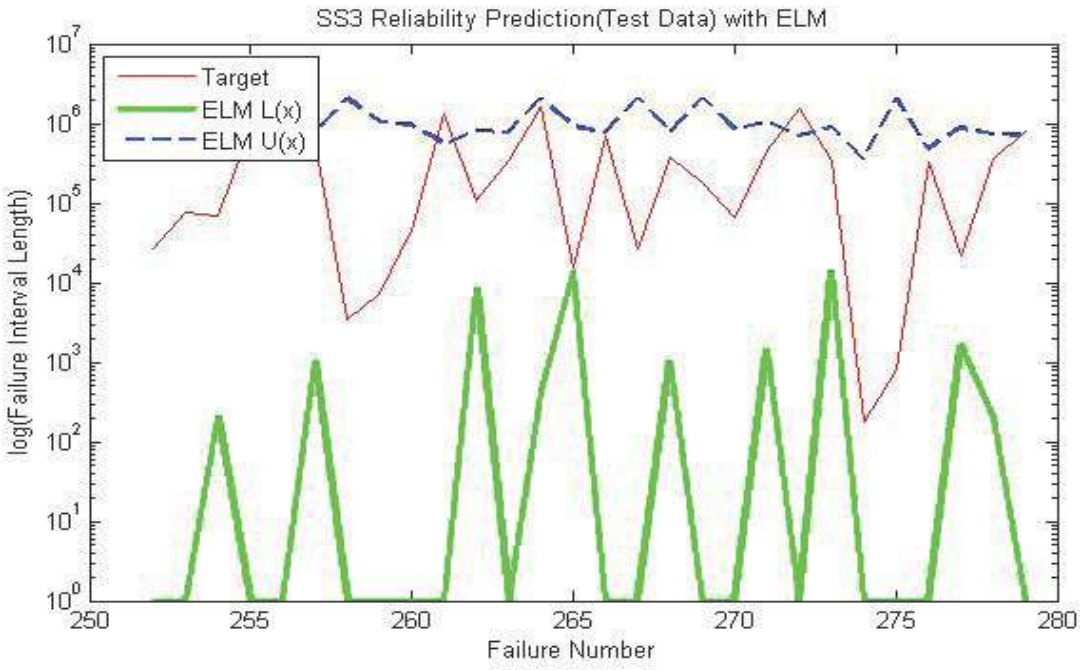


FIGURE 3.89: Plot of Target series, Lower bound L(X) and Upper bound U(X). (System SS3)[ELM (Test Data)]

both processes are independent, the regression task can be performed freely. The PICP and NMPIW care calculated as defined earlier.

For this model, we have also plotted the graphs for showing the Target series, Lower bound L(X), and Upper bound U(X). We have taken the log values for proper visualization of the graphs. Plots of Target series for the trained data for the three software systems is shown in Figure 3.84, 3.85,3.86 . The plots of Target series for the test data for the three software systems is shown in Figure 3.87, 3.88, 3.89.

From the Figures, we interpreted that ELM is also a good predictor as MOGA - NN. We also observed the target series being completely captured by the intervals. The PICP and NMPIW values for the three TBF series of corresponding software system are presented in Table 3.48, 3.49 and 3.50 (For Test Data). We have presented four best solutions as given by ELM+KNN Model. From the table, we observe that the PICP values for ELM + Nearest Neighborhood modelling are more accurate than MOGA-NN also. We also observe

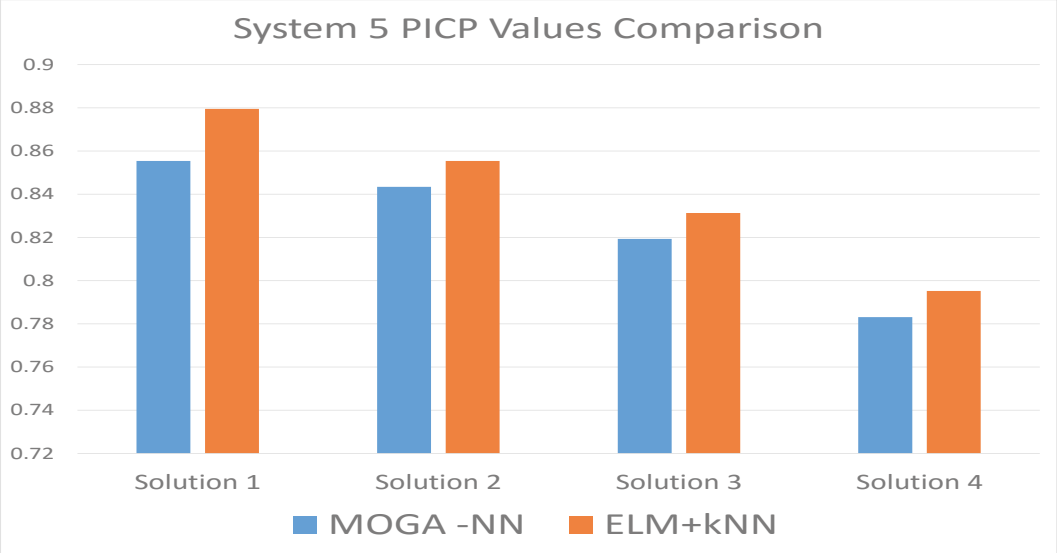


FIGURE 3.90: Comparison of PICP Values Between MOGA-NN and ELM +KNN[System 5]

that ELM gives better accuracy for TBF series with more number of data values.There is a significant difference in PICP values given by ELM + KNN and MOGA -NN model. The PICP of the former model significantly outperforms the MOGA -NN in case of System 5 and System SS3. ELM is also computationally less expensive than MOGA-NN. We have given a bar chart representation of PICP comparison between MOGA-NN and ELM for TBF series of the three software system respectively in Figure 3.90, Figure 3.91 and Figure 3.92.

3.24.2 External Validity Testing

External Validity: External validity testing is used for checking the performance of model across different software applications. It is required for making a generalizable model. In our case, we have also done external validity testing. We have trained the TBF series of System SS3 (90% Training Data) with MOGA-NN and tested it for TBF Series (10%

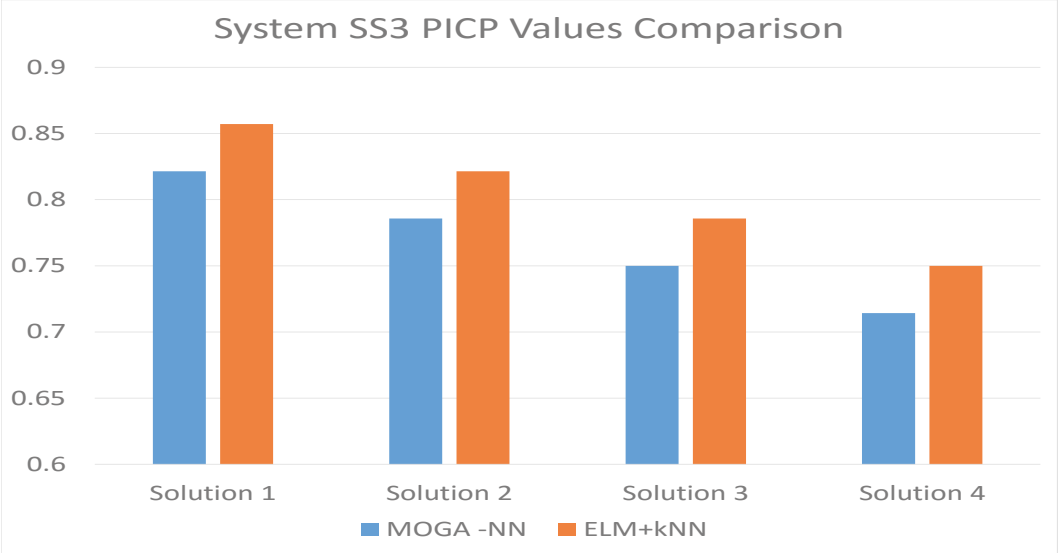


FIGURE 3.91: Comparison of PICP Values Between MOGA-NN and ELM +KNN[System SS3]

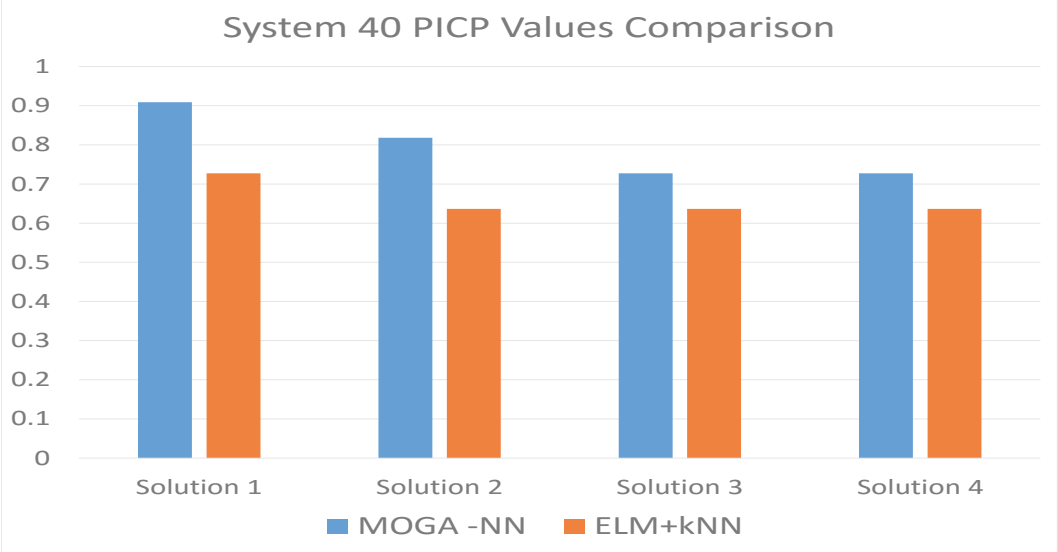


FIGURE 3.92: Comparison of PICP Values Between MOGA-NN and ELM +KNN[System 40]

TABLE 3.51: External Validity Testing (PICP, NMPIW)

Trained System	Tested System	MOGA - NN	ELM + KNN
System - 5	System - SS3	(0.7143, 0.4910)	(0.7500, 0.4152)
System - SS3	System - 5	(0.8916, 0.4495)	(0.8675, 0.3768)

Test Data) of the System 5 and vice versa. Similarly, we also apply ELM + Nearest Neighborhood modelling for training the TBF series of System SS3 (90% Training Data) and tested it on TBF Series (10% Test Data) of the System 5 and vice versa. The PICP, NMPIW values as produced by the models is presented in the Table 3.51. From the result, we observe that PICP values as given by both the models are good. So, these two models can be considered as generalized models for software reliability prediction. In this work, we have two advanced machine learning approaches for estimating the prediction intervals of TBF series of three different software systems. We found ELM + KNN as a better predictor than MOGA-NN. Another advantage is the use of interval based approach, which can eliminate the problem of associated errors due to uncertainties in the model parameters and noise in the input data. The models are both validated internally and externally. After testing the external validity of the models, we found that both the models have good generalization capability.

3.25 Conclusion

In this work, we have applied time series modelling for reliability prediction based on TBF pattern. From the results, we observed Hybrid ARIMA (ARIMA +ANN) as a good predictor of software reliability. An interval-based approach using MOGA –NN, and ELM, eliminated the problem of associated errors due to uncertainties in the model parameters and noise in the input data. Accurate prediction of software reliability will decrease the risk of failures which in turn improve the software quality.

Conclusion

The thesis presents a systematic approach to identify the temporal patterns that exist in various forms like Bugs, Clones, Failure Intervals, etc. across different versions of a software application. The thesis also used advanced time series analysis for modelling these increasing and decreasing patterns of a particular software characteristic (Bugs, Clones Failure Intervals) and also to give effective feedback to software developers and testing team in advance. This will reduce the effort invested in testing and maintenance. This will also help software managers to decide on resource allocation and effort investments. Predicting failure intervals in advance will also improve the software reliability if timely corrective measures are taken by the developers.

Chapter 2 presents a comprehensive survey of various models on defect prediction, clone prediction, and software reliability prediction. The chapter presents a description of different tools and techniques used for modelling the aforementioned software characteristics. It also presents a brief description of the result and also reports the remarks and suggestion by the authors.

Chapter 3 presents time series approach to predict the temporal bug numbers and patterns across different versions of a software application. We have applied advanced time series modelling techniques to model the temporal patterns in the bug numbers. Advance knowledge about bug numbers will help the software managers to decide on resource allocation and effort investments. The developers will be aware of the number of bugs in advance

and can take effective steps to reduce the number of bugs in the new version. The end-user can decide on adopting a particular software application among a variety of applications by knowing the bug growth patterns of the particular software application.

Chapter 4 presents a systematic approach for modelling the evolving clones across different versions of an open-source software application. It is observed that MOGA-NN based approach most promising and efficient in predicting clone evolution. Clone detection is useful for reducing the Corrective Maintenance[203] and Preventive Maintenance [200] which involves modification of code content to solve and prevent problems in the software respectively. Because if we can identify and detect the cloned areas, the defect in all the similar code fragments can be resolved at once. The software clone evolution prediction is immensely helpful in Perfective Maintenance[200], [23] and Adaptive Maintenance [200], [23] because the effort required to evolve a software is also dependent on the amount of cloned contents in the software.

Chapter 5 presents an application of time series modelling for reliability prediction based on TBF pattern. From the results, it is observed Hybrid ARIMA (ARIMA +ANN) as a good predictor of software reliability. An interval-based approach using MOGA –NN, and ELM, eliminated the problem of associated errors due to uncertainties in the model parameters and noise in the input data. Accurate prediction of software reliability will decrease the risk of failures which in turn improve the software quality.

In summary, the thesis used time series approach to improve Software Bug Prediction, Clone Evolution Prediction, and Software Reliability Prediction. Advanced Time Series Modeling using Statistical and Machine Learning Approaches have been used for modelling the temporal patterns. The dataset used are obtained from Open Source Software Repository and also Closed Source Software Repository. The results are evaluated using Standardized Evaluation Techniques and compared against other reported methods.

3.26 Future Scope of This Thesis

In the future, advanced multivariate time series techniques will be applied for modelling clones, bugs, and failure simultaneously. The objective is to find the inter-relationship between these three important characteristics and to identify the trend pattern. This will give an essential feedback to developers for improving the coding and to built software with least bugs and failures. This will not only improve software reliability but also reduce the effort invested in software testing and maintenance.

The future objective is to identify other evolving software characteristics and to model them using time series technique. This modelling approach is not only cost-effective but also does not require access to an internal characteristic of the software application. Earlier prediction of software characteristics will help the software manager for distribution of resource and project staff efficiently.

The future work also targets on applying an ensemble approach for modelling the evolving software characteristics. The ensemble model is a combination of both temporal pattern and some easily accessible software characteristic for improving the accuracy of the models. The ensemble models will be validated on different open and closed source software applications.