Software quality studies using analytical metric analysis

CECILIA RODRÍGUEZ MARTÍNEZ

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Software quality studies using analytical metric analysis

Cecilia Rodríguez Martínez

3 April 2013

Master Thesis

Industrial Supervisor: Hamidreza Yazdani
KTH Academic Supervisor: Gerald Q. Maguire Jr.
Presenter: Eloy Anguiano

Stockholm,Sweden

Abstract

Today engineering companies expend a large amount of resources on the detection and correction of the bugs (defects) in their software. These bugs are usually due to errors and mistakes made by programmers while writing the code or writing the specifications. No tool is able to detect all of these bugs. Some of these bugs remain undetected despite testing of the code. For these reasons, many researchers have tried to find indicators in the software's source codes that can be used to predict the presence of bugs.

Every bug in the source code is a potentially failure of the program to perform as expected. Therefore, programs are tested with many different cases in an attempt to cover all the possible paths through the program to detect all of these bugs. Early prediction of bugs informs the programmers about the location of the bugs in the code. Thus, programmers can more carefully test the more error prone files, and thus save a lot of time by not testing error free files.

This thesis project created a tool that is able to predict error prone source code written in C++. In order to achieve this, we have utilized one predictor which has been extremely well studied: software metrics. Many studies have demonstrated that there is a relationship between software metrics and the presence of bugs. In this project a Neuro-Fuzzy hybrid model based on Fuzzy c-means and Radial Basis Neural Network has been used. The efficiency of the model has been tested in a software project at Ericsson. Testing of this model proved that the program does not achieve high accuracy due to the lack of independent samples in the data set. However, experiments did show that classification models provide better predictions than regression models. The thesis concluded by suggesting future work that could improve the performance of this program.

Keywords: Bugs, Fuzzy c-means, Neuro-fuzzy hybrid model, Radial basis function neural network, Software metrics

Sammanfattning

Idag spenderar ingenjörsföretag en stor mängd resurser på att upptäcka och korrigera buggar (fel) i sin mjukvara. Det är oftast programmerare som inför dessa buggar på grund av fel och misstag som uppkommer när de skriver koden eller specifikationerna. Inget verktyg kan detektera alla dessa buggar. Några av buggarna förblir oupptäckta trots testning av koden. Av dessa skäl har många forskare försökt hitta indikatorer i programvarans källkod som kan användas för att förutsäga förekomsten av buggar.

Varje fel i källkoden är ett potentiellt misslyckande som gör att applikationen inte fungerar som förväntat. För att hitta buggarna testas koden med många olika testfall för att försöka täcka alla möjliga kombinationer och fall. Förutsägelse av buggar informerar programmerarna om var i koden buggarna finns. Således kan programmerarna mer noggrant testa felbenägna filer och därmed spara mycket tid genom att inte behöva testa felfria filer.

Detta examensarbete har skapat ett verktyg som kan förutsäga felbenägen källkod skriven i C ++. För att uppnå detta har vi utnyttjat en välkänd metod som heter Software Metrics. Många studier har visat att det finns ett samband mellan Software Metrics och förekomsten av buggar. I detta projekt har en Neuro-Fuzzy hybridmodell baserad på Fuzzy c-means och Radial Basis Neural Network använts. Effektiviteten av modellen har testats i ett mjukvaruprojekt på Ericsson. Testning av denna modell visade att programmet inte Uppnå hög noggrannhet på grund av bristen av oberoende urval i datauppsättningen. Men gjordt experiment visade att klassificering modeller ger bättre förutsägelser än regressionsmodeller. Exjobbet avslutade genom att föreslå framtida arbetet som skulle kunna förbättra detta program.

Keywords: Buggar, Fuzzy c-medel, Neuro-Fuzzy hybridmodell, Radial basis funktion neurala nätverk, programvarastatistik

Resumen

Actualmente las empresas de ingeniería derivan una gran cantidad de recursos a la detección y corrección de errores en sus códigos software. Estos errores se deben generalmente a los errores cometidos por los desarrolladores cuando escriben el código o sus especificaciones. No hay ninguna herramienta capaz de detectar todos estos errores y algunos de ellos pasan desapercibidos tras el proceso de pruebas. Por esta razón, numerosas investigaciones han intentado encontrar indicadores en los códigos fuente del software que puedan ser utilizados para detectar la presencia de errores.

Cada error en un código fuente es un error potencial en el funcionamiento del programa, por ello los programas son sometidos a exhaustivas pruebas que cubren (o intentan cubrir) todos los posibles caminos del programa para detectar todos sus errores. La temprana localización de errores informa a los programadores dedicados a la realización de estas pruebas sobre la ubicación de estos errores en el código. Así, los programadores pueden probar con más cuidado los archivos más propensos a tener errores dejando a un lado los archivos libres de error.

En este proyecto se ha creado una herramienta capaz de predecir código software propenso a errores escrito en C++. Para ello, en este proyecto se ha utilizado un indicador que ha sido cuidadosamente estudiado y ha demostrado su relación con la presencia de errores: las métricas del software. En este proyecto un modelo híbrido neuro-disfuso basado en Fuzzy c-means y en redes neuronales de función de base radial ha sido utilizado. La eficacia de este modelo ha sido probada en un proyecto software de Ericsson. Como resultado se ha comprobado que el modelo no alcanza una alta precisión debido a la falta de muestras independientes en el conjunto de datos y los experimentos han mostrado que los modelos de clasificación proporcionan mejores predicciones que los modelos de regresión. El proyecto concluye sugiriendo trabajo que mejoraría el funcionamiento del programa en el futuro.

Keywords: Error, Fuzzy c-means, modelo híbrido neuro-difuso, Red neuronal de función de base radial, métricas del software

Acknowledgements

First of all, I would like to thank my Ericsson's industrial supervisor Hamidreza Yazdani for trusting me and give me this great opportunity. I am also grateful to Gerald Maguire for being my examiner and for being always available to help, and to Eloy Anguiano for being my presenter making this possible.

I want to thank my friends from my home university because they pushed me here making my life amazing at the EPS, specially to Guillermo Gálvez for being the best friend and colleague during these five years. You guys brought me here!

I want to dedicate my PFC to Mónica Estevez and my girls for being always proud of me and being always there when I need it. I also want to acknowledge all my friends from Tres Cantos for supporting me and share their lives with me even in the distance. You simply make me happy.

I really want to thank my brother, Juan Rodríguez, for helping me and encouraging me every time I doubt, and to my family for helping me to achieve all my goals in my life.

I want to make a special mention to Jesus Paredes for his continuous support and confidence, to Nacho Mulas for listening to me every time I need it, to Alberto Fernandez, and to all the friends I met in Stockholm, you made warm this cold country!

Finally I also want to thank the people from COM for making me feel one more of the team, specially to Belén Valcarce-Pérez, Péter Dimitrov and the foosball team; I promise "I will do something".

Contents

List of Figures

List of Tables

List of Acronyms

ANN	Artificial Neural Network
BPNN	Back Propagation Neural Network
CBO	Coupling Between Objects
CCCC	C and C++ Code Counter
CK	Chidember and Kemerer
CS	Compact-Separate
DIT	Depth of Inheritance Tree
FCM	Fuzzy c-means
FLD	Fisher's Linear Discriminant
FS	Fukuyama.sugeno
GK	Gustafson-Kessel algorithm
HTML	HyperText Markup Language
IFc	Information Flow complexity
JMT	Java Measurement Tool
LCOM	Lack of COhesion in Mthods
LOC	Lines Of Code
NN	Neural network
NOA	Number Of Attributes
NOC	Number Of Children
NOM	Number Of Modules
OO	Object Oriented
PC	Partition.coefficient

PE	Partition.entropy
PHP	Hypertext Preprocessor
PL	Procedural language
RBF	Radial Basis Function
RBFNN	Radial Basis Function Neural Network
RFC	Response For a Class
RMS	Root Mean Square
ROC	Receiving Operating Characteristic
RSM	Resource Standard Metrics
SNNS	Stuttgart Neural Network Simulator
SQL	Structured Query Language
WMC	Weighted Methods per Class
XB	Xie.beni
XML	Extendible Markup Language

Chapter 1

Introduction

These days, the telecommunication industry is an open market accessible to all enterprises. In a market where many companies compete to be the market leaders companies need to offer a high level of quality and reliability in their products in order to maintain their market position. Delivering high quality and highly reliable software is a mandatory goal for software development companies, as this reduces the amount of resources needed to support the software after it has been delivered and this quality helps the firm to gain a larger portion of the market. Therefore, managers involved in the development and maintenance of software now focus on enhancing their software development process [10].

This software development process occurs whenever a company wants to create a new software product or improve an existing product. This development process consists, of at least, the following steps: specification, design, implementation, validation, documentation, delivery, and maintenance [31]. The waterfall model of this process is shown in Figure 1.1. Due to simplicity the waterfall model is explained but the scope of this thesis project are also applicable to different software development models. These steps are:

Specification The customers determine the requirements and the purpose of the software.

Design The managers of the project together with the programmers define the methodology and design of the product.

Development The programmers develop the new software.

Testing Here the programs are tested looking for any bug that could came up with present or future failures in the system. This stage is usually the most resource consuming.

Documentation and Maintenance In this step the programs are documented and maintenance starts. In the maintenance process programmers supervise

the proper operation of the product by fixing the bugs or improving the product's performance when needed.

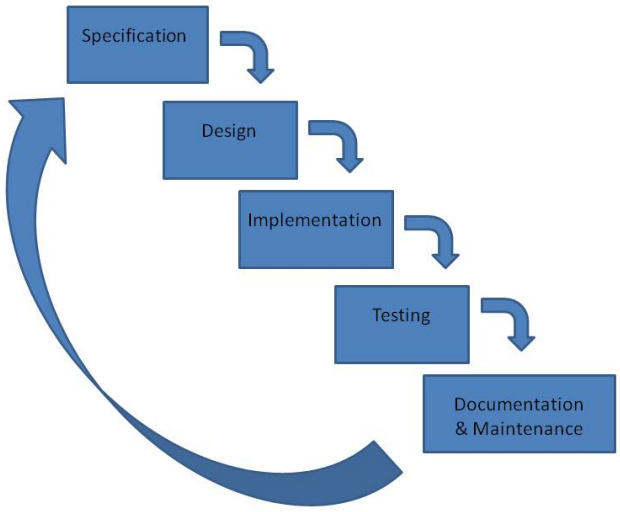

Figure 1.1. A software development process according to the waterfall model

1.1. Problem Statement

The testing stage is designed to ensure the quality of the software products, but it is the most resource consuming step in terms of time, effort, and costs. This testing activity represents 50 to 70 percent of the total costs of a project [21].

Human mistakes and human-computer misunderstandings in programming are common. When errors occur during the implementation stage they are typically not discovered until the code is tested. These errors force the developers to implement new code or to rewrite some of the code. Moreover, this additional code introduces its own bugs, hence the process is an exponential (although hopefully a dampened exponential) process.

The effort required to correct mistakes introduced during the implementation stage is less than the effort required to correct these errors during the testing stage. The worst scenario occurs when the end users find the errors at execution time. In many cases the problems caused by bugs cannot be solved and a new version of the software is necessary to remove one or more bugs. In each of these scenarios the company spends a lot of resources to resolve the issues caused by bugs. The resources that are expended include:

Capital Many different tests are needed to find the errors in the code. Developing and maintaining these tests also costs money.

Time Locating bugs usually takes a lot of time.

Effort Locating and correcting bugs is a hard task.

It is easy to see that the earlier a bug is detected and corrected, the lower the total cost. Consequently, many companies have decided to invest in new ways to detect and correct bugs as early in the overall process as possible [18], for example by adopting agile programming, deriving the program from formal specification, and other techniques.

1.2. Goals

The main goal of this thesis project is to improve the quality of software. Improving the quality of this software implies a reduction in development and maintenance costs, hence a reduction in the costs of the whole company. Higher quality software might also increase customer satisfaction and confidence in the company and its products [18]. To carry out this improvement this thesis project will focus on predicting the presence of bugs in the early stages of the development process. This prediction is based on computing software metrics over the software's source code.

With regard to studies of rapid bug prediction, this project will contribute by showing that there is a relationship between software metrics and the presence of bugs and that the presence of bugs can be estimated by using a neuro-fuzzy hybrid model.

1.3. Scope

The scope of this project is a tool to predict with high accuracy the existence of bugs in code. This tool will facilitate the detection of the most error prone modules in a programs *while* the software is being developed. Furthermore, this tool will allow users to:

- correct bugs during the development stage that would otherwise not be found until the testing stage (this will be done by refactoring of the source code);
- be aware of which modules have to be tested carefully during the testing process; and
- learn (by suitable training) a programming style which is less error prone. This can be done by learning which programming styles have good feedback

from the tool.

1.4. Target Audience

This thesis may be interesting for those programmers who want to improve the quality of their code. It should be interesting for all enterprises, especially those that develop software products. The project also contributes to research on software metrics and their relationship with bugs and to the investigations of several different (and useful) regression models.

1.5. Methodology

This thesis project started with an analysis of the problem statement (given in Section 1.1). Software metrics were previously identified to be predictors, thus this thesis project considered a number of different metrics and their contribution to bug prediction. A selection of the best means to compute these metrics was also done. Subsequently several different regression models were examined in order to find a model that successfully relates these metrics with bugs. Once the objective was clear, implementation of a prototype started.

A neuro-fuzzy hybrid model was chosen because, as the literature shows, this model has good capabilities together with regression models. Hybrid models are an improvement over neural network models which have no initial parameters defined. After the neuro-fuzzy hybrid model was chosed the following steps were taken:

1. Extraction of the software metrics of the source codes with the CCCC program [23].
2. Extraction of the number of known bugs for this source code from a repository of trouble reports.
3. Clustering of the metrics with the R program [26].
4. Implementation, training, and testing of a neural network in R.
5. Creation of a tool that would compute the software metrics and estimate the number of bugs in source code that is input to this tool.

1.6. Structure

Chapter 1 introduces the objectives and the motivation for this thesis project.

Chapter 2 gives the background necessary for reading this thesis and the knowledge needed to understand it. Chapter 2 starts with some definitions of object-oriented programming for those who are not familiar with this terminology. The chapter continues with an explanation of the most widely used software metrics. The chapter concludes with a description of some of the most widely used regression models, but focuses on neuro-fuzzy hybrid models, on neural networks, and clustering techniques.

Chapter 3 gives an overview of the model followed to create the tool.
Chapter 3 begins with a brief explanation of the desired characteristics of the tool and continues with the selection of the programs, techniques, and models used in this thesis project to relate software metrics and bugs.

Chapter 4 describes in detail the implementation of the neuro-fuzzy hybrid model to finalize analyzing its results and verifying its quality.

Chapter 5 explains the functionalities of the final tool and the results of using this tool.

Chapter 6 presents some conclusions.

Chapter 7 proposes future work which can be done to improve the tool.

Furthermore, appendixes A and B contain further information about the tool. To conclude appendixes C and D contain the introduction and conclusions of this master thesis in spanish.

Chapter 2

Background

This chapter presents the background of the thesis. It is divided into two sections. Section 2.1 gives the necessary knowledge to understand the selected software metrics and how they are estimated. Section 2.2 gives an overview of the different regression models that were considered, subsequently focusing on the neuro-fuzzy model. A number of different clustering techniques and artificial neural network models are explained.

2.1. Software Metrics

Software metrics are quantitative measures of the attributes of a program. There are three kinds of software metrics: procedure metrics, project metrics, and product metrics [16].

- Procedure metrics measure the resources (time and cost) that a program development effort will take. They are useful for the administration and management of the project.
- Project metrics give information about the actual situation of the project. These metrics include costs, effort, risks, and quality. These are used to improve the development process of the project.
- Product metrics assess quality information about the program. These metrics focus on reliability, maintainability, complexity, and reusability of all or part of the software developed for the program.

The reliability of these software metrics as predictors bugs has been studied and tested by many researchers [6, 9, 3], who have used different regression models applied to different languages. All of these researchers have claimed software metrics to have good capabilities as indicators of bugs.

2.1.1. Basic Concepts

We will use a number of basic concepts in this thesis. Below is a short description of each of these concepts for the reader's reference.

OOP (Object-Oriented Programming) OOP is a programming style based on the use of objects. Thus, programs are made of objects that are manipulated to perform a specific task.

Object An object is an instance of a class which has been defined with some properties, specifically: attributes and methods.

Class A class is the template used to create objects. Different objects of the same class have a similar structure.

Method A method is subroutine of a class that performs an operation.

Attribute An attribute is a member variable of a class.

Inheritance Inheritance is the process where by an object acquires some properties and methods of another class.

Children A child or subclass is the class that inherits properties and methods in an inheritance process.

Ancestor An ancestor or superclass is the class from which another class inherits properties and methods in an inheritance process.

2.1.2. Definition of Metrics

Currently, the goal of software developers is to offer the best quality in their programs without increasing the amount of resources needed for the development of this software. For this reason in this study we will focus on product metrics. These metrics are summarized in Table 2.1. In despite of the metric definitions given in the Table 2.1, many researchers claimed that software metrics lacked mathematical strictness and other desirable properties [33, 35]. Therefore, they considered the estimates of these metrics to be unreliable. Consequently, in 1994, Chidamber and Kemerer described in [6] six metrics which are now called 'CK metrics'. These six metrics have been successfully correlated with the likelihood of error in software in many investigation (in [30] for Java language and in [3] for C++ language).

Table 2.1. Object Oriented Metrics

Name/ Acronym	Description
Lines of Code (LOC)	Total number of lines of code. LOC may or may not take into consideration blank and comment lines. LOC is related with the size of the source code subject to bugs. The reliability of this metric has always been questioned because although the probability of mistakes increases with the opportunity to make them, it has been observed that when the complexity of the code is low, the number of bugs is also small even if there are many lines of code.
Number of attributes (NOA)	The number of public, private, and inherited attributes.
Number of methods (NOM)	The number of public, private, and inherited methods.
McCabe's cyclomatic complexity	This metric can be defined in a given class as the sum of the complexity of the methods defined for the class. It estimates the complexity as the number of linearly independent paths that can be followed in the class. In 1976 Thomas J.McCabe introduced this new metric in [25]. When the value of McCabe's metric exceeds some threshold for one module (according to [11] a value higher than 10), then the module should be split into smaller modules.
Fan in/ Fanout	These two metrics concern the number of classes which reference the class/the number of classes referenced by the class. Fan in and Fanout describe the relationship between a given class and its environment. These metrics are related to the flow structure of the system. Sallie Henry and Dennis Kafura introduced these new metrics in [15] in 1981.
Information flow complexity	This metric is a measure of the complexity of the code in terms of its fan in and its fan out. This metric is computed using equation 2.1. $$IFc = (Fanin * Fanout)^2 \quad (2.1)$$ The greater the complexity of the code the greater the probability of there being an error. Sallie Henry and Dennis Kafura introduced this new metric in [15] in 1981.

The CK metrics are described in Table 2.2 for a given class.

Table 2.2. Object Oriented CK Metrics

Name/ Acronym	**Description**
Weighted Methods Per Class (WMC)	This metric is the sum of the complexity of the methods defined in the class when the complexity value assigned to each method is 1. This metric is computed using equations 2.2 and 2.3. $$WCM = \sum_{i=1}^{n} C_i \quad (2.2)$$ $$\textbf{when } C_i = 1, WCM = n \quad (2.3)$$ This metric gives an approximation of the time and effort needed to design and maintain the class. This metric captures the observation that the more complex the methods of a class, the more error prone this class is.
Depth of Inheritance Tree (DIT)	In an inheritance tree DIT is the maximum length from a node in the tree to the root. High values of DIT means that the classes are reusing the methods of its ancestors, thus these classes were well defined which enabled their reuse. However, a high value of DIT implies that the structure of the code is more complex, and that it is more complicated to test.
Number of Children (NOC)	NOC is the number of direct subclasses which belong to a class. When NOC is high, a change in the class can potentially affect many different classes.
Coupling between object classes (CBO)	One class is coupled with another when it uses the methods or instance variables of the latter. CBO is the number of other classes which are coupled with a specific class. When CBO is high, then one error in the class could affect many classes in the program. CBO is a measure of the independency of the class.

Name/ Acronym	Description
Response For a Class (RFC)	RFC is the number of methods invoked when an object of the class receives a message. The value is computed according to equation 2.4. $$RFC = \|RS\|, \text{ where } RS = \{M\} \cup_{alli} \{R_i\} \quad (2.4)$$ Where M is the set of all methods in the class and Ri is the set of methods invoked by method i. Classes with a low value of RFC are simpler, hence it is easier to debug them. Classes with a high value of RFC have a large and potentially highly distributed effect, hence they are harder to debug as the number of messages triggered by one message can expand rapidly.
Lack of Cohesion in Methods (LCOM)	LCOM is the number of methods pairs belonging to the class that do not share instance variables minus the number of those which do. This metric is calculated using equation 2.5. $$LCOM = \begin{cases} \|P\| - \|Q\| & \text{if } \|P\| > \|Q\| \\ 0 & \text{otherwise} \end{cases} \quad (2.5)$$ Where P is the number of methods in the class that do not use the same instance variables (analyzed by pairs) and Q is the number of methods that share them. The larger the value of LCOM is, the lower the cohesion of the class. A high value of LCOM indicates that the class should be divided into subclasses, as its methods realize different objectives (as they represent computations over variables that are increasingly independent with increase LCOM value).

These metrics are only a sample of all the metrics which can be used (and have been used) to analyze the characteristics of source code. These metrics have being used in many studies that have attempted to predict bugs in software. Today these metrics are used by many enterprises to estimate the quality of their programs.

An additional metric called BUGFIXES is based on the results of the datasheet used by Marco D'Ambros and Romain Robbes in 2010. In [9] they described how they created a datasheet for five different systems written in java: Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn, and Apache Lucene. Their goal was to predict the efficiency of different types of metrics applied to different programs. For this prediction, they introduced a new metric called BUGFIXES (see Table 2.3.

Table 2.3. Object Oriented BUGFIXES Metric

Name/Acronym	Description
BUGFIXES	BUGFIXES is the number of the bugs that where fixed in the past for this code. The value for this metric can be extracted from the number of fixed trouble reports reported for the code. In [9] and [24] it was shown that BUGFIXES is correlated with the number of future bugs. Furthermore, the authors state that considering the effort needed to calculate the metrics of a large system a combination of CK and Object Oriented (OO) metrics with BUGFIXES is the best approach to forecast the error-prone behavior. The datasheet is public and is available at http://bug.inf.usi.ch.

2.1.3. Software metrics measurement programs

There are many programs that can be used to estimate the different software metrics. A brief explanation of some of these programs is given below:

Java Measurement Tool(JMT) JMT is a Java application developed by Ingo Patett and improved by Christian Kolbe. It analyzes different classes and their relationships. JMT can be used over files or over whole projects, but only for Java Code. The application can be downloaded from http://jmt.tigris.org/.

C and C++ Code Counter(CCCC) Created by Tim Littlefair, CCCC analyzes C/C++ and Java code. The program can be executed from the shell on both Unix and DOS/Windows family platforms. It generates reports in HTML and XML format. The program can be downloaded from http://cccc.sourceforge.net/.

Eclipse Metrics Plug-in This is an Eclipse plug-in which calculates a set of metrics every time the code is compiled. It advertises the user whenever the value of the metrics exceeds a certain threshold. However, this plug-in only parses source code written in Java language. The information can be exported in XML. The plug-in can be downloaded from http://metrics.sourceforge.net/.

SONAR SONAR was developed by Sonarsource and can be downloaded from http://www.sonarsource.org/. The program itself is written in Java, but has plug-ins for C, C#, PHP, Flex, Natural, PL/SQL, Cobol, and Visual Basic 6. However, it does not compute of the same metrics for all of these languages.

Resource Standard Metrics (RSM. RSM is a commercial tool which performs a quality analysis of code written in C, ANSI C++, C# and Java. RSM runs under both Windows and Linux. It creates reports in HTML and/or XML. This program can be found at http://msquaredtechnoatogies.com/.

SD Metrics SD is a commercial tool that computes metrics of C++, Java, Delphi and Smalltalk and generates HTML and/or XML report. The program is available for both Windows and Unix platforms. Information about the program is available from http://www.sdmetrics.com/.

Table 2.4 shows the metrics computed by each of the above program. Table 2.5 summarizes some of the most important features of these programs.

Table 2.4. Comparison between programs by the metrics they compute

Metric	JMT	CCCC	Eclipse plug-in	Sonar	RSM	SDM
WMC	✓	✓	✓			
DIT	✓	✓	✓	✓	✓	✓
RFC	✓			✓		
NOC	✓	✓	✓	✓	✓	✓
CBO	✓	✓				✓
LCOM			✓	✓		
Cyclomatic complexity		✓	✓	✓	✓	✓
Fan in		✓		✓		✓
Fan out		✓		✓		✓
LOC	✓	✓	✓	✓	✓	✓
NOA	✓				✓	✓
NOM	✓	✓	✓	✓	✓	

Table 2.5. Comparison between programs based upon their languages and license

	JMT	CCCC	Eclipse plug-in	Sonar	RSM	SDM
Language	Java	Java C/C++	Java	Java C/C++*	Java C/C++	Java C/C++
Free license	✓	✓	✓	✓		

* The original program does not implement a C/C++ parser, but there are plug-ins for the program that allow the users to compute metrics for those languages

2.2. Regression Models

A regression model can be used to find the relationship between metrics and bugs. Many different regression models have been studied in the literature.

In [27], Ratzinger, et al. used a liner regression algorithm to predict defects. Linear regression models relate the output as a linear equation of the input

attributes. In their study they create a predictor for defect densities by using data mining techniques. This predictor had good results (with a correlation coefficient between predictions and real values greater than 0.7 over 1) in the three software projects that they tested: ArgoUML and the Spring framework (both open source projects with 5,000 and 10,000 classes respectively), and one commercial system with more than 8,600 classes. All of these three programs were written in Java.

Today research about metrics and bugs no longer utilizes linear models, as recent studies has shown that linear models do not provide good performance due to the complexity of the relation between metrics and bugs [18, 8]. Today machine learning based models are widely used. These models are based on the learning capacity of algorithms which are trained with patterns. The most widely used learning algorithms to implement bug prediction models are decision trees, classifiers, and neural networks.

In 2012, Singh and Verma [30] utilized two different models of machine learning: J48 (a decision tree) and naïve Bayes algorithm (classifiers). With these models they got an accuracy of 98.15% and 95.58% (respectively) using the CK metrics as predictors. Furthermore, Ahsan and Wotawa [1] used regression models and decision trees to predict bugs in C language programs. In their work decision trees seemed to give better predictions (with an accuracy of 97%).

However, in [29] Gray and MacDonell claim that neuro-fuzzy hybrids are the best regression model in comparison with neural networks and fuzzy logic models. Gan and Harris use topology clustering techniques to define the neural networks and to give the initial parameters [13]. A similar model was used by Jin, et al. [18] where a Fuzzy c-means (FCM) clustering with a Fisher's Linear Discriminant (FLD) method is used to define the initial parameters for a Radial Basis Function (RBF) neural network. This model was used to determine the probability of errors of 70 C++ classes. The model was trained with 106 classes and it showed an accuracy of 90% in comparison with 87.14% for logistic regression.

In 2012, a new model was developed by Couto, et al. [8] where the Granger Causality test was used to establish the relationship. This test was suggested by Clive Granger to predict how the past events occurring in one series (a numerical sequence) influenced events in other series. The model uses the next bivariate auto-regressive model. Couto, et al. predicted with the Granger's Causality test the origin of 64% to 93% of the bugs detected in the systems studied by D'Ambros et al. in their datasheet [9]. The datasheet contains 1041 classes from Eclipse JDT Core, 1924 from Eclipse PDE UI, 444 from Equinox, and and 889 from Lucene with a total of 5028 (known) bugs in the four systems.

2.2.1. Artificial Neural Networks

Artificial Neural Networks (ANN), also called Neural Network (NN), is a mathematical model which aims to learn by training the behavior of a specific system. This model was inspired by the behavior of the brain when learning and this model has been used to establish relationships between a set of inputs and outputs [7, 14].

There are two main neural network architectures. The difference is the connection between the layers. These two architectures are:

- Feed-forward networks. In this architecture the signals advance from the input to the output only in one direction: forward. This architecture is widely used for pattern recognition.
- Feedback networks. These networks have feedback loops so the signals move both forward and backward until they reach an equilibrium. This equilibrium changes every time an input is modified.

A simple example of a neural network is shown in Figure 2.1. It consists of three layers: the input layer fed wth the input data, the hidden layer composed of multiple processing elements working in parallel, and the output layer. All the elements (analogous to neurons) are connected by links of different weights. Figure 2.2 shows one neuron in such a network.

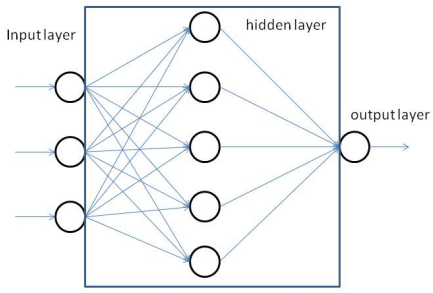

Figure 2.1. An example of an artificial neural network

When a neuron is activated its output contributes to the global output of the system. The activation state is decided by analyzing the result of the activation function over the inputs and their weights. The global output depends on the weight of the different links and these weights are adjusted such that the system gives the

correct outputs. The process responsible for this adjustment in weights is called the learning or training process.

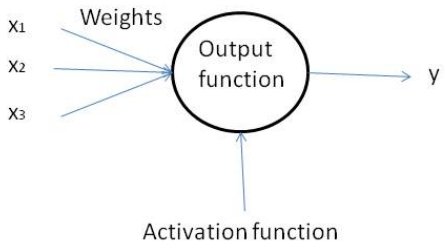

Figure 2.2. A neuron of a generic artificial neural network

There are three types of learning processes:

Supervised learning In supervised learning the system knows the input data and their expected output. Training consists of minimize the error between the current result and the desired output. The error is defined by equation 2.6.

$$E = \frac{1}{2} \sum_{1}^{k} ||y_i - y_i'||^2 \tag{2.6}$$

Unsupervised learning In unsupervised learning the system only has input data. These type of algorithms are commonly used for pattern classification.

Reinforcement learning In reinforcment learning the system knows the input data and which outputs are correct or incorrect.

2.2.1.1. Backpropagation Neural Network

Backpropagation Neural Networks uses the backpropagation (BP) algorithm, which is a supervised learning algorithm used in feed-forward architectures. Backpropagation is the most widely used algorithm in ANNs. The signals travel forward in the topology and sends backward the estimated error. A hidden neuron of this network is shown in Figure 2.3

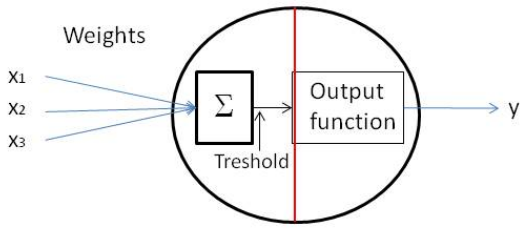

Figure 2.3. A neuron of a backpropagation neural network

If is $\vec{x}$ the input data and $W_{i,j}$ are the weights between the input neurons i and the hidden neurons j, then the activation function of this algorithm is given by equation 2.7.

$$h_{i,j}(x_i, W_{i,j}) = \sum_{1}^{n} x_i * W_{i,j} > \text{threshold} \tag{2.7}$$

The most used output function for the neuron is the sigmoidal function given in equation 2.8

$$\frac{1}{1 + e^{h_{i,j}(x_i, W_{i,j})}} \tag{2.8}$$

where h is the activation function. The BP algorithm minimizes the error function by using the method of gradient descent, i.e., in this algorithm the sum of the gradient of the error function for every hidden neuron is calculated in every iteration. Once the total error is computed, then the algorithm tries to modify the weights of the links between the hidden layer and the output layer to minimize this sum (as described by equaiton 2.9).

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial w_3}, \ldots, \frac{\partial E}{\partial w_n}) = 0 \tag{2.9}$$

The weights are incremented by

$$\Delta w_i = \gamma * \frac{\partial E}{\partial w_i}, i = 1, ..., n \tag{2.10}$$

More information about the backpropagation algorithm can be found in [28].

2.2.1.2. Radial Basis Function Neural Network

A Radial Basis Function (RBF) Neural Network is also a feed-forward network. In this topology, the neurons are defined to be activated when the input sample belongs to their cluster. Membership is decided by estimating the Euclidean distance between the input data and the weight of the links with the following layer. Unlike the BP algorithm, the RBF algorithm fixes the input weights of the hidden neurons and tries to optimize the weights of the following layers. In this topology hidden neurons use a non-linear activation function. There are many possibilities for this function, but the most usual it is the Gaussian function given in equation 2.11.

$$H(x) = \mathrm{e}^{-\beta \vec{x}^2} \text{ for some } \beta > 0 \tag{2.11}$$

This can be approximated as:

$$h(x) = \mathrm{e}-\frac{||\vec{x} - \vec{c_i}||^2}{\sigma^2} \tag{2.12}$$

This choice of function implies that when the distance to the center of the cluster is small, then and only then does the neuron have a perceptible value. This value decreases rapidly to zero as the distance from the center of the clusters increases.

The output function of the network is therefore of the form shown in equation 2.13.

$$F(x) = \sum_{i=1}^{k} \vec{c_i} * h(\vec{x}) \tag{2.13}$$

This algorithm can implement both supervised and unsupervised learning. It can be used for classification, times series prediction, approximation function, etc.

The main problem of neural networks is that the topology (specifically the number of neurons) should be defined *a priori*. For that reason, clustering techniques are frequently used to extract information which is hidden in the data, thus enabling us to make the best choice of the number of neurons. Clustering techniques also can give initial values for the links between the two first layers.

In addition, both BP and RBF neural networks have the same major drawback in that both are strongly dependent upon their initial parameters. This means that the network can come trapped in a local minimum instead of finding a global minimum if the network starts close to a local minima. In [22] both BP and RBF neural networks were studied and compared by Leonard and Kramer. Leonard and Kramer claim that a RBF NN performs better in terms of identifying samples (a given combination of inputs) located far from the training data. It was also claimed that in general, RBF NN are faster than BP NN by one decimal order of magnitude.

2.2.2. Clustering

Clustering is the task of divide a set of objects into clusters. Every cluster is formed by a group of objects which share similarities such that objects belonging to different cluster are as different as possible. Clustering is therefore an unsupervised classification mechanism which aims to reduce the dimension of the input set of data by discarding redundant information. Clustering techniques are widely used in pattern recognition and classification, and image processing.

The objects $\vec{x} \in R_n$ are usually observations of a phenomenon collected from n measurements, $\vec{x} = (x_1, x_2, \ldots, x_n)$. An example of a cluster in two dimensions is shown in Figure 2.4. In this example the data has been divided into three clusters.

There are two different techniques regarding applying this kind of partitioning to a data set: hard partitioning and soft partitioning.

Hard Clustering implements hard partitioning. In this technique the objects of the data set belong to one and only one cluster. Figure 2.4 shows this type of partitioning.

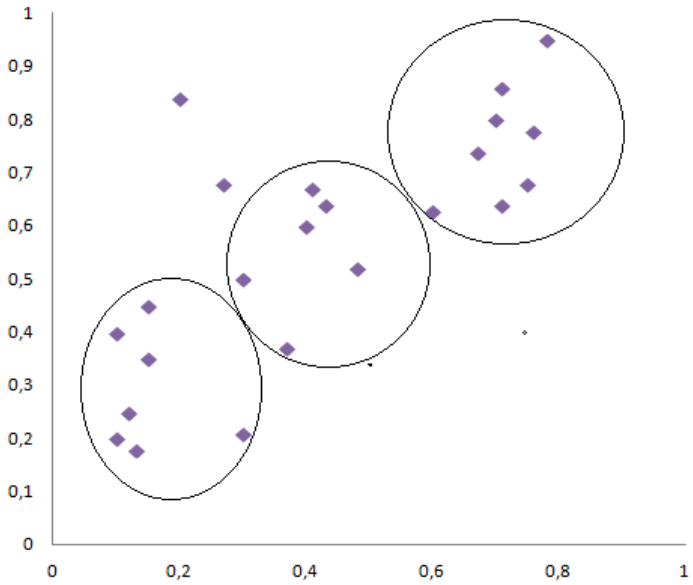

Figure 2.4. Hard partitioning over a data set

Soft Clustering utilizes soft partitioning (also called fuzzy partitioning) allowing objects to belong to multiple clusters. Each object can be a member of a cluster to a certain degree, ranging from 0 to 1. An example of this is shown in Figure 2.5.

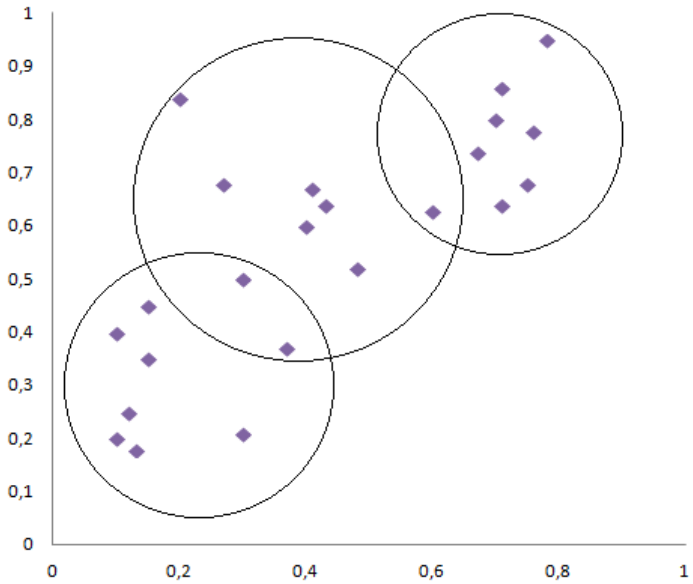

Figure 2.5. Soft partitioning over a data set

Regarding the method used to split up the cluster, multiple algorithms can be used. In this thesis the technique we will use is linear optimization algorithms. These algorithms are used to search for the local minima of an objective function.

2.2.2.1. Fuzzy c-means

A fuzzy clustering algorithm that is widely used in software metric analysis is fuzzy c-means (FCM). This agorithm divides the objects into equal spherical clusters by using as the distance norm the Euclidean distance. This technique aims to optimize the function shown in equation 2.14. This function was first proposed by Bezdek in [17].

$$J_m(Z;U,V) = \sum_{i=1}^{c}\sum_{k=1}^{N}(\mu_{ik}^m)||z_k - v_i||^2 \tag{2.14}$$

The algorithm is based upon the following steps [2]:

1. Initialize the membership matrix.
2. Compute the clusters model.
3. Compute the distance between the sample and the clusters
4. Update the membership matrix

The output of the algorithm it is the membership matrix and the centers of the clusters.

Due to the use of the Euclidean distance to define the clusters, this algorithm only gives good results when the data set forms spheroids of the same size or when the distance between the different clusters is large. As a result the applicability of fuzzy algorithms is dependent upon the shape of the clusters. In the case of FCM each cluster will have a spherical shape, but the cluster might also be elliptical or rectangular. The decision upon the shape is very important as it represents a limitation of the algorithm which forces the algorithm to find clusters with that specific shape *even when is not present in the data.*

One approach to solve this problem is the use of the Gustafson-Kessel algorithm.

2.2.2.2. Gustafson-Kessel Algorithm

In contrast to FCM the Gustafson-Kessel (GK) algorithm implements an adaptive distance norm so it can distinguish the different geometrical forms which the patterns perform [20, 2].

The operating of the algorithm is similar to FCM, but the distance used is defined in equation 2.15.

$$D^2_{ikA_i} = (z_k - v_i)^T A_i (z_k - v_i) \tag{2.15}$$

It should be noted that the GK distance is the same as the FCM distance when A is the identity matrix. In this case, A is described by the Lagrange multiplier method for every i cluster as:

$$A_i = [\rho_i \det(F_i)]^{\frac{1}{n}} F_i^{-1} \tag{2.16}$$

Where F_i is the fuzzy covariance matrix:

$$F_i = \frac{\sum_{k=1}^{N} (\mu_i k)^m (z_k - v_i)(z_k - v_i)^T}{\sum_{k=1}^{N} (\mu_i k)^m} \tag{2.17}$$

Thus the objective function is:

$$J(Z; U, V, A_i) = \sum_{i=1}^{c} \sum_{k=1}^{N} (\mu_{ik}^m) D^2_{ikA_i} \tag{2.18}$$

One example of the results that can be obtained from this algorithm is shown in Figure 2.6. Note that this is the same data set as used in Figures 2.4 and 2.5

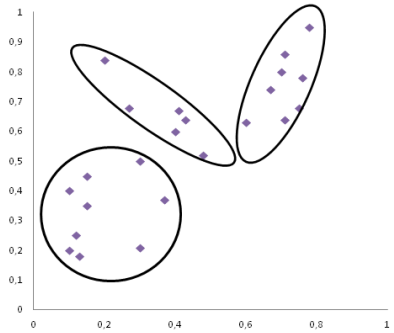

Figure 2.6. Results of Gustafson-Kessel Algorithm over a data set

The main problem of these algorithms is that the number of clusters, the fuzziness exponent, and the tolerance must be determined *a priori* [19].

Another problem due to the choice of algorithm is that they are strongly dependent on the initial parameters, thus they may converge to different local minima for different initializations. This also means that the resulting NN may not correctly predict the output for an input that is not nearly equivalent to a sample from the training set.

The number of clusters K is the most critical parameter. This parameter is hard to determine because it has to be defined in advanced, usually without knowledge of the actual number of cluster that are present in the data set. Once this number is defined, the algorithm will look for that number of cluster whether they exist or not.

The fuzzifier exponent, m, is related to the level of fuzziness of the algorithm. When $m \to \infty$, it is completely fuzzy and it becomes less fuzzy as m decreases. The most frequent value for this parameter is 2.

Finally the fuzzy algorithm may or may not converge to a clear minimum. For this reason a tolerance parameter is needed. This paramter defines the accepted difference between two successive iterations that reflects the fact that the algorithm has found an acceptable minimum

Chapter 3

Analysis

In this chapter an overview of the model followed to create the final tool. In Section 3.1 the specifications and limitations of the tool are given. In Section 3.2 the background material is analyzed and the software metrics and regression model are chosen.

3.0.3. Specification

The specifications of the tool are:

- The final tool should implement two functions: predict the actual number of bugs of a given set of source code; and allow the user to train the network to adapt it to different environments, i.e. different programming languages.
- Even though the program should work for different programming languages the predictor in this thesis project will analyze only source code written in C++.
- The tool should be run from the Linux shell in order to be easily incorporated into the existing project running within the department.
- The tool should be implemented by using license free or open source programs.

3.0.4. Selection of the model

The main goal which need to be reached in order to create the tool is the creation of a model which relates software metrics with the number of bugs in a set of C++ source code.

The background literature described in Chapter 2 has shown that software metrics and bugs have been related by many different models. There are also

different metrics that can be used. Thus, it is necessary to select the metrics and the model that are most useful for our goal.

Regarding the program, we have to take into consideration two main restrictions: the first restriction is that this program should analyze C/C++ language because this is the programming language used by the department; and the second restriction is that the program needs to include only be license free or open source as part of the tool. There are only two programs in Table 2.5 that fulfill these requirements: CCCC and Sonar. Sonar has a free plug-in that parses source code written in C/C++ but is very limited (it does not implement the same features as the original program) so we discarded this program. Consequently, the program selected to compute the metrics is CCCC-C and C++ Code Counter.

Regarding the metrics, CCCC calculates all of the metrics but three: RFC, LCOM, and NOA. RFC and LCOM were also excluded in the study carried out by Subramanyam and Krishnan in [32] due to the complexity of their computation. In their study they claim that the effect of exclude RFC was limited and that LCOM rather than providing useful information may cause alterations in the results owing to the lack of a solid definition. Lastly, NOA has not been considered in many studies. In this thesis project we will also not consider NOA. In addition, we also eliminate NOM from our list of metrics, because NOM has the same value as WCM when the complexity value assigned to each method is 1.

Concerning the regression model, in [22] Leonard and Kramer performed a comparison between BPNN and RBFNN. They claim that BPNN classifies the samples arbitrarily when the samples are far from the training data. In contrast RBFNN classifies samples according to the distance between the samples and the training data. Thus, Leonard and Kramer claim that a RBFNN results in better performance in terms of identifying new samples. Therefore, in this thesis project RBFNN is used to relate the software metrics of a code with its bugs. Furthermore, we agreed with Gray and MacDonell [29] that a neuro-fuzzy hybrid is the best option to implement regression models. Hybrid models implement neural networks with the improvement that the initial parameters are defined by the prior clustering. This model is also supported by [34], where Wang and Shang showed that fuzzy clustering RBFNNs have better capabilities in pattern classification than normal RBF networks.

Finally we decided to use FCM. The main reason was that RBFNN (as FCM) utilizes the Euclidean distance to determine which samples should activate the different hidden neurons. Consequently it will not be useful to look for any other type of shape in the input data.

Chapter 4

Model implementation

In this chapter the implementation of the model is explained in detail. Section 4.1 defines how clustering techniques are related with the neural networks to create neuro-fuzzy hybrid models. Section 4.1 finalizes with the definition of some parameters that can be used to ensure the quality of the model. In Section 4.2 the software metrics' extraction is explained. In Section 4.3 and 4.4 respectively the FCM and RBFNN implementation are specified. Finally in Section 4.5 the results of the model are analyzed.

4.1. Hybrid topology

The creation of the final tool started with the implementation of the neuro-fuzzy hybrid model. This hybrid topology is composed of a primary stage of clustering followed by a neural network. The methodology is the following: first we divided the data samples into training samples and testing samples. Each data sample contains the software metrics of one module. We use the training samples to train the model and the testing samples to analyze its performance. Once we have the training set we normalized it between 0 and 1, in order to limit the upper bound of the input data to the network. Subsequently we applied the clustering technique, in this case fuzzy c-means, over the training set and we obtained the center of the c clusters defined to the algorithm. Later we created the radial basis neural network where we use the centroids of the clusters to determine the number of hidden neurons and the initial weight of the links between the input layer and the hidden layer. Finally, as is shown in Figure 4.1, we used the training samples to train the network.

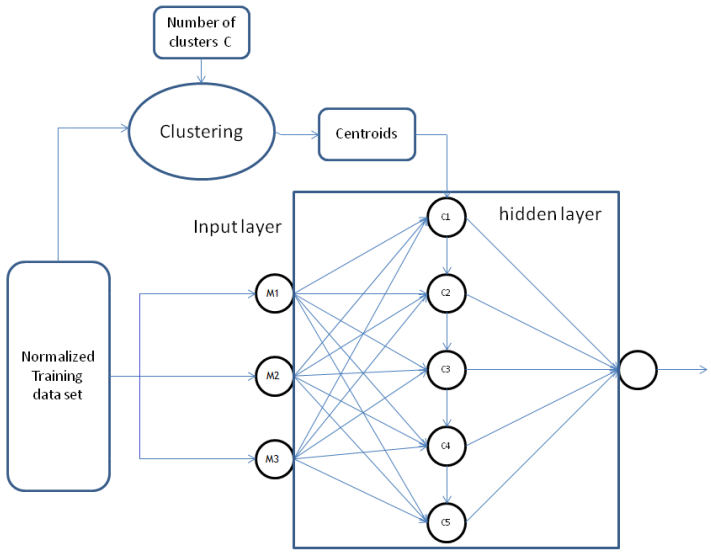

Figure 4.1. Example of an hybrid topology with 5 clusters

After training the model we used the testing data samples to verify its capability as a bug predictor. The quality of the model was evaluated in terms of the following indicators:

Confusion matrix The confusion matrix of a neural network shows its classification results over a data sample. This matrix is shown in Table 4.1. In this matrix a module is valuated as 0 when it is non buggy and as 1 when it is.

Table 4.1. Confusion Matrix

	Predicted module=0	**Predicted module=1**
Actual module=0	f 00	f 01
Actual module=1	f 10	f 11

From this matrix we can derive the following measures:

Accuracy Accuracy is the ratio of the correctly predicted modules to the total predicted modules.

$$Accuracy = \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}} \tag{4.1}$$

Precision Precision is the ratio of the correctly predicted buggy modules to the total modules that are predicted to be buggy. The lower the precision,

the more effort is wasted in testing free error modules.

$$Precision = \frac{f_{11}}{f_{01} + f_{11}} \tag{4.2}$$

Recall Recall is the ratio of the correctly predicted buggy modules to the total modules that are actually buggy. The lower the recall, the more buggy modules go undetected.

$$Recall = \frac{f_{11}}{f_{10} + f_{11}} \tag{4.3}$$

F-measure The F-measure is a combination between precision and recall. It considers the tradeoff between both measures.

$$F - measure = \frac{2 * Precision * recall}{Precision + Recall} \tag{4.4}$$

Receiving Operating Characteristic (ROC) The ROC curve of the network is also plotted. The ROC shows the tradeoff between correctly predicted buggy modules and wrongly predicted buggy modules. In [36] it is claim that the greater the area under the curve, the greater the quality of the neural network.

Root Mean Square (RMS) RMS shows the difference between the expected output value and the actual output value. The lower the RMS, the greater the quality of the network. RMS is calculated in [39] as shown in Equation 4.5.

$$RMS = \sqrt{\frac{\sum_{i=1}^{n} (y_{actualoutput} - y_{expectedoutput})^2}{n}} \tag{4.5}$$

4.2. Extraction of metrics and bugs

In this thesis project the extraction of the software metrics was done by two different methods. We used the CCCC program to extract the metrics which can be obtained from the source code and we created a simple script to extract from a repository of trouble reports the BUGFIXES metric. The actual number of bugs in this code was also extracted from the same repository. Note that here we are assuming that all of the bugs have been detected in this repository. We believe that this is a relatively safe assumption because the lifetime of the program is large enough to have reported all the bugs.

4.2.1. Extracting the metrics

In order to extract the metrics we used the CCCC- C and C++ Code Counter program. This program can be downloaded from http://cccc.sourceforge.net/.

CCCC extracts the metrics WMC, DIT, NOC, CBO, McCabe's Cyclomatic complexity, Fan in, Fan out, LOC, and information flow complexity from a given file or set of files of a directory.

In order to calculate these metrics the CCCC program divides the files into modules. Every class as well as every namespace is considered as a module (in the C++ environment). Functions which do not belong to any of these structures are merged as part of the module "Anonymous".

In our program we wanted to perform a class-level bug prediction because Object Oriented programs are built over classes so we did not use the information given by the "Anonymous" file to create our predictor. We did not use either the metrics given by modules for which any definition or member function had been identified or modules which triggered a parse failure in CCCC, i.e. protected classes.

It is important to notice that CCCC has some limitations in computing metrics, therefore, a more specific definition of these metrics (as they are actually computed) will be given in Table 4.2.

Although the measures of the CCCC program are not perfect, they are claimed to agree with the manually calculated values of these metrics, under the same definitions, within 2-3% (i.e., the automatic extraction of these metrics has an error of only 2-3%).

Table 4.2. CCCC metrics' definition

Name/ Acronym	**Description**
Lines of Code (LOC)	The total number of lines is calculated taking into account the number of non-blank and non-comment lines. The preprocessor lines are treated as blank lines and declarations of global data are ignored. However the number of lines may be overestimated as the program may count twice the number of lines in the class definitions. This could occur because the algorithm counts the lines of a module as the sum of the lines of the module itself plus the lines of its member functions. Thus, declarations and definitions of the member functions in the body of the class will be counted twice.
McCabe's cyclomatic complexity	McCabe's Cyclomatic Complexity value is approximated by counting the commands which create extra paths in the execution of a class. In the case of C++ this means that it counts the number of the following tokens: 'if', 'while', 'for', 'switch', 'break', '&&' and '||'.
Fan in/ Fanout	These two metrics concern the number of classes which reference the class/the number of classes referenced by the class. CCCC can only identify the following relationships in a class: inheritance, instance of a supplier class, and the existence of member functions which accept/return instances from/to a supplier class. However, it is often one of these relationships references a class, so both counts seems to be highly correlated.
Information flow complexity	IF_c is calculated as the square of the product of the fan in and fan out of the current module.
Weighted Methods Per Class (WMC)	WMC is computed as the multiplication of the number of functions of the current module times a weighting factor. The algorithm uses a nominal value of 1 as weighting factor.
Depth of Inheritance Tree (DIT)	DIT is given by the length of the longest path of the inheritance tree which ends at the current module.
Number of Children (NOC)	NOC is given by the number of modules that directly inherit the current module.
Coupling between object classes (CBO)	CBO is the number of modules coupled with the current module either as clients or suppliers.

4.2.2. Extracting information about bugs

In order to implement supervised learning in the neural network the number of actual bugs (the expected output of the system) has to be extracted from the repository. This thesis project trained of the network using a released version of a product developed in Ericsson. For this program we extracted the number of previously fixed bugs as well as the number of actual bugs from the repository based upon trouble reports recorded for the project.

We consider that every trouble reported indicates one bug in all the files which had to be changed to fix this bug.

Since CCCC parses the programs by modules and we extract the number of bugs by files there is a discrepancy that had to be resolved. Our solution was to assign to every module the sum of bugs reported for the files which compose the module.

4.3. FCM Clustering

We have implemented FCM in R using the package 'e1071' which can be downloaded from `http://cran.r-project.org/web/packages/e1071/index.html`. We applied the function 'cmeans' with the method 'cmeans' to the training set. There are three parameters that have to be defined *a priori*: the number of clusters, the fuzziness exponent, and the tolerance.

Number of clusters

The number of clusters is defined as the number of groups into which the data set is split during the clustering process. The number of clusters is the most critical parameter because the algorithm will look for this number of clusters whether they are present or not in the data set. Determining this parameter without previous study of the samples is, in general, a hard task. Therefore we will use some parameters which have been used in the literature to determine the optimum number of clusters in a data set. These parameters are:

Xie.beni Proposed by Xuanli Lisa Xie and Gerardo Beni in [37], Xie.beni (XB) gives the ratio of the total variation of the partition and the centroids, and the separation of the centroids vectors. Thus, it is function of the data set and the center of the clusters. The minimum value of xie.beni under comparison usually indicates the best partition.

$$XB = \frac{\sum_{i=1}^{c}\sum_{k=1}^{N}(\mu_{ik}^2)||z_k - v_i||^2}{N * min_{i\neq k}||v_i - v_k||^2} \tag{4.6}$$

Fukuyama.sugeno Fukuyama.sugeno (FS) index, as shown in Equation 4.7, computes the difference between two terms combined with the fuzziness in the membership matrix: the compactness of the representation of the data set, and the degree of separation between each cluster and the mean of the cluster centroids. Low values of fukuyama.sugeno indicate good partitions [38].

$$FS = \sum_{i=1}^{c}\sum_{k=1}^{N}(\mu_{ik}^{m})(||z_k - v_i||^2 - ||v_i - \bar{v_i}||^2) \tag{4.7}$$

Partition coefficient and Partition entropy Proposed by Bezdek, the partition coefficient (PC) and partition entropy(PE) indexes measure the fuzziness of the partition. This means that both parameters measure how much the overlapping is between the clusters in the data set.

The partition coefficient is given by Equation 4.8 (and takes on values in the range [1/c,1]).

$$PC = \frac{1}{N}\sum_{i=1}^{c}\sum_{k=1}^{N}\mu_{ik}^{2} \tag{4.8}$$

The partition entropy is given by Equation 4.9 (and takes on values in the range [0, log(c)]).

$$PC = -\frac{1}{N}\sum_{i=1}^{c}\sum_{k=1}^{N}\mu_{ik}\log_a \mu_{ik} \tag{4.9}$$

We look for a value for the number of clusters which maximizes PC and minimizes PE [38].

Partition separation Index (CS (Compact-Separate)Index) The partition separation index is created from the partition coefficient and the partition entropy. It identifies compact, separate clusters. Large values of this coefficient means that the clusters are more compact and that they are well separated from each other [12]. However, for large data sets this index is computationally infeasible since a distance matrix between all the data terms has to be computed.

$$PS(c) = \sum_{i=1}^{c}\sum_{k=1}^{N}\frac{(\mu_{ik}^{2})}{\mu_M} - e^{(-min_{j\neq i}\frac{||v_i - v_j||^2}{\beta_T})} \tag{4.10}$$

where

$$\mu_M = max_{1\leq i\leq c}\sum_{k=1}^{N}\mu_{ik}^{2} \tag{4.11}$$

and

$$\beta_T = \frac{\sum_{i=1}^{c}||v_i - \bar{v}||^3}{c} \tag{4.12}$$

Tolerance

Tolerance is determined by the program. The only value that is possible to change is the number of maximum iterations. This maximum number of iterations indicates to the program when to stop if the tolerance is not achieved. In this case, after running FCM several times, we observed that the algorithm always converged to a solution.

Fuzziness exponent

We set the fuzziness exponent to the value of 2, as this is its default value.

4.4. RBFNN

The RBFNN used in this thesis project was implemented in R by using the package RSNNS [5]. This package uses the libraries of the Stuttgart Neural Network Simulator (SNNS) [39] which allows the package to implement many different neural networks. A further description of the package is given in [4]. In order to implement our model we have used the low level interface of the package. The guide used to set up the RBFNN can be followed in [39].

The procedure was the following, first of all we created a SNNS object and we defined the neurons of the network. We defined one input neuron for each metric, one hidden neuron for each cluster, and one unique output since we only wanted one output.

The desired output function of the network is shown in Equation 2.13. To compute this output, we defined the activation and the output function of the neurons as shown in Table 4.3. Each of these functions is described below the table.

Table 4.3. Output functions of the neurons of the network

	Activation function	**Output function**
Input neurons	Act_Identity	Out_Identity
Hidden neurons	Act_RBF_Gaussian: $h(q,p) = exp(e^{-\beta_q})$ where $q = \lvert\bar{x} - \bar{t}\rvert^2$ and β is the bias of the neuron	Out_Identity
Output neurons	Act_IdentityPlusBias	Out_Identity

- Act_Identity leaves the neuron active all the time.
- Act_RBF_Gaussian activates the neuron with the distance between $\bar{x}$ (the sample) and $\bar{t}$ (the center of the cluster defined to the neuron). Values of $\bar{x}$

equal to $\bar{t}$ yield an output of 1.0, while larger distances yield an output of 0.0.

- Act_IdentityPlusBias activates the neuron with the weighted sum of all incoming activations and adds the bias of the neuron.

Subsequently, we initialized the weights of the links between the input layer and the output layer. First we initialized the network with the "RBF_Weights" procedure to copy unchanged the centroids of the clusters and the bias into the different hidden neurons. Afterward we initialize the network with the "RBF_Weights_Redo" procedure which initializes the link weights between the hidden and the output layer. The data used in this initialization process are the training data set and their actual outcome values.

Finally, we train the network with the learning function "RadialBasisLearning". This function can modify different parameters within the network: the center vectors of the hidden neurons, the bias of the hidden neurons, and the weights of all the links as well as the bias of the output network. Furthermore, this function prevents the overtraining of the network by limiting the tolerated error in the output neuron. This overtraining occurs when the network learns the output of specific training samples, but fails to learn the general behavior [22] -hence this overtraining should be avoided.

Besides the regression model used to predict the probable number of bugs in the modules, we have implemented a classification model. This classification model identifies whether there are or not bugs in a module. This implementation differs from the implementation of the regression model in the expected output value of the samples. Thus, in the classification model we assign an output value of 1 to all the buggy modules of the training data set and an output value of -1 to the bug free modules of the same set.

4.5. Experiments and results

In this thesis project we have parsed 1755 modules. From these modules, we randomly selected 1492 modules to train the model and 263 to test it (15%). The major drawback of this neuro-fuzzy hybrid model is that it is very dependent upon the samples used to train it, therefore we selected different random combinations of training and testing data sets.

We performed the clustering over the training samples giving different values of the number of clusters ranging from 3 to 10. These results are shown in Table 4.4 where the optimum values according to the definitions given in Section 4.3 are highlighted.

Table 4.4. Cluster validity indexes

N.clusters	XB	FS	PC	PE	CS
10	0.001236	-51.1354	0.60137	0.92128	0.00966
9	0.0013346	-48.7097	0.60091	0.89828	0.00119
8	0.0003734	-50.3111	0.63003	0.81634	**0.00974**
7	0.0004622	-51.1411	0.63922	0.77780	0.00461
6	0.0006059	-50.1556	0.65346	0.71733	0.00500
5	0.0001928	-49.5123	0.69760	0.63600	0.00795
4	**0.0000747**	**-51.7349**	**0.76467**	0.48368	0.00795
3	0.0002578	-38.9224	0.75922	**0.44721**	0.00877

Thus, based upon the indexes given by xie.beni, fukuyama.sugeno, and the partition coefficient we chose 4 as the number of clusters.

Based on the result of the clustering process we implemented a 10-4-1 network topology with 10 input neurons, 4 hidden neurons, and 1 output neuron. We used different values of bias for the hidden units and we allow the algorithm to modify the center of the clusters, the links, and the bias of the output neuron.

Table 4.5 shows the best results we obtained. The best performance was achieved with a bias value of 1.5.

Table 4.5. Quality validity 10-4-1

	Regression model	Classification model
Accuracy	0.21673	0.84791
Recall	1.00000	0.35088
Precision	0.21673	0.86956
F-measure	0.35625	0.50000
RMSE	0.93108	0.77998
AUC	0.50000	0.66816

The threshold for predicting classes as fault-prone or non fault-prone was 0 in the classification model since the value assigned to buggy modules was 1 and to non buggy modules the assigned value was -1. In the case of the regression model the threshold was 0.0125 since 0.2564103 is the output value assigned to 1 bug in a module.

The values in Table 4.5 show that none of these configurations could be used to predict bugs. The regression model does **not** predict any bug in any of the modules of the testing set, and the quality of the classification model is not good since almost 75% of the buggy modules go undetected.

Nevertheless, we compare the results obtained with a fuzzifier exponent of 2

with different values of bias and different number of clusters and we found that the best quality of the model was achieved with 8 neurons and a bias value of 2.

Table 4.6. Quality validity 8-2-1

	Regression model	**Classification model**
Accuracy	0.84790	0.85551
Recall	0.33333	0.43860
Precision	0.90476	0.80645
F-measure	0.48718	0.56818
RMSE	0.77998	0.76023
AUC	0.66181	0.70473

The results showed in Table 4.6 supports the notion that the separation index is the best indicator of the number of clusters in the data set.

These results are in accordance with the claims by Subramanyam and Krishnan in [32], where they stated that "Defects are not uniformly distributed across the modules in the system and that a few modules may account for most defects in the system". We though that since the clusters tend to be located in regions with a high concentration of data samples m then probably a small group of data samples that are responsible for the defects of the program remained hidden in the data. The idea was to find this "buggy" cluster by increasing the number of clusters. However, increasing the number of clusters may lead to a more complicated topology, thus making it more difficult for the learning process of the network -for this reason this trade-off should be controlled.

Our next experiment was to calculate the results of different topologies by changing the three parameters: the number of clusters, the fuzzifier exponent, and the bias of the hidden network. We chose the best topology based upon their F-measures. We decided to use the F-measure as indicator rather than the accuracy because the model seemed to have problems identifying modules with bugs; while working fine at identifying modules without bugs. Since the number of non buggy modules was much greater than the number of buggy modules in our data set, the accuracy did not provide a good evaluation of the quality of the model.

The simulations showed that the best F-measure in the regression model was obtained with 8 clusters, a fuzzifier exponent of 2, and a bias of 2. In the classification model the best performance was achieved with 10 clusters, a fuzzifier exponent of 1.9, and a bias of 2. These results are shown in Table 4.7.

Table 4.7. Quality validity

	Regression model	**Classification model**
Accuracy	0.84790	0.8593156
Recall	0.33333	0.4912281
Precision	0.90476	0.7777778
F-measure	0.48718	0.6021505
RMSE	0.77998	0.7501584
AUC	0.66181	0.7261966

Figures 4.2 and 4.3 show the ROC curve of the regression and the classification model respectively.

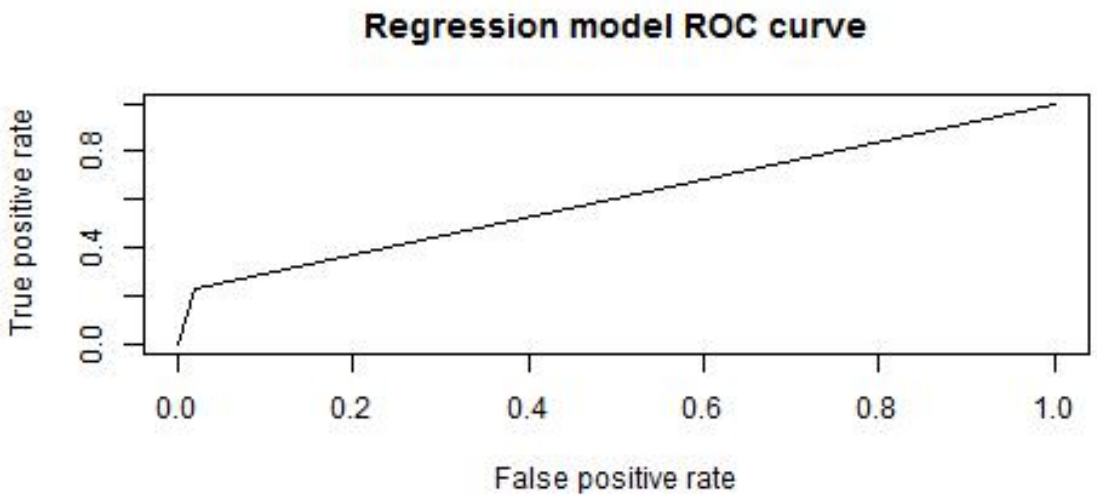

Figure 4.2. ROC curve of the regression model 10-8-1

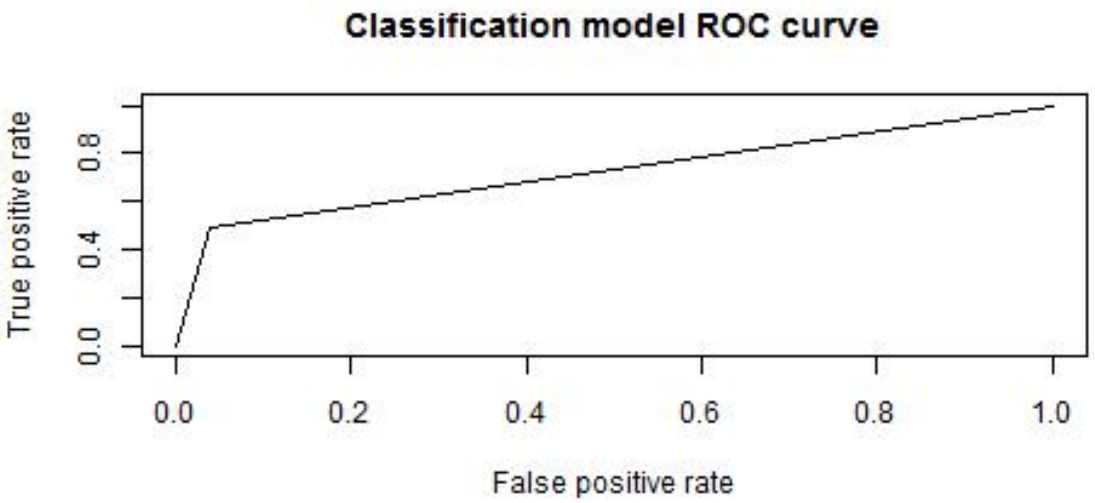

Figure 4.3. ROC curve of the regression model 10-10-1

Furthermore we tested the performance of the regression model by dividing the samples into five different levels depending upon their number of bugs. The accuarcy of the model is low therefore it does not provide enough data to check the capability of the network to distinguish between different levels of bugs and thus to inform the programers which modules should be tested more carefully.

To summarize, our results show that the classification model provides better predictions; however, the desired output of the tool is the number of bugs so we implemented the regression model into our tool.

Chapter 5

Design of the tool

The tool has been developed using bash scripting. The tool implements two functionalities:

- Predicts the number of bugs of a given file or files of a directory, and
- Implements a new neuro-fuzzy hybrid model and trains it.

The prediction mode uses the default regression model to estimate the predictions. The results can be given on a per file or per module basis. In both cases the results are generated for every module/file with the files ranked in descendent order of their number of bugs.

The training mode allows the user to implement the neuro-fuzzy hybrid model. In this new implementation the data set and the parameters (in this case the number of clusters, fuzzifier exponent, and bias of the hidden neurons) can be changed. The parameters can be defined by the user or be selected by the program. After training, the tool compares the new results with the original results and asks for the user's confirmation to save the network which achieves the best performance..

Further explanation of the commands used to run the tool can be found in Appendix A.

The requirements to run the tool can be found in Appendix B.

5.1. Analysis of the tool

We analyzed some of the features of the tool:

User-friendly

After show a demo of the tool to its future users the tool seemed to be easy to

use and the output information easy to understand.

Operation

The time execution of the program in both modes (prediction mode and training mode) depends upon the data set desired to be parsed. In our case, when we use as input data the whole project (1755 modules), it took to the program half an hour to estimate the predictions and more than one hour to train the network. The main reason of this waste of time was the necessity of the CCCC program to have all the files in the same directory (at the same level) to parse them properly as a whole. Therefore, the improvement of the execution time is related with the improvement in the features of the CCCC program. Regarding the operation of the program any problem was found in any of the modes.

Error handling

The tool implements error handling. The user is informed when the commands used are not properly typed and when there have been a problem during the execution of the program. In this case, an error message is displayed in the shell giving some information about where the problem occurred.

Portability

The tool could be incorporated as part of the c-make of any project. In our particular case the tool was incorporated in the c-make of the project in such way that users had the option to run it in background mode every time they compiled the project. When our tool was enabled all the code of the project was parsed and predictions were estimated. The increment of time while making the project was considerable, for that reason, by default, the tool is disables in the c-make. Future work out of the scope of this project is to integrate the tool as part of Jenkins. Thus, the project is monitored and the predictions are estimated every day.

Adaptability

The tool has been developed by using license free or open source programs. It uses the R tool and the CCCC program and it can be adapted without problems to new versions of both programs as far as the R version support the required packages and the CCCC program does not change completely their output XML report. Furthermore, it would be easy to add more metrics and/or new programming languages if the CCCC supports them.

Chapter 6

Conclusion

In this thesis project we have written a program to predict the number of bugs in source code. The program is based on a neuro-fuzzy hybrid model created from a first stage clustering, followed by a Radial Basis Function Neural Network. However, the program does not achieve high accuracy due to the lack of independent samples in the data set. Neuro-fuzzy hybrid models are very dependent upon the data set used to train them. Thus, large data sets across different source code should be used to get a high level of accuracy in predicting bugs. The user must use the functions of the program to simulate a new model and train it with different and larger data sets.

Regarding the experiments done within this thesis project, the results have shown that incorrect choices in the number of clusters leads to large changes in the results of the RBFNN. Bad initializations in the clustering process and hence in the neural network can easily lead to sub-optimal clustering and thus in not accurate predictions. The sensitivity to initialization becomes acute when the data set is not large enough.

It should also be mention that in our experiments the partition index seemed to be the best indicator of number of clusters in the data set was supported by the neural network results. In addition, should be mention that our experiments show that fuzzifier values close to 2 lead to better predictions. This value is the most frequent value used in previous studies.

To conclude, experiments showed that classification models produce better predictions than regression models. This is because of the accuracy required from the network to predict buggy/non buggy models in these two different types of models.

Chapter 7

Future work

This chapter suggests future work to improve upon the results shown this thesis project. Section 7.1 focuses on the improvements associated with the implementation of the model used to relate metrics with bugs. Section 7.2 focuses on the possible improvements of the tool.

7.1. Model improvements

The first task that should be done is to use larger data sets across different sets of source code to train the neuro-fuzzy hybrid model of the tool. Larger and more independent training data sets would lead to a more accurate tool with better abilities to detect error-prone source code.

Regarding the software metrics more metrics can be added, or the existent ones can be changed since the right combination of metrics could make a huge improvement to the early prediction of bugs. It is important to adapt the software metrics to the programming language so they can better define the quality of the code. Programs which estimate the value of the software metrics with greater accuracy could be used also to improve the accuracy of the model.

It would also a major improvement to modify the methodology followed when reporting bugs. In the tool we assign different bugs to the modules as a function of the bugs that we extract from the files, consequently, the number of bugs assigned to each module is *overestimated*. Thus, we propose to report the bugs indicating the exact modules that were necessary to modify in order to fix the bugs.

Regarding the neuro-fuzzy hybrid model, it would be interesting to have a neural network able to detect, as the Gustafson-Kessel Algorithm, different shapes of clusters in the data set. These shapes remain unknown and we forced them to be gather in circular clusters.

Lastly, future studies in early bug prediction may use genetic algorithms. Genetic algorithms can be used to set the weights of the links in a fixed architecture and to define the number of hidden neurons. Moreover, genetic algorithms have a more interesting application from this thesis project's perspective: we could select the training data of the data set which leads to the best learning by the neural network. These genetic algorithms discard non useful information and keep only the representative samples.

7.2. Tool improvements

The tool has multiple potential improvements that could lead make it a more useful and complete program. We propose three major improvements:

- The tool could be implemented to run on different platforms (such as Microsoft's Windows).
- The tool could have a more intuitive user interface.
- The tool could be improved in order to predict bugs in different programming languages.

Furthermore, it would be a major improvement to adapt the program to predict bugs in C language since there are many repositories with large data sets that can be used to train the model. It would be easy for users to find an open source code for which all the bugs have been previously reported. Thus, a better analysis of the accuracy of the model could be carried out.

Chapter 8

Required reflections

Nowadays companies and customers demand an improvement in the quality of the software products. This thesis project has shown that bug prediction is a feasible goal that would improve the quality of the software and decrease the costs of the software development process. In addition, bug prediction would also decrease the cost of the customers since they may skip the costs of their own testing and the cost of the maintenance of the product. Therefore, the tool developed in this thesis project must be used to improve the process of detecting and correcting bugs in the software development process.

Further development of this tool may provide

- more efficient developers since programmers using this tool will learn the least error-prone programming styles; and with
- more efficient development process since the location of the bugs will be given as soon as the code is written. This tool together with the new agile methods will allow the user to resolved the bugs very fast.

Bibliography

[1] S.N. Ahsan and F. Wotawa. Fault prediction capability of program file's logical-coupling metrics. In *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 257 –262, Nov. 2011.

[2] Robert Babuska. Fuzzy clustering with applications in pattern recognition and data-driven modeling. Technical report, Delft Center for Systems and Control, Delft University of Technology, Netherlands.

[3] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751 –761, Oct. 1996.

[4] Christoph Bergmeir and José M. Benítez. Neural networks in R using the stuttgart neural network simulator: RSNNS. *Journal of Statistical Software*, 46(7):1–26, 2012.

[5] Bergmeir C. and BenÃŋtez J. RSNNS: Neural networks in r using the stuttgart neural network simulator (snns). `http://cran.r-project.org/web/packages/RSNNS/index.html`, 2012.

[6] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476 –493, Jun. 1994.

[7] Dimitrios Siganos Christos Stergiou. Neural networks. `http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Asimpleneuron`. Accessed:1 December 2012.

[8] C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223 –232, March 2012.

[9] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories (MSR), 2010*, pages 31 –41, May 2010.

[10] S. Dick and A. Kandel. Fuzzy clustering of software metrics. In *Fuzzy Systems, 2003. FUZZ '03. The 12th IEEE International Conference on*, volume 1, pages 642 – 647 vol.1, May 2003.

[11] M. Dixon. An objective measure of code quality. Technical report, Energy group, 2008.

[12] C. Frelicot, L. Mascarilla, and M. Beithier. A new cluster validity index for fuzzy clustering based on combination of dual triples. In *IEEE International Conference on Fuzzy Systems, 2006*, pages 42 –47, 0-0 2006.

[13] Qiang Gan and C.J. Harris. A hybrid learning scheme combining em and masmod algorithms for fuzzy local linearization modeling. *Neural Networks, IEEE Transactions on*, 12(1):43 –53, Jan. 2001.

[14] Carlos Gershenson. Artificial neural networks for beginners. `http://arxiv.org/ftp/cs/papers/0308/0308031.pdf`, 2003. Accessed:1 December 2012.

[15] S. Henry and D. Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510 – 518, Sept. 1981.

[16] Tu Honglei, Sun Wei, and Zhang Yanan. The research on software metrics and software complexity metrics. In *Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on*, volume 1, pages 131 –136, Dec. 2009.

[17] J.C.Bezdek. *Fuzzy Mathematics in Pattern Classification*. PhD thesis, Applied Math. Center, Cornell University, Ithaca, 1973.

[18] Cong Jin, Shu-Wei Jin, Jun-Min Ye, and Qing-Guo Zhang. Quality prediction model of object-oriented software system using computational intelligence. In *Power Electronics and Intelligent Transportation System (PEITS), 2009 2nd International Conference on*, volume 2, pages 120 –123, Dec. 2009.

[19] Uzay Kaymak and Magne Setnes. Extended fuzzy clustering algorithms. ERIM report series Research in Management, Erasmus nRS-2000-5 1 -LIS, Research Institute of Management, Erasmus University Rotterdam, Netherlands, Nov. 2000.

[20] R. Krishnapuram and Jongwoo Kim. A note on the gustafson-kessel and adaptive fuzzy clustering algorithms. *IEEE Transactions on Fuzzy Systems*, 7(4):453 –461, aug 1999.

[21] Hua Jie Lee, Lee Naish, and K. Ramamohanarao. Study of the relationship of bug consistency with respect to performance of spectra metrics. In *2nd IEEE International Conference on Computer Science and Information Technology, 2009. ICCSIT 2009.*, pages 501 –508, Aug. 2009.

[22] J.A. Leonard and M.A. Kramer. Radial basis function networks for classifying process faults. *Control Systems, IEEE*, 11(3):31 –38, April 1991.

[23] Tim Littlefair. CCCC: C and C++ code counter. `http://cccc.sourceforge.net/`, 1997. Accessed:1 December 2012.

[24] Michele Lanza Marco D'Ambros and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering (EMSE)*, 17(4):531–577, 2012.

[25] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308 – 320, Dec. 1976.

[26] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[27] J. Ratzinger, H. Gall, and M. Pinzger. Quality assessment based on attribute series of software evolution. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 80 –89, Oct. 2007.

[28] R. Rojas. Neural networks, a systematic introduction. Technical report, Springer-Verlag, Berlin, New-York, 1996.

[29] A.R. Gray S.G. MacDonell. A comparison of modeling techniques for software development effort prediction. In *International Conference on Neural Information Processing and Intelligent Information Systems*, pages 869–872. Springer-Verlag, 1997.

[30] P. Singh and S. Verma. Empirical investigation of fault prediction capability of object oriented metrics of open source software. In *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on*, pages 323 –327, 30 May 2012-1 June 2012 2012.

[31] Ian Sommerville. *Software Engineering*. Addison Wesley, 7 edition, May 2004.

[32] R. Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297 – 310, april 2003.

[33] Iris Vessey and Ron Weber. Research on structured programming: An empiricist's evaluation. *Software Engineering, IEEE Transactions on*, SE-10(4):397 –407, July 1984.

[34] Yongxue Wang and Yan Shang. Fuzzy clustering rbf neural network applied to signal processing of the imaging detection. In *Measuring Technology and Mechatronics Automation (ICMTMA), 2010 International Conference on*, volume 2, pages 321 –324, March 2010.

[35] E.J. Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357 –1365, Sep 1988.

[36] K. Woods and K.W. Bowyer. Generating roc curves for artificial neural networks. *Medical Imaging, IEEE Transactions on*, 16(3):329 –337, June 1997.

[37] X.l. Xie and G. Beni. A validity measure for fuzzy clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):841 –847, Aug 1991.

[38] Miin-Shen Yang and Kuo-Lung Wu. A new validity index for fuzzy clustering. In *The 10th IEEE International Conference on Fuzzy Systems*, volume 1, pages 89 –92, 2001.

[39] Andreas Zell. *SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2*. University of Stuttgart and WSI, University of Tubinga, 1998.

Appendix A

Sinopsis of the tool

It is possible to specify a path to a file/directory if the user wants to run the program over one file or one directory of the project, rather than over the whole project.

For prediction:

```
>ZMetriX -p [-file/-module] [path to file/directory] (path to
    git repository)
```

Options:

-file: shows the results by file in PredictedBugreportbyfile

-module: shows the results by module in PredictedBugreportbymodule

For training:

```
>ZMetriX -t [-default] [path to file/directory] (path to git
    repository)
```

Options:

-default: The program defines the number of clusters, fuzzifier exponent, and bias of the neuro-fuzzy hybrid model. If the default mode is not selected, the program will ask the user to define these parameters.

After the model is implemented the program compares the original model with this new model. The program recommends saving the best network (based upon the value of its F-measure), but the user must decide which model they want to keep. If "yes" is typed, then the original network is deleted and the new network is saved as the new default regression model. If "no" is typed then the new network is

deleted.

To run the program in training mode is necessary to have a file called Bugs.txt with the actual number of bugs associated with each module of the data set. If the user does not have this file, they should run first the command:

```
>sh Extractbugs (path to git repository)
```

Appendix B

Requirements of the tool

The requirements for the correct operation of the program are:

- The CCCC program should be installed
- The R program should be installed with the following packages: "e1071", "RSN"NS", "ROCR". The minimum version required is 2.15.
- Xmlstarlet should be installed on Linux.
- CCCC program should be installed
- The distributed version control system where the bugs are reported must be GIT and bugs should be reported with the word "artif".

Appendix C

Introducción

Actualmente, la industria de la telecomunicación es un mercado abierto en el que numerosas empresas compiten por el liderazgo; es por tanto un requisito obligatorio para las compañías que compiten en este tipo de mercados ofrecer una alta calidad y confiabilidad en sus productos para mantener su posición dentro del mismo. Más específicamente, este objetivo es obligatorio para las compañías dedicadas al desarrollo de software debido a que un software de alta calidad reduce la cantidad de recursos necesarios para su mantenimiento y ayuda a la compañía a obtener una porción mayor del mercado. Consecuentemente, managers relacionados con el desarrollo y el mantenimiento de software han decidido apostar por mejorar el proceso de desarrollo del software en pos de obtener una mayor calidad en su software con un coste más reducido [10].

Este ciclo de vida o proceso del desarrollo de software es emulado cada vez que una compañía decide crear un nuevo producto software o mejorar uno ya existente. Este proceso consiste, de al menos, los siguientes pasos: especificación, diseño, desarrollo o implementación, prueba o validación, documentación, entrega, y mantenimiento [31]. El modelo en cascada de este proceso puede verse en la Figura 1.1. Por simplicidad en este projecto se explica el modelo en casacada, pero el alcance de este proyecto es también aplicable a otros modelos de desarrollo software. Estos pasos son:

Especificación Los clientes determinan los requerimientos y el propósito del software

Diseño Los responsables del proyecto en conjunto con los programadores definen la metodología y el diseño del producto.

Desarrollo Los programadores desarrollan el nuevo software.

Prueba Los programas son probados en busca de cualquier posible error que pueda producir fallos en el sistema. Esta etapa suele ser la que más recursos consume.

Documentación y Mantenimiento En esta etapa los programas son documentados y comienza el proceso de mantenimiento. En este proceso se supervisa el correcto funcionamiento del producto; cuando es necesario se corrigen errores o/y se implementan mejoras.

C.1. Descripción del problema

El proceso de prueba está diseñado para asegurar la calidad de los productos software, pero es el proceso que más recursos consume en términos de tiempo, esfuerzo, y coste. Esta actividad supone entre un 50 y un 70 por ciento de los costes totales del proyecto [21]. Los errores humanos y los malentendidos entre el hombre y la máquina son comunes en la programación. Estos errores son producidos durante la implementación del código, pero no son descubiertos hasta que éste es probado. Una vez descubiertos, estos errores obligan a los desarrolladores a implementar código nuevo o a rescribir parte del código, lo que puede a su vez introducir nuevos errores convirtiendo este proceso en un proceso exponencial. El esfuerzo necesario para corregir estos errores mientras el código está siendo implementado es mucho menor que el necesario para corregirlos cuando el producto está siendo probado; en el peor de los casos son los usuarios finales lo que detectan estos errores en tiempo de ejecución. En muchos casos los defectos introducidos por estos errores no se pueden resolver y es necesaria la creación de una nueva versión del código para eliminarlos. En ambos escenarios, la compañía gasta una gran cantidad de recursos en corregir los errores. Es fácil comprobar por tanto que cuánto antes un error es encontrado y corregido, menor es el coste total. Así mismo, incrementaría la satisfacción de los consumidores y su confianza en la empresa y sus productos [18]. En consecuencia numerosas compañías han decidido invertir en nuevas formas de detección y corrección de errores en las primeras etapas del proceso de desarrollo [18], por ejemplo el adoptando métodos ágiles, haciendo uso de especificaciones formales u otras técnicas.

C.2. Objetivos

El objetivo principal de este proyecto es mejorar la calidad del software. Para llevar a cabo esta mejora, este proyecto se centrará en la predicción de la presencia de errores en las primeras etapas del proceso de desarrollo. Así, en este proyecto se creará une herramienta capaz de predecir la existencia de errores en un código software basándose en sus métricas. Esta predicción facilitará la detección de los módulos más propensos a tener errores en un programa *mientras* este está siendo desarrollado. Más específicamente, esta herramienta permitirá al usuario a:

- corregir errores durante la etapa de desarrollo que de otra forma no serían

encontrados hasta la etapa de pruebas (esto sería posible mediante la refactorización del código fuente);

- conocer que módulos deben ser probados más cuidadosamente y cuales no; y
- aprender que estilos de programación son menos propensos a errores a través de los resultados de la herramienta.

C.3. Estructura

El capítulo 1 introduce los objetivos y la motivación del proyecto.

El capítulo 2 presenta los antecedentes y conocimientos necesarios para la lectura y entendimiento del proyecto. Este capítulo comienza con algunas definiciones básicas de la programación orientada a objetos, continúa con la descripción de las métricas del software más utilizadas, y concluye con una descripción de algunos de los modelos de regresión más empleados centrándose en los modelos híbridos neuro-difusos, en las redes neuronales y en las técnicas de agrupamiento.

El capítulo 3 ofrece una visión general del modelo utilizado para la creación de la herramienta. Este capítulo comienza con una breve explicación de las especificaciones de la herramienta y continúa con la selección de los programas, técnicas, y modelos empleados en este proyecto para relacionar las métricas del software y los errores.

El capítulo 4 describe en detalle la implementación del modelo híbrido neuro-difuso. El capítulo finaliza analizando los resultados del modelo y su calidad.

El capítulo 5 explica las funcionalidades de la herramienta y los resultados del uso de esta herramienta.

El capítulo 6 presenta las conclusiones del proyecto.

El capítulo 7 propone trabajo futuro para mejorar la herramienta.

Finalmente los apéndices A y B contienen información más detallada de la herramienta.

Appendix D

Conclusiones

En este proyecto se ha creado un programa basado en un modelo híbrido neuro-difuso que predice el número de errores de un código fuente. Este modelo está compuesto por una etapa de agrupación seguida de una red neuronal de función de base radial por lo que es muy dependiente del conjunto de datos usado para entrenarlo. Por ello, grandes bancos de datos provenientes de diferentes códigos fuente deben ser utilizados para alcanzar un elevado nivel de precisión. Debido a la falta de muestras independientes en el conjunto de datos utilizado en este proyecto, el resultado del análisis del programa ha concluido que este no ha alcanzado alta precisión en sus predicciones. Se deja por tanto en manos del usuario, utilizar las funciones de la herramienta para simular un nuevo modelo y entrenarlo con grandes conjuntos de datos.

En cuanto a los experimentos realizados en este proyecto, los resultados han mostrado que una elección incorrecta del número de grupos en el proceso de agrupación produce grandes alteraciones en los resultados de la RBFNN, siendo el índice de partición el mejor indicador de este valor. A su vez, estos resultados mostraron que valores del exponente de difusión cercanos a 2 (que como se vio previamente es valor más utilizado en estudios previos) resultaban en mejores predicciones. Malas inicializaciones en el proceso de agrupación, y por tanto, en la red neuronal, conducen a agrupaciones sub-óptimas y a predicciones poco fiables. Esta sensibilidad a la inicialización se ve incrementada cuando el conjunto de datos no es lo suficientemente grande.

Por último, los experimentos han mostrado que los modelos de clasificación hacen mejores predicciones que los modelos de regresión debido a la precisión requerida para clasificar una muestra como errónea o no en ambos casos.

TRITA-ICT-EX-2013:52

www.kth.se