SOFTWARE RELIABILITY PREDICTION OF MOBILE APPLICATIONS USING MACHINE LEARNING TECHNIQUES

By

SUMAYA HOOSEN

(20928758)

Submitted in fulfilment of the requirements for the Degree of

MASTER OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

in the

DEPARTMENT OF INFORMATION TECHNOLOGY

in the

FACULTY OF ACCOUNTING AND INFORMATICS

April 2021

DECLARATION

I, Sumaya Hoosen, declare that this dissertation is a representation of my own work both in conception and execution. This work has not been submitted in any form for another degree at any university or institution of higher learning. All information cited from published or unpublished works have been acknowledged.

30 April 2021

Sumaya Hoosen

Approved for final submission

30 April 2021

Supervisor
Dr Alveen Singh (D.Tech:IT)

ACKNOWLEDGEMENTS

At the outset, a special thank you and appreciation goes out to Dr Alveen Singh, my supervisor, for his invaluable support and contribution to this dissertation. Without his guidance, this would not have been possible.

I would also like to express my gratitude to my closest family and friends for their support and encouragement in helping me achieve this goal.

A very special thank you goes out to my mum for her unconditional love, strength and support throughout this journey.

ABSTRACT

Software reliability is an important aspect for evaluating the quality of a software product. In a growing global software industry of increasingly complex systems, reliability becomes crucial urging software engineers to strive toward the development of failure free software and to ensure high reliability before delivery. This positions software reliability as one of the key attributes required to achieve high quality software products. In response to this stature, software companies invest considerable resources boosting apps development into a multi-billion Rand global industry. In recent times smart devices are established as one of the most used electronic device with apps being the more popular medium for bringing a multitude of functionalities to a wide user base. However, current literature portrays a far from ideal reliability rate for apps. Despite the availability of a wide range of approaches focused on improved reliability these mostly remain cumbersome and costly to implement from a software management perspective. Hence, there is a need to investigate approaches beyond current dominant thinking that underpins reliability measurements in the mobile app development space. At the same time, Machine Learning (ML) is a recent recipient of much attention from researchers and practitioners that offers a bouquet of tools and techniques that when applied correctly could potentially improve reliability prediction. In line with the above, the overall aim of this study is to provide a ML modelling approach to assist with the reliability prediction of mobile apps. It is hoped that the findings of this study may provide a useful ML modelling approach to help developers increase the reliability rates of apps. For this study ML techniques were applied to 3 feature sets of data extracted from the Eclipse JDT core dataset. These feature sets based on software systems and their histories, include the source code metrics set, process metric sets, and a combination of both metric sets. All metric sets went through stages of data cleaning and pre-processing before they were modelled using five machine learning algorithms, namely, Random Forest, Support Vector Machine, Naïve Bayes, Decision Trees and Neural Networks. During the modelling process, all the results were evaluated using ML evaluation scores to determine which ML modelling approach is most useful for reliability prediction. The results indicate that Random Forest generated better results in all cases and can be used for predicting app reliability since it predicted reliability more accurately and precisely compared to the other ML algorithms. Random Forest also achieved the highest evaluation score when it was applied to the combined metric set of data. This means that the modelling approach of applying Random Forest to a combination of source code and process metrics generated the highest prediction performance.

This further implies that developers should consider these selected features within the combined metric set, as they could serve as useful indicators for predicting reliability of apps.

Keywords: Software reliability, Mobile apps, Machine Learning, Reliability prediction, Machine learning algorithms, Feature Selection

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE SNIPPETS

ACRONYMS AND ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
AUC	Area Under Curve
CK	Chidamber and Kemerer
DT	Decision Tree
EA	Experimental Algorithmics
GP	Gaussian Process
Apps	Mobile Applications
MAD	Mobile Application Development
MADLC	Mobile Application Development Life Cycle
ML	Machine Learning
NB	Naïve Bayes
NN	Neural Network
OO	Object Oriented
OS	Operating System
RF	Random Forest
RFC	Random Forest Classifier
ROC	Receiver Operating Characteristic
SFS	Sequential Forward Selection
SRGM	Software Reliability Growth Models
SVM	Support Vector Machine

CHAPTER 1: INTRODUCTION

This dissertation applies Machine Learning (ML) techniques for the reliability prediction of mobile applications (apps). This chapter provides an overview of the material covered in the dissertation. It begins with a discussion of the background context related to mobile application reliability and ML. It then provides an overview of the research problem, followed by the research aims and objectives, the research process and the significance and contribution of the study. This chapter concludes with a brief outline regarding the organisation of the rest of the dissertation.

1.1 Background

Recent decades have shown a significant development of mobile apps. The scale and complexity of these apps have grown and will continue to do so in the future (Tian, Xiang, Zhenxiao, Yi, & Yunqiang, 2019). The popularity of apps can be seen permeating across various fields of academia, industries and organisations due to their dependability in a wide range of uses and through services provided. Recent market research shows that for the Google platform of Android based apps, there are over 1.5 million apps that exist within 42 different categories (Prashanth & Premaratne, 2020). The high demand for apps requires rapid development of complex functionality. This places exorbitant demands on mobile app development teams who need to deliver high quality software products within limited project constraints (Pandey, Litoriya, & Pandey, 2020).

Although there are several ways to define software quality, software reliability emerges as an important quality attribute. Software reliability aids in measuring the correctness of software, with the process of reliability engineering focussing on analysing and measuring the quality of software. As defined in literature, a reliable software product is one that is able to execute without producing defects (Barack & Huang, 2020; Wang & Zhang, 2018). The nature of mobile apps, in terms of their diverse functionalities and features, is very different from that of server and desktop based applications. This difference requires that developers spend more time and effort to ensure mobile app reliability. Unfortunately, due to the high demand for apps, ensuring such reliability poses a huge challenge. Even though there are several ways to define software reliability, an app with many defects is perceived as an unreliable product. Defects are described as bugs or programming errors that can negatively affect the reliability of a software product, with corresponding costs and efforts to resolve the issues. Therefore, the early prediction of software defects, also known as Software Defect Prediction, becomes an

essential aspect of software engineering (Alsaeedi & Khan, 2019). Software defect prediction is intended to assist the development team to produce high quality, reliable apps within a limited timeframe, before being released (Iqbal et al., 2019; Tian, Xiang, Zhenxiao, Yi, & Yunqiang, 2019). Over the years many software reliability models have been developed and successfully applied to desktop apps but were not as effective with mobile apps (Li & Pham, 2019). It appears that the dynamic and unstable nature of mobile apps cannot be accommodated by these reliability models. As a result, literature consistently calls for new models, tools and techniques to improve on the overall reliability in the mobile app development space (Barack & Huang, 2020).

There has been a growing interest in the application of ML in various aspects of software engineering to assist developers and engineers to obtain reliable software, especially in the area of apps development (Durelli et al., 2019; Wang & Zhang, 2018). ML, a branch of Artificial Intelligence (AI), has emerged as an efficient approach for building predictive models (Antonopoulos et al., 2020). Through the use of statistical techniques, ML is able to extract hidden patterns from data and map this to some predicted outcome and thus serves as an effective way for predicting defects in software (Iqbal et al., 2019). Despite sustained research in this area there still are unresolved challenges for app reliability with no "one size fits all approach" and ML research continues to evolve (Tian, Xiang, Zhenxiao, Yi, & Yunqiang, 2019). This study adopts the standpoint that the application of ML techniques to app development can significantly contribute towards the early reliability prediction of apps, and thereby prevent the related consequences.

1.2 Research problem

In order to generate high revenue levels and remain competitive, apps need to be developed to satisfy a wide range of users in the app market (Pandey, Litoriya, & Pandey, 2020). The fast-paced industry of mobile or smartphone development and related apps' innovations are forcing developers to focus on stability. A further complexity is that modern apps need to run on different operating systems' platforms to ensure interoperability. Apps that are unable to exhibit characteristics of reliability, scalability and interoperability impact negatively on user experience (Xavier, Matteussi, França, Pereira, & De Rose, 2017).

It is reported that a consequence of the aggressive *time to market* strategy of developing apps is a decline in overall quality (Ławrynowicz, Legut, & Jóźwiak, 2020). This is mostly attributed to developers reducing certain time and resource extensive stages, like testing, in order to stay

within cost, time and resource constraints of the project. Mobile apps are versatile, dynamic and have varying execution conditions. As a result, various popular software reliability approaches fall short of effectively evaluating the reliability of smartphone apps (Martinez & Lecomte, 2017).

Another major challenge for app development is the high number and the diversity of mobile platforms. An app can only be developed for one platform at a time (Li & Pham, 2019), which implies that there cannot be a high rate of code reuse. The challenge of monitoring, analysis and testing support for apps poses another difficult task for developers. There are limited automated testing tools available and most of the current tools do not support mobile features such as mobility, location services, sensors and various gestures and inputs (Ahmad, Feng, Tao, Yousif, & Ge, 2017). Further challenges are that of inconsistency across platforms, the API, usability and handling changes and crashes (Cruz, Abreu, & Lo, 2019).

In summary, existing reliability measures seem ill-equipped to cater for the complexities inherent in apps development. Older, well ingrained approaches may not be ideal due to the complexities of Mobile Application Development (MAD) and as a result the inherent quality of smartphone apps is being compromised. Hence, there is a need for a new approach to be considered for mobile app reliability.

1.3 Research aim and objectives

Software reliability is an important and indispensable part of software quality with an increasing need to develop highly reliable software (Barack & Huang, 2020; Arunima Jaiswal & Ruchika Malhotra, 2018; Li & Pham, 2019). The rapid evolution of mobile app development portrays mobile app reliability as a growing area of interest for practitioners and academics and remains a challenge (Ahmad, Feng, Tao, Yousif, & Ge, 2017; Biørn-Hansen, Grønli, & Ghinea, 2018). ML techniques have shown favourable results with their ability to capture complexities of software reliability behaviour through data driven learning methods to train software and make generalised predictions (Boutaba et al., 2018; Diwaker, Tomar, Poonia, & Singh, 2018; Yang, Chen, Hu, & Deng, 2017a).

This study aims to provide a ML modelling approach for predicting the reliability of mobile apps. This research is guided by the following research question.

Research Question: Can ML assist in identifying useful predictors for reliability of mobile apps?

Research Objectives:

RO1: To review the challenges of existing software reliability models for the reliability prediction of apps

RO2: To identify suitable ML modelling techniques for app reliability prediction

RO3: To architect and conduct an experiment to compare ML modelling approaches

RO4: To evaluate the prediction performance of ML modelling approaches applied in the study

1.4 Overview of the Research Process

A quantitative, Experimental Algorithmics (EA) approach was adopted for this study. In quantitative research, numerical data is collected and analysed using mathematically based methods (Creswell, 2003). EA refers to the study of algorithms and their design, for faster and better problem solving (McGeoch, 2012). Both these approaches are suited for this study that applies ML to app reliability prediction.

All data for the study is extracted of a numerical dataset of software systems in terms of software features and their measures, known as software metrics. These metrics are categorised into 3 datasets of source code, process and a combination of both metric sets. Different ML algorithms are applied to each metric set and results compared, to determine the algorithm and the features most useful for reliability prediction. ML techniques applied to this study, all follow multiple computational methods, and apply statistical dependencies of variables and probabilities to select these relevant features of data, motivated through this quantitative approach. ML algorithms are applied and evaluated to provide further insights into selecting these app reliability indicators prescribed through the EA approach. All the data analysis and observations are estimated using statistical models and mathematical functions. There are no participants used in this study.

The research paradigm selected for this study is post-positivism. Post-positivism acknowledges and understands that all findings contribute to the development of knowledge (Fischer, 1998). The knowledge that develops through this lens is based on careful pattern recognition and measurements of features in data through ML techniques. Possible solutions are intended to provide some relief to the challenges faced by developers in app development field, thus providing some contribution to this body of knowledge. This study uses multiple ML techniques to explore all angles of data and provide a more flexible approach to maximise and

improve app reliability rates. Such flexibility is prescribed through post-positivist research. Another characteristic of post-positivism states that the theory and practice cannot be compartmentalised just for collecting the facts. Both aspects need to be considered. This study adopts an extensive literature review into the fields of software engineering in app development to gain sufficient knowledge to help select suitable ML techniques for the research process. These techniques are then experimentally applied to different feature sets of data to help identify useful app predictors for reliability prediction. All these adopted approaches guide the research process for this study which aims to determine a modelling approach most suited for the reliability prediction of apps.

1.5 Significance of the study

a) This study is of significance in the field of software engineering as it extends the knowledge base that presently exists in this field. Apps play a significant role in personal, business, educational and public communication (Al-Janabi, Al-Shourbaji, Shojafar, & Abdelhag, 2017). Software reliability is an important characteristic and a key component of software quality. (Mohsin, Naqvi, Khan, Naeem, & AsadUllah, 2017). According to the International Standards Organisation (ISO) 8402, quality is defined as: "The totality of features and characteristics of a product or service that bears on its ability to satisfy specified or implied needs" (Glass, 1998, p. 103). Some recent studies set out to improve on the different ambits of app reliability which is an indicator of growing interest in the field. However, currently, the literature signals a far from ideal reliability rate for apps. Some approaches are limited in that they are cumbersome and costly to implement from a project development perspective.
b) In light of the above, this study presents a different approach to help developers with the reliability prediction of apps. This study proposes the use of ML techniques applied to different feature sets of data to determine the ML modelling approach that is most suited for the reliability prediction of apps.
c) The study also identifies some major challenges experienced by developers in app development in an attempt to alleviate the impact of these challenges through applying this ML approach.
d) The results of the study provide useful information to help and guide developers increase the reliability rates of apps through the use of ML.
e) The findings are also expected to provide value for future ML research in which researchers may want to improve on the accuracy scores obtained for prediction.

Therefore, the findings of this study can provide a foundation to assist both the developers and researchers interested in improving the reliability rates of mobile apps.

1.6 Contribution of the study

The intended contribution of this study is to provide a ML modelling approach to assist with the reliability prediction of apps. The research attempts to add significantly to the knowledge and practice in the area of software engineering and ML, specifically to app development by shedding more light on the factors that have been identified as having an effect on app development in literature. The contributions of this study to the body of knowledge and to practice are listed below:

a) Identifying some major challenges of within MAD.
b) Reviewing the effectiveness of existing software reliability models.
c) The analysis and synthesis of selecting ML algorithms and defining useful software metrics for app reliability prediction.
d) Applying and evaluating ML techniques for the reliability prediction of apps.

This study proposes a ML modelling approach that can help developers early predict the reliability of apps. The findings in this study can be easily generalised to the field of MAD, providing developers with useful reliability indicators to assist in predicting app reliability.

1.7 Organisation of this dissertation

This dissertation is organised into six chapters, as follows:

Chapter 1: Introduction

This chapter starts by presenting the background to the study, followed by the research aim, objectives and the research question. It goes on to provide an overview of the research process applied followed by the significance of the research, contributions and the structure of the study.

Chapter 2: Literature Review

This chapter provides a literature review in terms of the definitions and perspectives of software reliability, mobile apps and ML. It reviews and interprets current literature in an attempt to clearly define aspects relating to software reliability, mobile apps, mobile app development, challenges of mobile app development, existing reliability models and applications of ML for software reliability prediction.

Chapter 3: Research Methodology

This chapter presents the methodology adopted for the execution of the research. This includes the research design aspects of the research paradigm, strategy and the methodology. It then progresses to describe all the components applied in conducting the experiments for the study which includes the dataset, software metrics, feature selection, ML algorithms and evaluations.

Chapter 4: Description of the experiment

This chapter discusses the actual administration of the experiments for the study. Experiments were conducted using the Python programming language. All the steps of the ML process were applied to the experiments that include the data collection, data cleaning and pre-processing steps, dimensionality reduction and feature selection, ML modelling and evaluation. The results at each stage of the experiments were presented in this chapter.

Chapter 5: Discussion of the results and findings

This chapter discusses all the findings obtained after conducting the experiments. It summarises the methods applied within the experiment and the output that was generated during this process. All results are discussed in detail and the overall findings presented.

Chapter 6: Conclusion

This chapter provides a summary of the study with contributions, observations and limitations of the study. It revisits the research aims and objectives and provides recommendations for further research in this area.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

This chapter presents a critical review of relevant literature for this study. These include the main theoretical concepts, definitions and key characteristics regarding apps, MAD and ML. This chapter also identifies aspects within the ML process required for the development of a prediction system such as the ML algorithms, software metrics and ML evaluation measures. Section 2.2 and 2.3 defines software reliability and apps. Section 2.4 reviews MAD in terms of its development life cycle, the software engineering aspects of its development, and the development challenges experienced. Section 2.5 reviews existing software reliability techniques defined as Software Reliability Growth Models (SRGMs) describing how these are used to assess the reliability of software systems. Section 2.6 defines the concept of ML. Section 2.7 reviews ML as a software reliability prediction technique and compares the prediction accuracy of various ML algorithms applied, to help select the better performing algorithms for this study. Section 2.8 further describes each selected algorithm in terms of its design and capabilities. Section 2.9 provides a definition and review of software metrics applied in literature as software reliability indicators to help identify appropriate metrics for this study. Section 2.10 describes ML evaluation measures that are used to evaluate the performance of ML techniques.

2.2 A definition of software reliability

This section provides a definition of software reliability and its importance in the software industry. Software reliability is described as a property that measures how software correctly and continuously performs its service without any faults or failures (Okamura & Dohi, 2015). Software reliability is of vital concern given the dependency of society and industry on a multitude of software products (Park, Lee, & Baik, 2014). Reliability is more often than not considered one of the main attributes required to achieve exact and decisive software products (Tariq et al., 2018).

In recent times, the software industry has been faced with many challenges in the development of highly reliable software. Reliability forms one of the most crucial aspects of software development. It affects the quality of software and is defined as the probability of software being failure free for a specific time period in which defects are found and removed (Utkin & Coolen, 2018). This differs from hardware reliability where the focus is on ensuring robust manufacturing processes. Reliability together with aspects of performance, usability and fault

prediction all form important attributes of software quality (Capretz, Meskini, & Bou, 2013). Other software quality attributes include availability, interoperability, maintainability, manageability, performance and reusability, with reliability being one of the major quality attributes (Rao, Sahitha, Sireesha, & Manoj, 2018). In summary of this, software reliability can therefore be defined as a measure of a software product's ability to achieve its full desired functionality, without the existence of any defects, failures or bugs.

2.3 Mobile applications: a definition

This section defines apps in terms of their categories, significance, the effects of their constant evolution and their popularity. Mobile apps are defined as software programs that are designed to run on mobile devices (Zein, Salleh, & Grundy, 2016). Apps have become an important aspect of people's lives, playing a significant roles in personal, business and public communication (Al-Janabi, Al-Shourbaji, Shojafar, & Abdelhag, 2017), with many software companies investing considerably in app development whilst targeting various available opportunities in the app market (Yang, Chen, Hu, & Deng, 2017b). The development of apps is increasingly evolving with millions of apps being downloaded and used across the globe. The Google Play Store and the Apple App Store form the leading platforms for app distribution with more than 1.5 million apps in the former and more than one million apps in the latter (Prashanth & Premaratne, 2020).

This evolution is attributed to their popularity and ubiquity among the end users. These apps are not only for entertainment but they are also developed to serve more critical areas such as payment systems, the military and many other critical sectors, thus increasing the development complexity (Zein, Salleh, & Grundy, 2016). Falling under the ambit of ubiquitous computing, apps have been responsible for changing the ways in which users interact with technology daily. Ubiquitous apps to be able to work anywhere and at any time (Weichbroth, 2020). In 1991, ubiquitous computing was identified as the main computing technology for the 21st century. (Weiser, 1991, p. 94) asserted,

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

Now, almost three decades later, apps have dynamically changed the computing environment with regards to its users, devices and software components, allowing systems to integrate and adapt to the physical world (Carvalho, de Castro Andrade, & de Oliveira, 2018). Apps continue to allow for the development and offering of powerful, ubiquitous service delivery for a variety

of products and services to users. The estimated number of smartphone users is expected to grow to an approximate figure of 5.5 billion users globally by the year 2022 (McLean, Al-Nabhani, & Wilson, 2018).

Apps are intended to serve users with functionality similar to those offered through software systems in personal computers (PCs). PC software systems offer an integrated functionality which differs very much from app functionality. Apps provide isolated functionality in which an app may focus on either areas of gaming, browsing or productivity(Fasano, Martinelli, Mercaldo, & Santone, 2018).

Therefore, apps are computer programs designed to run on mobile devices. Due to their popularity and ubiquity, mobile apps serve growing industries all over the world.

2.4 Mobile application development

This section discusses aspects of MAD, in terms of its design and processes as compared to traditional desktop applications. MAD refers to the development of apps developed for mobile devices and offer specific services to them. Apps are developed for mobile platforms, including both the hardware and software environment for service delivery (Mushtaq, Kirmani, & Saif, 2016). MAD has greatly evolved over the years mainly through ubiquity and popularity as millions of apps are downloaded across the world. Due to this, fields of software engineering are required to adopt new approaches to MAD (Fathi, Hashim, Ibrahim, & Hassan, 2017).

MAD differs from traditional software development in which the delivery time of apps is very short with frequent changes due to expectations and requirements of the users and the app market. The app industry, therefore, requires an upgrade from the traditional software engineering processes and techniques (Patel & Patel, 2017). MAD requires the creation of a more assessable user interface, an architecture for dealing with multiple platforms, context design and more clearly defined requirements. Organisations are constantly faced with making decisions to utilise favourable strategies to serve and satisfy user requirements. This rapid growth and development of apps have a direct impact on its development life cycle in its format and design (Anwar et al., 2020).

2.4.1 Mobile application development life cycle

This section highlights the six phases of the mobile applications development life cycle (MADLC) model as the identification phase, design phase, development phase, prototyping phase, testing phase and maintenance phase.

MAD is referred to as a special case of software development as various aspects need to be considered in terms of the relatively short development cycle. Each phase of the MADLC is explained below (Aldayel & Alnafjan, 2017; Ehrler, Lovis, & Blondon, 2019; Fathi, Hashim, Ibrahim, & Hassan, 2017).

1. Identification Phase

   The main objectives for the app are defined at this phase. The idea usually comes from the users and the developers.
2. Design Phase

   At this phase the feasibility of developing the app is determined and the target platform is identified.
3. Development Phase

   This is where the application is coded for functionality and the User Interface (UI).
4. Prototyping Phase

   The functional requirements of the app for each prototype is analysed and tested and sent to the client for feedback.
5. Testing Phase

   During this phase the prototype is applied and tested on a real device.
6. Maintenance Phase

   In this final phase of the MADLC, feedback is collected from users and changes are made to fix bugs to improve apps.

Global competitive app markets call for manufacturers to develop software products within shorter timeframes while striving for higher customer satisfaction, thus causing a vested interest in software reliability (Alsina, Chica, Trawiński, & Regattieri, 2018). Increase in the pressure for time to market also increases the complexity of software projects which in turn jeopardises quality (Jayanthi & Florence, 2018). These growing complexities and dependencies of software have led to an increased need to deliver high quality software at a lower cost (Malhotra, 2015). Apps are versatile, dynamic and have varying execution conditions when compared to desktop apps. As a result, existing approaches fail to evaluate the reliability of apps adequately (Meskini, Nassif, & Capretz, 2013). Figure 2.1 below summarises each phase of the MADLC.

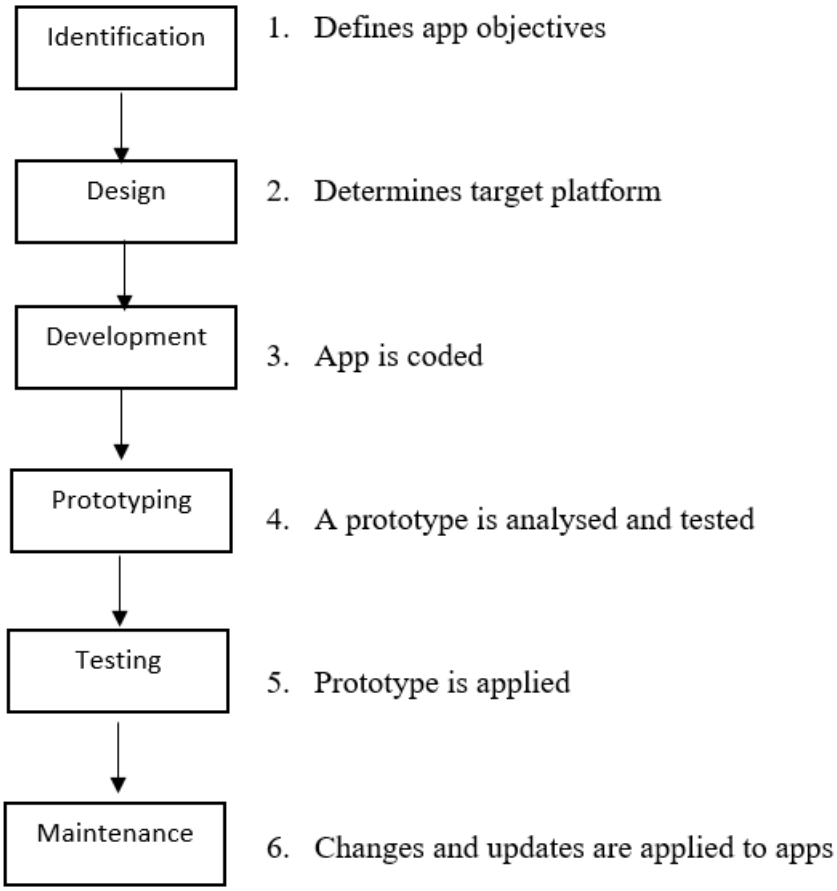


Figure 2.1 The six phases of MADLC

Therefore, the development of apps is shown to be relatively different to the development of traditional desktop software, due to app popularity and ubiquity. Software industries constantly call for upgrades in development processes and techniques. The MADLC includes the implementation of a very brief development cycle that requires app development to occur within very short timeframes to keep up with the dynamics of apps in the app markets.

2.4.2 Software engineering for mobile application development

This section defines how apps are developed and implemented. Over the years, the demand for apps has caused an increase in the development of projects for these apps. As a result, both the quantity and quality of these apps have created great concerns for the software engineering industry (Aldayel & Alnafjan, 2017). Apps are developed in three ways: native, web and hybrid apps (Ahmad, Feng, Tao, Yousif, & Ge, 2017; Patel & Patel, 2017). Native apps are platform dedicated, in that the apps are implemented for a certain OS platform using SDKs and platform tools. This type of development limits the number of mobile devices using the app but benefits from the platform features (Ali & Mesbah, 2016).

Web apps run on web services and are implemented through web technologies such as Hypertext Mark-up Language, Cascading Style Sheet and JavaScript. Web apps are portable across different platforms, thus reducing development costs and time. However, these apps cannot utilise device capabilities nor can they work offline. Hybrid apps combine the advantages of both native and web apps. These are also available within the browser control and have access to the device's hardware features. They are also available for download through a distribution store (Joorabchi, Mesbah, & Kruchten, 2013).

The common practice for industry is to build native mobile apps. These native apps are built for a specific platform, which means that the same app cannot be implemented on another platform (Fasano, Martinelli, Mercaldo, & Santone, 2018). This poses a huge challenge for developers due to the increasing number of diverse platforms and implies that these apps need to be independently developed across each target platform. Many other challenges regarding MAD are discussed below.

2.4.3 Challenges of mobile application development

This section identifies some major challenges experienced by developers in the development of apps. These challenges include: fragmentation, stability and interoperability, monitoring analysis and testing support, change management and keeping up with frequent changes, user interface design, re-use of code and the lack of resources and tools support. Each challenge is discussed below.

The first major challenge for MAD is the number and the diversity of mobile platforms, as apps can only be compiled for one platform at a time. This means that there cannot be a significant re-use of code but the same app needs to be re-designed from the start. This is referred to as fragmentation and is deemed one of the most reported challenges in literature (Ahmad, Feng, Tao, Yousif, & Ge, 2017). Such fragmentation not only occurs across platforms but also within the same platform. Fragmentation across platforms shows that each mobile platform is different in terms of the user interface, experience, expectations, interactions, programming languages, API/SDK and supported tools. Within the same platform there are issues of device memory, CPU speed, screen sizes and resolutions. Fragmentation is a challenge for both development and testing as applications need to be tested against different OSs and screen sizes to ensure functionality (Joorabchi, Mesbah, & Kruchten, 2013). Due to the variations in device environments and platforms, these apps may behave differently and in order to ensure reliability they need to be validated (Bhojan, Vivekanandan, Ramyachitra, & Ganesan, 2018).

In order to validate apps, it must be noted that app development is a special case of software development and follows a different development life cycle approach as compared to the traditional approaches (El-Kassas, Abdullah, Yousef, & Wahba, 2017). With these apps being released through app stores and with the competitiveness of the market, it is the task of the developer to constantly release new and improved versions of an app. This may result in poor quality and unreliable apps being produced as a result of not following the prescribed app development life cycle (Xia, Shihab, Kamei, Lo, & Wang, 2016). Therefore, being able to identify defects or problems early during development can help developers to create more reliable apps to ensure a greater app success rate and also maintain their user base.

The second challenge to MAD reliability is ensuring an application maintains stability and interoperability. These two properties also contribute to the overall reliability concept such that apps that are unable to ensure stability and interoperability impact negatively on the user experience (Xavier, Matteussi, França, Pereira, & De Rose, 2017). Stability in software engineering is a software property that refers to long term characteristics that provide good support to the design and implementation of a product, to allow it to withstand environmental changes (Salama, Bahsoon, & Lago, 2019). It forms an important aspect for evaluating product reliability and is defined as the ability of an application to function and remain unchanged despite exterior changes, such as, multiple OS platforms and new business requirements (Goyal & Srivastava, 2018). Stability poses a huge challenge for apps due to the diverse number of mobile platforms that currently exist. Organisations are required to deliver the same apps across platforms separately, to ensure consistent app functionality (Ahmad, Feng, Tao, Yousif, & Ge, 2017). This issue of stability also feeds into another reliability concept known as interoperability.

Interoperability of software systems refers to the ability of systems to work with other products and systems without encountering any problems or restrictions. It is the measure of the capabilities of different applications to exchange data using common exchange formats and protocols (Duffy & Katajamäki, 2018). The basic software system layers are made up of the OS, libraries and applications. The capabilities of smartphones together with the various user-centred apps need to form a reliable system that runs and performs a given set of instructions in a timely manner with an efficient use of resources to adequately complete a task. In order to provide such reliability, the system needs to be assessed *holistically* and this includes all these system's layers mentioned above. In order for apps to benefit from the platform level reliability

mentioned above, they need to be designed and programmed in a way to ensure users will benefit from this (Dantu, Ko, & Ziarek, 2017).

A third challenge is that of monitoring, analysis and testing support for native applications, which poses another difficult task for developers. There are limited automated testing tools available. The current tools do not support mobile features such as mobility, location services and sensors, as well as various gestures and inputs. A testing issue is that of consistency across platforms, API, usability and handling changes and crashes (Ahmad, Feng, Tao, Yousif, & Ge, 2017; Joorabchi, Mesbah, & Kruchten, 2013). Existing approaches seem mainly suited for desktop apps and fail to evaluate the reliability of mobile apps due to their versatility and dynamic configuration.

A fourth challenge is change management in which developers need to keep up with frequent requests for change from customers. These apps in themselves are challenging enough due to the multiple platform environments. With these changes, either functional or not, developers are challenged to make these app changes and apply them across all the different platforms. Developers are therefore required to familiarise themselves and keep up to date with more programming languages and application programming interfaces (APIs) from the different platform environments. This implies extra efforts, costs and constant development of expertise in these areas for the developers (Joorabchi, 2016). Mobile devices, applications and their associated services are constantly changing thus affecting businesses strategies and models, user behaviour and the way employees work in corporate organisations (Lachgar & Abdali, 2017). These changes to apps' development, especially in the critical domains, require robust testing techniques in order to build high quality and more reliable apps (Zein, Salleh, & Grundy, 2016). The problems of quality and behaviour predictions of such apps are of utmost importance in assessing reliability (Ivanov, Reznik, & Succi, 2018). Smartphones with mobile apps offer very advanced computing and connectivity in terms of processors, memory, high resolution screens, rich sensors, as well as high speed Wi-Fi. Compared to personal computers, desktops and web applications, smartphone resources are still limited due to the issues of diverse operating systems and their need for frequent upgrades that pose constant challenges to mobile apps. These challenges create complexities not only in application development but also in application testing and validation (Zein, Salleh, & Grundy, 2016).

A fifth challenge is the user interface design. This poses a huge challenge for developers to ensure maximum use of the limited screen size of mobile devices positioning this as an

extremely important aspect of app usage. Traditional applications offer the best user interfaces but these need to be redesigned to focus on the more common functionalities within mobile apps. The mobile user interface needs to satisfy both the user input and the location and motion information of the device (Mushtaq, Kirmani, & Saif, 2016). Studies estimate that a total of 2 billion smartphones are in use globally and a 50% increase is expected to occur over the next five years (Martinez & Lecomte, 2017). It was in 2011 that smartphones overtook traditional computers in sales (Orsini, Bade, & Lamersdorf, 2015). With smartphones portrayed as one of the most used electronic devices, apps of the highest quality standards need to be designed to suite the mobile user. Poor reliability in these apps can be damaging to the company's reputation, especially if not detected early and resolved. The intense evolution in mobile device capabilities spurred on by competition among manufacturers, has focused developers' attention on app reliability. This rapid growth has also caused the app markets to become increasingly competitive and hence require different techniques and tools within the developmental life cycle of the application (Amalfitano et al., 2015). The app market demands high quality products that meet the user requirements adequately to succeed in this highly competitive domain, thus generating higher revenues for software companies (Corral & Fronza, 2015). Currently there are hundreds of different types or brands of mobile devices with varying software and hardware capabilities. Apps are expected to run on these devices successfully without impairing its existing functionality (Bhojan, Vivekanandan, Ramyachitra, & Ganesan, 2018).

The sixth challenge relates to the reuse of code. Re-using the code of apps across multiple platforms will compromise the functionality and quality of an app. Each platform requires different UIs, APIs and constraints. Hence, developers need to re-write code from the beginning each time, whenever an app needs to be applied to a new platform. This calls for extensive tool support and frameworks for developers to apply and implement (Patel & Patel, 2017).

The seventh challenge refers to the lack of tools and resources. A major challenge posed by developers is that there is an absence of sufficient tools and resources to assist them with the recoding of apps for the different platforms. There is a desperate need for some uniform tool support that can assist developers to adapt apps to new platforms with less effort and also to prevent the need for having to develop expertise across all mobile platforms (Biørn-Hansen, Grønli, & Ghinea, 2018)

With the mass production of software trying to meet the demands of modern technology-based apps, software quality still remains largely an issue requiring more consideration (Manjula & Florence, 2018). With these challenges, it becomes important to be able to find appropriate strategies for managing and coping with such complexities as these add further pressure to reliability constraints. In software engineering, the early detection of defects or bugs in parts of the software has an advantage of allowing developers to manage their time for other areas of the development phases. Also, the cost of correcting defects in software increases exponentially over time. (Malhotra, 2016). Literature advocates the use of ML to create prediction models to detect bugs prior to the release of software, to reduce costs and better allocate resources (Rhmann, Pandey, Ansari, & Pandey, 2020). Therefore, the ability to predict the reliability of an app, before it is deployed, becomes a necessary requirement for the development process. Many issues surface after deployment, creating further complexities. This study proposes the use of ML techniques to attempt to accurately predict the reliability of apps at the early phases of its development which can be adopted prior to deployment.

2.5 Reliability testing techniques: a review of existing reliability models

This section reviews existing software reliability models that have been developed and applied to various aspects of software engineering for assessing the reliability of a software product. These models are known as Software Reliability Growth Models (SRGMs). There are different types of SRGM's and these include, early prediction models, input domain models, architectural models, black box models, white box models.

Over the years, testing techniques for the reliability of software applications have become increasingly vital and complex due to the nature of the mobile app markets (Amalfitano et al., 2015). Recent trends show the importance of testing in terms of performance, reliability and efficiency of a software product in which software is tested with the use of special tools and specific techniques, to ensure it is reliable (Mohsin, Naqvi, Khan, Naeem, & AsadUllah, 2017). Testing techniques are applied through SRGMs and have been successful in improving the reliability of desktop apps (Capretz, Meskini, & Bou, 2013). Various SRGMs have been developed, verified and applied in many software projects to help estimate the reliability of software, using information from the software testing process (Utkin & Coolen, 2018). After exposing software to these various testing techniques, failure rates are minimised. Practitioners estimate the parameters of the SRGM using failure data during testing, and the SRGM uses this data to estimate future failure occurrence times, failure intensity and determine the release time for the software product (Lakshmanan & Ramasamy, 2017).

SRGMs are usually categorised into static and dynamic models in which the dynamic model may be applied to the temporary behaviour of error detection at the testing phase, while the static model may be used to analyse logic on the same code. Both of these models seem to assist in the development of a reliable product within a reasonable time frame (Tariq et al., 2018). SRGMs may be applied quantitatively to reliability by estimating current bugs residing in the system, thus providing an estimated software release date to the market (Okamura & Dohi, 2015). This means that SRGMs are statistically analysed for bug detection and provide future behaviour prediction for these detected bugs. With software reliability being crucial to the measure of software quality, many SRGMs have been applied and provided for by researchers for software reliability measurement. These include: early prediction models, input domain models, architectural models, black box models and white box models (Gupta, Gupta, Garg, & Kumar, 2019; Singh, Verma, & Kumar, 2016). Each model is briefly discussed below.

Early prediction models focus on characteristics right from the outset of software development, starting from the requirements gathering and on to the testing phases of the software development process (Gupta, Sharma, Goyal, & Rashid, 2020). Early prediction models are required to predict the reliability of software early. These models are applied to the software development life cycle models such as the waterfall model, right from the requirement phase. (Özakıncı & Tarhan, 2018).

Input domain based models operate by generating test cases based on variable input distribution that represents the operational usage of software. The input domain is separated into classes and reliability is gathered from observed failures during the execution using test cases (Moharil, Jena, & Thakare, 2019). In other words, reliability is determined based on the input provided to the different modules of the software.

Architectural models estimate software reliability based on the architecture of software. Models are estimated independently and all estimators are combined to provide a full estimation of the whole software system. Many architecturally based models have been proposed to estimate software reliability in the testing phase of its development. The motivation for use of this model is based on an understanding that software reliability is dependent on the architectural components of the system, such as the interfaces and components (Sridhar, 2016).

Black box models estimate reliability based on systems execution and fault data. Generally, these models are applied to the various phases that exist with traditional software development methods. However, black box models are not suited to component based apps. Component

based apps are ones in which the system processes are placed into separate components and work through interoperability between the components (Kaliraj, Vivek, Kannan, Karthick, & Lydia, 2020).

White box models estimate reliability based on the architecture of the system. Unlike black box models, they consider all the internal structures of the system which include component reliabilities. With white box models, all component modules are identified and failure rates and reliabilities for each module and associated components are defined. With white box reliability models, the internal coding structure and the modular operating system interaction is evaluated to estimate the reliability of the entire software product (Pai, 2013).

SRGMs require optimal parameter estimation techniques, failing which creates inaccurate reliability predictions, especially with the more complex software systems (Chaudhary, Agarwal, Rana, & Kumar, 2019). There is also a substantial dependence on time as a modelling factor and these models do not consider the dynamic behaviour of software (Utkin & Coolen, 2018). This is evident with all the models discussed above. All apply testing techniques at various phases of the development process which are estimated over a period of time. These estimations are further strained due to increased system complexities of MAD.

Capretz, Meskini, and Bou (2013) attempted to apply SRGMs to two mobile applications, one for iOS and the other for a Windows mobile phone, to test the applications on different OS platforms. Their findings showed that SRGMs failed to fit all the data and were unsuccessful in predicting the reliability of the mobile apps. Yang, Chen, Hu, and Deng (2017b) proposed SRGMs to evaluate the reliability of mobile systems by using the workload and failure data from the Apache server log of the application. Their findings showed that the SRGMs were too simple to be able to depict the relationship between the workload data and time, to estimate its reliability.

Yazdanbakhsh, Dick, Reay, and Mace (2016) applied SRGMs to open source Android and Mozilla datasets and concluded that these reliability models were not able to account for the chaotic behaviour and nature of mobile data and failed to estimate its reliability. Therefore, even though there have been many models created and used to assess the reliability of software systems, some of these reliability models are successful in certain cases of software development, but struggle with the complexities of app development to accurately predict its reliability. Given the challenges of MAD as discussed in 2.4 above, these models fail to fit its complex and dynamic nature. MAD requires its software to be developed on multiple sites,

with multiple functions and features and therefore, pose a great challenge and impact on the existing SRGM's reliability prediction techniques. Due to this, these SRGM's are inadequate and more specialised tools and techniques are needed to handle the complexities of MAD. This study proposes the use of ML which is discussed in the section below.

2.6 A definition of machine learning

Machine Learning (ML) is a popular sub-field of AI that involves statistical learning for comparisons and matchmaking (Meinke & Bennaceur, 2018). It refers to a set of methods that can be used to predict and improve the behaviours of systems. ML is based in the collection of some historical data without being explicitly programmed (Molnar, 2020). It can be viewed as an automated process that focusses on an output variable and explores and maps patterns within the data to help predict an outcome (Immaculate, Begam, & Floramary, 2019). In other words, ML refers to all the statistical techniques that help build systems that can learn from data. It is an automated process of identifying some meaningful patterns in data, and through the use of ML tools and techniques, programs are *empowered* to learn and adapt on their own (Osisanwo et al., 2017).

There are two main types of ML, i.e. supervised and unsupervised learning. In supervised learning, the learning algorithm is trained using labelled input data. The ML algorithm is then able to learn and make predictions based on this data. There are two types of supervised learning namely, classification and regression. Both techniques are related to prediction: but classification predicts discrete class labels while regression predicts a continuous quantity. In the case of classification, predicting two categorical class labels is referred to as binary classification (Pospieszny, Czarnacka-Chrobot, & Kobylinski, 2018).

In contrast, unsupervised learning algorithms are trained with unlabelled data and the algorithm is required to draw inferences from data into clusters, known as clustering. Clustering is a common example of unsupervised learning (Shanthamallu, Spanias, Tepedelenlioglu, & Stanley, 2017). ML techniques are guided through a learning process of exploratory data analysis of acquiring knowledge and learning from examples similar to humans (Basavaraju & Varde, 2017). Therefore, ML is an automated process that applies statistical techniques to help build ML based models that can learn from data and predict certain outcomes. The purpose of ML is to help improve the behaviours of software systems.

This study uses supervised ML with binary classification. The next section discusses some reasons that point towards the effectiveness of applying ML to serve as an effective software reliability prediction technique.

2.7 Machine learning as a software reliability prediction technique

ML has emerged as a more effective technique to aid in software reliability prediction using models such as artificial neural networks (ANNs) and support vector machines (SVMs) which have shown promising results for predicting reliability from historical data (Alsina, Chica, Trawiński, & Regattieri, 2018). Non-parametric refers to an algorithm that accepts a flexible number of parameters. These algorithms then grow and learn from data. ML algorithms use computational methods allowing them to learn patterns in data. As the number of data samples increase, algorithms are able to improve their performance adaptively.

Locating and detecting a software defect can be a complex undertaking due to the nature and size of the software application. In cases like this a combination of ML and data mining techniques may be effectively implemented (Jayanthi & Florence, 2018). ML models may be defined as data-driven learning techniques that train software and make generalised predictions from historical data (Alsina, Chica, Trawiński, & Regattieri, 2018). ML uses a variety of tasks to process and model data and various algorithms have been designed to support these tasks. These algorithms maintain their own set of requirements for data with varying levels of complexities (Manjula & Florence, 2018).

For bug detection, data mining is a technique that will classify the dataset into faulty and non-faulty datasets as a bug prediction model using classification techniques (Jayanthi & Florence, 2018). The next section looks at how several researchers have implemented ML in their studies relating to software reliability prediction.

2.7.1 Applications of machine learning for software reliability prediction

This section reviews how researchers have adopted ML techniques into their studies for software reliability prediction. In this growing software industry, developers and engineers need to ensure that software products are failure free and reliable (Kumaresan & Ganeshkumar, 2019). According to research, ML techniques for software reliability prediction have shown very positive results according to (Alsina, Chica, Trawiński, & Regattieri, 2018; Arunima Jaiswal & Ruchika Malhotra, 2018; Rashid, Patnaik, & Bhattacherjee, 2014). Literature proposes ML techniques for software reliability modelling and prediction with varying levels of design, complexity and prediction accuracy (Hammouri, Hammad, Alnabhan, & Alsarayrah,

2018; Arunima Jaiswal & Ruchika Malhotra, 2018), with the aim of finding a way to identify software faults (Rashid, Patnaik, & Bhattacherjee, 2014).

(Iqbal et al., 2019) contend that the ML approach is effective in identifying software faults through a process of exploring hidden patterns within software attributes. They used several ML classification techniques to predict software defects in systems. With mobile devices being an integral part in people's lives, it has become crucial for improving the user experiences on these devices (Basavaraju & Varde, 2017). Xia, Shihab, Kamei, Lo, and Wang (2016) used an ML technique to help predict the crashing rate of mobile apps so that mobile developers are able to predict the success of their apps.

Hammouri, Hammad, Alnabhan, and Alsarayrah (2018) presented a software bug prediction model based on ML techniques to help developers predict software faults early in the development phase, so as to improve the quality of the software product. Singh and Chug (2017) identified and analysed the most widely used and popular ML techniques for reliability prediction in which classification was performed and validated on a total of seven software datasets. Lamba and Mishra (2019) did a comparative analysis of different ML techniques using many performance parameters to determine the best performing model used for reliability prediction.

Therefore, many studies reveal that ML shows promising results when predicting and estimating the behaviour of software systems during their development. Studies also show the positive impact of applying ML techniques to help identify patterns in the behaviour of software. The literature proposes many ML techniques for software reliability prediction but due to the rapid development of mobile apps, development complexities and challenges are constantly on the rise, there is no "one size fit all" approach. The literature thus calls for new ML approaches and techniques to deal with the complexities of the real world challenges faced within the app development industry(Wang et al., 2018).

2.7.2 Supervised machine learning

This section describes how various supervised ML algorithms have been applied in literature and compares their performance based on prediction accuracy. Supervised learning is firstly described and then a discussion into how this learning technique is implemented in previous studies is provided, with an attempt to help select a list of the better performing algorithms that may be useful and applied to this study for reliability prediction.

2.7.2.1 A definition of supervised machine learning

Supervised learning, as discussed in section 2.6 above, is a sub category of ML that consists of algorithms, also known as classifiers, that try to develop a function by learning a model from externally known, or labelled, data and known responses referred to as the training data (Ortiz & Peru, 2018). Training data is made up of training examples in which each example relates to an input object together with the desired output. The machine learns the given function based on this input data, and the learned pattern can then be applied to predict future instances over new sets of data (Sahli, 2020).

The techniques of supervised learning as mentioned in section 2.6 of classification constructs a predictive model for a discrete range of a function, referred to as categorical ML, while regression is for more continuous range of values (Charte, Charte, García, & Herrera, 2019). Categorical ML, defines two categories of predicted outputs, referred to as binary classification.

Unsupervised learning, on the other hand, learns from observation through a process of grouping based on certain guidelines or properties of an object. This type of learning does not require a supervisor or teacher (Berry, Mohamed, & Yap, 2019). The literature shows that many pieces of research focus on supervised learning with an application of a variety of ML classifiers, also referred to as ML algorithms, used for reliability prediction (Basavaraju & Varde, 2017; Hammouri, Hammad, Alnabhan, & Alsarayrah, 2018; Ortiz & Peru, 2018; Rashid, Patnaik, & Bhattacherjee, 2014).

2.7.2.2 A review and performance comparison of supervised ML algorithms applied in literature

Basavaraju and Varde (2017) compared supervised ML algorithms that showed relevance to mobile apps. These included SVM, AdaBoost, ANNs, Gaussian Process, Decision Tree (DT) and Naïve Bayes(NB). Kaur and Arora (2013a) employed supervised ML classifiers for their study on fault prediction in software modules. The classifiers used were Bayes Net, NB, Sequential Minimal Optimisation, Logistic Regression, NBTree, Random Trees, and Random Forest(RF).

Hammouri, Hammad, Alnabhan, and Alsarayrah (2018) for their study on software bug prediction, proposed a list of popular ML techniques used to predict faulty modules according to the literature. Their selected ML classifiers were NB, DT, and ANNs. Ghotra, McIntosh, and Hassan (2015) selected the following classification techniques based on their performance

on defect prediction models: Statistical Techniques, Clustering Techniques, Rule Based Techniques, Neural Networks (NN), Nearest Neighbour, SVM, DT and Ensemble Methods.

Rosenfeld, Kardashov, and Zang (2018) constructed a classifier using the Weka tool to discover functional bugs in various Android apps. This classifier was trained using various classification algorithms and the ones with the highest prediction accuracy were listed as: K-star, Multi-Layer Perceptron, RF, Logistic Regression, K-Nearest Neighbours and DT. Ortiz and Peru (2018) evaluated ML techniques to help predict the quality of software developed in IBM RPG. The algorithms were Linear Discriminant Analysis, Classification and Regression Trees, K-Nearest Neighbours, SVM and RF. RF had the highest prediction accuracy.

Table 2.1 provides a comparison of the predictive performance of ML algorithms as applied to various ML studies based on their prediction accuracies.

Table 2.1 A comparison of the predictive performance of ML algorithms applied in research for reliability prediction

Author	**The Study**	**The dataset/sample**	**ML Algorithms Applied**	**Selected Algorithms/Prediction Accuracy**
Basavaraju and Varde (2017)	Supervised learning techniques in mobile device apps in Android.	Publications in the digital libraries of ACM and IEEE (2006-2016)	SVM, AdaBoost, NN, GP, DT, NB.	AB, NN, NB
Kaur and Arora (2013b)	Adaptive Approach of Fault Prediction in Software Modules by using Discriminative and Generative Model of Machine Learning	4 NASA datasets from the Promise repository (CM1, JM1, KC1, PC1)	Bayes Net , NB, Sequential Minimal Optimisation, Logistic Regression, NBTree, Random Trees, RF.	RF: 96.7% NB: 93.7%

Hammouri, Hammad, Alnabhan, and Alsarayrah (2018)	Software bug prediction using the ML approach.	Three software faults datasets(DS1, DS2, DS3)	NB, DT, NN	DT: 97.1% NN: 95.1%
Ghotra, McIntosh, and Hassan (2015)	Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models	NASA dataset	Statistical Techniques (NB, Simple logistic), Clustering Techniques, Rule Based Techniques, NN, Nearest Neighbour, SVM, DT, Ensemble Methods.	The statistical Techniques using NB and Simple logistic outperformed the other techniques
Rosenfeld, Kardashov, and Zang (2018)	Automation of Android Applications Functional Testing Using Machine Learning Activities Classification	Features extracted off an Android Application	K-Star, Multi-Layer Perceptron (MLP), RF, Logistic Regression. K-Nearest Neighbour (KNN), DT.	K-Star: 86.25% MLP: 83.75% RF: 82.5%
Ortiz and Peru (2018)	IBM RPG Software Quality Prediction using Machine Learning Techniques	A sample of 5490 programs	Linear Discriminant Analysis, Classification and Regression Trees, KNN, SVM, RF.	RF: 97.5% SVM: 97%
Total Algorithms Compared			**20**	

2.8 A description of machine learning algorithms selected for this study

Based on the synthesis and analysis of the information on the twenty algorithms in 2.7 above, algorithms with the higher prediction scores were selected for this study. These include algorithms with scores over 90%. These algorithms are RF, SVM, NN, DT and NB. More details regarding the design and capabilities of each selected algorithm is discussed below.

NB: The Naïve Bayesian classifier was proposed by Thomas Bayes. It is also regarded as a probabilistic classifier with independence assumptions between features. NB is not a single algorithm but rather a family of algorithms based on some common principles and very strong independence assumptions of the presence or absence of features (Ghotra, McIntosh, & Hassan, 2015; Kaur & Arora, 2013a). It is very simplified in its design and assumptions but still works very well in complex real life situations (Arunima Jaiswal & Ruchika Malhotra, 2018). The algorithm also assumes the interdependence of various attributes, meaning that the presence or absence of a particular feature does not relate to the presence or absence of any other feature (Belavagi & Muniyal, 2016). Naïve Bayes is also regarded as a very computationally efficient ML technique (Singh & Chug, 2017).

RF: These are learning methods for classification. During training time, many decision trees are constructed. RFs are classifiers containing many tree structured classifiers (Kaur & Arora, 2013a). This combination of tree classifiers is also known as an ensemble. An ensemble combines all the decisions through a voting process which is then applied to unknown samples for prediction. This type of algorithm is found to be most accurate when compared to each individual classifier that makes up the ensemble (Ham, Chen, Crawford, & Ghosh, 2005; Pal, 2005). RF is able to combine multiple models for better prediction (Lamba & Mishra, 2019). It is also fast to train, runs efficiently on datasets and has the ability to rank and identify the most important independent variable when compared to other classifiers (Malhotra, 2016). It can be described as a collaborative approach that provides an easily understandable model (Belavagi & Muniyal, 2016).

NN: are referred to as non-linear predictive models that resemble biological neural networks and are able to learn through a training process. Also called an Artificial Neural Network (ANN), this mathematical model forms an adaptive system imitating the biological neural system consisting of nodes and links (Hammouri, Hammad, Alnabhan, & Alsarayrah, 2018). These classifiers model complex relationships between sets of input and output. Neurons send signals to other neurons in the network, thus producing output using the non-linear function of

all the neurons' inputs (Basavaraju & Varde, 2017). NN are used to correlate inputs with outputs, finding patterns and predicting defects (Ghotra, McIntosh, & Hassan, 2015).

DT: This is a common learning technique that is used in data mining. It is a hierarchical structure of decision nodes in which many branches and leaf nodes represent a decision. This serves as an efficient. non-parametric method that may be used for both classification and regression (Ghotra, McIntosh, & Hassan, 2015). DT algorithm is a learning system that is able to map observations to conclusions on items and their target values. Classification of instances are performed by following a path from the root to the leaf nodes of a tree, checking feature values against the rules at every node (Bhojan, Vivekanandan, Ramyachitra, & Ganesan, 2018). DT is able to cope with missing data and errors in datasets, but it tends to be an over-sensitive algorithm in the training set of data (Ghotra, McIntosh, & Hassan, 2015).

SVM: These are kernel-based learning algorithms used as a method for classifying data. SVMs develop models in which the data are separated by decision planes or hyperplanes that are defined by a number of support vectors. A *hyper plane* refers to a straight line that parts data points into two categories (Ghotra, McIntosh, & Hassan, 2015; Li, Sun, Li, Turrini, & Zhang, 2019).The learning process is made up of firstly training the SVM and then cross validating the classifier. The hyperplanes with the maximum margins from all classes form the best classification model. (Elish & Elish, 2008; Arunima Jaiswal & Ruchika Malhotra, 2018). SVMs are able to handle complex nonlinear functions (Ge, Liu, & Liu, 2018). The number of features selected for training has no effect on this technique; thus making the SVM a good fit for modelling. An SVM works well with a small sample, nonlinear and high dimensional pattern recognition for prediction analysis (Prasad, Florence, & Arya, 2015; Singh & Chug, 2017).

Studies show that there are differences in the performance of prediction models trained with various classification techniques and researchers are encouraged to explore various other techniques available.(Alsaeedi & Khan, 2019). There are many available ML algorithms that have been proposed and adopted in literature and it does become difficult to determine a superior choice of one technique over the other (Aydin & Samli, 2020). Therefore, researchers apply a choice of different models that are then evaluated in order to determine the best performing ML algorithm for a study (Gupta, Sharma, Goyal, & Rashid, 2020; Arunima Jaiswal & Ruchika Malhotra, 2018). For this study, after evaluating algorithms based on

similar studies and accuracy scores, the selected algorithms for this study is summarised in Table 2.2 below.

Table 2.2 A summary of ML classification techniques selected for the study

ML Algorithm	Description	Properties
RF	An ensemble learner method that combines multiple models for prediction.	User friendly, robust, efficient, collaborative.
SVM	Supervised ML technique that uses a hyperplane for classification.	Handles complex non-linear functions and serves a good fit for modelling.
NN	Non-linear predictive method that resembles the biological neural network.	Able to model complex relationships between inputs and outputs
NB	Probabilistic classifier that is based on the Bayes theorem.	Simple in its design, robust, easy to interpret and computationally efficient.
DT	Learning system that maps observations to conclusions following the path from the root node to the leaf node of a tree structure.	Copes well with missing data and errors in the dataset, cost effective, simple and comprehensible.

Once the choice of ML algorithms has been selected, the next step is to identify the types of features of data that will be analysed and used for reliability prediction. ML techniques are applied to data and the results are evaluated and analysed to assist with reliability prediction. In ML, data is extracted off a dataset containing features or columns of information and these features represent a measurable piece of data used for the analysis process (Zhao, Da, & Yan, 2021). With software system datasets, these features are referred to as software metrics (Padhy, Panigrahi, & Neeraja, 2019).

2.9 Software features as reliability indicators for mobile applications

This section provides a definition of software features, also known as, software metrics and highlights their importance in the field of reliability prediction. It presents a review of software metrics identifying how they are categorised and how they are applied to software systems as reliability indicators. This review aims to identify categories of metrics that will serve as essential contributors for assessing the reliability of apps, specific to this study.

Prasad, Florence, and Arya (2015) define software metrics as a measure of the quality of software in terms of the software product and the software process. Ge, Liu, and Liu (2018) define software metrics as a continuous quantitative units of measure used to define and analyse a system in terms of its development, processes and products. Software industries constantly strive to produce good quality software. This may be achieved through consistent bug removal, bug prediction and prediction of fault prone modules (Jayanthi & Florence, 2018).

A measure of the source code quality for apps is done through the use of software metrics, referred to as source code metrics. Source code metrics are extracted from the source code of an application and their value indicates a measure of quality of the product. Ceylan, Kutlubay, and Bener (2006) state that the main aim of prediction models is to determine and estimate the reliability of a software system and usually, focussing on a single metric or process is deemed insufficient for accurate predictions. Instead, a more suitable way to predict reliability is through observing the system throughout the software life cycle.

Studies show that in many cases, a combination of significant metrics is required to form better predictive systems for identifying reliable software. Therefore, the choice of metrics is an important precursor for creating predictive models in ML. Software metrics form a common approach for software reliability prediction leading to an early detection of faults and thus improving the reliability and quality of a mobile app (Singh, Monika, & Nidhi, 2016). This study proposes the application of ML techniques to a combination of significant sets of software metrics for mobile apps. Results obtained after evaluating these techniques will then be analysed and the best predicted results will be chosen to help predict the reliability of mobile apps.

There are a number of studies in the literature that analyse the relationship between the different types of software metrics and software quality, which is discussed further in the next section.

2.9.1 A review of software metrics used in previous studies

Lamba and Mishra (2019) stress that software metrics are useful for the evaluation of the complexity, reliability and fault proneness of a software system with the Chidamber and Kemerer set of metrics, also referred to as the "CK" metrics suite, is most widely used for measuring the code quality for object oriented systems. This source code metrics suite consists of six design level metrics (Catal & Diri, 2009) (Corral & Fronza, 2015).

- Weighted Methods per Class (WMC) refers to the number of methods in all classes.
- Depth of Inheritance Tree (DIT) is the longest path from a class to the root.
- Number of Children (NOC) is the number of children per parent class.
- Response for a Class (RFC) refers to the number of methods executed to respond to a message.
- Coupling between Objects (CBO) is the number of non-inherited classes to which a class is coupled.
- Lack of cohesion in methods (LCOM) relates to the access ratio of attributes.

These six metrics cover the aspects of complexity, coupling and cohesion of object oriented systems. These indicators are complemented by many other indicators to help measure the quality of a product based on the source code of the app as discussed in the studies below.

(Corral & Fronza, 2015) complemented these CK metrics by using indicators such as Cyclomatic Complexity (CC) that measured the number of independent linear paths through source code and Logical Lines of Code (LOC) which refers to the quantity of executable lines of code. The model used here characterises the metrics in terms of complexity, coupling, cohesion and size. (Catal & Diri, 2009) added the following to the set of CK metrics: Percent_Pub_Data (the percentage of protected data), Access_to_Pub_Data (the amount of public data for a class), Dep_On_Child whether a class depends on a child) and Fan_In (the number of calls by higher modules). Malhotra (2016), through a detailed literature survey, suggested a combination of the CK metrics with the QMOOD (Quality Model for Object Oriented Design) metrics. Two structured coupling metrics called Ca-Afferent couplings and Ce-Efferent couplings were added to the CK metrics. Afferent couplings refer to the number of other classes that use a specific class and efferent couplings refer to the number of other classes used by the class.

Apart from the use of the popular CK metrics suits, other metrics were used as software reliability indicators. (Manjula & Florence, 2018) utilised the PROMISE repository of

software defect datasets consisting of traditional software metrics namely, process metrics and object oriented metrics for the development of defect prediction models. The authors various measures or features, also known as attributes, were contained in these datasets. A genetic algorithm (GA) was applied for feature selection (i.e. the selection of the best predictors for software defects). The selected features were: LOC – Lines of code, Iv(g) – McCabe's design complexity analysis, Ev(g) – McCabe's essential complexity, N – Number of operators, v(g) – McCabe's cyclomatic complexity measurement, D – Measurement difficulty, B – Estimation of effort, L – Program length, I – Intelligence in measurement, E – Measurement effort, uniq_op – Number of unique operators, uniq_opnd – Number of unique operands, Branchcount – Total number of branches in the software module, and Locodeandcomment – number of lines and comments.

Sathya and Sudhakar (2016) created a predictive system using reliability relevant metric list (RRML). These metrics related to the various phases of the software development life cycle. These include the Requirement Phase Metrics, the Design Phase metrics and the Coding Phase metrics. (Rashid, Patnaik, & Bhattacherjee, 2014) performed a study using ML and software quality prediction as an expert system in an attempt to predict the quality of a software product. They mentioned the use of software metrics data to be useful clues regarding the position of possible faults in a programming module. Software reliability measures how frequently these failures occur for a given time interval. In their study the following metrics were used for their model: Size Metrics (LOC), Code Documentation metrics (comment and blank lines), Number of functions, Difficulty level of software, Experience of programmer in years, Development time and Number of variables.

Shihab, Jiang, Ibrahim, Adams, and Hassan (2010) compared simple models with fewer metrics to the more complex models with a large number of metrics. It was mentioned that even though the addition of more metrics to a model may increase the overall prediction capabilities, the complexity of the models creates difficulty in applying them practically. So, by using statistical models they were able to identify the most significant metrics for defect prediction. These were extracted from three different releases of Eclipse and categorised under process metrics and code metrics. Under process metrics the following were used: Total Prior Change (TPC) – the total number of changes to a file; Prior Bug Fixing Changes (BFC) – the number of bug fixing changes to a file; Pre-Release Defects (PRE) – the number of pre-release defects; and Post-Release Defects (POST) – the number of post-release defects, with this being the dependent variable in the model. Under source code metrics the following were selected:

Total Lines of Code (TLOC); Fan Out (FOUT) – measures the number of method calls; Method Lines of Code (MLOC); Nested Block Depth (NBD); Number of Parameters (PAR), McCabe Cyclomatic Complexity; Number of Fields (NOF); Number of Methods (NOM); Number of Static Fields (NSFG); Number of Static Methods (NSM); Anonymous Type Declaration (ACD); Number of Interfaces (NOI); and Number of Classes (NOT).

Malhotra (2015) conducted a systematic review of ML techniques for reliability prediction, identified 64 studies and identified various metrics in software engineering to measure software characteristics. These included procedural metrics; the traditional Halstead and McCabe metrics; Object Oriented (OO) metrics such as cohesion, coupling and inheritance; hybrid metrics, which is a combination of procedural and object oriented metrics; and miscellaneous metrics such as requirement metrics, change metrics, network metrics, and other miscellaneous metrics. Of those mentioned, the Procedural metrics are the most commonly used and the most common OO metric suite used is the Chidamber and Kemerer (CK) metric suite.

2.9.2 Motivating the choice of metrics for the study

Corral and Fronza (2015) indicate that the use of the CK metrics suite has been widely accepted and validated for the evaluation of Object Oriented Systems. The source code metrics in this dataset combines CK metrics with the OO metrics and LOC metrics. Change metrics, also known as process metrics, are often considered as accurate predictors that relate to the changes made during the software life cycle. In this life cycle, the churn metrics illustrate how the code has evolved which is relevant to the dynamic nature of mobile apps (Choudhary, Kumar, Kumar, Mishra, & Catal, 2018).

Process metrics are useful measures over time, while static source code metrics relate to the specific attributes of a software product (Okutan, 2018). Discussions from 2.9.1 above show significant use of source code metrics for predicting errors in software. It is stated that source code metrics serve as essential contributors to the measurement of the quality of a software product (Nuñez-Varela, Pérez-Gonzalez, Martínez-Perez, & Soubervielle-Montalvo, 2017). Fasano, Martinelli, Mercaldo, and Santone (2018) also suggested the use of source code metrics to assess the reliability of a mobile app from a Java-based, object oriented perspective.

Compared to traditional desktop software, mobile apps develop at a much faster rate and therefore, are proven to change more frequently as discussed in section 2.4. Changes, mainly through user requirements, need to occur rapidly implying rapid code development with less attention to design practices. Continuous changes made to these mobile apps leads to a process

of evolution. This evolution is what adds complexity to the mobile apps as it continues to change and grow (Al-Msie'deen & Blasi, 2018). These changes are measured through change or process metrics. Process metrics are considered to assess the reliability of a mobile app through software changes over time, gathered from the revisions of software products and analysed to improve the software process (Kaur, Kaur, & Kaur, 2016).

It is mentioned that a combination of these source code metrics and process metrics can serve as an effective combination to help detect software defects in mobile applications and serve as useful reliability prediction indicators (Gupta, Suri, & Bhat, 2019; Kaur & Arora, 2013a).

Based on this and for the purpose of this study, ML techniques will be applied to each of the selected source code metric set, the process metric set as well as a combination of both metric sets extracted off the Eclipse dataset. The selected ML algorithms will be applied to these 3 metric sets of data. The results obtained after applying and evaluating these ML algorithms on all 3 metric sets will then be analysed to determine which ML modelling approach (i.e. ML algorithm and metric set) generates the highest predicted results which may serve useful for the reliability prediction of mobile apps. The next section explains ML evaluation measures needed to evaluate the prediction performance of ML techniques to determine if they are useful for software reliability prediction, or not.

2.10 Machine learning evaluation measures

In order to compare the prediction accuracies of different ML techniques applied, meaningful performance measures are required to quantify the most accurate predictions (Zheng, 2009). These performance measures also evaluate the learning process and indicate its effectiveness in making predictions (Malhotra, 2016). A good prediction system produces high values for performance measures. The performance measures discussed a use a technique of measurement known as the confusion matrix (Belavagi & Muniyal, 2016; Gupta, Sharma, Goyal, & Rashid, 2020; Hammouri, Hammad, Alnabhan, & Alsarayrah, 2018).

With the confusion matrix analysis, the performance of the correctly classified and incorrectly classified classes is measured for a given dataset. Actual classes and predicted classes are stored in the matrix to obtain classification results that help compute accuracy, performance, sensitivity and other aspects for a proposed approach (Manjula & Florence, 2018). The list below defines the popular performance measures used in various studies (Asim & Khan, 2018; Gupta & Saxena, 2017; Malhotra, 2015; Manjula & Florence, 2018; Sun et al., 2019).

Accuracy: This defines the percentage of correctly classified instances. Mathematically it is represented as:

$$Accuracy = \frac{correctly\_classified\_samples}{total\_samples} * 100$$

In other words, it refers to the proportion of the total number of correct predictions divided by the total number of incorrect as well as correct predictions.

Precision/Correctness: This refers to the probability of the predicted positive records being correctly classified. For example, precision may refer to the total number of correctly classified fault prone classes divided by the total number of classified fault prone classes.

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

TP is a true positive: correctly classified or predicted instances and FP is a false positive: incorrectly classified instances.

Recall/Sensitivity/TPR (true positive rate): This is the measure of probability that a record related to each class is correctly classified.

$$Recall = \frac{TP}{TP + FN}$$

FN is a false negative: the number of incorrectly predicted records by the classifier. It is also defined as the ratio of correctly predicted defective classes to the total number of classes.

Specificity/TNR (true negative rate): This refers to the measurement of correctly classified defective modules or classes.

$$Specificity = \frac{TN}{TN + FP}$$

where TN is a true negative: the number of records correctly predicted as negative.

Pf/FPR (false positive rate): This is a measure of the probability of non-fault prone modules being classified incorrectly as fault prone.

$$FPR = \frac{FP}{FP + TN}$$

F-measure (F1 Score): This evaluates the balance of the precision and recall. It is also referred to as the harmonic mean of precision and recall.

$$F1\ Score = 2 * \frac{(recall * precision)}{(recall + precision)}$$

AUC (area under the curve) / ROC (receiver operating characteristic curve):

The AUC_ROC curve is also a useful metric for evaluating prediction performance of a ML algorithm visually (Lee et al., 2020). The ROC is a probability curve while the AUC represents the ML algorithms ability to distinguish between classes as a measure of separability. The higher the AUC, the better the model's performance. The ROC curve is plotted with the True positive rate (y-axis) against the False positive rate (x-axis) for all ML algorithms applied. An excellent predictive result is obtained by the algorithm that has an AUC nearest to 1 which means it is a good measure of separability while an AUC near 0 has a low measure of separability (Narkhede, 2018). Hence the term area under the ROC curve. The ROC does not depend on class distribution and this makes it a useful evaluation measure for classification techniques even for imbalanced datasets (Ballabio, Grisoni, & Todeschini, 2018).

These measures are useful in selecting the most reliable learning process with the highest prediction accuracy. The results obtained after evaluation indicate the usefulness of the prediction system and its effectiveness when applied in the real world.

2.11 Summary of the chapter

This chapter reviewed the literature in major areas of this research, namely, the challenges in MAD, existing software reliability models and ML techniques adopted for software reliability prediction. Software reliability is identified as an important measure of quality that is defined as the probability of software being failure free for a specific time period. Software reliability poses a huge challenge for developers and an even greater challenge for app development. Apps need to be developed quicker and changed more frequently to satisfy the demands of the app markets and users.

The ability to identify problems or defects early during development is said to improve its reliability and also ensure greater success within the app market. Section 2.4.1 highlighted the six phases of the MADLC that illustrate the short development cycle of apps to satisfy its short development timeframes. Section 2.4.3 pointed out some major challenges experienced by developers that include fragmentation, stability and interoperability, monitoring analysis and testing support, change management and keeping up with frequent changes, user interface design, re-use of code and the lack of resources and tools support. Each challenge was further discussed and it was established that major challenges existed as a result of the nature of the

mobile environment, the functionalities and features of the mobile environment, the mobile operating system and the design of mobile devices. The next section (section 2.5) provided a review of how the existing reliability models, known as SRGMs, have been applied towards the development of reliable software. These models offer various testing techniques at various phases of software development to estimate the reliability of the product.

It was discovered that these models require optimum estimating techniques, especially for the more complex systems. Also, testing techniques are applied at phases that exist within the traditional software development life cycle models such as, the waterfall model with a substantial dependence on time. Given the complex and dynamic natures of MAD, these models failed to fit all the data and were not successful in predicting the reliability of apps implying that other tools and techniques are required for this process. Section 2.6 defined the concept of ML and described it as an automated process that uses tools and techniques to identify meaningful patterns in data, that is mapped to an output, to help predict an outcome.

Section 2.7 reviewed literature that have adopted ML in various studies and described ML as a field that provides a set of useful statistical techniques to help predict and improve the reliability and behaviour of a software system. This then advocated the use of ML as a more suitable technique for the predicting the reliability of apps. This study applies a sub category of ML, known as supervised learning, that has been discussed in section 2.7.2 with an extensive review and performance comparison of supervised ML algorithms.

The synthesis and analysis from the information gathered in 2.7.2 help select ML algorithms with the higher prediction scores for this study, namely, NB, RF, NN, DT, SVM. After selecting the algorithms, the next step is to identify the software features or metrics to serve as reliability indicators for the study. Section 2.9 provides a review of software metrics used in previous studies and show the significant use of source code metrics for predicting errors in software. Source code metrics are measures extracted off the source code of software. It must be noted that compared to regular software, mobile apps develop at a much faster rate that leads to a process of evolution. This process measures software changes over time gathered from the revisions of a software product, referred to as process metrics.

Therefore, just using source code metrics may not be sufficient to predict the dynamics of app reliability, so including the use of process metrics may help improve the app reliability prediction. This study proposes the use of source code metrics combined with process metrics to serve as the reliability indicators for apps. However, a comparison of applying ML

techniques to each metric set will be conducted through ML evaluation measures, to determine which of the source code metrics, the process metrics or the combination will serve as the most accurate reliability indicators for apps. These ML evaluation measures are discussed in section 2.10 and serve to quantify predictions to determine the most accurate predictions. Evaluating the learning process created by applying selected ML techniques to sets of data and comparing the results, help select the most effective techniques for making predictions.

CHAPTER 3: RESEARCH METHODOLOGY

3.1 Introduction

This chapter describes elements of the research design and methodology adopted for this study. Each aspect of the design is described in detail in the different sub-sections. The research design is carefully devised to provide an adequate scaffolding to supporting the aim of the study which is to apply ML techniques to assist with the reliability prediction of apps. The research satisfies the characteristics of post-positivist paradigm, with a quantitative research design strategy. To support the research endeavour of applying ML techniques to aid in the reliability prediction of apps, the study subscribes to Experimental Algorithmics (EA) as the research methodology. The chapter progresses as follows: section 3.2 provides motivation for situating this research in the post-positivist paradigm; 3.3 explains why this research is quantitative; 3.4 discusses the choice of experimental algorithmics as the research methodology; 3.5 describes the experiment; and 3.6 discusses the credibility of the research in terms of validity, reliability and generalisability. Table 3.1 below provides an overview of the selected research design aspects for the study.

Table 3.1 An overview of the research design

Research Design Aspect	**Choice**
Paradigm	Post-positivism
Methodology	Experimental Algorithmics
Strategy	Quantitative

A motivation of choice for each of the paradigm, methodology, strategy is given below.

3.2 An overview of candidate research paradigms

A research paradigm can be defined as a "loose collection of logically related assumptions, concepts, or propositions that orient thinking and research" (Bogdan & Biklen, 1998, p. 22). Research is often described as a systematic investigation in which data is analysed and interpreted in an effort to predict, understand or describe a phenomenon within a context (Burns 1997) and the process occurs within certain established frameworks and within the prescribed guidelines (Williams, 2007). A number of research paradigms are evident in the literature

(Rehman & Alharthi, 2016). Positivism and post-positivism are the candidate paradigms for this study which is further elaborated on below, with a motivation of preferred choice.

3.2.1 Positivist research

Positivism assumes that science is the way to get to the truth and understand the world well enough so that it can be predicted or controlled (Krauss, 2005). Positivism aims to test a theory quantitatively, through observation and measurements of independent facts in which the data being observed does not change. In other words, positivism is an approach that is based on scientific evidence acquired from experiments and statistics. Positivists view the world through a "one way mirror" (Healy & Perry, 2000), applying the philosophy that the world is deterministic operating under the laws of cause and effect (Krauss, 2005). With the positivist paradigm, all studies are based on facts. The world is regarded as external and objective in which human interests are somewhat irrelevant.

3.2.2 Post-positivist research

Both positivist and post-positivist research is based on the quantitative methodology of data collection and analysis (Mackenzie & Knipe, 2006). However, positivism was later replaced by post-positivism as researchers started to notice problems with the positivist theory (Clark, 1998; Guba, 1990; Henderson, 2011). The limitation to positivism is that it strives to find the absolute truth based only on observable and empirical facts. It does not recognise the existence of errors and neither does it allow for revisions or improvements in the study as opposed to the post-positivist approach (Creswell, 2003). A post-positivist researcher understands that the world is variable in which everything is not completely knowable. Understanding is the main objective, rather than explaining, in which scientific *reasoning and common sense* go together. Post-positivism together with quantitative analysis, strives to explore clear details and directions on phenomena based on historical, comparative and philosophical perspectives (Fischer, 1998).

3.2.3 Motivation for the use of post-positivist research to underpin this study

A major flaw of positivist research is its claim to certainty that only scientific knowledge observed is valid and accurate in revealing the truth about a reality and everything else is false (Houghton, 2011). Post-positivist research, on the other hand, offers a more flexible approach that allows a researcher to use multiple techniques to ensure that a study is investigated from different angles to maximise reliability (Panhwar, Ansari, & Shah, 2017). This dissertation does not aim to disapprove the scientific elements of positivism, but rather to emphasise a

proper understanding of the directions and perspectives of the study as indicated by post-positivism. This is evident through the aim of the study.

The study aims to apply and evaluate ML techniques to assist with the reliability prediction of apps. This is done through adopting in-depth investigations into the fields of software engineering for mobile app development and ML techniques to gain sufficient knowledge in the area so as to select suitable techniques to drive the study. These ML techniques are then applied to historical data extracted from software systems made up of software metrics. This follows a characteristic of post-positivist research that states that the theory and practice cannot be kept separate for the sake of collecting just the facts (Ryan & students, 2006).

Software metrics are units of measure that impact software reliability, yet the value of these measures change across different software systems, implying that the data is not a collection of complete facts, but rather variable measures that need to be considered in the broad area of software development. Software systems applied are also designed by developers of varying levels and skills sets implying the adoption of a more intuitive approach toward the analysis of data and an acknowledgement that there is no absolute truth as stated through positivism.

3.3 A brief comparison of research strategies

The research strategy refers to the overall plan that is followed by the researcher in order to conduct a study serving as a guide, starting from the planning phase and extends all the way to the execution phase of the study (Johannesson & Perjons, 2014). It follows a systematic approach in which the choice of strategy for a research project becomes vital for the quality and value of the project. Determined within the context of the research, the selection of this strategy provides a “best way” approach to conduct the research, given the assertion that there are a large number of methodologies that may be applicable to a particular study (Jenkins, 1985). A research strategy refers to how data is collected and analysed including representations derived from the data, as well as reporting (Mackenzie & Knipe, 2006). Three common approaches are qualitative, quantitative and mixed methods. Each is discussed briefly below, followed by a motivation of choice of the selected strategy for this study.

3.3.1 Qualitative method

Krauss (2005) defines qualitative research as a function of meaning which includes a face-to face interaction with a participant and being able to understand not only the words but the underlying meaning of the words, as well as being able to acquire some social knowledge

regarding the study. Becker (1996) states that qualitative research is a way for scientists to implicitly or explicitly analyse and interpret the actions of people. This can be achieved through formal and informal interviews, through observations and through questionnaires. Derived from inductive reasoning, qualitative research commonly includes case studies, ethnographic study, phenomenological study, grounded theory study, and content analysis. Qualitative data analysis is sometimes described as a unique and powerful tool to help with the understanding of human experiences in a natural setting. This study does not gather information for analysis through the interaction of participants. There are no questionnaires, interviews or case studies applied in this research for data collection, but rather an application and analysis of ML techniques applied on data extracted from software systems. Therefore, this study does not follow the qualitative research approach.

3.3.2 Mixed method

According to (Creswell, 2003), there has been a notable change in research methodologies in which certain approaches have become more complex in design but more flexible in their application. This has called for a combined approach that resembles a mix of both the qualitative and quantitative approaches, known as the mixed method approach to research. Gorard and Taylor (2004) stipulated that the combination or mixed method has shown great improvements in research and argue that this approach requires greater skill, but with little wastage of useful information. The ability to design research which combines data collection from both the quantitative and qualitative approaches allows researchers to test and build theories with the ability to apply both deductive and inductive analysis in the study (Williams, 2007). Usually a mixed method approach begins with a qualitative collection and analysis to derive major themes, followed by quantitative analysis to test and generalise these themes. This study does not use participants to gather information to be analysed. All data is extracted of datasets and analysed and applied through the use of ML techniques. Hence, without the use of participants in this study, the mixed method cannot be applied to this research.

3.3.3 Quantitative method

Quantitative research methods are used to collect data that can be represented numerically and analysed statistically using mathematical methods (Goertzen, 2017). This approach quantifies data by creating meaning through analysing the collected data (Williams, 2007). The intention of quantitative research is to seek explanations and validate relationships to explain some phenomena (Creswell, 2017). Defining this further, explaining some phenomena is a key element in research in which researchers set out to explain something, while applying

mathematical methods imply that all data has to be in numerical form in order to be analysed (Sukamolson, 2007). Therefore, quantitative research refers to the numerical representation and the analysis of observations used to describe and explain some phenomena. Having now described the above research strategies, motivation for use of the quantitative method is presented next.

3.3.4 Motivation for the use of the Quantitative method

This study sets out to determine if ML techniques can assist with the reliability prediction of mobile apps. This phenomena of reliability prediction follows a process that applies ML techniques to data in an attempt to learn from this data and make predictions on unseen data. The use of ML techniques within this research follows a sequence of statistical data processing steps at different stages of the experiment implying that all data to be processed has to be numerical.

The quantitative research strategy is applied at various stages of the research. This is evident right from the first step of selecting and preparing the data, to applying the ML algorithms to this data for training and testing, to the evaluation and comparison of algorithms for predictive performance. All ML techniques involve a series of statistical dependencies of variables and probability scoring through the use of mathematical functions and formulae for evaluating prediction performance. All of this indicates that the quantitative research strategy is appropriate and necessary for this study. It uses measurement and applies a range of methods to analyse data for trends and relationships in order to help with the estimation and prediction of the reliability of an app. The next section defines the research methodology and motivates the use of EA as the appropriate methodology for this study.

3.4 Research methodology

This sub-section explores the areas of design science research and experimental design and algorithmics in an attempt to motivate for the most appropriate methodology relating to this study.

3.4.1 Design science research

Peffers, Tuunanen, and Niehaves (2018) describe design research as a process involved in the formulation and validation of models and theories about the aspects of design. It involves the development and validation of knowledge, methods and tools used to improve the design process (Blessing & Chakrabarti, 2009). Peffers, Tuunanen, and Niehaves (2018) designed a methodology for researchers to use, based on the various design research principles that involve

five activities: awareness, suggestion, development, evaluation and conclusion. This has been structured sequentially, but researchers are able to start at any step and move forward. The design science research methodology focusses on the design and development of artefacts. Such artefacts include the design of a system, application, methods or anything that will be useful to an organisation. Design science research follows a methodology for solving important real world problems characterised by the relationship between understanding the nature of the problem and designing an artefact to solve this problem (Butgereit, Niland, Schmidt, & Rheeder, 2018). This poses a challenge within complex, interconnected technical systems such as software systems (Bisandu, 2016). The challenges and complexities of app development create unstable interactions between the problem and the solution design, thus requiring a more flexible design of artefacts to deal with such complexities.

3.4.2 Experimental algorithmics as the planned research methodology

Experimental Algorithmics (EA) refers to an approach of design and analysis of algorithms and their properties in an attempt to discover certain truths (McGeoch, 2007). In other words, it refers to the study of algorithms and their performance by experimental needs to gain some insight into solving a problem. EA is a discipline that forms part of the algorithmic engineering process that features a cyclic iterative process, made up of different phases (Bartz-Beielstein & Mernik, 2016). The first phase involves a clear understanding of the problem, the second deals with the design of the algorithm, the third analyses the algorithm, the fourth implements the algorithm and the fifth phase evaluates the algorithm (Angriman et al., 2019). EA is an area that lends its focus to experiments with algorithms and then analyses the theory behind them. (Bartz-Beielstein, 2016; Mihelic & Cibej, 2017).

3.4.2.1 A short review of experimental algorithmics employed in other studies

Bartz-Beielstein and Mernik (2016) looked into an experimental methodology for on-line ML algorithms for streaming data. Their study illustrated how the experimental tools from EA can be applied to on-line or streaming data settings. The results revealed that the statistical methods derived from EA were relevant and applicable to the on-line setting. However, it was also mentioned that certain aspects of the algorithm, such as the run time plots, needed to be complemented by some other, more advanced statistical tools. Mihelic and Cibej (2017) used the concept of experimental algorithmics to examine Maxeler's dataflow architecture (Maxeler, 2015). Part of their study included a focus on the techniques and guidelines to be followed to perform successful experiments. Two phases of the experimental process were highlighted, that is planning and execution of the experiment. Before experimentation and as

part of the planning phase, the type of test subject or algorithm needs to be determined. This implies that at the first phase all the experimental requirements, goals and motivations need to be clearly defined for the study. Many alignment constraints arose when EA was applied to the dataflow architecture, mainly with regards to the test data and how it should be transformed to fit the architecture. It was concluded that the implementation needs to be as general as possible to clearly highlight the algorithm behaviour. Further complexities may be considered when measuring the performance of the algorithms.

3.4.2.2 Experimental algorithmics: A motivation for this study

McGeoch (2012) mentioned 3 main aspects in algorithmic studies, namely, analysis, design and models of computation. Analysis aims to predict the performance of algorithms under the given assumptions of input and output. Design focuses on the application of effective algorithms to solve computational problems. Model of computation considers how changes in the system can affect the design and analysis of algorithms. Bartz-Beielstein and Mernik (2016) suggest that before any experimental runs are performed using EA, the scientific question that motivates the experiment needs to be clearly stated.

This study is based on the following research question: *Can ML assist in identifying useful predictors for reliability of mobile apps?* In an attempt to answer this question through applying the aspects of EA, this study proposes the use ML techniques. These techniques combine multiple computational methods, that when applied to data extracted from software systems, has the ability to learn directly from this data. ML algorithms applied are intended to find natural patterns in data, thereby generating some insight into the issues of software reliability in an attempt to help make better decisions and predictions during the design process. EA requires experimental setups, and for this study sets of ML algorithms are selected and applied to the data.

Results generated for each algorithm is evaluated using measures of precision, accuracy recall and F1 scores help determine the algorithm that best fits data, to be applied to assess and predict app reliability. Therefore, with the careful planning and execution of the experiments in this research, this study proposes the use of EA as the planned research methodology. The next section describes the components of the experiment.

3.5 Describing the experiment

This section describes the components of the experiment used in the study. The aim of this study is to apply ML techniques to assist with the reliability prediction of mobile apps. In order

to facilitate this aim, an experimental setup, that follows the steps of the ML workflow process, is needed. These steps are highlighted in Figure 3.1 below:

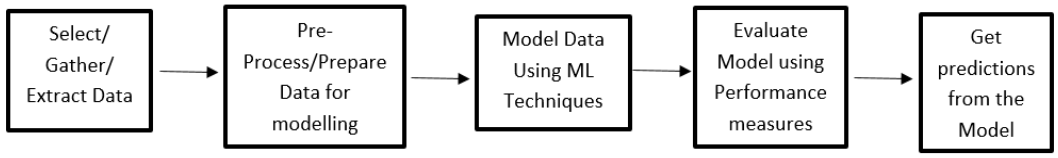


Figure 3.1 Process flow diagram for the experiment. Source: (Lantz, 2013; Raschka, 2015)

Following these steps may help ascertain if ML can assist with the reliability prediction of apps. The sub-sections that follow provide details of the experiment regarding the chosen dataset, the choice of metrics, the need for feature selection, the choice of classification techniques and the ML evaluation methods required for this part of the study.

3.5.1 Describing the dataset

In order to make predictions in ML, a collection of data is required. This is referred to as a dataset. All ML algorithms learn from the data which it is provided with; the data represents the main input to ML. Different datasets characterise different problem domains. This study focuses on software reliability and hence, uses a bug prediction dataset. This dataset groups features of software with values, referred to as software metrics, as discussed in section 2.9. These metrics are extracted from software systems historical data, allowing researchers to apply ML to this data to predict the outcome of software products through the early detection of bugs (Kalaivani & Beena, 2018). Bug prediction datasets are categorised into two types: public and private. Public datasets are available in many repositories and are freely available for public use. Private datasets, on the other hand, are privately prepared by researchers for their own studies (Bhuiyan, Jabiullah, & Felix, 2020). This study used a public online dataset; the reason for using this dataset as opposed to a manual one is discussed below.

The dataset used contains a total of 997 software class elements with metrics and other bug related information. The dataset also depicts a total of 91 versions of the Eclipse JDT core system with 9 135 transactions and 463 post-related defects. A deeper look into this dataset, and a reason for benchmarking this for this study, indicates that the selected bug prediction dataset contains change, bug and version information for the Eclipse JDT core software system. This dataset contains extracted data from the change log files and defects from the defect repository of systems linked to the system classes that reference them. Biweekly, different versions of the system were parsed into object oriented models. Systems are all Java-based to ensure that all code measures were consistently defined for each system class (Boucher &

Badri, 2016; Ferenc, Tóth, Ladányi, Siket, & Gyimóthy, 2018; Lamba, Kumar, & Mishra, 2019). Hence, this dataset provides extensive opportunities to create and evaluate ML prediction models for software bug prediction which would be very difficult to achieve through the creation of a manual dataset.

The dataset used for this study is a public *bug* prediction dataset for the Eclipse JDT core software system. Eclipse JDT core is a Java-based system made up of the compiler and Java tools. This dataset comprises a collection of models, metrics and historical information of the Eclipse JDT core software system which is designed to perform bug prediction at the class level. The dataset is publicly available at http://bug.inf.usi.ch. The dataset contains change, bug and version information of the Eclipse JDT system as shown in Table 3.2 below.

Table 3.2 Details of the Eclipse JDT core Dataset

System	Release	#Classes	#Versions	#Transactions	#Post-rel. defects
Eclipse JDT Core www.eclipse.org/jdt/core/	3.4	997	91	9135	463

This dataset makes it possible to utilise and apply various types of metrics to assist in the prediction of post-release defects at the class level for software systems. Applying ML algorithms to these metric sets of data create ML based models that can then be evaluated for precision accuracy by comparing the predicted results against the actual results for post-release defects which is provided as part of this dataset (D'Ambros, Lanza, & Robbes, 2010). Metrics included in this dataset used as predictors are Change metrics (Moser, Pedrycz, & Succi, 2008), CK metrics, (Basili, Briand, & Melo, 1996), OO metrics (Basili, Briand, & Melo, 1996), Number of previous defects (Kim, Zimmermann, Jr., & Zeller, 2007), Complexity of code change (Hassan, 2009), Churn of CK and OO metrics (D'Ambros, Lanza, & Robbes, 2010), and Entropy of CK and OO metrics (D'Ambros, Lanza, & Robbes, 2010).

There is no tool available to be able to test the reliability of an application automatically, so it is therefore necessary to be able to find a reliable source of metrics that are influential in the success of the app, also referred to as the best set of predictors (Riegler & Holzmann, 2018). This data source was used by other authors for similar research (Alsolai & Roper, 2019; Boucher & Badri, 2016; Choudhary, Kumar, Kumar, Mishra, & Catal, 2018; Ferenc, Tóth,

Ladányi, Siket, & Gyimóthy, 2018; Gupta, Sharma, Goyal, & Rashid, 2020; Lamba & Mishra, 2019). Table 3.3 below describes studies that used this data source (Eclipse JDT core dataset).

Table 3.3 Studies that used the Eclipse JDT core Dataset

Authors	**Description of the Study**
Alsolai and Roper (2019)	To determine the best accuracy of software maintainability models using Auto –WEKA tool.
Boucher and Badri (2016)	Used software metrics threshold to predict fault prone classes in object oriented software.
Lamba, Kumar, and Mishra (2019)	A comparative study of bug prediction techniques on software metrics.
Choudhary, Kumar, Kumar, Mishra, and Catal (2018)	An empirical analysis of change metrics for SFP.
Ferenc, Tóth, Ladányi, Siket, and Gyimóthy (2018)	A public unified dataset for Java.
Gupta, Sharma, Goyal, and Rashid (2020)	A novel XG-Boost tuned Machine Learning model for software bug prediction.

3.5.2 Describing the features of the Eclipse JDT dataset

This sub-section describes all the metrics available in the Eclipse JDT dataset.

1. Change or Process Metrics

   Change or process metrics refer to the changes that were made to during the software development process. These metrics are extracted from the CVS/SVN logs from multiple releases of the software product. These metrics include:

   NR Number of revisions

   NREF Number of times file has been refactored

   NFIX Number of times file was involved in bug fixing

   NAUTH Number of authors who committed the file

   LINES Lines added and removed (sum, max, average)

CHURN	Code churn (sum, maximum and average)
CHGSET	Change set size (maximum and average)
AGE	Age and weighted age

2. Previous Defects

   Proposed by (Zimmermann, Premraj, & Zeller, 2007), this bug prediction approach correlates the number of past bug fixes extracted from a repository to the number of future bug fixes with the rationale that past defects predict future defects. A variant is used to categorise the number of bugs according to severity and priority. There are five variant bug categories, i.e. all bugs, non-trivial bugs, major bugs, critical bugs and high priority bugs. This approach uses a single metric to perform the prediction.

3. Source Code Metrics

   The source code metrics used in this dataset combine the CK metrics with the OO metrics and LOC metrics (D'Ambros, Lanza, & Robbes, 2010). Based on the rationale that complex components are harder to change and hence error prone, the source code metrics used in this dataset are:

Type	**Metric**	
CK	WMC	Weighted Method Count
CK	DIT	Depth of Inheritance Tree
CK	RFC	Response for Class
CK	NOC	Number of Children
CK	CBO	Coupling Between Objects
CK	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of other classes that reference the class
OO	FanOut	Number of other classes referenced by the class
OO	NOA	Number of attributes
OO	NOPA	Number of public attributes
OO	NOPRA	Number of private attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods

OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

4. Entropy of Changes

   Proposed by (Hassan, 2009), this approach measures over time how changes made were distributed throughout the system. The greater the spread of changes, the greater the complexity of the system; the rationale being that complex changes are more error prone than simpler ones. The use of entropy code change for bug prediction is based on the History of Complexity Metric (HCM) of a file. Two-week intervals are used to collate this information. Three variants of the HCM are used with additional weights based on the periods in the past. These are the Exponentially Decayed HCM, the Linearly Decayed HCM and the Logarithmically Decayed HCM.

5. Churn of Source Code Metrics

   Churn of source code metrics are used to predict post-release defects. The history of the source code is sampled every two weeks and the deltas of source code metrics are computed for each consecutive pair of samples. The rationale is that higher level metrics may create better models using code churn metrics as compared to simple metrics like the addition or deletion of lines of code (D'Ambros, Lanza, & Robbes, 2010).

6. Entropy of Source Code Metrics

   This bug prediction approach extends the code change entropy to the source code metrics listed in the dataset. Entropy measures the complexity of variants of metrics over different sample versions. The higher the distribution of the variants of the metrics, the higher the complexity and hence the higher the entropy. The entropy of source code metrics is defined for every source code metric. To determine how much the class has changed, the history of weighted entropy (HWH) is defined. Three variants are considered for the decay of entropy over time and are expressed as the exponential (EDHH), linear (LDHH) and the logarithmic (LDGHH) weighted entropies.

The selected dataset consists of 91 versions of the Eclipse JDT core system. For each class, within each system version, it contains 6 CK metrics, 11 OO metrics, and 15 change metrics. It also provides data on past defect counts categorised with severity and priority, post-release

defect counts categorised with severity and priority, class history over all versions, churn measures for all CK and OO metrics, entropy measures for all CK and OO metrics, complexity of code change measures as well as the weighted linear, exponential and logarithmic variants for churn, entropy and complexity of code change.

Not all of these metrics will be applied to this study. The metrics applied are that of the source code and process metric sets, extracted off the Eclipse dataset. The selection of metrics for the study has been discussed and explained in much detail in section 2.9.2 with a motivation for its use for this research. Now that the relevant data for use in the study has been identified, the next step refers to the application of ML data pre-processing techniques known as feature selection which is described below.

3.5.3 The need for dimensionality reduction and feature selection

Once the metrics that been identified for software reliability prediction, (Asim & Khan, 2018; Bhojan, Vivekanandan, Ramyachitra, & Ganesan, 2018; Choudhary, Kumar, Kumar, Mishra, & Catal, 2018; Ge, Liu, & Liu, 2018; Lamba & Mishra, 2019; Malhotra, 2015; Malhotra, 2016) stress the importance of selecting the relevant features or attributes to be applied to the ML process. Also known as dimensionality reduction, feature selection reduces the number of considered features by creating a smaller, more relevant set of features. There are two types of dimensionality algorithms, i.e. *Feature Selection* and *Feature Extraction*. Feature selection aims to reduce the *overfitting problem*, improve prediction accuracy and reduce training time and helps reduce computational overheads (Khaire & Dhanalakshmi, 2019). The method of feature selection evaluates the correlation between the independent variables and then follows the process of selecting those independent variables that are highly correlated with the dependent variable and also uncorrelated with each other to reduce redundancy (Cai, Luo, Wang, & Yang, 2018).

In this study the independent variable refers to the software metrics, and the dependent variable is a two-value (0 or 1) variable stating whether there exists a post-release bug or not. Feature selection methods included in this study are the *filter methods* and *wrapper methods*. Filter methods are intended to help eliminate all irrelevant, redundant, constant, duplicated and correlated features. Wrapper methods identify and evaluate subsets of features through ML techniques. They detect the interaction between the feature variables to find the optimal feature subset that may deem useful for prediction. It follows a search approach by evaluating all possible combination of features using accuracy scores and comparing these scores to determine

the best performing, or the optimal subset of features to be used for prediction within the dataset. Once the data has been cleaned, and pre-processed using the feature selection techniques, this data is read to be trained using the selected ML algorithms as discussed in section 2.8. and the results evaluated for prediction accuracy to assist with the reliability prediction of mobile apps.

3.5.4 A choice of suitable Machine Learning algorithms for the study

Section 2.6 defines ML as a division of AI that has the ability to build systems that *learn* from data. For software products a prediction system, also called binary prediction, classifies data into one of two states i.e., the output contains bugs or does not contain bugs. Hence, the term *classification*. A classifier refers to a ML algorithm that has the ability to sort data into categories of labelled classes. In other words, a classifier refers to an algorithm that maps input data into specific categories, e.g. classifying raw emails into spam or not spam. This study adopts supervised ML techniques, described in section 2.7.2.1, of binary classification. Therefore, all the ML algorithms applied are classifiers. With this ML technique, models are developed and trained to learn how to map labelled inputs into two categories or groups of outputs, which in this case is, either the existence of bugs in a mobile app or not. This technique is illustrated in Figure 3.2 below:

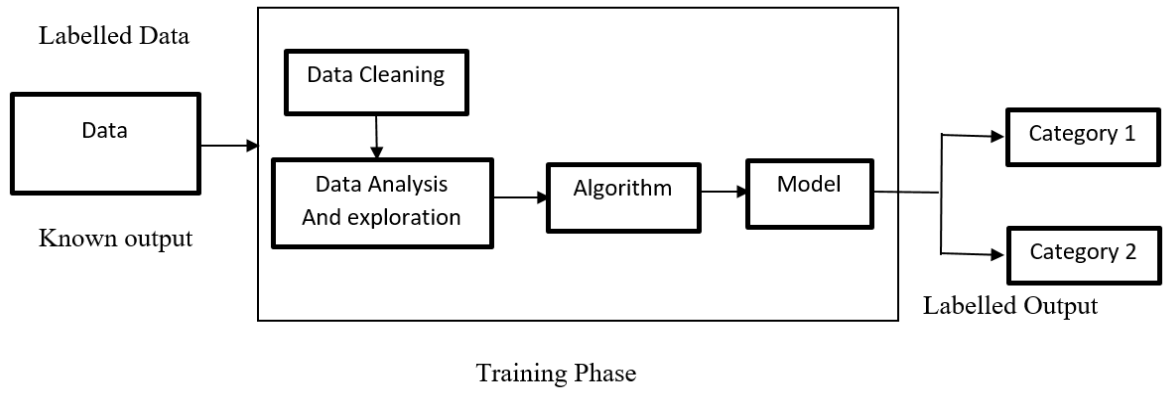


Figure 3.2 The supervised classification process of machine learning

The process of classification will help to determine the reliability of a mobile app through the existence of bugs or not. The selected ML algorithms or classifiers for this study are: RF, SVM, NN, NB and DT, described in section 2.8. These classifiers were evaluated and selected according to their relevance to software reliability prediction and prediction accuracy scores when applied to previous studies detailed in section 2.7 and 2.8. These classifiers learn through many training instances of the data of known labels known as supervised learning which has been discussed in section 2.6. Supervised learning follows a process of training a model, as indicated by the training phase in Figure 3.2 above, to categorise data as indicated by labelled

data in the same figure. The rules adopted through this training process can be applied to unseen data for prediction.

A further description and properties for each selected ML classifier was discussed and tabulated in section 2.8. So once the classifiers are selected, they need to be applied to data and trained to recognise patterns within the data so that they can be applied to unseen data to be evaluated for its prediction accuracy. The next section discusses some evaluation measures that will be used in this study.

3.5.5 Selected machine learning performance evaluation measures

In order to evaluate the learning processes of the ML algorithms and techniques applied to the dataset in this study, various performance measures need to be applied and analysed. Performance is evaluated through measures adapted from the confusion matrix. These measures are: True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN) (Iqbal et al., 2019), as defined in section 2.10. This confusion matrix can then be used to evaluate other performance measures such as accuracy, precision, recall and F-measure scores.

In addition to these measures, another commonly used method to evaluate the effectiveness of the approach is used, namely, the AUC_ROC. These were shortlisted as they have been adopted in many studies as presented in 2.10, and with ML classification techniques for comparing the performance of prediction models (Choudhary, Kumar, Kumar, Mishra, & Catal, 2018; Elish & Elish, 2008; Hammouri, Hammad, Alnabhan, & Alsarayrah, 2018; Rahman & Devanbu, 2013; Wang, Liu, Nam, & Tan, 2018). Therefore, the evaluation measures used for the purpose of the study are defined below in terms of TP, TN, FP and FP.

- *Accuracy* measures the rate of correct classification. Normally denoted by ACC as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FN}$$

- *Precision* (positive predicted value) is calculated as the number of correct positive predictions divided by the number of positive predictions with the best precision being 1 and the worst being 0. It is calculated as:

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

- *Recall* (True positive rate) is calculated as the number of positive predictions divided by the total number of positives. The best recall is 1 and the worst is 0. The formula for recall is:

$$Recall = \frac{TP}{TP + FN}$$

- F-measure (F-score) is the weighted average of precision and recall that takes into account both false positives and false negatives, usually more useful than accuracy, especially with imbalanced datasets. The formula is:

$$F1\ Score = 2 * \frac{Recall * Precision}{Recall + Precision}$$

- AUC_ROC curve for a visual representation of all the ML algorithms prediction accuracies.

Accuracy is the most commonly used evaluation metric for classification models but can be misleading at times with large class imbalances, hence the other measures are needed to evaluate the techniques more accurately (Picek, Heuser, Jovic, Bhasin, & Regazzoni, 2019). Imbalanced datasets have a higher instance of one of the classes compared to the other. For example, a higher instance of more buggy classes than non-buggy classes in a dataset implies an unbalanced dataset. Applying and evaluation ML techniques to unbalanced datasets often show poor results when generalized to unseen data and therefore affect the prediction performance of the prediction process (Thabtah, Hammoud, Kamalov, & Gonsalves, 2020). This study does show an imbalance in the dataset, therefore, based on the discussion above it will not be feasible to apply just the single evaluation measure of accuracy. This study will also apply the measures of Precision, Recall, F1 Score and the AUC_ROC curve analysis.

3.5.6 A summary of selected components, methods and experimental plan for this study

Table 3.4 below lists each required component and the selected items for the study.

Table 3.4 A summary of selected components

Component	Selected Items/Methods for this study
The Dataset	Eclipse JDT core bug prediction dataset
Software Metrics	Source Code Metrics, Process Metrics
Dimensionality Reduction to remove redundant features	Feature Selection using filter and wrapper methods
ML algorithms for classification	RF, SVM, NN, NB, DT
Performance evaluation methods	accuracy, precision, recall, F1 score AUC_ROC curve

The plan for conducting the experiment is made up of five steps or stages. The first step describes the metric sets used for the data collection. The second step includes details with regards to data cleaning methods applied. The third step applies the process of dimensionality reduction to remove redundant features and details the process of feature selection for selecting the best subset of features modelling. The fourth step applies ML modelling techniques for the training and testing processes, and finally, evaluation performance measures are applied to the ML models to help select the most accurate model for prediction.

The flow diagram in Figure 3.3 below briefly describes the steps followed for the experiment.

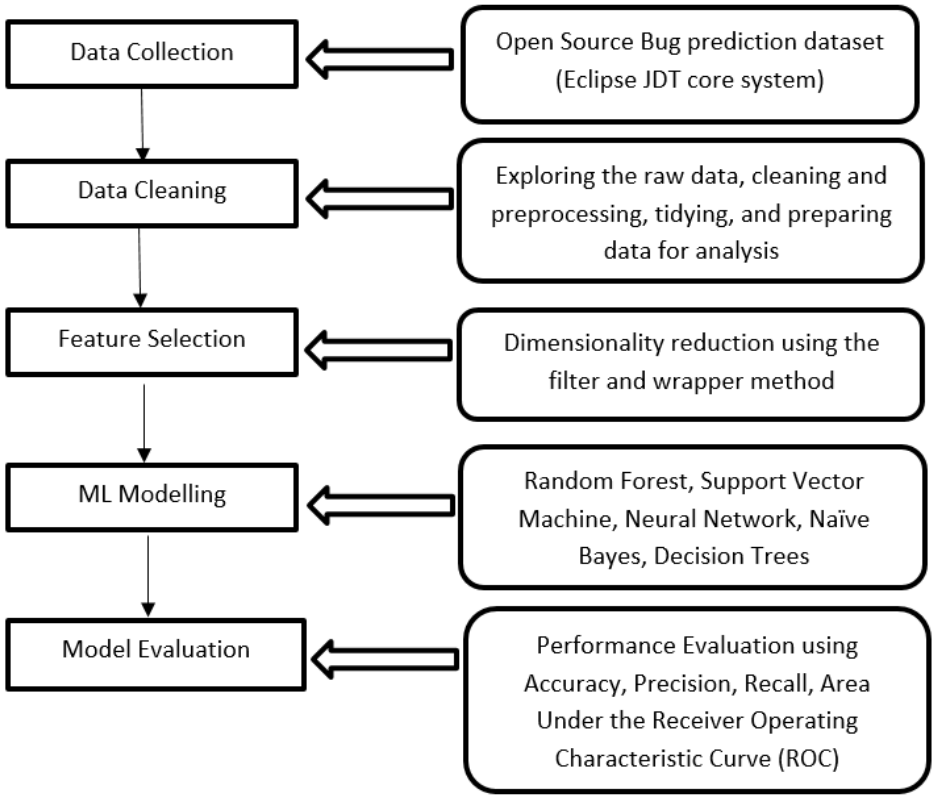


Figure 3.3 Steps for the experiment

3.6 On the credibility of the research

Due to the complexity of the experiment, it becomes vital to ensure validity, reliability and generalisability of results. The following sub-sections describe how this was achieved for the study.

3.6.1 Validity

In research validity made up of two essential parts: internal and external validity (Lakshmi & Mohideen, 2013). Internal validity examines the trustworthiness of a study in terms of how it was designed, conducted and analysed. External validity refers to whether the findings of the study are able to generalise to other contexts (Andrade, 2018).

This study follows a sequence of activities, highlighting aspects of the ML process, discussed in section 3.5, needed to aid with successful predictions in software reliability. Among various technologies (Zhai, Yin, Pellegrino, Haudek, & Shi, 2020) advocates ML as a useful technology for scientific assessment. It has the ability to analyse complex constructs and make predictions on complex data without much human effort in analysis and scoring. In this study ML is used to find patterns in software data and categorise software into having bugs or not

and thereby predict if software is either reliable or not. Unlike traditional statistics, ML can analyse large amounts of data with a multitude of variables using complex mathematical functions repeatedly and at high speeds (Zhai, Krajcik, & Pellegrino, 2020).

This study applies ML analysis to a dataset that contains 997 software classes with metrics and 91 versions of the Eclipse JDT core system with 9135 transactions and 463 post-related defects as described in section 3.5.1. In such a large and diverse dataset, methods beyond the traditional statistical linear regression is required. ML provides a solution that applies complex algorithms to data, minimising human intervention to automate processes and reduce tedious manual workloads. Therefore, applying ML processes and techniques to this study ensures its internal validity.

For this study, to obtain unbiased results and generalise to other contexts, the findings need to be applied to unseen data. In order to achieve this, it is important to apply a resampling procedure to evaluate ML models on a limited data sample. For this study the 10-fold cross-validation method was employed. This method divides the input dataset into 10 folds of equal size. Each time nine parts are used for training and one part is used for testing. It must be noted that in each iteration of this process, the training data is different from the testing data. This is repeated 10 times for further validation, with a total of 100 iterations in which results from each fold are combined to produce the validation (Wei, Hu, Chen, Xue, & Zhang, 2019).

The 10-fold cross validation method ensures external validity to estimate how the machine learning model is expected to perform in general when making predictions. Therefore, the ML techniques and processes applied in this study ensure the validity of the research through the use of complex algorithms and mathematical functions to find optimal solutions to aid in the reliability prediction of apps.

3.6.2 Reliability

Reliability measures the stability and accuracy of methods used in a study with the ability to provide consistent results each time it is applied (Souza, Alexandre, & Guirardello, 2017). There are many studies in literature regarding software reliability prediction which states that such software problems are extremely complex and requires specialised tools to handle such complexities (Hammouri, Hammad, Alnabhan, & Alsarayrah, 2018).

Studies point to the use of ML techniques, which are essentially, specialised algorithms built to handle such complexities in a customised manner (Singh & Chug, 2017). This study follows the steps of the ML workflow process explained in section 3.5 which is customised through the

use of selected ML techniques applied to data for software reliability prediction as listed in Table 3.4. Explained further, these include the univariate and multivariate filter methods for data cleaning and pre-processing, wrapper methods for feature selection, applying ML algorithms to data for training and testing for prediction accuracy using the 10-fold cross validation method.

Prediction accuracy is evaluated through scoring measures of precision, accuracy, recall and also through AUC_ROC curve analysis technique. Applying these ML techniques ensures consistent measures over time, hence ensuring reliability of the processes applied and the results. It is unlikely that the same exact results will be given every time due to different data samples and changes over time, however, applying these techniques provide consistency and should show a strong correlation between the results and software reliability prediction, thus ensuring reliability for this study.

3.6.3 Generalisability

Generalisability relates to how confidently the results gathered from the study can be extended and applied to the greater population in other settings (Ferguson, 2004). If the results are broadly applicable to different types of situations, then the study is deemed to display good generalisability. This study applies ML techniques to data extracted off a dataset that contains a large number of class files used for bug prediction, i.e. 997 classes over 91 versions. Data is categorised based on specific features of source code metrics and process metrics as described in section 2.9.2 and 3.5.2.

Following through on the ML process by applying its techniques to these sets of data, help determine which ML models, of data and algorithms, provide an optimal solution to assist with reliability prediction of apps. As explained in section 3.6.1, applying complex algorithms with reduced human effort and automation ensures validity, which can also be extended to generalisability (Muijs, 2004). Also, this study uses cross validation techniques, as discussed in section 3.6.1, to ensure results are unbiased and can generalise to other contexts, through resampling techniques to assess how the results will generalise in practice. This is achieved through this technique by splitting the selected data into training (known data) and testing (unknown or first seen data) sets. Training data is used for the purpose of learning and testing data is used to validate the results obtained for prediction. Combining all these ML processes and applying these techniques, ensure generalisability of the results obtained for this study.

3.7 Summary of the chapter

This chapter presented the research design aspects for the research paradigm, methodology and strategy underpinning the study. A description for each aspect was presented together with a motivation for the selected choices. These include the post-positivist paradigm with a quantitative research strategy, applying the EA research methodology. The chapter went on to describe the experimental approach used in the study with regards to the dataset, dataset features, ML techniques of feature selection, ML algorithms and evaluation measures. These have been identified and summarised in Table 3.4.

The plan to conduct the experiment depicting the various stages was defined and summarised in Figure 3.3. The last section of the chapter outlined the credibility of the research in terms of validity, reliability and generalisability of the results. This forms a vital component of the study due to its complexity in applying ML for the purpose of software reliability prediction. This chapter intended to identify all aspects needed for the research methodology and experimental design for the purpose of applying ML techniques to data with an intention to obtain useful and valid results to assist with reliability prediction of apps. The next chapter focusses on a discussion of the experiments conducted in the study.

CHAPTER 4: DISCUSSION OF THE EXPERIMENTS

This chapter includes sections that discuss the application of the ML techniques selected to conduct the experiments focussed on achieving RO4. In keeping with the research aim of applying ML techniques to assist with the reliability prediction of apps, the study conducts three experiments that apply the same ML techniques to three metric sets of data to determine which category of metrics are most influential to the app reliability prediction.

Metric sets are defined as the source code metric set, the process metric set and a combination of both. All experiments were conducted using the Python programming language. The first section describes the features of metric sets used for the data collection. The second section includes details of data cleaning methods used. The third section discusses the process of dimensionality reduction to remove redundant features and details the process of feature selection that was used to select the best subset of features from a metric set based on the observations, features and target responses. The fourth section explains the ML modelling techniques used for the training and testing of the filtered dataset mentioned above, and finally there is an explanation of the ML evaluation methods used to compare the performance of the selected classifiers best suited for prediction. The flow diagram in Figure 4.1 below lists each section that is discussed in this chapter.

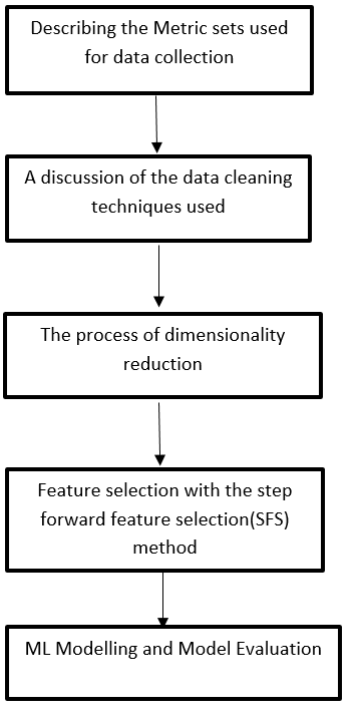


Figure 4.1 Stages for this chapter

4.1 Data collection – metric sets

The dataset used for this study is a public *bug* prediction dataset for the Eclipse JDT core software system as described in section 3.5.1. Selected data is extracted off this dataset in terms of two categories of software measures obtained from the software classes within the dataset. These are referred to as the source code metrics and the process metrics. The motivation of choice for the use of these metrics is explained in section 2.9.2. For each metric set there are a total of 997 of the *same* observations and target responses, but each has its own feature set. Each observation has a target response that exists in one of two states, that is, 0 if no bugs exist and 1 if bugs do exist. As a result, these labelled metric sets use supervised ML techniques with binary classification as defined in section 2.7.2.1 and section 3.5.4. Table 4.1 below lists the features within each metric set used in the study.

Table 4.1 Details of metric sets used in this study

Metric Set	Features	#Features
Source Code Metrics	cbo, dit, fanIn, fanOut, lcom, noc, numberOfAttributes, numberOfAttributesInherited, numberOfLinesOfCode, numberOfMethods, numberOfMethodsInherited, numberOfPrivateAttributes, numberOfPrivateMethods, numberOfPublicAttributes, numberOfPublicMethods, rfc, wmc	17
Process Metrics	numberOfVersionsUntil, numberOfFixesUntil, numberOfRefactoringsUntil, numberOfAuthorsUntil, linesAddedUntil, maxLinesAddedUntil, avgLinesAddedUntil, linesRemovedUntil, maxLinesRemovedUntil,	15

	avgLinesRemovedUntil, codeChurnUntil, maxCodeChurnUntil, avgCodeChurnUntil, ageWithRespectTo, weightedAgeWithRespectTo	
Combined Metrics	A combination of all features above	32

To progress with this research all metric sets need to be imported into the Python programming environment for further processing and analysis. Tables 4.2, 4.3 and 4.4 below illustrates a part of each metric set after applying the necessary code to import and display content.

Table 4.2, for the source code metric set, shows a part of 997 observations represented by the rows and eighteen columns.

Table 4.2 A part of the Metric Set for Source Code Metrics

numberOfMethodsInherited	numberOfPrivateAttributes	numberOfPrivateMethods	numberOfPublicAttributes	numberOfPublicMethods	rfc	wmc	bugs
19	0	0	1	5	34	20	0
8	0	0	2	1	1	1	0
8	0	1	3	19	156	176	1
207	0	0	0	4	18	12	0
8	0	2	7	1	174	115	0
...	...	...	...	...	...	...	...
8	0	0	2	2	2	1	0
95	0	1	3	3	27	14	0
27	2	0	6	6	94	88	1
30	2	0	0	19	47	29	0
25	22	0	0	39	105	100	1

Seventeen columns represent the seventeen features in this set of source code metrics, with the last column representing the target responses labelled *bugs*. The *bugs* column is populated with 0's and 1's: 0 for the existence of no bugs and 1 for the existence of bugs. This occurs as the target response column for all datasets used.

Table 4.3 below, for the process metric set, shows part of the dataset, again with the same 997 observations represented by the rows from Table 4.2.

Table 4.3 A part of the Metric Set for Process Metrics

	numberOfVersionsUntil	numberOfFixesUntil	numberOfRefactoringsUntil	numberOfAuthorsUntil	linesAddedUntil	maxLinesAddedUntil	avgLinesAddedUntil
0	65	4	0	8	608	158	9.35385
1	2	0	0	2	10	10	5.00000
2	120	10	0	12	1361	99	11.34170
3	28	4	0	5	138	39	4.92857
4	93	17	0	8	1870	684	20.10750
...	...	...	...	...	...	...	...
992	58	14	0	6	1017	246	17.53450
993	35	5	0	9	322	48	9.20000
994	34	5	0	7	690	119	20.29410
995	3	3	0	1	32	27	10.66670
996	51	8	0	10	2071	317	40.60780

This dataset shows different features relating to process metrics, represented by sixteen columns. Fifteen columns represent the features of this dataset, and the last column contains the target response values, as mentioned from Table 4.2 above.

Table 4.4 shows a part of the combined metric set, and once again, it is made up of the 997 observations as represented by the rows.

Table 4.4 A part of the Metric set for Combined Metrics (Source code and Process metrics)

	cbo	dit	fanIn	fanOut	lcom	noc	numberOfAttributes	numberOfAttributesInherited	numberOfLinesOfCode	numberOfMethods	...	avgLinesAddedUntil
0	9	2	1	9	15	0	1	8	122	6	...	9.35385
1	1	1	1	0	0	0	2	0	4	1	...	5.00000
2	114	1	102	18	190	6	131	249	484	20	...	11.34170
3	5	6	1	4	10	0	0	61	33	5	...	4.92857
4	23	2	1	22	820	0	7	416	673	41	...	20.10750
...	...	...	...	...	...	...	...	...	...	...	...	...
992	0	1	0	0	1	2	3	0	3	2	...	17.53450
993	9	6	2	7	15	1	3	386	48	6	...	9.20000
994	35	3	25	10	153	9	11	52	306	18	...	20.29410
995	7	2	0	7	190	0	2	6	87	20	...	10.66670
996	8	2	4	4	780	0	49	20	347	40	...	40.60780

The combination dataset is made up of a total of 32 columns from the source code and process metrics datasets. The last column again represents the target responses for the existence of bugs or not.

4.2 Data cleaning and pre-processing

Bowes, Hall, and Petrić (2018) clearly demonstrated the significant impact of data on the performance of a prediction system. The quality of data used to build models plays an important role in the model's prediction performance. Datasets are often termed as being *noisy*. Noisy data relates to various aspects of the dataset in terms of its arrangements, formats, construction, duplications, extra spaces and missing values, all of which can skew and compromise the

prediction system (Jiang, Lin, Cukic, & Menzies, 2009). For these reasons, data needs to undergo cleaning and pre-processing before being applied for prediction.

4.2.1 Dimensionality reduction with filtering methods

Features are also an important part that needs to be considered when developing prediction models. Particularly in the case of repeated and related feature attributes, these have shown to over fit models, creating bias prediction systems. Such features not only affect the performance of the algorithm but also slow down the training time of the algorithm. Applying filtering methods to a dataset seems to improve the performance of ML models (Bowes, Hall, & Petrić, 2018; Menzies, Greenwald, & Frank, 2006; Shepperd, Bowes, & Hall, 2014; Singh & Chug, 2017).

Filter methods carry out the feature selection pre-processing step that selects features independently of ML models, making it one of the faster methods of feature selection. This method uses statistical dependencies and other general characteristics of the training data to select features. The drawback, however, is that this method tends to select subsets with a high number of features, therefore a *variance threshold* is needed. Variance threshold in feature selection applies a simple threshold or baseline value that is used to remove features. For example, zero variance features are features that have the same value in all samples and need to be removed. This section discusses the filtering methods that have been applied for dimensionality reduction. Two types of filtering methods have been used, namely, univariate and multivariate, and each is described below.

4.2.1.1 Univariate filter method

The best features are selected through univariate statistical tests. These tests are more specifically known as ANOVA (Analysis of Variance) and they measure the dependence of two variables. In the case of dataset X, each feature in X is compared to the binary target variable y. This is done one feature at a time to establish some statistically significant relationship between the features and the target variable. With the univariate filter method, each feature is treated individually and is independent of the feature space. The feature space refers to all the other dimensions of features in the dataset. As per Figure 4.5 below, X refers to the dataset matrix of feature values and y refers to the target response values, which in this case is the existence of bugs or not.

```
In [8]: X=data.drop('bugs',axis = 1)
        y=data['bugs']
        X.shape,y.shape

Out[8]: ((997, 17), (997,))
```

Code Snippet 1: Python code illustrating matrix X with response value y

In Code Snippet 1, X is extracted from the dataset, removing the response column called 'bugs' and y is then set as the 'bugs' column. The output shows the size of both X and y. The output translates to X having a total of 997 observations with 17 features that make up the matrix of the dataset. The variable y just has one column with 997 target responses which is either a 0 for no bugs and 1 if bugs exist.

The Variance Threshold feature selection algorithm was used for the removal of all low variance features relating to constants and quasi constants. Constant features refer to those that contain one value for all outputs in the dataset and are of no use to classification. Quasi constant features are those that have the same value for a large subset of dataset outputs and are not useful for predictions. Duplicate features were also removed to prevent over fitting of the predictive model. Duplicate features have attributes that are identical to each other.

4.2.1.1.1 Constant feature removal

The Python code in Code Snippet 2 below illustrates how constant features are removed from the dataset.

```
In [54]: #Constant feature removal
         constant_filter=VarianceThreshold(threshold=0)
         constant_filter.fit(X)

Out[54]: VarianceThreshold(threshold=0)

In [55]: constant_filter.get_support().sum()

Out[55]: 17

In [56]: X_filter=constant_filter.transform(X)

In [51]: X_filter.shape, X.shape

Out[51]: ((997, 17), (997, 17))
```

Code Snippet 2: Python code illustrating constant feature removal

The variance set at 0 means that there is 0% non-similarity and all features that are 100% similar will be removed. This filter method is then applied to matrix X and the output display of 17 suggests that after removing constant features, a total of 17 features remain. Hence this implies that there were no constant features in this dataset as 17 was the initial number of features listed to begin with. The process also requires the constant features be filtered out of the dataset **X**, and this is obtained by using the *transform* method. The new filtered dataset now exists as **X_filter**. The last statement compares the size of the filtered dataset and the original dataset and the output clearly shows that the size of both datasets is the same as there were no constant features to remove.

4.2.1.1.2 Quasi-constant feature removal

Quasi constants refer to features that are almost constant. These are features that have the same value for a very large subset of outputs. In other words, if the variance is low, i.e. there is a small amount the feature varies from the mean, then that feature is regarded as a quasi-constant. The features are not useful when making predictions and may cause overfitting and overheads on the ML models.

Code Snippet 3 below illustrates how quasi-constant features are removed from the dataset.

```
In [14]: #Quasi Constant
         quasi_constant_filter=VarianceThreshold(threshold=0.01)

In [52]: quasi_constant_filter.fit(X_filter)
Out[52]: VarianceThreshold(threshold=0.01)

In [53]: quasi_constant_filter.get_support().sum()
Out[53]: 17

In [57]: X_quasi_filter=quasi_constant_filter.transform(X_filter)

In [58]: X_quasi_filter.shape, X.shape
Out[58]: ((997, 17), (997, 17))
```

Code Snippet 3: Python code illustrating quasi-constant feature removal

Once again, the *Variance Threshold* function is used and this time the threshold is set at 0.01. This means that the variance is set at 1% of non-similarity and all features that are 99% similar will be removed. In other words, if the variance of values within a feature set or column is less

than 0.01 then this column or feature set will be removed. The rest of the steps are very similar to those described in the previous section on constant feature removal and again it can be seen that there were no quasi-constants in this dataset.

4.2.1.1.3 Duplicate feature removal

Duplicate features refer to features that have very similar values. The features merely delay the training time of an algorithm and add no value to the prediction system. These therefore need to be removed from the dataset before training occurs. Unlike constant removal and quasi-constant removal, with duplicate feature removal a *pandas dataframe* is used to remove duplicate rows. Pandas is a widely used Python library that allows for the manipulation of data into tables of columns and rows, known as a dataframe. Since features exist as columns in a dataset, these now need to be transposed into rows. Code Snippet 4 below illustrates how duplicate features are removed from the dataset using the *pandas dataframe* and the transpose method.

```
In [64]: # Duplicate Features
         X_T = X_quasi_filter.T

In [65]: type(X_T)
Out[65]: numpy.ndarray

In [66]: X_T = pd.DataFrame(X_T)

In [67]: X_T.shape
Out[67]: (17, 997)

In [68]: X_T.duplicated().sum()
Out[68]: 0
```

Code Snippet 4: Python code illustrating duplicate feature removal

Code Snippet 4 above transposes the dataset and stores it into **X_T** dataframe. Looking at the shape of the dataset it can clearly be seen that the transposed dataset now has 17 rows and 997 columns. Pandas contains a method called **duplicated()** which helps determine duplicate rows from the dataframe. The **sum()** method is then used to find the total number of duplicated features in the dataset and these can then be dropped using the **drop_duplicates()** method. The

sum of duplicates above displays 0, meaning that there were no duplicate features in this dataset.

The univariate filter method used here shows how a dataset passes through a kind of filtration system in which each algorithm used acts like a filter, removing features as it passes from one feature removal algorithm to the next. This pre-processing method is carried out to select the best features for greater prediction. Each feature is ranked according to a criterion and the highest ranking features are selected for training the predictive models. The problem with univariate filter methods is that features are considered individually and this method does not consider the relationship between other features within the dataset.

4.2.1.2 Multivariate filter method

The multivariate filter method, on the other hand, does consider the entire feature space and takes into account the relationship between all features in the dataset. Apart from duplicates and constant features, a dataset may also contain correlated features. Correlated features refer to those features that are close to each other within a linear space. If two or more features are closely correlated, they are bound to convey redundant information to the model creating overfitting and inaccurate predictions. Therefore, correlated features need to be removed and just one of the correlated features should be retained in the dataset. The following subsection discusses the Pearson correlation coefficient that has been selected as the multivariate method to measure the correlation between features.

4.2.1.2.1 Pearson correlation coefficient

This measure is commonly used in ML to summarise the linear relationship between two variables with a value between 1 and -1. The Pearson correlation method is used to remove all correlated features by a certain point. For this study, if the correlation coefficient is more than 0.85 or less than -0.85 this means that those features are highly correlated and are not useful for prediction. Also, removing these correlated features should improve and reduce the training time of the ML model. Code Snippet 5 below illustrates the use of the Pearson correlation method to help identify the highly correlated features and eventually remove them from the dataset. This was applied to all three metric sets used in this study.

```
In [137]: X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.2,random_state=0, stratify=y)

In [134]: def get_correlation(data,threshold):
              corr_column=set()
              corrmatrix=data.corr()
              for i in range(len(corrmatrix.columns)):
                  for j in range(i):
                      if abs(corrmatrix.iloc[i,j])>threshold:
                          colname=corrmatrix.columns[i]
                          corr_column.add(colname)
              return corr_column

In [135]: corr_features=get_correlation(X_train,0.85)
          corr_features
```

Code Snippet 5: Python code illustrating use of Pearson Correlation Coefficient

Before any feature removal can take place, the data has to be split into a training and testing set. To identify the correlated features, the **get_correlation()** function is designed. This function makes use of the **corr()** method from the Pandas dataframe that returns a correlation matrix. This correlation matrix contains the correlation between all the columns of this dataframe. A loop is used to compare columns in the correlation matrix to the threshold correlation, which is set to a value of 0.85, as seen in the script above. A highly correlated column is then added to the set of correlated columns which is returned to **corr_features,** as illustrated above. These correlated columns can then be removed from the dataset.

A heatmap makes it easy to visualise these correlation matrices. The heatmaps below have been plotted in Python using the Seaborn library. Each heatmap illustrates the correlation coefficients of features across the three metric sets used in this study, i.e. source code metrics, process metrics and a combination of both. The results are illustrated below.

Pearson correlation applied to the source code metrics dataset

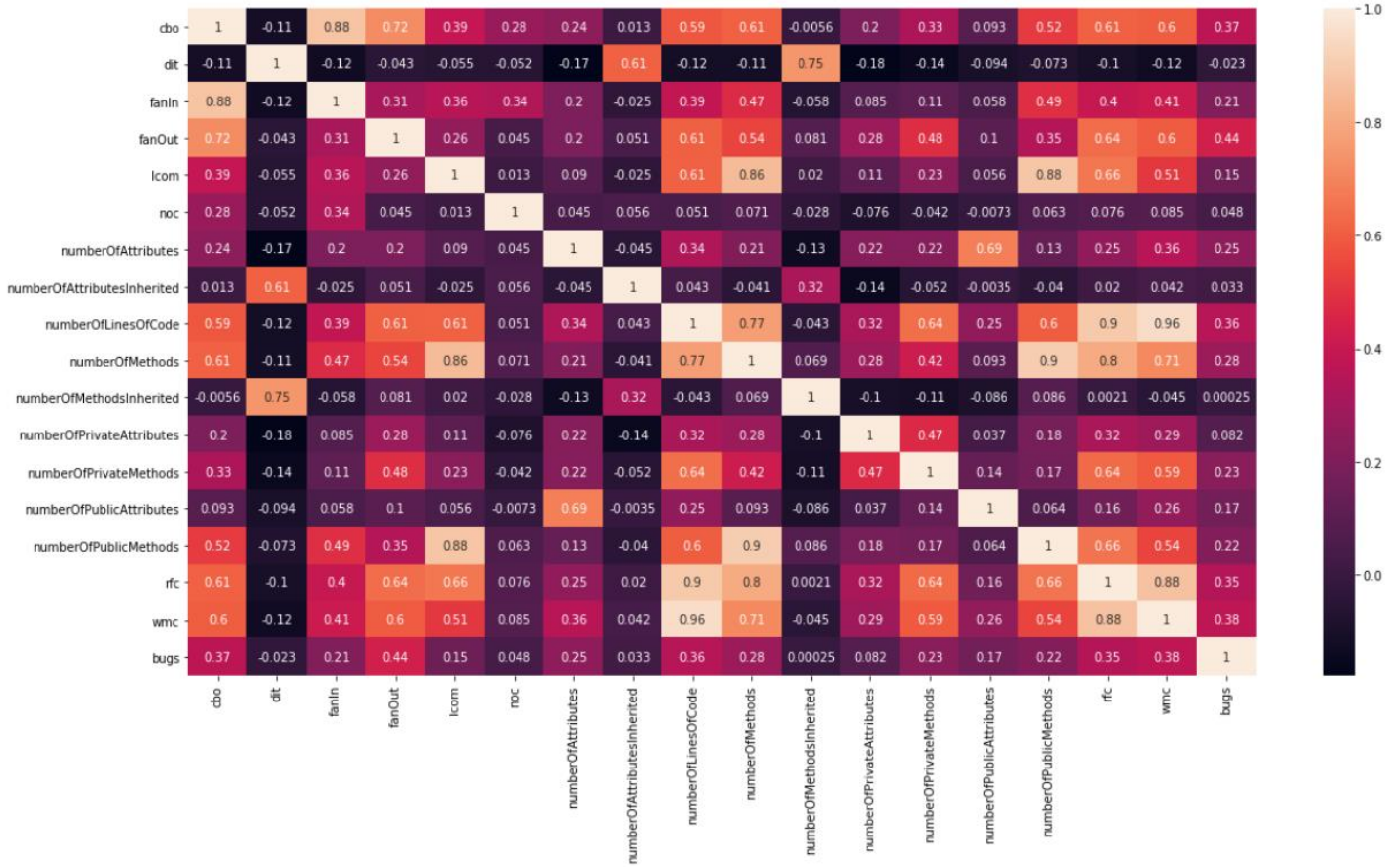


Figure 4.2 Heatmap of correlated features for Source Code Metrics

For the source code metrics dataset, looking at the last row called *bugs,* in the heatmap in Figure 4.2 above, the highly correlated features are fanIn (0.21), rfc (0.35), wmc (0.38), numberOfMethods (0.28), numberOfPublicMethods (0.22) and these need to be removed from the dataset. Using Pearson Correlation as illustrated in Code Snippet 5 above, these features are identified. A call is made to the **get_correlation()** method, with the training data and the threshold of 0.85, passed through as parameters to this method. Features are then displayed from **corr_features** and then removed using the **.drop()** method. This is illustrated in the Code Snippet 6 below. The output clearly shows the five correlated features, as visualised in the heatmap above. These features are then dropped or removed from the dataset which now has a total of 12 features. The initial dataset contained 17features.

```
In [135]: corr_features=get_correlation(X_train,0.85)
          corr_features

Out[135]: {'fanIn', 'numberOfMethods', 'numberOfPublicMethods', 'rfc', 'wmc'}

In [138]: len(corr_features) # number of highly correlated features

Out[138]: 5

In [139]: X_train_uncorr=X_train.drop(labels=corr_features, axis=1)
          X_test_uncorr=X_test.drop(labels=corr_features, axis=1)

In [140]: X_train_uncorr.shape, X_test_uncorr.shape

Out[140]: ((797, 12), (200, 12))
```

Code Snippet 6: Python code to display and remove correlated features for the source code metrics

Pearson correlation applied to the process metrics dataset

This exact process was applied to the process metrics dataset and to the combination of both source code and process datasets and the results were as follows:

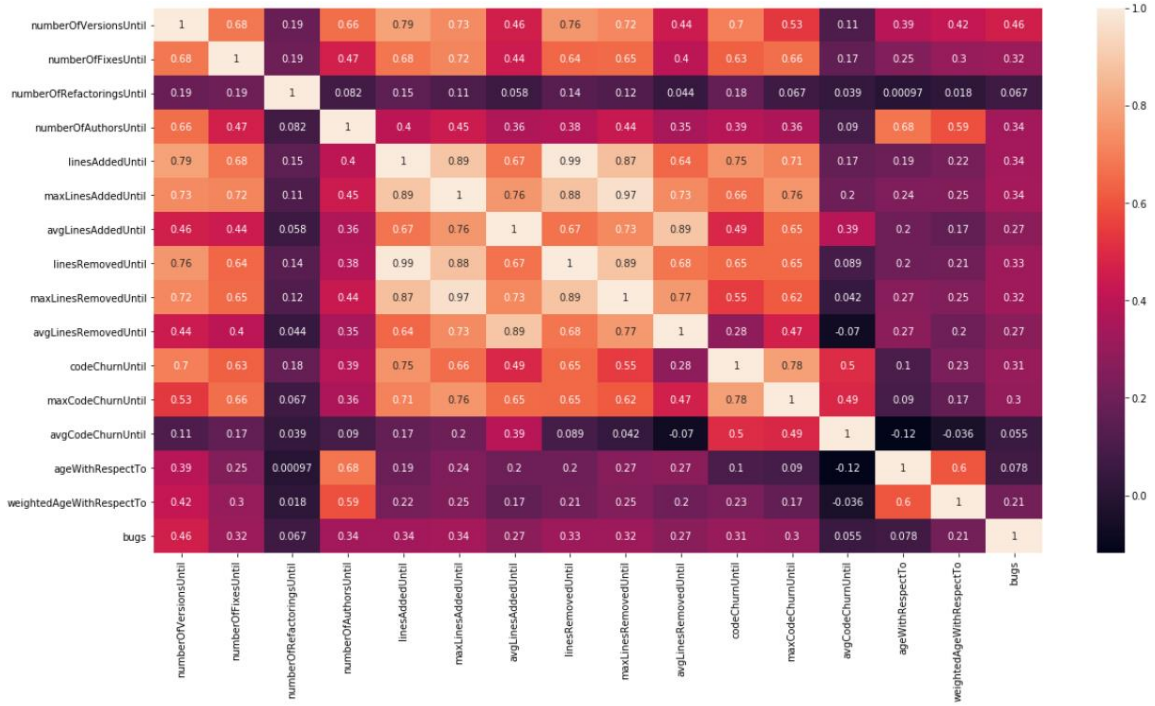


Figure 4.3 Heatmap of correlated features for Process Metrics

The heatmap in Figure 4.3 above shows that the highly correlated features with respect to the bugs row, are avgLinesRemovedUntil (0.27), linesRemovedUntil (0.33) and maxLinesRemovedUntil (0.32). Using the Pearson correlation method illustrated in the Python code below, these features are then dropped from this dataset. There was an initial total of 15 features and after applying the Pearson correlation filter method to this dataset, there is now a total of 12 features. This can be seen in Code Snippet 7 below.

```
In [27]: corr_features=get_correlation(X_train,0.85)
         corr_features
Out[27]: {'avgLinesRemovedUntil', 'linesRemovedUntil', 'maxLinesRemovedUntil'}

In [52]: len(corr_features) # number of higly correlated features
Out[52]: 3

In [53]: X_train_uncorr=X_train.drop(labels=corr_features, axis=1)
         X_test_uncorr=X_test.drop(labels=corr_features, axis=1)

In [56]: X_train_uncorr.shape, X_test_uncorr.shape
Out[56]: ((797, 12), (200, 12))
```

Code Snippet 7: Python code to display and remove correlated features for the Process Metrics

Pearson correlation applied to the combination dataset

With the combined feature dataset, the highly correlated features with respect to bugs are, avgLinesRemovedUntil (0.26), fanIn (0.21), linesRemovedUntil (0.32), MaxLinesAddedUntil (0.33), maxLinesRemovedUntil (0.26), numberOfMethods (0.20), numberOfPublicMethods (0.22), rfc (0.35) and wmc (0.38). Applying the Pearson correlation method, these 9 features are then removed from this dataset reducing the total number of features from a total of 32 to 23. This is the largest dataset as it combines both the source code and process metrics datasets. The Figure 4.4 below shows the heatmap highlighting the correlated features.

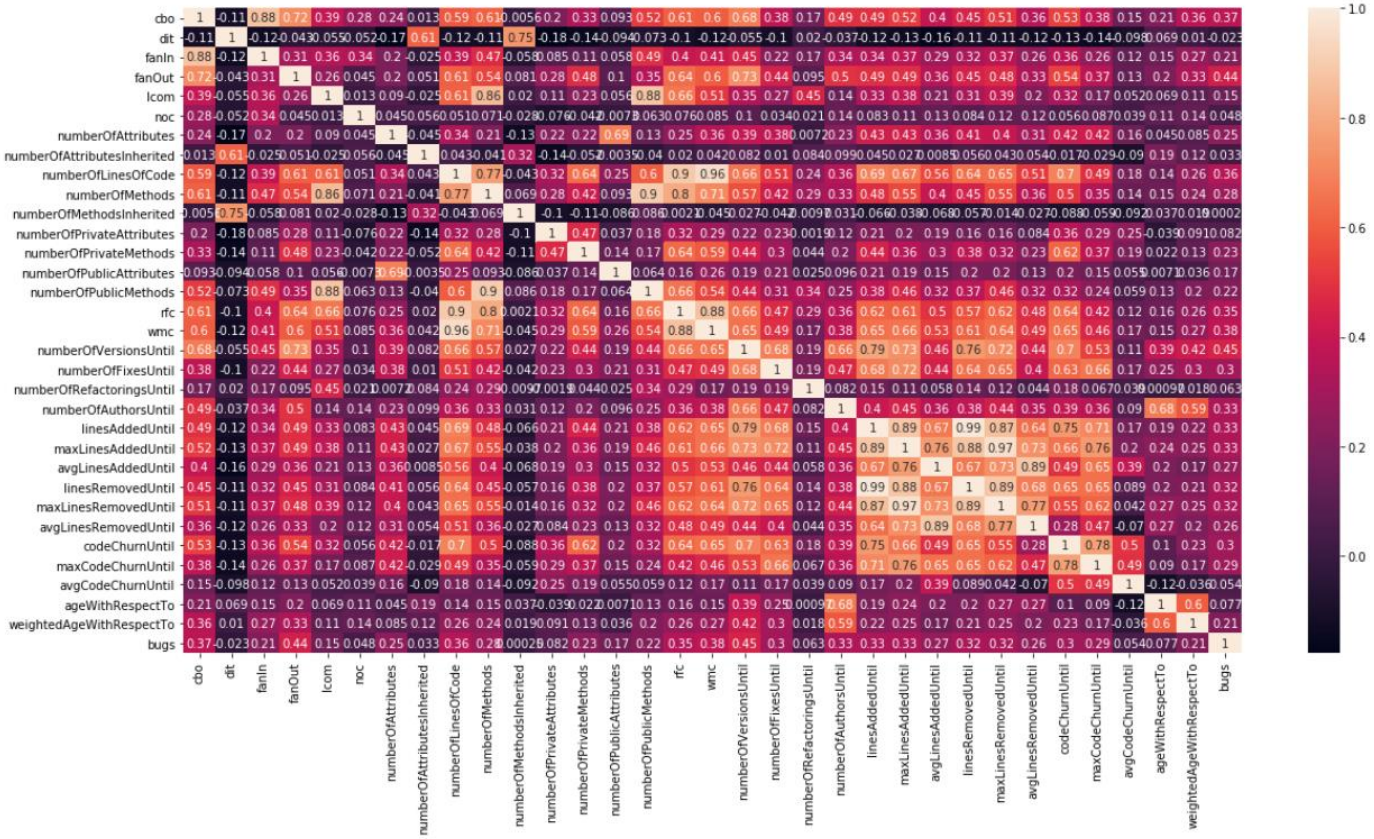


Figure 4.4 Heatmap showing correlated features for Combination Metrics

Code Snippet 8 implements the Pearson correlation for identifying and removing the correlated features from this dataset.

```
In [27]: corr_features=get_correlation(X_train,0.85)
         corr_features

Out[27]: {'avgLinesRemovedUntil',
          'fanIn',
          'linesRemovedUntil',
          'maxLinesAddedUntil',
          'maxLinesRemovedUntil',
          'numberOfMethods',
          'numberOfPublicMethods',
          'rfc',
          'wmc'}

In [28]: len(corr_features) # number of higly correlated features

Out[28]: 9

In [29]: X_train_uncorr=X_train.drop(labels=corr_features, axis=1)
         X_test_uncorr=X_test.drop(labels=corr_features, axis=1)

In [30]: X_train_uncorr.shape, X_test_uncorr.shape

Out[30]: ((797, 23), (200, 23))
```

Code Snippet 8: Python code to display and remove correlated features for the Combination Metrics

Therefore, the filter methods selected for this study helped eliminate all irrelevant, redundant, constant, duplicated and correlated features. This then resulted in dimensionality reduction for

each of the three datasets used in this study. The next part of the study implements a wrapper method known as the Step Forward Feature Selection (SFS) method that helps select the best performing subset of features for the predictive model.

4.2.2 Feature selection using the step forward feature selection wrapper method

This section looks at the application of the feature selection process using the SFS technique. The purpose of applying this technique will select, through a stringent iterative process, the best subset of features for ML modelling.

The SFS method is an iterative method that starts by evaluating each feature in the model. Iteratively, features are added and evaluated until a subset of the best performing features are selected based on accuracy scores. Unlike the filter methods used above, the wrapper method includes the use of a ML algorithm to help in selecting the best features.

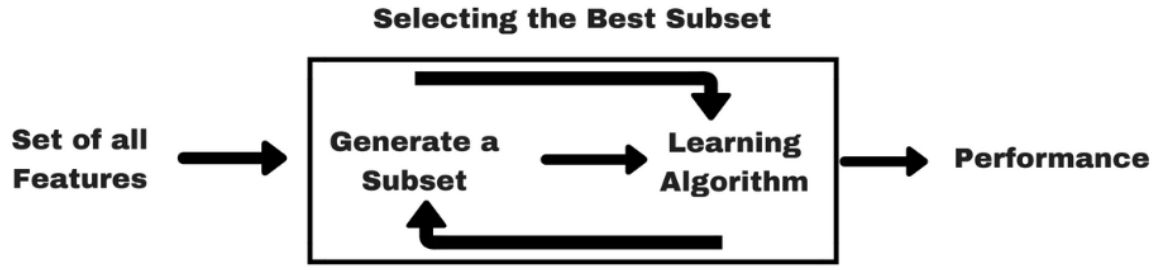


Figure 4.5 Wrapper method for feature selection.

As can be seen in Figure 4.5 above, the wrapper method uses a subset of features and trains the model using them. In this iterative process, inferences are drawn from the previous model and features are then added or removed from the feature subset to obtain the best performing subset. Wrapper methods are computationally expensive and are therefore not recommended for a very large dataset with a large number of features. The diagram in Figure 4.6 below illustrates how the SFS method is implemented to obtain the best subset of features from a dataset.

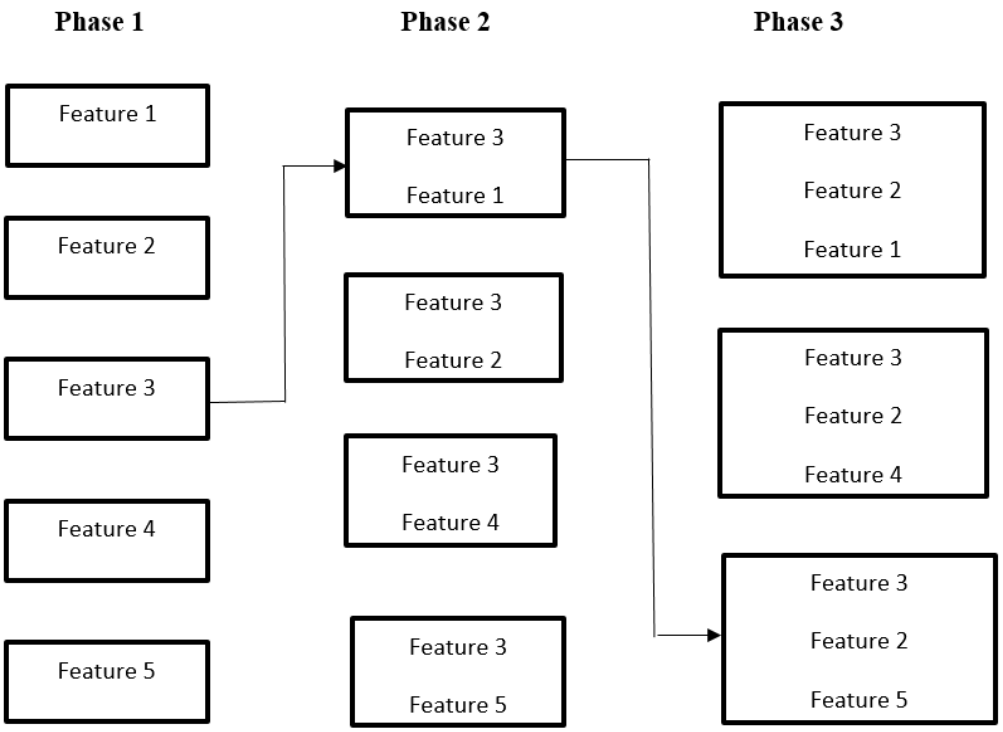


Figure 4.6: A representation of the step forward feature selection(SFS) method. Adapted from: Matlab mlxtend library

In the first phase of this method, the performance of the classifier is evaluated on each feature and the best performing feature is selected. In the second phase the first feature is combined with the other features and then the combination of the best two features is selected; this process continues until it reaches the desired number of best performing features for the model.

In order to perform the SFS process in Python, the mlxtend library is required and must be installed. This library includes the implementation of Sequential Feature Selection Algorithms (SFAs) used to select relevant features. The Random Forest classifier is used for this feature selection process and for model building. The Python code for this process is shown in Code Snippet 9 below:

```
In [40]: #Step forward feature Selection
         sfs=SFS(RandomForestClassifier(n_estimators=100, random_state=0, n_jobs=-1),
                 k_features=12,
                 forward=True,
                 floating=False,
                 verbose=2,
                 scoring='accuracy',
                cv=4,
                n_jobs=-1).fit(X_new,y)
```

Code Snippet 9: Python code illustrating the step forward feature selection method

The SFS algorithm is able to add or remove features one at a time based on the performance of the Random Forest Classifier(RFC) until it reaches a desired subset of k features. This method is applied to all three datasets used in this study and the results are discussed below. Using the Panda's data frame, there are feature sets selected at each iteration of the SFS together with measures such as accuracy scores, feature names and indexes and others listed in the diagrams below for each of source code metrics, process metrics, and the combination metrics.

4.2.2.1 Source code metrics: SFS results

Code Snippet 10 below show the results obtained after applying the SFS method to the source code metrics dataset. It shows the input function applied and the resulting output, displayed in a dataframe. It can be seen that the highest average accuracy score (represented by the **avg_score** column) of 0.858582 is obtained when all 12 features (represented by the **feature_idx** column) are selected.

```
In [44]: pd.DataFrame.from_dict(sfs.get_metric_dict()).T
```

Out[44]:

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
1	(2,)	[0.84, 0.8554216867469879, 0.8152610441767069,...	0.838514	(fanOut,)	0.0234001	0.0145979	0.00842809
2	(2, 11)	[0.844, 0.8152610441767069, 0.8232931726907631...	0.826462	(fanOut, numberOfPublicAttributes)	0.0170611	0.0106434	0.00614496
3	(2, 4, 11)	[0.824, 0.8514056224899599, 0.8433734939759037...	0.830498	(fanOut, noc, numberOfPublicAttributes)	0.0298776	0.0186387	0.0107611
4	(0, 2, 4, 11)	[0.844, 0.8313253012048193, 0.8313253012048193...	0.825458	(cbo, fanOut, noc, numberOfPublicAttributes)	0.0292228	0.0182303	0.0105252
5	(0, 2, 4, 6, 11)	[0.84, 0.8514056224899599, 0.8232931726907631,...	0.833494	(cbo, fanOut, noc, numberOfAttributesInherited...	0.0207357	0.0129357	0.00746844
6	(0, 2, 3, 4, 6, 11)	[0.86, 0.8514056224899599, 0.8473895582329317,...	0.85255	(cbo, fanOut, lcom, noc, numberOfAttributesInh...	0.00737858	0.00460304	0.00265757
7	(0, 1, 2, 3, 4, 6, 11)	[0.86, 0.8473895582329317, 0.8473895582329317,...	0.855562	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0137677	0.00858882	0.00495876
8	(0, 1, 2, 3, 4, 6, 10, 11)	[0.872, 0.8714859437751004, 0.8313253012048193...	0.85555	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0275086	0.0171609	0.00990785
9	(0, 1, 2, 3, 4, 6, 7, 10, 11)	[0.864, 0.8674698795180723, 0.8433734939759037...	0.854554	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0180301	0.0112478	0.00649393
10	(0, 1, 2, 3, 4, 5, 6, 7, 10, 11)	[0.844, 0.8755020080321285, 0.8514056224899599...	0.854574	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0198188	0.0123637	0.00713819
11	(0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11)	[0.844, 0.8634538152610441, 0.8433734939759037...	0.85357	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0158471	0.00988602	0.00570769
12	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)	[0.852, 0.8875502008032129, 0.8353413654618473...	0.858582	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.030238	0.0188636	0.0108909

Code Snippet 10: SFS applied to the Source Code Metrics dataset: results at each iteration

This is further presented in the plot in Figure 4.7 below. This plot displays the average performance of the model (y-axis) as per the number of features (x-axis) that form the best subset of features. It can be seen that the number of features that form the subset with the highest average prediction accuracy is 12 for this dataset.

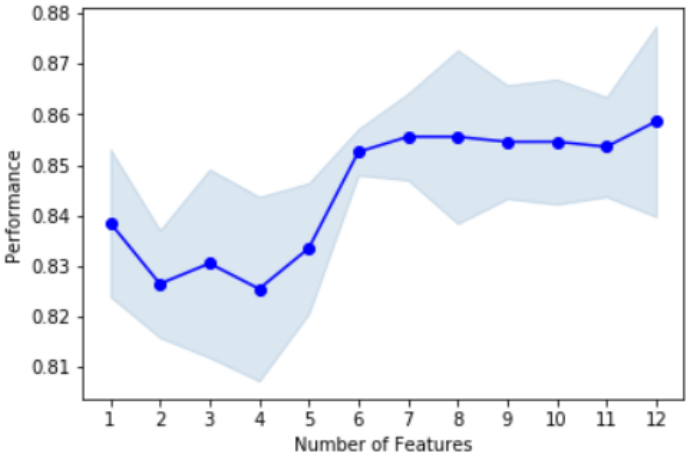


Figure 4.7 SFS Performance plot for Source Code Metrics

Selected features and the score for this dataset using the SFS method can be seen in Code Snippet 11 below. It lists all 12 features for this dataset as mentioned in this section above.

```
In [41]: sfs.k_feature_names_

Out[41]: ('cbo',
          'dit',
          'fanOut',
          'lcom',
          'noc',
          'numberOfAttributes',
          'numberOfAttributesInherited',
          'numberOfLinesOfCode',
          'numberOfMethodsInherited',
          'numberOfPrivateAttributes',
          'numberOfPrivateMethods',
          'numberOfPublicAttributes')

In [42]: sfs.k_score_

Out[42]: 0.8585823293172691
```

Code Snippet 11: Displays the selected best subset of features for source code metrics using SFS

4.2.2.2 Process metrics: SFS results

Code Snippet 12 below displays the results of applying SFS to the process metrics dataset, and it shows that the highest average accuracy score is 0.848554. This score is obtained with a subset selection of 8 out of the 12 features for this metric set.

```
In [87]: pd.DataFrame.from_dict(sfs.get_metric_dict()).T
```

Out[87]:

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
1	(3,)	[0.824, 0.8072289156626506, 0.8192771084337349...	0.824474	(numberOfAuthorsUntil,)	0.0233638	0.0145752	0.00841501
2	(2, 3)	[0.824, 0.8072289156626506, 0.8192771084337349...	0.82347	(numberOfRefactoringsUntil, numberOfAuthorsUntil)	0.0208662	0.0130171	0.00751545
3	(1, 2, 3)	[0.808, 0.8032128514056225, 0.8152610441767069...	0.81947	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0303452	0.0189305	0.0109295
4	(1, 2, 3, 10)	[0.8, 0.8273092369477911, 0.835341365461847 3, ...	0.823494	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0222146	0.0138583	0.00800111
5	(1, 2, 3, 4, 10)	[0.812, 0.8353413654618473, 0.8273092369477911...	0.83553	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0324721	0.0202573	0.0116956
6	(1, 2, 3, 4, 5, 10)	[0.824, 0.8514056224899599, 0.8313253012048193...	0.840538	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0211753	0.0132099	0.00762675
7	(1, 2, 3, 4, 5, 8, 10)	[0.84, 0.8313253012048193, 0.8353413654618473,...	0.844538	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0254205	0.0158583	0.00915578
8	(1, 2, 3, 4, 5, 6, 8, 10)	[0.84, 0.8393574297188755, 0.8433734939759037,...	0.848554	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0213633	0.0133272	0.00769448
9	(1, 2, 3, 4, 5, 6, 8, 10, 11)	[0.84, 0.8273092369477911, 0.8393574297188755,...	0.838514	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.0115382	0.00719795	0.00415574
10	(1, 2, 3, 4, 5, 6, 8, 9, 10, 11)	[0.844, 0.8313253012048193, 0.8433734939759037...	0.841522	(numberOfFixesUntil, numberOfRefactoringsUntil...	0.00974955	0.00608213	0.00351152
11	(0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11)	[0.84, 0.8273092369477911, 0.8313253012048193,...	0.834498	(numberOfVersionsUntil, numberOfFixesUntil, nu...	0.00861856	0.00537658	0.00310417
12	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)	[0.828, 0.8313253012048193, 0.8393574297188755...	0.83451	(numberOfVersionsUntil, numberOfFixesUntil, nu...	0.00799555	0.00498792	0.00287978

Code Snippet 12: SFS applied to the Process Metrics Dataset: Results at each iteration

This is also represented in the plot in Figure 4.8 below. The plot depicts the highest performance score (y-axis) is obtained with a subset of 8 features (x-axis).

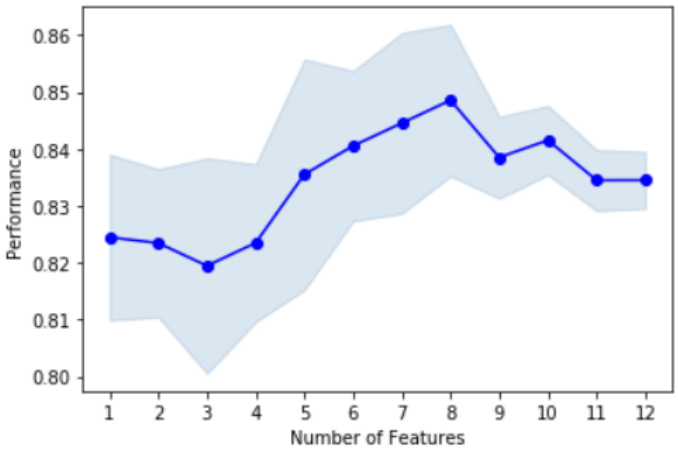


Figure 4.8 SFS Performance plot for Process Metrics

All details reported above are also illustrated in Code Snippet 12 below for the process metrics dataset.

```
In [63]: sfs.k_feature_names_

Out[63]: ('numberOfFixesUntil',
          'numberOfRefactoringsUntil',
          'numberOfAuthorsUntil',
          'linesAddedUntil',
          'maxLinesAddedUntil',
          'avgLinesAddedUntil',
          'maxCodeChurnUntil',
          'ageWithRespectTo')

In [64]: sfs.k_score_

Out[64]: 0.84855421686747
```

Code Snippet 13: Displays the selected best subset of features for Process Metrics using SFS

The results show that the 8 features selected as the best subset of features using the SFS wrapper method for this dataset are:

numberOfFixesUntil; numberOfRefactoringsUntil; numberOfAuthorsUntil; linesAddedUntil; maxLinesAddedUntil; avgLinesAddedUntil; maxCodeChurnUntil; and ageWithRespectTo.

4.2.2.3 Combined metrics: SFS results

The combined metrics dataset contains a combination of features of both the source code metrics and the process metrics. Applying SFS to this dataset, Code Snippet 13 below shows that out of the total of 23 features a subset of 10 features yields the highest accuracy score of 0.866606. This is also illustrated in the plot represented by Figure 4.9 below. Finally, Code Snippet 14 shows the Python code implementing the SFS method to display the selected feature names as well as the accuracy score relating to these selected features. The 10 features selected as the best subset of features for this dataset are: dit; fanOut; noc; numberOfAttributes; numberOfAttributesInherited; numberOfPrivateMethods; numberOfPublicAttributes; numberOfRefactoringsUntil; linesAddedUntil; and maxCodeChurnUntil.

In [155]:
```
pd.DataFrame.from_dict(sfs.get_metric_dict()).T
```

Out[155]:

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
1	(2,)	[0.84, 0.8554216867469879, 0.8152610441767069,...	0.838514	(fanOut,)	0.0234001	0.0145979	0.00842809
2	(2, 14)	[0.844, 0.8594377510040161, 0.8152610441767069...	0.839514	(fanOut, numberOfRefactoringsUntil)	0.0254126	0.0158533	0.00915292
3	(2, 11, 14)	[0.848, 0.8112449799196787, 0.8273092369477911...	0.828466	(fanOut, numberOfPublicAttributes, numberOfRef...	0.0209129	0.0130462	0.00753225
4	(2, 4, 11, 14)	[0.824, 0.8433734939759037, 0.8393574297188755...	0.825478	(fanOut, noc, numberOfPublicAttributes, number...	0.0303404	0.0189275	0.0109278
5	(2, 4, 11, 14, 16)	[0.856, 0.8313253012048193, 0.8313253012048193...	0.838498	(fanOut, noc, numberOfPublicAttributes, number...	0.0164096	0.0102369	0.0059103
6	(2, 4, 6, 11, 14, 16)	[0.852, 0.8514056224899599, 0.8353413654618473...	0.847538	(fanOut, noc, numberOfAttributesInherited, num...	0.0112946	0.007046	0.00406801
7	(2, 4, 6, 11, 14, 16, 19)	[0.856, 0.8594377510040161, 0.8473895582329317...	0.855566	(fanOut, noc, numberOfAttributesInherited, num...	0.00789472	0.00492502	0.00284346
8	(1, 2, 4, 6, 11, 14, 16, 19)	[0.852, 0.8674698795180723, 0.8473895582329317...	0.858582	(dit, fanOut, noc, numberOfAttributesInherited...	0.0144842	0.00903579	0.00521682
9	(1, 2, 4, 5, 6, 11, 14, 16, 19)	[0.848, 0.8674698795180723, 0.8514056224899599...	0.858586	(dit, fanOut, noc, numberOfAttributes, numberO...	0.0143703	0.00896476	0.00517581
10	(1, 2, 4, 5, 6, 10, 11, 14, 16, 19)	[0.86, 0.8835341365461847, 0.8473895582329317,...	0.866606	(dit, fanOut, noc, numberOfAttributes, numberO...	0.0223645	0.0139518	0.00805508
11	(1, 2, 4, 5, 6, 10, 11, 14, 16, 19, 22)	[0.844, 0.8875502008032129, 0.8594377510040161...	0.86361	(dit, fanOut, noc, numberOfAttributes, numberO...	0.025028	0.0156134	0.00901442
12	(1, 2, 4, 5, 6, 10, 11, 14, 16, 19, 20, 22)	[0.848, 0.8835341365461847, 0.8473895582329317...	0.858586	(dit, fanOut, noc, numberOfAttributes, numberO...	0.0236385	0.0147466	0.00851394
13	(1, 2, 4, 5, 6, 9, 10, 11, 14, 16, 19, 20, 22)	[0.848, 0.8795180722891566, 0.8473895582329317...	0.858586	(dit, fanOut, noc, numberOfAttributes, numberO...	0.0208434	0.0130029	0.00750722
14	(1, 2, 4, 5, 6, 9, 10, 11, 14, 16, 18, 19, 20,...	[0.836, 0.8875502008032129, 0.8554216867469879...	0.860606	(dit, fanOut, noc, numberOfAttributes, numberO...	0.0296273	0.0184826	0.010671
15	(0, 1, 2, 4, 5, 6, 9, 10, 11, 14, 16, 18, 19, ...	[0.852, 0.8634538152610441, 0.8594377510040161...	0.859586	(cbo, dit, fanOut, noc, numberOfAttributes, nu...	0.0074968	0.00467679	0.00270014
16	(0, 1, 2, 4, 5, 6, 8, 9, 10, 11, 14, 16, 18, 1...	[0.868, 0.8795180722891566, 0.8594377510040161...	0.866598	(cbo, dit, fanOut, noc, numberOfAttributes, nu...	0.0132047	0.0082376	0.00475598
17	(0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 14, 16, 18...	[0.852, 0.8634538152610441, 0.8554216867469879...	0.858582	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.00804608	0.00501945	0.00289798
18	(0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 14, 16...	[0.86, 0.8714859437751004, 0.8393574297188755,...	0.85757	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0185341	0.0115623	0.00667548
19	(0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 14, 16...	[0.86, 0.8795180722891566, 0.8514056224899599,...	0.86259	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0165871	0.0103476	0.00597421
20	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14,...	[0.848, 0.8755020080321285, 0.8473895582329317...	0.858586	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0187499	0.0116969	0.00675321
21	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...	[0.852, 0.8714859437751004, 0.8393574297188755...	0.854566	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0183627	0.0114554	0.00661376
22	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...	[0.84, 0.8634538152610441, 0.8554216867469879,...	0.85257	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0135533	0.00845503	0.00488151
23	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...	[0.848, 0.8714859437751004, 0.8473895582329317...	0.853566	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0165891	0.0103489	0.00597496

Code Snippet 14: SFS applied to the Combined Metrics dataset: Results at each iteration

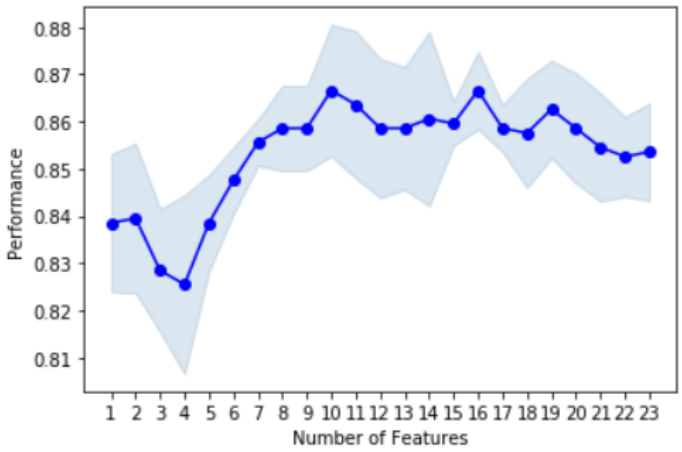


Figure 4.9 SFS Performance plot for Combined Metrics

```
In [37]: sfs.k_feature_names_

Out[37]: ('dit',
          'fanOut',
          'noc',
          'numberOfAttributes',
          'numberOfAttributesInherited',
          'numberOfPrivateMethods',
          'numberOfPublicAttributes',
          'numberOfRefactoringsUntil',
          'linesAddedUntil',
          'maxCodeChurnUntil')

In [38]: sfs.k_score_

Out[38]: 0.8666064257028112
```

Code Snippet 15: Displays the selected best subset of features for Combined Metrics using SFS

In summary, Table 4.5 below shows the subset of best features for each dataset selected through the SFS wrapper feature selection method.

Table 4.5 Subset of selected best features for each metric set selected through SFS wrapper method

Metric Set	Selected Best Features	#Features	Accuracy Score
Source Code Metrics	cbo, dit, fanOut, lcom, noc, numberOfAttributes, numberOfAttributesInherited, numberOfLinesOfCode, numberOfMethodsInherited, numberOfPrivateAttributes,	12	85.85

	numberOfPrivateMethods, numberOfPublicAttributes		
Process Metrics	numberOfFixesUntil, numberOfRefactoringsUntil, numberOfAuthorsUntil, linesAddedUntil, maxLinesAddedUntil, avgLinesAddedUntil, maxCodeChurnUntil, ageWithRespectTo.	8	84.86
Combined Metrics	dit, fanOut, noc, numberOfAttributes, numberOfAttributesInherited, numberOfPrivateMethods, numberOfPublicAttributes, numberOfRefactoringsUntil, linesAddedUntil, maxCodeChurnUntil.	10	86.66

This section of the experiment revealed the best subset of features per metric set of the proposed dataset. The SFS wrapper method of feature selection was applied to each metric set and results were generated and tabulated in Table 4.5 above. Now that the data has been through stringent cleaning and pre-processing steps, it is ready to be run through ML algorithms and follow the process of evaluation to be applied for reliability prediction. This is outlined in the next section.

4.3 Machine learning modelling and evaluation results

This section provides a comparative analysis for the prediction performance of the selected ML algorithms as applied to each of the three metric sets. The ML algorithms are evaluated according to accuracy, precision, recall and F1 scores, as well as through the AUC_ROC probability curve analysis explained in section 2.10 and section 3.5.5. The results obtained per algorithm as applied to each of the three metric sets are then tabulated and discussed further.

4.3.1 Ten-fold cross validation with accuracy, recall and precision scoring

The subsets of each dataset using the selected features displayed in Table 4.5 above, were extracted and applied to the selected ML algorithms (RF, SVM, NN, NB and DT) from section 2.8 using the 10-fold cross validation method. This method divides each dataset into 10 parts. Nine parts are used for training and 1 tenth is used for testing. This process is repeated 10 times

reserving a different tenth for testing each time. Cross-validation is a technique that is used to assess the predictive performance of a model as well as to test the generalisability of the model by giving an indication of the model's performance on unseen data, in terms of its prediction accuracy. These are the steps followed for the 10-fold cross validation method (Berrar, 2019; Tandon, Tripathi, Saraswat, & Dabas, 2019)

1. Split the dataset into 10 equal partitions or folds.
2. Use one fold as the testing set and the union of the other folds as the training set.
3. Calculate the testing accuracy.
4. Repeat steps 2 and 3 ten times using a different fold for testing each time.
5. Use the average testing accuracy as the estimate for the out-of-sample accuracy.

Code Snippet 16 below indicates how this process is applied to the selected subset of features for source code metrics using the Random Forest Classifier (RFC) with the evaluation results of accuracy, recall, precision and F1 score displayed below:

```
In [55]: #10 fold cross Validation and scoring
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
y = np.array([number[0] for number in lb.fit_transform(y)])

rfc=RandomForestClassifier(n_estimators=100, max_depth=10, random_state=1)

accuracy=cross_val_score(rfc,X_best,y,cv=10,scoring='accuracy')
a_float = np.mean(accuracy)
formatted_float = "{:.2f}".format(a_float)
print('Accuracy', formatted_float, accuracy)

recall = cross_val_score(rfc, X_best, y, cv=10, scoring='recall')
a_float = np.mean(recall)
formatted_float = "{:.2f}".format(a_float)
print('Recall', formatted_float, recall)

precision = cross_val_score(rfc, X_best, y, cv=10, scoring='precision')
a_float = np.mean(precision)
formatted_float = "{:.2f}".format(a_float)
print('Precision', formatted_float, precision)

f1 = cross_val_score(rfc, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)

Accuracy 0.85 [0.89       0.81       0.86       0.89       0.83       0.86
 0.88       0.78787879 0.8989899  0.83838384]
Recall 0.96 [0.975      0.92405063 0.94936709 0.97468354 0.93670886 0.98734177
 0.96202532 0.92405063 0.97468354 0.94936709]
Precision 0.87 [0.89655172 0.84883721 0.88235294 0.89534884 0.86046512 0.85714286
 0.89411765 0.82954545 0.90588235 0.86206897]
F1 Score 0.91 [0.93413174 0.88484848 0.91463415 0.93333333 0.8969697  0.91764706
 0.92682927 0.8742515  0.93902439 0.90361446]
```

Code Snippet 16: Python code illustrating modelling using cross validation with the RFC

ML classifiers and cross_val_score methods are imported from *sklearn*, also known as Scikit-learn, is a library in the Python programming language that offers many tools for ML modelling. The results in Code Snippet 16 show that after applying evaluation measures of accuracy, recall, precision and F1 score to the ML modelling of the RFC on source code metric set, the average prediction scores are 0.85, 0.96, 0.87 and 0.91 respectively. These values are represented by the first score in each row of the output for each measure which is obtained from the 10 prediction scores, displayed in square brackets, for each fold of the cross validation process for accuracy, recall and precision for the applied ML algorithm. This process is repeated 5 times for each ML algorithm applied to each of the 3 metric sets of data.

The average prediction scores for each ML algorithm of accuracy, recall, precision and F1 score across each dataset is represented in Tables 4.6, 4.7 and 4.8 below.

Table 4.6 Accuracy, Recall, Precision and F1 scores for Source Code Metrics

ML Algorithm	Accuracy	Recall	Precision	F1 Score
RF	0.85	0.96	0.87	0.91
SVM	0.82	0.98	0.82	0.90
NN	0.80	0.91	0.86	0.89
NB	0.83	0.95	0.85	0.90
DT	0.81	0.86	0.90	0.88

Table 4.7 Accuracy, Recall, Precision and F1 scores for Process Metrics

ML Algorithm	Accuracy	Recall	Precision	F1 Score
RF	0.84	0.42	0.71	0.52
SVM	0.84	0.33	0.75	0.45
NN	0.81	0.37	0.62	0.51
NB	0.82	0.31	0.63	0.40
DT	0.76	0.42	0.40	0.41

Table 4.8 Accuracy, Recall, Precision and F1 scores for Combination Metrics

ML Algorithm	Accuracy	Recall	Precision	F1 Score
RF	0.87	0.95	0.88	0.92
SVM	0.84	0.97	0.85	0.91
NN	0.78	0.81	0.86	0.84
NB	0.82	0.95	0.84	0.89
DT	0.80	0.81	0.88	0.88

The scores listed in the Tables 4.6, 4.7 and 4.8 above evaluate the ML algorithms in terms of accuracy, recall, precision and F1 scores. To recap on the evaluation measures discussed in section 2.10 and 3.5.5, the accuracy score refers to the fraction of correct overall predictions, i.e. number of correctly predicted observations over the total number of observations. The recall score also known as the sensitivity score, refers to the proportion of actual positives that were identified correctly. The precision score attempts to measure what proportion of positive identifications was actually correct and F1 score is the harmonic mean between the precision and recall scores.

In simplified terms, assume there are 8 observations in a dataset. 3 are *buggy* and 5 are not *buggy.*

Accuracy refers to how many out of the 8 observations were predicted correctly in terms of the number of buggy and the number of non-buggy classes. This is like an overall score.

Recall refers to the proportion of accurately predicted observations, e.g. if 2 out of the 3 classes were predicted as buggy, the recall = 2/8, where 2 refers to the number predicted correctly and 8 refers to the total number of observations.

Precision refers to the proportion of positive identifications, e.g. f 2 out of the 3 classes were predicted as buggy, precision = 2/3, where 2 refers to the number predicted correctly and 3 represents the total number of correct observations.

The F1 score combines the precision and recall of a classifier into a single metric by taking their mean. This is used to compare the performance of classifiers. For example, if classifier A

and classifier B have high precision, the F1 score can be used to determine which will produce better results.

4.3.2 Area Under Curve (AUC) Receiver Operating Characteristic (ROC) analysis

Below in Figures 4.10, 4.11 and 4.12 are the AUC_ROC curves for each ML classifier across each of the three datasets used in this study.

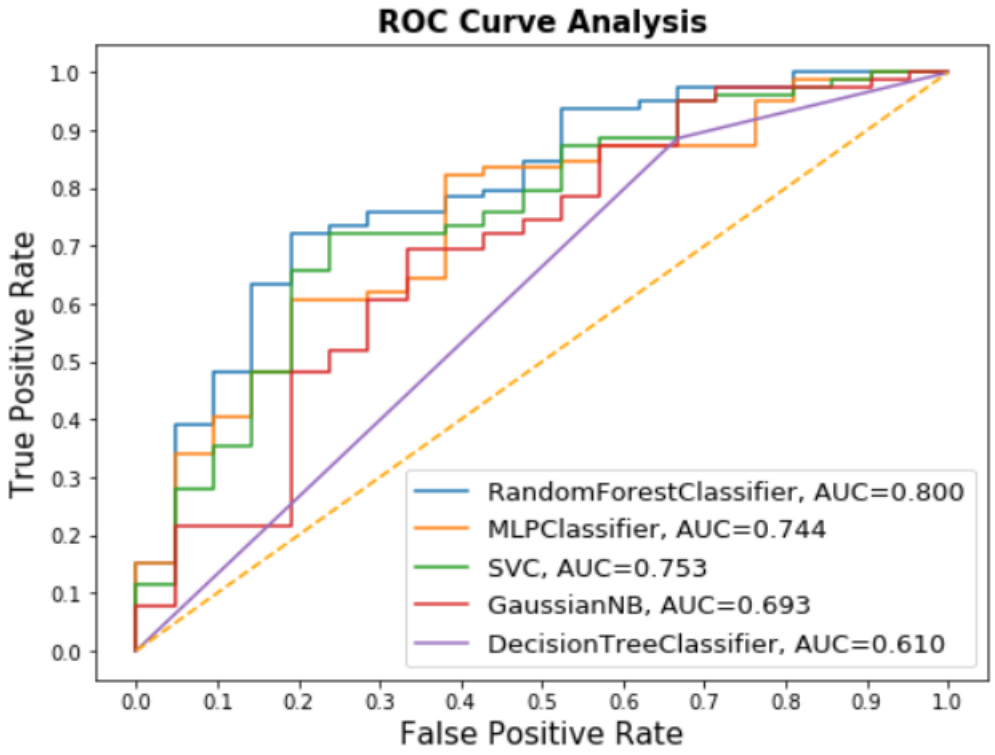


Figure 4.10 AUC-ROC curve analysis for Source Code Metrics

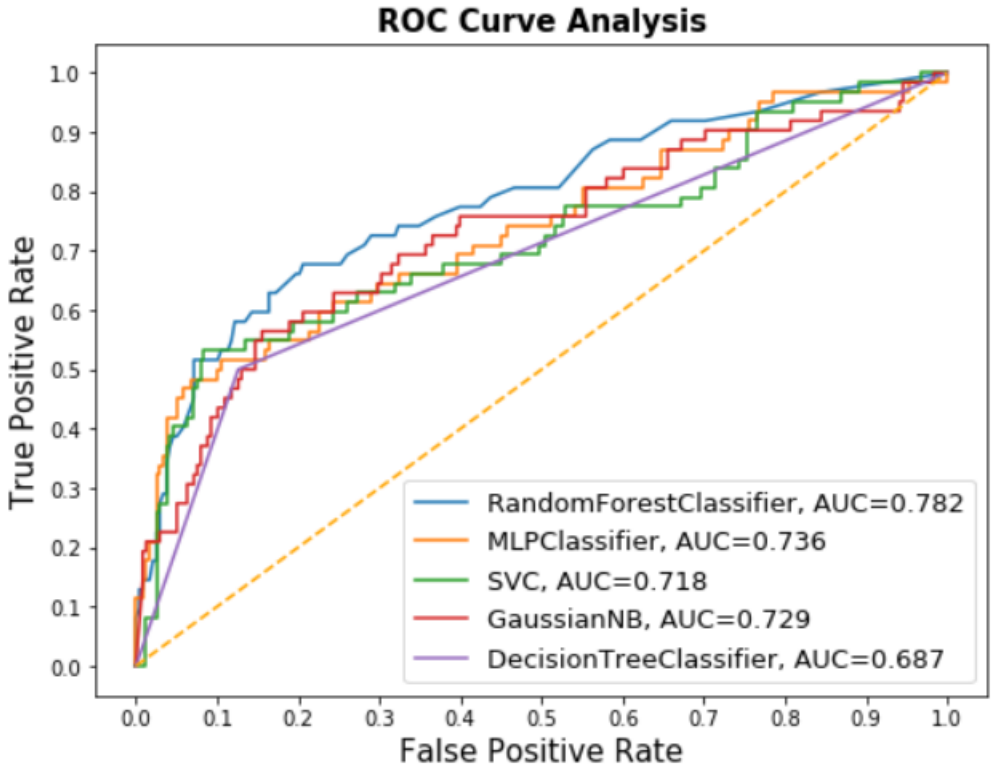


Figure 4.11 AUC-ROC curve analysis for Process Metrics

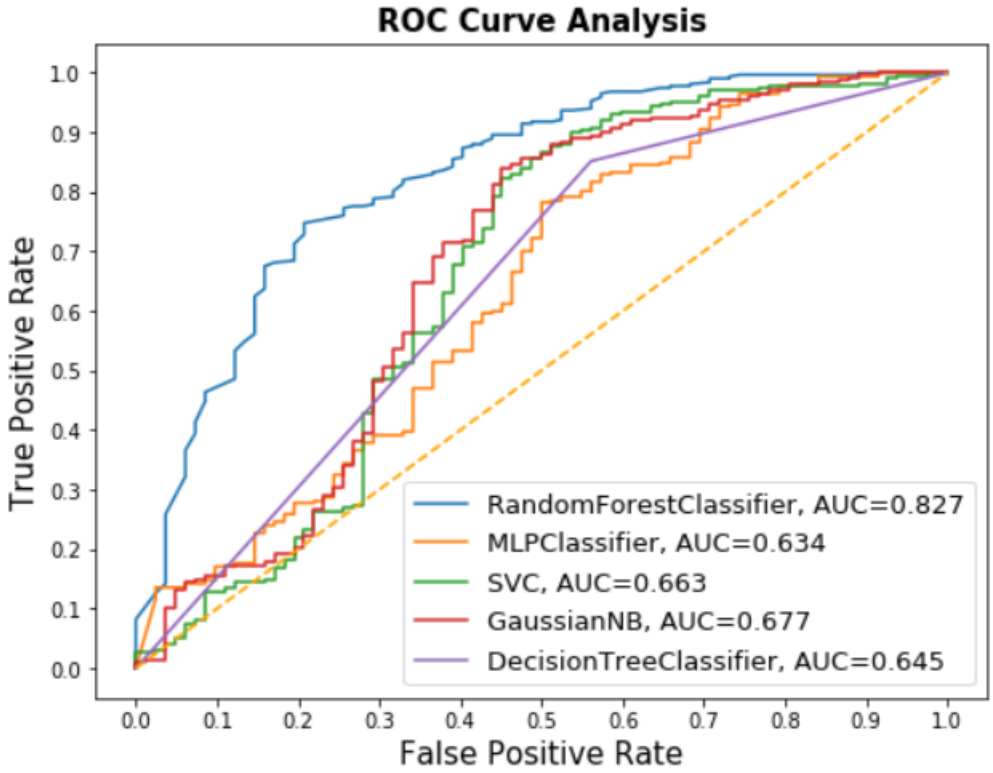


Figure 4.12 AUC-ROC curve analysis for Combined Metrics

In order to compare the different ML modelling techniques, the performance of each classifier is summarised by calculating the area under the ROC curve that is AUC. All The evaluation results are discussed further in Chapter 5.

4.4 Summary of the chapter

This chapter discussed how the experiments were conducted for the study. All steps of the ML process described in Figure 3.1, were applied to the experiments. The first section defined all the metric sets used for the experiments. These included the source code metric set and the process metric set and a combined metric set. All data belonged to the same number of 997 classes that pointed to the same target response values.

The next part demonstrated the use of dimensionality reduction. Dimensionality reduction is a ML technique that applies filtering methods to data as a data pre-processing technique to remove redundant and unwanted data. The filtering methods applied were the univariate and the multivariate filter methods. The multivariate Pearson correlation technique was applied to data to remove correlated features. Correlated features share a very close linear relationship that add no value to the prediction performance and need to be removed. Feature selection was applied to all metric sets and an updated list of features was presented after dimensionality reduction. These updated sets of features were then passed through another data pre-processing step that applied a stringent iterative process known as the SFS method. This ML technique helped identify the best subset of features to be applied to ML modelling through accuracy scores.

The best subset of features selected through SFS implies that these are the best suited predictors for ML modelling. Once again, all metric sets were updated containing only the selected best subsets of features selected through SFS. The next section discussed the final part of the experiment, that is, ML modelling and the evaluation results. The selected subset of data for each metric set was applied and run through the selected ML algorithms and evaluation was done using the 10-fold cross validation technique that splits the data into training and testing sets. This technique serves as useful for testing the generalisability of the results as though it were applied to unseen data. Results were evaluated using the evaluation measures of precision, accuracy and recall and F1 score. Further to this, the AUC_ROC curve analysis technique was also applied to each metric set for evaluation.

All these ML techniques were applied three times for each metric set making up a total of three experiments for this study. These experiments were conducted to determine which of the three

metric sets and ML algorithm provide a ML modelling approach most suited for the reliability prediction of apps. The AUC_ROC curve was applied to visualise the performance of classifiers. The next section discusses all the results obtained from the experiments conducted and presents the research findings for the study.

CHAPTER 5: DISCUSSION OF THE RESULTS

This chapter discusses the results obtained at each stage of the experiments from Chapter 4. Section 5.1 provides a quick recap of the selected data used in the study. Section 5.2 tabulates the number of features for each metric set used. Section 5.3 discusses the results after applying feature removal for constants, quasi constants and duplicate features. This section also provided a discussion of the results after applying dimensionality reduction using the Pearson correlation coefficient for each dataset. Section 5.4 looks at the results after applying the SFS method to each dataset and section 5.5 provides the evaluation results of the modelling processes applied. The results were compared using measures of accuracy, precision, recall and F1 score. Section 5.6 concludes the chapter. The diagram in Figure 5.1 below summarises the steps that were followed for the experiment together with a brief overview of the techniques that were applied at each step.

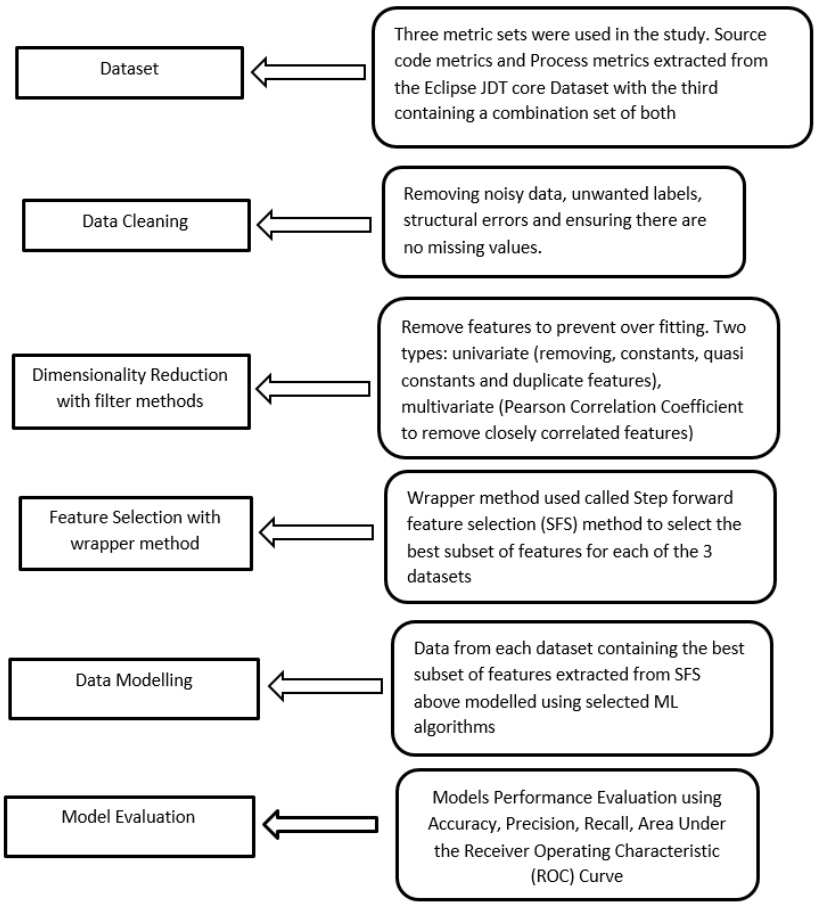


Figure 5.1 Summary of steps carried out for the experiment

5.1 The datasets

The data used in this study has been extracted from the bug prediction dataset for the Eclipse JDT core software system. This bug prediction dataset contains a collection of models and metrics of software systems and their histories. This dataset has been created by (D'Ambros, Lanza, & Robbes, 2010) to allow researchers and programmers to improve on prediction systems and be able to evaluate new ML models for better modelling techniques. The data extracted from this dataset was the source code metric set and the process metric set. These two sets were then combined to create a third called the combined metric set. The same ML techniques were applied to each dataset to compare the effects of ML on different feature sets of bug prediction data for software reliability prediction. This was achieved through the use of ML modelling and evaluation processes. The study was therefore made up of 3 experiments, one for each dataset, and the results, after applying these processes for each experiment, was interpreted, compared and discussed.

5.2 Data cleaning and pre-processing

The dataset contained a total of 997 observations for each of the three metric sets used in this study. Each metric set contained its own category of features as displayed in Table 5.1 below:

Table 5.1 Total number of features per metric set; Source: Eclipse JDT core dataset

Metric Set	Number of Features
Source Code	17
Process	15
Combined	32

5.3 Dimensionality reduction with filter methods

In the case of the univariate filtering methods for constant feature removal, quasi-constant feature removal and duplicate feature removal across all datasets, the system indicated that there were no constants or duplicate features present. This meant that none of the features during this data pre-processing stage needed to be removed or dropped from the datasets.

In the case of multivariate filter methods, the Pearson correlation coefficient was used to identify the closely correlated features. These features then were removed from the datasets as they could have been responsible for overfitting the model and thus creating incorrect predictions. The correlated features for each dataset are discussed below.

5.3.1 Correlated features identified and removed for source code metrics

For the source code metrics dataset, 5 out of 17 features were identified as being closely correlated with each other and removed. These identified 5 correlated features were:

1) fanIn: refers to the measure of the number of methods that call some other method.
2) rfc: response for class refers to the total number of methods invoked by a class.
3) wmc: weighted method for class defines the number and complexity of methods.
4) numberOfMethods: the total number of methods in a class.
5) NumberOfPublicMethods: the number of methods that can be accessed by any object from any class.

From the definition for each feature, it can be seen that all these features have some relationship to the number of methods within a class and hence, the correlation. These features share the same linear space in the dataset and are clearly redundant and therefore needed to be removed. After removal of these 5 correlated features, 12 remained as part of this dataset.

5.3.2 Correlated features identified and removed for process metrics

For the Process Metrics Dataset, 3 out of 15 features were identified to be closely correlated with each other and removed. These features were:

1) avgLinesRemovedUntil: This refers to the average number of lines of code that were removed from a class.
2) linesRemovedUntil: This refers to the total number of lines of code that were removed from a class.
3) maxLinesRemovedUntil: This refers to the maximum number of lines of code that were removed.

All of the features above refer to lines of code that were removed during the revisions or changes to the classes. Once again, as with the case of the source code metrics, it can be seen that these features share a close correlation and are probable candidates for overfitting the model and hence, these had to be removed. After removal of these features, 12 features were retained for this dataset.

5.3.3 Correlated features identified and removed for combination metrics

The combined metrics dataset containing a combination of both the source code metrics and the process metrics datasets, had a cumulative total of 32 features. Of these 32 features, the

number of correlated features included the exact same 5 from source code metric set in section 5.3.1 and the 3 from the process metric set in section 5.3.2, but it also identified and additional feature called maxLinesAddedUntil from the process metric set. This then brought the total number of correlated features for the combined set to a total of 9 features. These are listed below.

1) fanIn
2) rfc
3) wmc
4) numberOfMethods
5) NumberOfPublicMethods
6) avgLinesRemovedUntil
7) linesRemovedUntil
8) maxLinesRemovedUntil
9) maxLinesAddedUntil: Refers to the maximum number of lines added to the class.

After removing these nine features from the combined metrics dataset a total of 23 features were left for this dataset.

After applying the feature selection process with filtering methods and removing all features that were closely correlated to prevent overfitting, across all 3 datasets, the number of features that remained are displayed in Table 5.2 below.

Table 5.2 Number of features before and after applying feature selection with filtering

Metric Set	**Number of features before data filtering**	**Number of features after data filtering**
Source Code	17	12
Process	15	12
Combined	32	23

Therefore, this decrease in the number of features in each dataset, is a result of applying the data pre-processing and dimensionality reduction steps for the experiment.

5.4 Feature selection with the wrapper method (SFS)

The SFS iteration method was used to select the higher performing subset of features for each dataset. Using multiple iterations and a combination of features, the subset was selected through the comparisons of accuracy scores. The subset that produced the highest accuracy score was selected for each dataset. The results indicated the following for each of the source code metrics, process metrics and combination metrics.

5.4.1 SFS applied to source code metrics

A combination of all the features within this source code metric set generated the highest accuracy score of 0.86 compared to the results generated for other iterations and combinations of features. This implies that all 12 features combined for the source code metrics dataset produce highest accuracy scoring when applied to predicting bugs in classes of software. These 12 features include:

1) cbo: Coupling between objects refers to the number of classes that are coupled to a particular class, that is, where the methods of one class call the methods or access the variables of the other.
2) dit: The depth of inheritance tree, dit, is specific to object-oriented programming. It measures the maximum length between a node and the root node in a class hierarchy.
3) fanOut: refers to the number of other classes referenced by a class.
4) lcom: The lack of cohesion of methods metric measures the correlation between the methods and the local instance variables of a class. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity.
5) noc: Number of children refers to the number of immediate subclasses of a class.
6) numberOfAttributes: refers to the number of attributes present in a class.
7) numberOfAttributesInherited: refers to the number of attributes inherited from other classes.
8) numberOfLinesOfCode: the total number of lines of code in a class.
9) numberOfMethodsInherited: the number of methods inherited from other classes.
10) numberOfPrivateAttributes: the number of attributes only accessible within the class.
11) numberOfPrivateMethods: the number of methods accessed within a class.
12) numberOfPublicAttributes: the number of attributes accessible outside of the class.

The features selected above show that most of the measures relate to numbers of attributes, methods, children, lines of code, access of private or public attributes and methods. There is

also the measure of cohesion of the methods and their ability to correlate with each and the level of inheritance between the parent and child classes. These measures all seem to relate to the complexity implying that the higher these values, the more complex is the software. The more complex the software, then the greater chance of the existence of bugs which impacts on reliability.

5.4.2 SFS applied to process metrics

Eight out of the 12 features for this dataset were selected after obtaining the highest accuracy score of 0.85 compared to other combinations of features. The selected features are:

1) numberOfFixesUntil: refers to the number of fixes applied to the class until it was functional again.
2) numberOfRefactoringsUntil: refers to the number of times the code was refactored or restructured until completion.
3) numberOfAuthorsUntil: the number of authors that revised and changed code until it was functional.
4) linesAddedUntil: the number of lines added to the class until functional.
5) maxLinesAddedUntil: maximum number of lines that were added to fix the product.
6) avgLinesAddedUntil: the average number of lines that were added until functional
7) maxCodeChurnUntil: refers to the maximum rate at which the code evolved.
8) ageWithRespectTo: the time taken to fix all bugs in the code ensuring full functionality.

Process metrics are usually those characteristics that can be used to improve the development and maintenance activities of software. These measures refer to the processes that are involved with the removal of defects and bugs in software. In other words, these measures ensure complete and correct functionality of a software product. The mix of features above shows a close relationship to predicting the existence of bugs in an application. Most of the measures relate to the number of overall changes that were made to the product in terms of the number of lines of code added, the number of times the product was revised and fixed, and the time taken to fix the entire product. The higher these numbers, the greater the chance of the existence of bugs which again relates to the reliability of the product.

5.4.3 SFS applied to combined metrics

Ten out of 23 features of the combined metrics produced the highest accuracy score of 0.87 after applying SFS to this dataset. Even though this dataset is a combination of the other two datasets, the wrapper method did not just combine the selections from the source code and

process metrics above, but rather created a new subset of features based on how the features correlate when applied to detecting bugs in a software. The following subset of features were identified:

1) dit
2) fanOut
3) noc
4) numberOfAttributes
5) numberOfAttributesInherited
6) numberOfPrivateMethods
7) numberOfPublicAttributes
8) numberOfRefactoringsUntil
9) linesAddedUntil
10) maxCodeChurnUntil

Within this combined set of metrics, it can be seen that seven features were selected from the source code metrics dataset (dit; fanOut; noc; numberOfAttributes; numberOfAttributesInherited; numberOfPrivateMethods; and numberOfPublicAttributes) and three from the Combined metric dataset (numberOfRefactoringsUntil; linesAddedUntil; and maxCodeChurnUntil).These features show that combining the code complexities, the number and types of methods and attributes with the changes in terms of the revisions, number of lines added, and the rate of evolution of the code to achieve full functionality, creates the highest prediction accuracy for predicting software reliability through detecting bugs in software. Hence, in predicting its reliability. Table 5.3 below shows how many features have been selected after feature selection using the wrapper SFS method.

Table 5.3 Number of features before and after applying feature selection with wrapping

Metric Set	**Number of features before applying SFS**	**Number of features after applying SFS**
Source Code	12	12
Process	12	8
Combined	23	10

After applying feature selection methods to the original datasets resulted in the creation of three cleaned, pre-processed and more relevant features for each of source code metrics, process

metrics and combination metrics datasets, for reliability prediction. After these pre-processing steps were applied, the datasets were trained and tested using selected ML algorithms and cross validation methods, and the results were evaluated. The results for each dataset is discussed below.

5.5 Evaluation results of the ML modelling processes

Each of the three metric sets, discussed in 5.4 above, containing the selected subset of features, were trained with the machine learning algorithms, namely, RF, SVM, NN, NB, DT and the results tested and validated using the 10-fold cross validation method which was then evaluated according to accuracy, recall, precision and F1 scores. Accuracy refers to the overall score and works well with balanced data in which the number of positive and negative values are approximately the same.

However, with imbalanced data in which there is a large difference between the positive and negative values, as in the case with these datasets, the other scores of precision and recall and F1 score is needed to ensure a more accurate evaluation of the results as described in section 3.5.5. The evaluation scores obtained should serve as useful indicators for app reliability prediction in terms of ML algorithms when applied to different feature sets of data in this study. The evaluation results for each metric set is discussed below:

5.5.1 The Source Code Metric Set

The results for this dataset displayed in Table 5.4 below, reveal the *highest scores* achieved for each evaluation measure of accuracy, precision, recall and F1 score together with its corresponding ML algorithm for the source code metric set extracted off Table 4.6.

Table 5.4 Highest prediction scores obtained for the source code metric set

Evaluation Measures	ML Algorithms	Highest Scores
Accuracy	RF	0.85
Precision	DT	0.90
Recall	SVM	0.98
F1 score	RF	0.91

The accuracy score shows that RF when applied to this dataset was able to correctly classify 85% of the data into their categories of either containing bugs or not, which was the highest score achieved compared to the scores of the other algorithms for this dataset. Accuracy works

well if there are an equal number of samples in each category. For example, if 50% of the classes contain bugs and 50% do not contain bugs, then this relates to a perfect balance of samples, which is not always the case. Imbalances could create biased results with high prediction accuracy scores but poor predictive performance.

Precision on the other hand, is a measure that quantifies the number of correct positive predictions made. It is calculated as the number of true positives divided by the total number of predicted positives (true positives and false positives), as explained in section 2.10. Precision shows how precise or accurate the results are based on the predicted positive. The results here show that the highest score of 0.90 was obtained with the DT algorithm implying that 90% of the classes were correctly identified with having bugs out of all the predicted positives.
This score comments only on all positive predictions but does not provide an indication of a missed positive prediction. A missed positive prediction refers to a class that does have bugs but was incorrectly classified as not having any bugs.

Recall, on the other hand, is a measure that quantifies the number of correct positive predictions made out of all the positive classes in the dataset and therefore provides an indication of the missed positive predictions. Thus, for all the classes containing bugs in the dataset, recall measures how many classes were correctly classified as containing bugs. The results show a recall score of 0.98 was obtained with SVM implying that out of all the classes containing bugs, 98% of them were predicted correctly.

Even though precision and recall scores are important evaluation measures, it is not possible to maximise both measures at the same time as one measure increased at the expense of the other and vice versa. The F1 score provides a single score that balances both concerns of precision and recall as mentioned in section 4.3 and can therefore be used to compare the performance of classifiers. The highest F1 score of 0.91 was obtained with the RFC, implying that this classifier outperformed all the other classifiers when applied to this dataset.

5.5.2 The Process Metric Set

Applying the same measures from 5.5.1 above to the process metric set, Table 5.5 below displays the highest accuracy scores obtained from Table 4.7.

Table 5.5 Highest prediction scores obtained for the process metric set

Evaluation Measures	ML Algorithms	Highest Scores
Accuracy	RF, SVM	0.84
Precision	SVM	0.75
Recall	RF/DT	0.42
F1 score	RF	0.52

The accuracy score shows that highest score was achieved for both RF and the SVM. Both classifiers were able to correctly classify 84% of the data into their respective categories of containing bugs or not. Again, considering the imbalance issues, the highest precision score was obtained through the SVM of 0.75, implying that with SVM 75% of classes were correctly identified with having bugs out of all the predicted positives.

The recall score shows that RF/DT produce the highest score of 0.42 implying that 42% of positive classes were predicted correctly out of all the positive classes in the dataset. Balancing the weights of precision and recall, the F1 score revealed the highest evaluation measure of 0.52 was achieved with the RFC, outperforming the other classifiers that were applied to this dataset.

5.5.3 The Combination Metric Set

Applying the same measures from 5.5.1 and 5.5.2 above to the combination metric set, Table 5.6 below displays the highest accuracy scores obtained from Table 4.8.

Table 5.6 Highest prediction scores obtained for the combination metric set

Evaluation Measures	ML Algorithms	Highest Scores
Accuracy	RF	0.87
Precision	RF/DT	0.88
Recall	SVM	0.97
F1 score	RF	0.92

The accuracy score shows that highest score was achieved by the RF classifier of 0.87 in which 87% of the data was correctly classified into their respective categories of containing bugs or not. The highest precision score was obtained through both RF and DT classifiers of 0.88,

implying that when applying both these classifiers to this dataset, 88% of classes were correctly predicted with having bugs over all the predicted positives. The recall score shows that the SVM produced the highest score of 0.97 implying a 97% positive prediction of classes out of all the positive classes in the dataset. Balancing the weights of precision and recall, the F1 score revealed the highest evaluation measure of 0.92 was achieved with the RF classifier, once again outperforming all the other classifiers that were applied to this dataset.

The evaluation results across all three metric sets, show that the RFC outperforms the other algorithms that were selected for this study. The results also reveal that the ML modelling approach most suited for software reliability prediction in terms, of the combining an algorithm to a dataset is when the RF algorithm is applied to the combination metrics dataset. This modelling approach produced the highest F1 score of 92% compared to the other software metric sets.

This means that combining source code metrics and process metrics together, modelled with RF algorithm can serve as useful predictors for the reliability of apps. This was followed very closely by applying RF with the source code metric set with an F1 score of 91%. However, applying RF to the process metric set produced the lowest score of 52%. This means that by applying just the process metrics alone to software reliability, may not serve as useful indicators for the reliability prediction of apps compared to the other datasets. The same can be seen with the AUC_ROC curve in section 4.3.2. The highest AUC value of 0.83 was achieved for the combined metrics dataset with RF. For the other two datasets, AUC values were 0.80 for source code metrics and 0.78 for process metrics also with RF. Again this reveals that RF outperformed the other classifiers, but produced the highest score when modelled with the combined metrics dataset.

5.6 A summary of the chapter

In summary the results for this study show that the combination of Source Code metrics and Process metrics modelled through RF generates the highest F1 score when compared to the results of each dataset individually. This combination dataset lists the following selected features for the reliability prediction of apps. To list them again,
these features include:

1) **dit**: The depth of inheritance tree (DIT is specific to object-oriented programming). It measures the maximum length between a node and the root node in a class hierarchy.
2) **fanOut**: refers to the number of other classes referenced by a class.
3) **noc**: number of children refers to the number of immediate subclasses of a class.
4) **numberOfAttributes**: refers to the number of attributes present in a class.
5) **numberOfAttributesInherited**: refers to the number of attributes inherited from other classes.
6) **numberOfPrivateMethods**: the number of methods accessed within a class.
7) **numberOfPublicAttributes**: the number of attributes accessible outside of the class.
8) **numberOfRefactoringsUntil**: refers to the number of times the code was refactored or restructured until completion.
9) **linesAddedUntil**: the number of lines added to the class until functional.

Figure 5.2: A list of selected features for app reliability prediction

This means that the above listed features generated from the combination of the source code metrics and process metrics feature sets, modelled through the RFC provides useful predictors for the reliability apps. This modelling approach can now be applied to future mobile app projects to help with its reliability prediction.

The aim of this study is to provide a machine learning modelling process to assist with the reliability prediction of apps. The results of this chapter depict the achievement of this aim as it presents a ML modelling approach that applies RF to a combined metric set of data to serve as useful reliability indicators for app reliability prediction. The next chapter provides a summary and concludes the dissertation.

CHAPTER 6: CONCLUSION

This chapter reflects on the entire dissertation by summarising all aspects of the study and concludes the dissertation. It revisits the research objectives, and attempts to provide answers to these objectives, identifies limitations of the study, and suggests recommendations for future research in this area.

6.1 Summary of the study

There has been a significant increase in the growth and development of apps over the years. There is a high demand for apps in the app market and developers need to design high quality products within short periods of time. This places a severe strain on the quality of the software product delivered. Software reliability is regarded as an important quality attribute that measures the correctness of software. This requires developers to spend more time and effort to ensure such reliability. However, due to the high demand of apps in the competitive app market, reliability becomes an issue. Apart from this high demand, there are also challenges within app development that has an impact on app reliability. MAD refers to the development of apps for mobile platforms and differs from traditional software development approaches. The rapid growth and development of apps impact the development lifecycle of MAD in its format and design which is shown to be different to the traditional software development life cycle models.

Some major challenges in MAD include, fragmentation, stability and interoperability, monitoring analysis and testing support, change management and keeping up with frequent changes, user interface design, re-use of code and lack of resources and tools support. These challenges call for appropriate techniques and strategies to help manage and cope with these complexities affecting app reliability. Literature suggests the use of ML to help deal with these issues through predicting the reliability of apps prior to its release. This can assist in reducing costs and better allocate resources as the early detection of bugs or defects in software allow developers to apply their time to other areas of the development phases. Therefore, to predict reliability before deployment becomes an essential requirement for the development process. ML provides opportunities to create predictive models to help detect bugs in software prior to its release.

There are other existing approaches that provide testing techniques to help predict reliability of software systems. These are known as SRGM's. These reliability models have been developed to assess reliability of a software product using information gathered from the testing phase. It follows a process in which practitioners estimate the parameters of the SRGM using failure data during testing, to estimate future failures. This means that SRGMs require optimal parameter estimation techniques which is challenging with the more complex systems like app development, and can thus produce inaccurate results. Therefore, these models are shown to be unsuccessful in predicting the reliability of apps as they fail to fit all the data correctly. ML, on the other hand, provides a more automated process that focuses on a target variable and explores patterns in data which can be applied to unseen data to predict the outcome.

This study applies ML techniques to different feature sets of data extracted off a dataset. These feature sets include source code metrics, process metrics and a combination of both. These software metrics are quantitative measures to assist practitioners observe the efficiency of software products and processes. ML techniques were applied to each of these datasets that made up three experiments for this study. Each experiment applied the same ML techniques to each dataset and the results were analysed. All the experiments followed the ML process of data extraction, data cleaning and pre-processing, dimensionality reduction, feature selection, ML modelling and evaluation.

For ML modelling, five ML algorithms, through a performance comparison of supervised ML algorithms applied in literature, were selected for the study. These were RF, SVM, NB, NN and DT. The results across all three metric sets of data show that RF outperformed all of the other ML algorithms applied to this study and obtained the highest score (92%) when modelled with the combined metric set. The same was noted using the AUC_ROC curve, where the highest accuracy of (83%) was achieved for the combined metric set of data modelled with RF.

This therefore, shows that RF outperformed all other classifiers and produced the highest accuracy scores when modelled with the combined metric set. A closer look into the list of features selected within the combined metric set also provide a useful list of reliability indicators for apps. The main contribution of this study is to provide a ML modelling approach to assist developers with the reliability prediction of apps. In light of this, the study proposes the use of RF with the combined feature set of both source code and process metrics to be used for the reliability prediction of apps.

6.2 Answering the research question

This study revolved around the central research question that reads:

Can ML assist in identifying useful predictors for reliability of mobile apps?

The purpose of answering this research question was to gain a better understanding of ML processes and techniques available to improve app reliability. Building reliable apps is a huge concern for developers as they try to meet increasing app demands and deal with the complexities that exist within MAD. Existing reliability models are not able to fit the complexities of apps. The following research objectives serve to structure this study and guide the activities of the research in an attempt to answer this research question.

The research objectives for the study are:

RO1: To review the challenges of existing software reliability models for the reliability prediction of apps

RO2: To identify suitable ML modelling techniques for app reliability prediction

RO3: To architect and conduct an experiment to compare ML modelling approaches

RO4: To evaluate the prediction performance of ML modelling approaches applied in the study

RO1: To review the challenges of existing software reliability models for the reliability prediction of apps

This objective was featured in Chapter 2 and helped gain a better understanding of the challenges that exists within MAD, existing software reliability models and its impact on app reliability. This objective was set out to try and provide some solutions to these issues, through the use of ML techniques.

ML applied in this study, provide an automated process. ML techniques focus on learning automatically through recognising patterns to gain insights into data. This helps make intelligent decisions, for example, if bugs exist in an application or not. This automation also implies that the system can adapt and learn whenever it changes its structure, program, or data and can still provide solutions to assist with reliability prediction. Both aspects of automation and adaptation provided through the use of ML techniques in this study can therefore assist with the challenges of monitoring and tool support, change management and keeping up with frequent changes, lack of resources and tools and reuse of code. The ML processes applied in

this study focus on the exploratory analysis of data and provides a tool that can assist with these challenges of MAD to help predict app reliability.

However, the automation and adaptation provided through ML in this study is unable to assist with the challenges brought on by external factors and entities of the environment such as fragmentation, stability and interoperability, the issues of user interface design, user experience and device specifications, the ability of systems to work with other products and limited screen sizes.

A further review of literature in Chapter 2, to examine the effectiveness of existing reliability models, pointed to the use of SRGMs. These are models developed and applied to estimate the reliability of software. Findings in literature show that all these models require estimation techniques with a substantial dependence on time. Testing techniques are applied at various phases of software development and estimated over time. This does not fit well with the dynamic nature of apps that need to be developed quickly to meet app demands. Also, SRGMs require optimal parameter estimations, set up by practitioners, to accurately predict reliability.

Complexities in MAD severely impact these estimation techniques, making it difficult for developers to fit all the data into the parameters of estimation and thus failing to estimate app reliability. This implied that more specialised tools and techniques are needed to handle the complexities of MAD and hence the need for ML as it provides an automated, non-parametric technique to help predict app reliability, as compared to the SRGMs parameter estimation techniques.

RO2: To identify suitable ML modelling techniques for app reliability prediction

Now that the need for this study had been outlined in RO1, this research objective focused on ML and the techniques required for ML modelling specifically to the ML algorithms and software feature sets. ML algorithms were analysed from literature in similar studies of bug prediction and mobile applications. Twenty algorithms were compared and five were selected based on a performance comparison of prediction accuracy scores in which the higher performing algorithms were selected. These included RF, SVM, NN, DT, NB.

Software feature sets, also known as software metrics, were selected to determine which metric set serves as essential contributors for assessing app reliability. A review of software metrics

was conducted identifying categories of features and how these categories are applied as software reliability indicators. Eight studies were reviewed and these studies reveal that combining significant metrics is needed to build reliable predictive models for software systems. Source code metrics such as the CK metrics suite, was shown to be widely accepted for the evaluation of software systems. To accommodate the rapid changes made during MAD, process metrics need to be considered. Therefore, information gathered from this, point to the use of source code metrics and process metrics to help assess the reliability of apps. Also, combining significant features from both sets, may serve more useful to app reliability prediction. Thus, this study provided experiments of modelling these sets of data with the selected ML algorithms to identify the ML modelling process most suited for the reliability prediction of apps. This objective featured in Chapter 2 and Chapter 3.

RO3: To architect and conduct an experiment to compare ML modelling approaches

This RO featured in Chapter 3 and Chapter 4, focussed on the components of the experiment and how they need to be applied to determine the ML modelling approach useful to app reliability prediction. All components selected were based on the ML process shown in Figure 3.1. The reasons for selecting these components were described in chapter 3 and further summarised in Table 3.4. The plan for conducting the experiments was made up of 5 stages illustrated in Figure 3.3. The ML techniques applied at each stage for each ML process featured in Chapter 4. All the ML techniques for this study was implemented and executed using the Python programming language. Each section of Chapter 4 presented the results obtained from Python, after applying and executing the ML techniques at each stage of the experiments for each dataset. Achievement of this RO provided the results to proceed with the next step of comparing and identifying the ML modelling approach most suited to app reliability prediction.

RO4: To evaluate the prediction performance of ML techniques applied in the study

RO4 featured in Chapter 4 and Chapter 5. In keeping with the aim for this study, this research objective sets out to establish if applying the ML techniques from RO3 to different feature sets of data can provide a ML modelling approach for predicting the reliability of apps.

The ML modelling process that produced the highest score of 92% occurred when the combined metric set of data was modelled with the RF algorithm. This score may not be a perfect score given the stochastic nature of data and algorithms but it is closest to 100%, implying that this ML modelling approach can serve useful and provide relevant insights into predicting reliability of apps. A deeper look into the features selected within the combined metric dataset through ML techniques, provided a list of 9 combined features from the source code metric set and the process metric set, that could serve as useful predictors for the reliability of apps.

Therefore, the answer to the research question, derived from these sought objectives, is that ML *can* assist in identifying useful predictors for the reliability of apps. This study suggests the use of the RFC as the learning algorithm, and the combination of selected features of source code and process metrics from software systems, through ML, for the reliability prediction of apps. These findings also point to the achievement of the overall aim for this study which is to provide a useful ML modelling approach for the reliability prediction of apps.

6.3 Contribution of the study

This study provides a ML modelling approach for the reliability prediction of apps. This adds knowledge and practice in the area of application development allowing developers to apply these findings to help build more reliable apps. The contribution adds to the growing discourse on app reliability for which, even though it has received much attention, challenges still persist. In line with this, this study proposed a new lens to be applied to this issue of app reliability. Other contributions to the body of knowledge include, a review of some major challenges within MAD as well as discussions on how this study may address some of these challenges as explained in RO1. Also this study provided a review of the effectiveness of existing software reliability models for reliability prediction of apps. Part of the study also did an analysis and synthesis of the ML algorithms and defined software metrics most suited to app reliability prediction. This study can be used as a springboard or starting block for further studies in this

area. Researchers can build on this model by integrating other reliability techniques into the application. The results can be further refined by combining more feature sets and more ML algorithms to the experiment and comparing the results.

6.4 Limitations

Even though this research has set out and achieved the overall objectives, there are some limitations. The first limitation refers to the challenges in MAD described in section 2.4.3. From the discussions around RO1, this study is able to address only four of the seven challenges through the use of applying ML techniques. The challenges of fragmentation, stability and interoperability, and user interface design form external entities that have not been considered in this study. This study focuses on aspects within the software design itself and using ML techniques for the modelling and analysis of software features for reliability prediction. External entities such as device specifications such as screen sizes, CPU speed, device memory, as well as user experiences and expectations are not addressed and therefore provide limitations to this study.

A second limitation is on the ML techniques applied to this study, particularly that the feature selection techniques used in this will only work well on smaller datasets due to their high computation time. These are computationally very expensive and may take too long with large multi-dimensional datasets and be totally unfeasible.

A third limitation is the imbalanced nature of the dataset. Imbalanced datasets are a special case for classification problems in which the class distribution is not uniform. This dataset showed more negative classes than positive ones.

6.5 Future works

The results from this study could be validated by applying the experiment to datasets in related domains. This can then be used to determine if the results obtained from the dataset hold true for other datasets as well. Another future extension could include the impact of applying other feature selection techniques, other than the ones used in this study, to select an optimal set of features for app reliability prediction and to compare the results. The results could also be packaged into a more user-friendly tool for developers to use.

6.6 Conclusion

This study set out to determine if ML can assist in identifying useful predictors for reliability in apps to assist developers in dealing with some major challenges experienced during app development. The findings show that ML offers significant insights into app reliability through learning algorithms and feature selection techniques, and can therefore assist in identifying these predictors. This study presents a list of software features that may serve as useful app reliability indicators that developers can refer to during app development. This study also revealed that the RFC outperformed all the other classifiers applied in this study and is therefore the most suited algorithm for app reliability prediction for future research in this area.

Validity of the results stem from the use of complex learning algorithms and mathematical formulas prescribed through the use of ML, without much human intervention, and through the comparison of ML accuracy scores was able to provide a useful solution for app reliability prediction. Cross validation methods provided opportunities to assess the generalisability of the results through resampling and iterative processes to estimate its performance on unseen data. Reliability of the results is shown through the application of all the ML techniques within the study. These techniques show consistent measures over time and if applied to another set of data, may not provide the exact same results but should show some correlation to app reliability. Post-positivism views ML as an approach that provides flexibility through its multiple processes of exploratory analysis of the data toward a more scientific method of conducting research and analysing results. This contributes to the field of MAD assisting developers and researchers improve app reliability rates.

REFERENCES

Ahmad, A., Feng, C., Tao, M., Yousif, A., & Ge, S. (2017). *Challenges of Mobile Applications Development: Initial Results.* Paper presented at the Software Engineering and Service Science (ICSESS), 2017 8th IEEE International Conference on.

Al-Janabi, S., Al-Shourbaji, I., Shojafar, M., & Abdelhag, M. (2017). *Mobile Cloud Computing: Challenges and Future Research Directions.* Paper presented at the Developments in eSystems Engineering (DeSE), 2017 10th International Conference on.

Al-Msie'deen, R. F., & Blasi, A. (2018). The Impact of the Object-Oriented Software Evolution on Software Metrics: The Iris Approach. *arXiv preprint arXiv:.09823*.

Aldayel, A., & Alnafjan, K. (2017). *Challenges and Best Practices for Mobile Application Development.* Paper presented at the Proceedings of the International Conference on Compute and Data Analysis.

Ali, M., & Mesbah, A. (2016). *Mining and Characterizing Hybrid Apps.* Paper presented at the Proceedings of the International Workshop on App Market Analytics.

Alsaeedi, A., & Khan, M. Z. (2019). Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study. *Journal of Software Engineering and Applications, 12*(5), 85-100.

Alsina, E. F., Chica, M., Trawiński, K., & Regattieri, A. (2018). On the Use of Machine Learning Methods to Predict Component Reliability from Data-Driven Industrial Case Studies. *The International Journal of Advanced Manufacturing Technology, 94*(5-8), 2419-2433.

Alsolai, H., & Roper, M. (2019). *Determining the Best Prediction Accuracy of Software Maintainability Models Using Auto-Weka.* Paper presented at the International Conference on Computing.

Amalfitano, D., Amatucci, N., Fasolino, A. R., Tramontana, P., Kowalczyk, E., & Memon, A. M. (2015). *Exploiting the Saturation Effect in Automatic Random Testing of Android Applications.* Paper presented at the Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems.

Andrade, C. (2018). Internal, External, and Ecological Validity in Research Design, Conduct, and Evaluation. *Indian journal of psychological medicine, 40*(5), 498-499.

Angriman, E., van der Grinten, A., von Looz, M., Meyerhenke, H., Nöllenburg, M., Predari, M., & Tzovas, C. (2019). Guidelines for Experimental Algorithmics: A Case Study in Network Analysis. *Algorithms, 12*(7), 127.

Antonopoulos, I., Robu, V., Couraud, B., Kirli, D., Norbu, S., Kiprakis, A., . . . Wattam, S. (2020). Artificial Intelligence and Machine Learning Approaches to Energy Demand-Side Response: A Systematic Review. *Renewable Sustainable Energy Reviews*

130, 109899.

Anwar, N., Rizal, M. A. M., Mustamum, H. A., Taib, K. M., Razak, A. A., & Nordin, Z. (2020). *Mobile Application Development: A Preliminary Study.* Paper presented at the 2020 International Conference on Information Management and Technology (ICIMTech).

Asim, M., & Khan, Z. (2018). Mobile Price Class Prediction Using Machine Learning Techniques. *International Journal of Computer Applications, 975*, 8887.

Aydin, Z. B. G., & Samli, R. (2020). *Performance Evaluation of Some Machine Learning Algorithms in Nasa Defect Prediction Data Sets.* Paper presented at the 2020 5th International Conference on Computer Science and Engineering (UBMK).

Ballabio, D., Grisoni, F., & Todeschini, R. (2018). Multivariate Comparison of Classification Performance Measures. *J Chemometrics Intelligent Laboratory Systems*

174, 33-44.

Barack, O., & Huang, L. (2020). Assessment and Prediction of Software Reliability in Mobile Applications. *Journal of Software Engineering, 13*(9), 179-190.

Bartz-Beielstein, T. (2016). Easd-Experimental Algorithmics for Streaming Data.

Bartz-Beielstein, T., & Mernik, M. (2016). Experimental Algorithmics Applied to on-Line Machine Learning. 94-104.

Basavaraju, P., & Varde, A. S. (2017). Supervised Learning Techniques in Mobile Device Apps for Androids. *ACM SIGKDD Explorations Newsletter, 18*(2), 18-29.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761. doi:10.1109/32.544352

Becker, H. (1996). The Epistemology of Qualitative Research. *Ethnography human development: Context meaning in social inquiry, 27*, 53-71.

Belavagi, M. C., & Muniyal, B. (2016). Performance Evaluation of Supervised Machine Learning Algorithms for Intrusion Detection. *Procedia Computer Science, 89*(2016), 117-123.

Berrar, D. (2019). Cross-Validation. *Encyclopedia of bioinformatics computational biology, 1*, 542-545.

Berry, M. W., Mohamed, A., & Yap, B. W. (2019). *Supervised and Unsupervised Learning for Data Science*: Springer.

Bhojan, R. J., Vivekanandan, K., Ramyachitra, D., & Ganesan, S. (2018). A Machine Learning Based Approach for Detecting Non-Deterministic Tests and Its Analysis in Mobile Application Testing. *International Journal of Advanced Research in Computer Science, 9*(1).

Bhuiyan, T., Jabiullah, M. I., & Felix, E. A. (2020). *Prevalence of Machine Learning Techniques in Software Defect Prediction.* Paper presented at the Cyber Security and

Computer Science: Second EAI International Conference, ICONCS 2020, Dhaka, Bangladesh, February 15-16, 2020, Proceedings.

Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2018). A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. *ACM Computing Surveys, 51*(5), 1-34.

Bisandu, D. (2016). Design Science Research Methodology in Computer Science and Information Systems. *International Journal of Information Technology*.

Blessing, L. T., & Chakrabarti, A. (2009). Drm: A Design Reseach Methodology. *DRM, a Design Research Methodology*, 13-42.

Bogdan, R., & Biklen, S. (1998). Introduction to Qualitative Research in Education. *England: Pearson*.

Boucher, A., & Badri, M. (2016). *Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software.* Paper presented at the 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD).

Boutaba, R., Salahuddin, M. A., Limam, N., Ayoubi, S., Shahriar, N., Estrada-Solano, F., & Caicedo, O. M. (2018). A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities. *Journal of Internet Services Applications, 9*(1), 1-99.

Bowes, D., Hall, T., & Petrić, J. (2018). Software Defect Prediction: Do Different Classifiers Find the Same Defects? *Software Quality Journal, 26*(2), 525-552.

Butgereit, L., Niland, M., Schmidt, J., & Rheeder, W. (2018). *A Design Science Model for the Application of Data Mining and Machine Learning Models on Constrained Devices in Low Bandwidth Areas.* Paper presented at the 2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD).

Cai, J., Luo, J., Wang, S., & Yang, S. (2018). Feature Selection in Machine Learning: A New Perspective. *Neurocomputing, 300*, 70-79.

Capretz, L. F., Meskini, S., & Bou, A. (2013). *Reliability Models Applied to Mobile Applications.* Paper presented at the IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C).

Carvalho, R. M., de Castro Andrade, R. M., & de Oliveira, K. M. (2018). Aquarium-a Suite of Software Measures for Hci Quality Evaluation of Ubiquitous Mobile Applications. *Journal of Systems Software, 136*, 101-136.

Catal, C., & Diri, B. (2009). Investigating the Effect of Dataset Size, Metrics Sets, and Feature Selection Techniques on Software Fault Prediction Problem. *Information Sciences, 179*(8), 1040-1058.

Ceylan, E., Kutlubay, F. O., & Bener, A. B. (2006). *Software Defect Identification Using Machine Learning Techniques.* Paper presented at the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06).

Charte, D., Charte, F., García, S., & Herrera, F. (2019). A Snapshot on Nonstandard Supervised Learning Problems: Taxonomy, Relationships, Problem Transformations and Algorithm Adaptations. *Progress in Artificial Intelligence, 8*(1), 1-14.

Chaudhary, A., Agarwal, A. P., Rana, A., & Kumar, V. (2019). *Crow Search Optimization Based Approach for Parameter Estimation of Srgms.* Paper presented at the 2019 Amity International Conference on Artificial Intelligence (AICAI).

Choudhary, G. R., Kumar, S., Kumar, K., Mishra, A., & Catal, C. (2018). Empirical Analysis of Change Metrics for Software Fault Prediction. *Computers Electrical Engineering, 67*, 15-24.

Clark, A. M. (1998). The Qualitative-Quantitative Debate: Moving from Positivism and Confrontation to Post-Positivism and Reconciliation. *Journal of advanced nursing, 27*(6), 1242-1249.

Corral, L., & Fronza, I. (2015). *Better Code for Better Apps: A Study on Source Code Quality and Market Success of Android Applications.* Paper presented at the Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems.

Creswell, J. W. (2003). Research Design: Qualitative. *Research design: Qualitative, quantitative, mixed methods approaches*.

Creswell, J. W. (2017). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*: Sage publications.

Cruz, L., Abreu, R., & Lo, D. (2019). To the Attention of Mobile Software Developers: Guess What, Test Your App! *Empirical Software Engineering, 24*(4), 2438-2468.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). *An Extensive Comparison of Bug Prediction Approaches.* Paper presented at the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010).

Dantu, K., Ko, S. Y., & Ziarek, L. (2017). Raina: Reliability and Adaptability in Android for Fog Computing. *IEEE Communications Magazine, 55*(4), 41-45.

Diwaker, C., Tomar, P., Poonia, R. C., & Singh, V. (2018). Prediction of Software Reliability Using Bio Inspired Soft Computing Techniques. *Journal of medical systems, 42*(5), 1-16.

Duffy, D. J., & Katajamäki, M. (2018). Software Interoperability in Computational Finance, Part I: Foundations for Applications Using C++ 11 and C# in The. Net Framework. *Wilmott, 2018*(96), 46-53.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimaraes, M. P. (2019). Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Reliability, 68*(3), 1189-1212.

Ehrler, F., Lovis, C., & Blondon, K. (2019). A Mobile Phone App for Bedside Nursing Care: Design and Development Using an Adapted Software Development Life Cycle Model. *JMIR mHealth, 7*(4), e12551.

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., & Wahba, A. M. (2017). Taxonomy of Cross-Platform Mobile Applications Development Approaches. *Ain Shams Engineering Journal, 8*(2), 163-190.

Elish, K. O., & Elish, M. O. (2008). Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems, 81*(5), 649-660.

Fasano, F., Martinelli, F., Mercaldo, F., & Santone, A. (2018). *Measuring Mobile Applications Quality and Security in Higher Education.* Paper presented at the 2018 IEEE International Conference on Big Data (Big Data).

Fathi, N. A. M., Hashim, M., Ibrahim, N., & Hassan, S. N. S. (2017). Applying Mobile Application Development Life Cycle in the Development of Fasting Tracker Android Application. *E-Journal Penyelidikan Dan Inovasi, 4*(2), 267-284.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., & Gyimóthy, T. (2018). *A Public Unified Bug Dataset for Java.* Paper presented at the Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering.

Ferguson, L. (2004). External Validity, Generalizability, and Knowledge Utilization. *Journal of Nursing Scholarship, 36*(1), 16-22.

Fischer, B. F. (1998). Postpositivism: The Critique of Empiricism. *Policy Studies Journal, 26*(1), 129-146.

Ge, J., Liu, J., & Liu, W. (2018). *Comparative Study on Defect Prediction Algorithms of Supervised Learning Software Based on Imbalanced Classification Data Sets.* Paper presented at the 2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).

Ghotra, B., McIntosh, S., & Hassan, A. E. (2015). *Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models.* Paper presented at the Proceedings of the 37th International Conference on Software Engineering-Volume 1.

Glass, R. L. (1998). Defining Quality Intuitively. *IEEE software, 15*(3), 103-104.

Goertzen, M. J. (2017). Introduction to Quantitative Research and Data. *Library Technology Reports, 53*(4), 12-18.

Gorard, S., & Taylor, C. (2004). *Combining Methods in Educational and Social Research*: McGraw-Hill Education (UK).

Goyal, N., & Srivastava, R. (2018). Stability Evaluation Model for Object Oriented Software. *International Journal of Advanced Research in Computer Science, 9*(2), 871.

Guba, E. G. (1990). *The Paradigm Dialog.* Paper presented at the Alternative Paradigms Conference, Mar, 1989, Indiana U, School of Education, San Francisco, CA, US.

Gupta, A., Gupta, N., Garg, R., & Kumar, R. (2019). *Analysis of Software Reliability Models for Reliability Estimation.* Paper presented at the 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence).

Gupta, A., Sharma, S., Goyal, S., & Rashid, M. (2020). *Novel Xgboost Tuned Machine Learning Model for Software Bug Prediction.* Paper presented at the 2020 International Conference on Intelligent Engineering and Management (ICIEM).

Gupta, A., Suri, B., & Bhat, V. (2019). *Android Smells Detection Using Ml Algorithms with Static Code Metrics.* Paper presented at the International Conference on Recent Developments in Science, Engineering and Technology.

Gupta, D. L., & Saxena, K. (2017). Software Bug Prediction Using Object-Oriented Metrics. *Sādhanā, 42*(5), 655-669.

Ham, J., Chen, Y., Crawford, M. M., & Ghosh, J. (2005). Investigation of the Random Forest Framework for Classification of Hyperspectral Data. *IEEE Transactions on Geoscience, 43*(3), 492-501.

Hammouri, A., Hammad, M., Alnabhan, M., & Alsarayrah, F. (2018). Software Bug Prediction Using Machine Learning Approach. *International Journal of Advanced Computer Science Applications, 9*(2).

Hassan, A. E. (2009). *Predicting Faults Using the Complexity of Code Changes*. Paper presented at the Proceedings of the 31st International Conference on Software Engineering. https://doi.org/10.1109/ICSE.2009.5070510

Healy, M., & Perry, C. (2000). Comprehensive Criteria to Judge Validity and Reliability of Qualitative Research within the Realism Paradigm. *Qualitative market research: An international journal*.

Henderson, K. (2011). Post-Positivism and the Pragmatics of Leisure Research. *Leisure Sciences, 33*(4), 341-346.

Houghton, T. (2011). Does Positivism Really "Work" in the Social Sciences. *E-International Relations*.

Immaculate, S. D., Begam, M. F., & Floramary, M. (2019). *Software Bug Prediction Using Supervised Machine Learning Algorithms.* Paper presented at the 2019 International Conference on Data Science and Communication (IconDSC).

Iqbal, A., Aftab, S., Ali, U., Nawaz, Z., Sana, L., Ahmad, M., & Husen, A. (2019). Performance Analysis of Machine Learning Techniques on Software Defect Prediction Using Nasa Datasets. *Int. J. Adv. Comput. Sci. Appl, 10*(5).

Ivanov, V., Reznik, A., & Succi, G. (2018). Comparing the Reliability of Software Systems: A Case Study on Mobile Operating Systems. *Information Sciences, 423*, 398-411.

Jaiswal, A., & Malhotra, R. (2018). Software Reliability Prediction Using Machine Learning Techniques. *International Journal of System Assurance Engineering, 9*(1), 230-244.

Jaiswal, A., & Malhotra, R. (2018). Software Reliability Prediction Using Machine Learning Techniques. *International Journal of System Assurance Engineering and Management, 9*(1), 230-244.

Jayanthi, R., & Florence, L. (2018). Software Defect Prediction Techniques Using Metrics Based on Neural Network Classifier. *Cluster Computing*, 1-12.

Jenkins, A. M. (1985). Research Methodologies and Mis Research. *Research methods in information systems, 103*, 117.

Jiang, Y., Lin, J., Cukic, B., & Menzies, T. (2009). *Variance Analysis in Software Fault Prediction Models.* Paper presented at the 2009 20th International Symposium on Software Reliability Engineering.

Johannesson, P., & Perjons, E. (2014). Research Strategies and Methods. In *An Introduction to Design Science* (pp. 39-73). Cham: Springer International Publishing.

Joorabchi, M. (2016). *Mobile App Development: Challenges and Opportunities for Automated Support.* University of British Columbia,

Joorabchi, M. E., Mesbah, A., & Kruchten, P. (2013). *Real Challenges in Mobile App Development.* Paper presented at the Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on.

Kalaivani, N., & Beena, R. (2018). Overview of Software Defect Prediction Using Machine Learning Algorithms. *International Journal of Pure Applied Mathematics*

118(20), 3863-3873.

Kaliraj, S., Vivek, D., Kannan, M., Karthick, K., & Lydia, M. D. (2020). Critical Review on Software Reliability Models: Importance and Application of Reliability Analysis in Software Development. *Materials Today: Proceedings*.

Kaur, A., Kaur, K., & Kaur, H. (2016). Application of Machine Learning on Process Metrics for Defect Prediction in Mobile Application. In *Information Systems Design and Intelligent Applications* (pp. 81-98): Springer.

Kaur, V., & Arora, A. (2013a). Adaptive Approach of Fault Prediction in Software Modules by Using Discriminative and Generative Model of Machine Learning. *International Journal of Computer Applications, 74*(12).

Kaur, V., & Arora, A. (2013b). Adaptive Approach of Fault Prediction in Software Modules by Using Discriminative and Generative Model of Machine Learning. *International Journal of Computer Applications*

74(12).

Khaire, U. M., & Dhanalakshmi, R. (2019). Stability of Feature Selection Algorithm: A Review. *Journal of King Saud University-Computer Information Sciences*

Kim, S., Zimmermann, T., Jr., E. J. W., & Zeller, A. (2007). *Predicting Faults from Cached History*. Paper presented at the Proceedings of the 29th international conference on Software Engineering. https://doi.org/10.1109/ICSE.2007.66

Krauss, S. E. (2005). Research Paradigms and Meaning Making: A Primer. *The qualitative report, 10*(4), 758-770.

Kumaresan, K., & Ganeshkumar, P. (2019). Software Reliability Modeling Using Increased Failure Interval with Ann. *Cluster Computing*, 1-8.

Lachgar, M., & Abdali, A. (2017). Decision Framework for Mobile Development Methods. *Int J Adv Comput Sci Appl, 8*(2).

Lakshmanan, I., & Ramasamy, S. (2017). Improving Software Reliability Estimation Using Multi-Layer Neural-Network Combination Model. *International Journal of Innovative Computing Applications, 8*(2), 113-121.

Lakshmi, S., & Mohideen, M. A. (2013). Issues in Reliabilityand Validity of Research. *International journal of management research reviews, 3*(4), 2752.

Lamba, T., Kumar, D., & Mishra, A. (2019). Comparative Study of Bug Prediction Techniques on Software Metrics. *Proceedings of the 11th INDIACom; INDIACom-2017; IEEE Conference ID: 40353*.

Lamba, T., & Mishra, A. (2019). Optimal Machine Learning Model for Software Defect Prediction. *International Journal of Intelligent Systems, 11*(2), 36.

Lantz, B. (2013). *Machine Learning with R*: Packt publishing ltd.

Ławrynowicz, M., Legut, J., & Jóźwiak, I. J. (2020). Research on Reliability of Mobile Applications in a Distributed Environment. *Zeszyty Naukowe. Organizacja i Zarządzanie/Politechnika Śląska*(144), 327-337.

Lee, J., Kim, Y. W., Ha, A., Kim, Y. K., Park, K. H., Choi, H. J., & Jeoung, J. W. (2020). Estimating Visual Field Loss from Monoscopic Optic Disc Photography Using Deep Learning Model. *Scientific reports, 10*(1), 1-10.

Li, Q., & Pham, H. (2019). A Generalized Software Reliability Growth Model with Consideration of the Uncertainty of Operating Environments. *IEEE Access, 7*, 84253-84267.

Li, Y., Sun, X., Li, Y., Turrini, A., & Zhang, L. (2019). *Synthesizing Nested Ranking Functions for Loop Programs Via Svm.* Paper presented at the International Conference on Formal Engineering Methods.

Mackenzie, N., & Knipe, S. (2006). Research Dilemmas: Paradigms, Methods and Methodology. *Issues in educational research, 16*(2), 193-205.

Malhotra, R. (2015). A Systematic Review of Machine Learning Techniques for Software Fault Prediction. *Applied Soft Computing, 27*, 504-518.

Malhotra, R. (2016). An Empirical Framework for Defect Prediction Using Machine Learning Techniques with Android Software. *Applied Soft Computing, 49*, 1034-1050.

Malhotra, R., & Jain, A. (2012). Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality. *Journal of Information Processing Systems, 8*(2), 241-262.

Manjula, C., & Florence, L. (2018). Deep Neural Network Based Hybrid Approach for Software Defect Prediction Using Software Metrics. *Cluster Computing*, 1-17.

Martinez, M., & Lecomte, S. (2017). *Towards the Quality Improvement of Cross-Platform Mobile Applications.* Paper presented at the Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on.

Maxeler. (2015). Multiscale Dataflow Programming.

McGeoch, C. C. (2007). Experimental Algorithmics. *Communications of the ACM, 50*(11), 27-31.

McGeoch, C. C. (2012). *A Guide to Experimental Algorithmics*: Cambridge University Press.

McLean, G., Al-Nabhani, K., & Wilson, A. (2018). Developing a Mobile Applications Customer Experience Model (Mace)-Implications for Retailers. *Journal of Business Research, 85*, 325-336.

Meinke, K., & Bennaceur, A. (2018). *Machine Learning for Software Engineering: Models, Methods, and Applications.* Paper presented at the Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings.

Menzies, T., Greenwald, J., & Frank, A. (2006). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering, 33*(1), 2-13.

Meskini, S., Nassif, A. B., & Capretz, L. F. (2013). *Reliability Prediction of Smartphone Applications through Failure Data Analysis.* Paper presented at the Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on.

Mihelic, J., & Cibej, U. (2017). Experimental Algorithmics for the Dataflow Architecture: Guidelines and Issues. *IPSI BgD Transactions on Advanced Research, 13*(1), 1-8.

Moharil, P., Jena, S., & Thakare, V. (2019). Enhancement in Software Reliability Testing and Analysis. *International Journal of Computer Sciences and Engineering*.

Mohsin, A., Naqvi, S. I. R., Khan, A. U., Naeem, T., & AsadUllah, M. A. (2017). *A Comprehensive Framework to Quantify Fault Tolerance Metrics of Web Centric Mobile Applications.* Paper presented at the Communication Technologies (ComTech), 2017 International Conference on.

Molnar, C. (2020). *Interpretable Machine Learning*: Lulu. com.

Moser, R., Pedrycz, W., & Succi, G. (2008). *A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction*. Paper presented at

the Proceedings of the 30th international conference on Software engineering, Leipzig, Germany. https://doi.org/10.1145/1368088.1368114

Muijs, D. (2004). Validity, Reliability and Generalisability. *Doing quantitative research in education with SPSS*, 64-84.

Mushtaq, Z., Kirmani, M., & Saif, S. M. (2016). Mobile Application Development: Issues and Challenges. *International Research Journal of Engineering and Technology (IRJET)*, 1096-1099.

Narkhede, S. (2018). Understanding Auc-Roc Curve. *J Towards Data Science, 26*, 220-227.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., & Soubervielle-Montalvo, C. (2017). Source Code Metrics: A Systematic Mapping Study. *Journal of Systems Software, 128*, 164-197.

Okamura, H., & Dohi, T. (2015). *Towards Comprehensive Software Reliability Evaluation in Open Source Software.* Paper presented at the Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on.

Okutan, A. (2018). Use of Source Code Similarity Metrics in Software Defect Prediction. *https://arxiv.org/abs/1808.10033*.

Orsini, G., Bade, D., & Lamersdorf, W. (2015). *Computing at the Mobile Edge: Designing Elastic Android Applications for Computation Offloading.* Paper presented at the IFIP Wireless and Mobile Networking Conference (WMNC), 2015 8th.

Ortiz, G. B., & Peru, L. (2018). Ibm Rpg Software Quality Prediction Using Machine Learning Techniques. *Universidad César Vallejo*.

Osisanwo, F., Akinsola, J., Awodele, O., Hinmikaiye, J., Olakanmi, O., & Akinjobi, J. (2017). Supervised Machine Learning Algorithms: Classification and Comparison. *International Journal of Computer Trends and Technology (IJCTT) – Volume 48 Number 3 June 2017, 48*(3), 128-138.

Özakıncı, R., & Tarhan, A. (2018). Early Software Defect Prediction: A Systematic Map and Review. *ScienceDirect: The Journal of Systems & Software, 144*, 216-239.

Padhy, N., Panigrahi, R., & Neeraja, K. (2019). Threshold Estimation from Software Metrics by Using Evolutionary Techniques and Its Proposed Algorithms, Models. *Evolutionary Intelligence*, 1-15.

Pai, G. J. (2013). A Survey of Software Reliability Models. *https://arxiv.org/abs/1304.4539*.

Pal, M. (2005). Random Forest Classifier for Remote Sensing Classification. *International Journal of Remote Sensing, 26*(1), 217-222.

Pandey, M., Litoriya, R., & Pandey, P. (2020). Validation of Existing Software Effort Estimation Techniques in Context with Mobile Software Applications. *Wireless Personal Communications, 110*(4), 1659-1677.

Panhwar, A. H., Ansari, S., & Shah, A. A. (2017). Post-Positivism: An Effective Paradigm for Social and Educational Research. *International Research Journal of Arts Humanities, 45*(45).

Park, J., Lee, N., & Baik, J. (2014). *On the Long-Term Predictive Capability of Data-Driven Software Reliability Model: An Empirical Evaluation.* Paper presented at the Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on.

Patel, D., & Patel, A. (2017). Mobile Applications Testing Challenges and Related Solutions. *International Journal of Advanced Research in Computer Science, 8*(3).

Peffers, K., Tuunanen, T., & Niehaves, B. (2018). Design Science Research Genres: Introduction to the Special Issue on Exemplars and Criteria for Applicable Design Science Research. *European Journal of Information Systems, 27*(2), 129-139. doi:10.1080/0960085X.2018.1458066

Picek, S., Heuser, A., Jovic, A., Bhasin, S., & Regazzoni, F. (2019). The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-Channel Evaluations. *IACR Transactions on Cryptographic Hardware Embedded Systems, 2019*(1), 1-29.

Pospieszny, P., Czarnacka-Chrobot, B., & Kobylinski, A. (2018). An Effective Approach for Software Project Effort and Duration Estimation with Machine Learning Algorithms. *Journal of Systems Software*

137, 184-196.

Prasad, M., Florence, L., & Arya, A. (2015). A Study on Software Metrics Based Software Defect Prediction Using Data Mining and Machine Learning Techniques. *International Journal of Database Theory Application, 8*(3), 179-190.

Prashanth, D., & Premaratne, S. (2020). Reliability Analysis of Mobile Application Based on the User Reviews of Health and Fitness. *IJESC*.

Rahman, F., & Devanbu, P. (2013). *How, and Why, Process Metrics Are Better.* Paper presented at the 2013 35th International Conference on Software Engineering (ICSE).

Rao, S. S., Sahitha, I. N., Sireesha, G., & Manoj, P. (2018). Evaluating Software System Reliability Using Architecture Based Approach. *International Journal of Intelligent Information Systems, 7*(1), 1.

Raschka, S. (2015). *Python Machine Learning*: Packt publishing ltd.

Rashid, E. A., Patnaik, S. B., & Bhattacherjee, V. C. (2014). Machine Learning and Software Quality Prediction: As an Expert System. *International Journal of Information Engineering Electronic Business, 6*(2), 9.

Rehman, A. A., & Alharthi, K. (2016). An Introduction to Research Paradigms. *International Journal of Educational Investigations, 3*(8), 51-59.

Rhmann, W., Pandey, B., Ansari, G., & Pandey, D. (2020). Software Fault Prediction Based on Change Metrics Using Hybrid Algorithms: An Empirical Study. *Journal of King Saud University-Computer Information Sciences, 32*(4), 419-424.

Riegler, A., & Holzmann, C. (2018). Measuring Visual User Interface Complexity of Mobile Applications with Metrics. *30*(3), 207-223.

Rosenfeld, A., Kardashov, O., & Zang, O. (2018). *Automation of Android Applications Functional Testing Using Machine Learning Activities Classification.* Paper presented at the 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft).

Ryan, A. B., & students, W. y. T. a. g. f. p. (2006). Post-Positivist Approaches to Research. *Researching Writing your Thesis: a guide for postgraduate students*

12-26.

Sahli, H. (2020). An Introduction to Machine Learning. *TORUS 1–Toward an Open Resource Using Services: Cloud Computing for Environmental Data*, 61-74.

Salama, M., Bahsoon, R., & Lago, P. (2019). Stability in Software Engineering: Survey of the State-of-the-Art and Research Directions. *IEEE Transactions on Software Engineering*.

Sathya, R., & Sudhakar, P. (2016). Improve Software Quality Using Defect Prediction Models. *International Journal of Engineering Management Research, 6*(6), 24-29.

Shanthamallu, U. S., Spanias, A., Tepedelenlioglu, C., & Stanley, M. (2017). *A Brief Survey of Machine Learning Methods and Their Sensor and Iot Applications.* Paper presented at the 2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA).

Shepperd, M., Bowes, D., & Hall, T. (2014). Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering, 40*, 603-616. doi:10.1109/TSE.2014.2322358

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., & Hassan, A. E. (2010). *Understanding the Impact of Code and Process Metrics on Post-Release Defects: A Case Study on the Eclipse Project.* Paper presented at the Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.

Singh, Monika, & Nidhi. (2016). A Critical Review on Software Metrics in Mobile Applications. *International Journal of Innovative Research in Computer and Communuication Engineering, 4*(9), 7.

Singh, N., Verma, P., & Kumar, A. (2016). Software Reliability Models. *SSRN*.

Singh, P. D., & Chug, A. (2017). *Software Defect Prediction Analysis Using Machine Learning Algorithms.* Paper presented at the 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence.

Souza, A. C. d., Alexandre, N. M. C., & Guirardello, E. d. B. (2017). Psychometric Properties in Instruments Evaluation of Reliability and Validity. *Secretaria de Vigilância em Saúde - Ministério da Saúde do Brasil, 26*, 649-659.

Sridhar, S. (2016). A Study on Various Software Models as Inclusive Technology for Sustainable Solutions. In: IJITR.

Sukamolson, S. (2007). Fundamentals of Quantitative Research. *Language Institute Chulalongkorn University, 1*, 2-3.

Sun, B., Ban, T., Chang, S.-C., Sun, Y. S., Takahashi, T., & Inoue, D. (2019). *A Scalable and Accurate Feature Representation Method for Identifying Malicious Mobile Applications.* Paper presented at the Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing.

Tandon, S., Tripathi, S., Saraswat, P., & Dabas, C. (2019). *Bitcoin Price Forecasting Using Lstm and 10-Fold Cross Validation.* Paper presented at the 2019 International Conference on Signal Processing and Communication (ICSC).

Tariq, I., Maqsood, T. B., Hayat, B., Hameed, K., Nasir, M., & Jahangir, M. (2018). *The Comprehensive Study on Software Reliability.* Paper presented at the Computing, Mathematics and Engineering Technologies (iCoMET), 2018 International Conference on.

Thabtah, F., Hammoud, S., Kamalov, F., & Gonsalves, A. (2020). Data Imbalance in Classification: Experimental Evaluation. *Information Sciences, 513*, 429-441.

Tian, Z., Xiang, J., Zhenxiao, S., Yi, Z., & Yunqiang, Y. (2019). *Software Defect Prediction Based on Machine Learning Algorithms.* Paper presented at the 2019 IEEE 5th International Conference on Computer and Communications (ICCC).

Utkin, L. V., & Coolen, F. P. (2018). A Robust Weighted Svr-Based Software Reliability Growth Model. *Reliability Engineering & System Safety*.

Wang, J., Cao, B., Yu, P., Sun, L., Bao, W., & Zhu, X. (2018). *Deep Learning Towards Mobile Applications.* Paper presented at the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).

Wang, J., & Zhang, C. (2018). Software Reliability Prediction Using a Deep Learning Model Based on the Rnn Encoder–Decoder. *Reliability Engineering & System Safety, 170*, 73-82.

Wang, S., Liu, T., Nam, J., & Tan, L. (2018). Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering, 46*.

Wei, H., Hu, C., Chen, S., Xue, Y., & Zhang, Q. (2019). Establishing a Software Defect Prediction Model Via Effective Dimension Reduction. *Information Sciences, 477*, 399-409.

Weichbroth, P. (2020). Usability of Mobile Applications: A Systematic Literature Study. *IEEE Access, 8*, 55563-55577.

Weiser, M. (1991). The Computer for the 21 St Century. *Scientific American, 265*(3), 94-105.

Williams, C. (2007). Research Methods. *Journal of Business & Economics Research (JBER), 5*(3).

Xavier, M. G., Matteussi, K. J., França, G. R., Pereira, W. P., & De Rose, C. A. (2017). *Mobile Application Testing on Clouds: Challenges, Opportunities and Architectural Elements.* Paper presented at the Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on.

Xia, X., Shihab, E., Kamei, Y., Lo, D., & Wang, X. (2016). *Predicting Crashing Releases of Mobile Applications.* Paper presented at the Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.

Yang, J., Chen, J., Hu, W., & Deng, Z. (2017a). *Web-Based Software Reliability Growth Modelling for Mobile Applications.* Paper presented at the 2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP).

Yang, J., Chen, J., Hu, W., & Deng, Z. (2017b). *Web-Based Software Reliability Growth Modelling for Mobile Applications.* Paper presented at the Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2017 14th International Computer Conference on.

Yazdanbakhsh, O., Dick, S., Reay, I., & Mace, E. (2016). On Deterministic Chaos in Software Reliability Growth Models. *Applied Soft Computing, 49*, 1256-1269.

Zein, S., Salleh, N., & Grundy, J. (2016). A Systematic Mapping Study of Mobile Application Testing Techniques. *Journal of Systems and Software, 117*, 334-356.

Zhai, X., Krajcik, J., & Pellegrino, J. W. (2020). On the Validity of Machine Learning-Based Next Generation Science Assessments: A Validity Inferential Network. *Journal of Science Education Technology*, 1-15.

Zhai, X., Yin, Y., Pellegrino, J. W., Haudek, K. C., & Shi, L. (2020). Applying Machine Learning in Science Assessment: A Systematic Review. *Studies in Science Education, 56*(1), 111-151.

Zhao, Y., Da, J., & Yan, J. (2021). Detecting Health Misinformation in Online Health Communities: Incorporating Behavioral Features into Machine Learning Based Approaches. *Information Processing Management*

58(1), 102390.

Zheng, J. (2009). Predicting Software Reliability with Neural Network Ensembles. *Expert Systems with Applications, 36*(2), 2116-2122.

Zimmermann, T., Premraj, R., & Zeller, A. (2007). *Predicting Defects for Eclipse.* Paper presented at the Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007).

APPENDICES

The code for the experiment conducted on the Source Code Metric dataset is given below. For the rest of the code for the other experiments, please contact the author on hoosens@dut.ac.za.

In [2]:
```
!pip install mlxtend
```

```
Requirement already satisfied: mlxtend in c:\users\hoosens\anaconda3\lib\s
ite-packages (0.17.2)
Requirement already satisfied: numpy>=1.16.2 in c:\users\hoosens\anaconda3
\lib\site-packages (from mlxtend) (1.18.1)
Requirement already satisfied: setuptools in c:\users\hoosens\anaconda3\li
b\site-packages (from mlxtend) (45.2.0.post20200210)
Requirement already satisfied: pandas>=0.24.2 in c:\users\hoosens\anaconda
3\lib\site-packages (from mlxtend) (1.0.1)
Requirement already satisfied: matplotlib>=3.0.0 in c:\users\hoosens\anaco
nda3\lib\site-packages (from mlxtend) (3.1.3)
Requirement already satisfied: scikit-learn>=0.20.3 in c:\users\hoosens\an
aconda3\lib\site-packages (from mlxtend) (0.22.1)
Requirement already satisfied: joblib>=0.13.2 in c:\users\hoosens\anaconda
3\lib\site-packages (from mlxtend) (0.14.1)
Requirement already satisfied: scipy>=1.2.1 in c:\users\hoosens\anaconda3\
lib\site-packages (from mlxtend) (1.4.1)
Requirement already satisfied: pytz>=2017.2 in c:\users\hoosens\anaconda3\
lib\site-packages (from pandas>=0.24.2->mlxtend) (2019.3)
Requirement already satisfied: python-dateutil>=2.6.1 in c:\users\hoosens\
anaconda3\lib\site-packages (from pandas>=0.24.2->mlxtend) (2.8.1)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\hoosens\anaco
nda3\lib\site-packages (from matplotlib>=3.0.0->mlxtend) (1.1.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
 c:\users\hoosens\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlx
tend) (2.4.6)
Requirement already satisfied: cycler>=0.10 in c:\users\hoosens\anaconda3\
lib\site-packages (from matplotlib>=3.0.0->mlxtend) (0.10.0)
Requirement already satisfied: six>=1.5 in c:\users\hoosens\anaconda3\lib\
site-packages (from python-dateutil>=2.6.1->pandas>=0.24.2->mlxtend) (1.14
.0)
```

In [3]:
```
#feature selection with filtering method-constant, quasi constant and dupl
icate feature removal
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [4]:
```
from sklearn.model_selection import train_test_split,cross_val_score
from sklearn.ensemble import RandomForestClassifier,RandomForestRegressor
from sklearn.metrics import make_scorer,accuracy_score, precision_score,re
call_score, roc_auc_score,roc_curve,auc
from sklearn.feature_selection import VarianceThreshold
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

In [5]:
```
n=['cbo','dit','fanIn','fanOut','lcom','noc','numberOfAttributes','numberO
fAttributesInherited','numberOfLinesOfCode','numberOfMethods','numberOfMet
hodsInherited','numberOfPrivateAttributes','numberOfPrivateMethods','numbe
rOfPublicAttributes','numberOfPublicMethods','rfc','wmc','bugs']
data=pd.read_csv('SourceCodeMetrics.csv',names=n)
data
```

Out[5]:

	cbo	dit	fanIn	fanOut	lcom	noc	numberOfAttributes	numberOfAttributesInherited	numberOfL
0	9	2	1	9	15	0	1	8	
1	1	1	1	0	0	0	2	0	
2	114	1	102	18	190	6	131	249	
3	5	6	1	4	10	0	0	61	
4	23	2	1	22	820	0	7	416	
...	...	...	...	...	...	...	...	...	
992	0	1	0	0	1	2	3	0	
993	9	6	2	7	15	1	3	386	
994	35	3	25	10	153	9	11	52	
995	7	2	0	7	190	0	2	6	
996	8	2	4	4	780	0	49	20	

997 rows × 18 columns

In [6]:
```
data.keys()
```

Out[6]:
```
Index(['cbo', 'dit', 'fanIn', 'fanOut', 'lcom', 'noc', 'numberOfAttributes
',
       'numberOfAttributesInherited', 'numberOfLinesOfCode', 'numberOfMeth
ods',
       'numberOfMethodsInherited', 'numberOfPrivateAttributes',
       'numberOfPrivateMethods', 'numberOfPublicAttributes',
       'numberOfPublicMethods', 'rfc', 'wmc', 'bugs'],
      dtype='object')
```

In [7]:
```
print(data.bugs)
```

```
0       0
1       0
2       1
3       0
4       0
       ..
992     0
993     0
994     1
995     0
996     1
Name: bugs, Length: 997, dtype: int64
```

In [8]:
```
#X.isnull().sum
```

In [9]:
```
X=data.drop('bugs',axis = 1)
y=data['bugs']
X.shape,y.shape
```

Out[9]: ((997, 17), (997,))

In [10]:
```
X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.2,random
_state=0, stratify=y)
```

In [11]:
```
#Constant feature removal
constant_filter=VarianceThreshold(threshold=0)
constant_filter.fit(X)
```

Out[11]: VarianceThreshold(threshold=0)

In [12]:
```
constant_filter.get_support().sum()
```

Out[12]: 17

In [13]:
```
X_filter=constant_filter.transform(X)
```

In [14]:
```
X_filter.shape, X.shape
```

Out[14]: ((997, 17), (997, 17))

In [15]:
```
#Quasi Constant
quasi_constant_filter=VarianceThreshold(threshold=0.01)
```

In [16]:
```
quasi_constant_filter.fit(X_filter)
```

Out[16]: VarianceThreshold(threshold=0.01)

In [17]:
```
quasi_constant_filter.get_support().sum()
```

Out[17]: 17

In [18]:
```
X_quasi_filter=quasi_constant_filter.transform(X_filter)
```

In [19]:
```
X_quasi_filter.shape, X.shape
```

Out[19]: ((997, 17), (997, 17))

In [20]:
```
# Duplicate Features
X_T = X_quasi_filter.T
```

In [21]:
```
type(X_T)
```

Out[21]: numpy.ndarray

In [22]:
```
X_T = pd.DataFrame(X_T)
```

In [23]:
```
X_T.shape
```

Out[23]: (17, 997)

In [24]:
```
X_T.duplicated().sum()
```

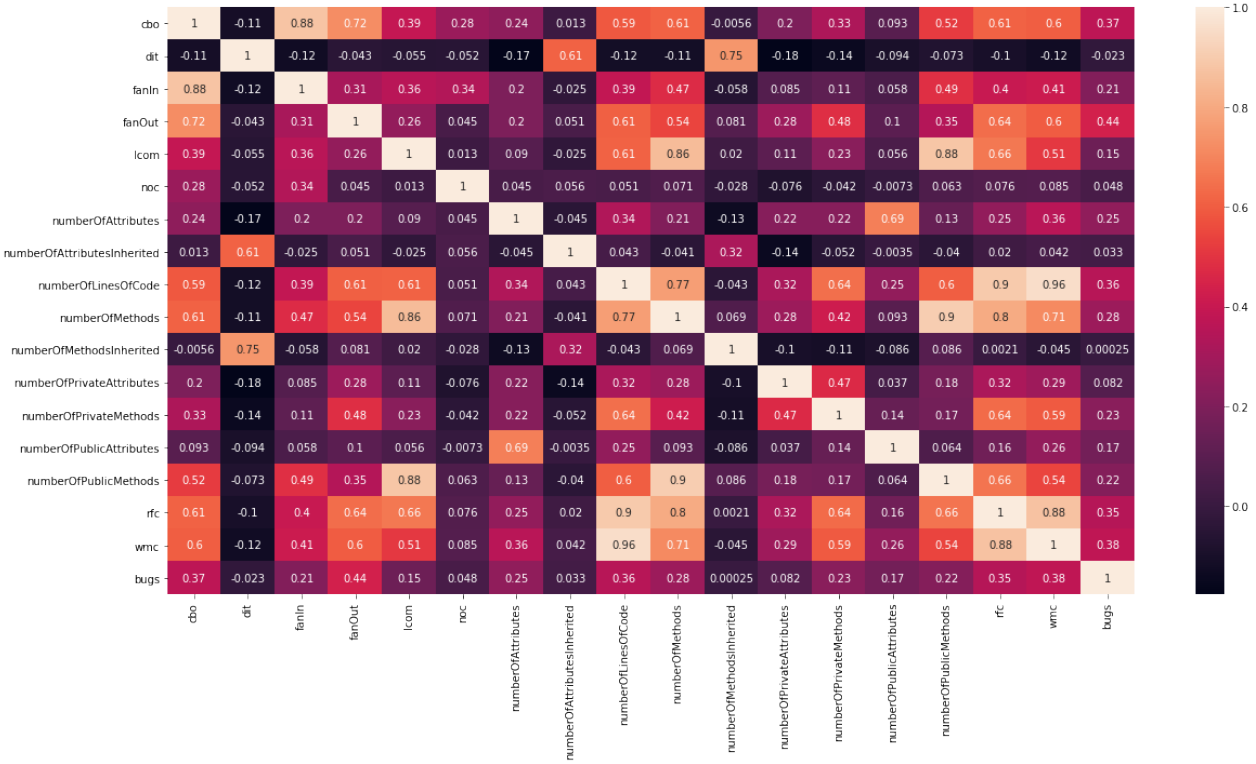
Out[24]: 0

In [25]:
```
#Build a ml model using random forest
def run_Random_Forest(X_train, X_test, y_train, y_test):
    clf=RandomForestClassifier(n_estimators=100, random_state=0,n_jobs=-1)
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_test)
    print('Accuracy on test set')
    print(accuracy_score(y_test,y_pred))
```

In [26]:
```
run_Random_Forest(X_train,X_test,y_train,y_test)
```

```
Accuracy on test set
0.845
```

In [27]:
```
#Pearson Correlation<0.85
corrmat=data.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20,10))
#sns.heatmap(corrmat)
#plot heat map
g=sns.heatmap(data[top_corr_features].corr(),annot=True)#,cmap="RdYlGn")
```



In [28]:
```
X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.2,random
_state=0, stratify=y)
```

In [29]:
```
def get_correlation(data,threshold):
    corr_column=set()
    corrmatrix=data.corr()
    for i in range(len(corrmatrix.columns)):
        for j in range(i):
            if abs(corrmatrix.iloc[i,j])>threshold:
```

```
                colname=corrmatrix.columns[i]
                corr_column.add(colname)
    return corr_column
```

In [30]:
```
corr_features=get_correlation(X_train,0.85)
corr_features
```

Out[30]: {'fanIn', 'numberOfMethods', 'numberOfPublicMethods', 'rfc', 'wmc'}

In [31]:
```
len(corr_features) # number of highly correlated features
```

Out[31]: 5

In [32]:
```
X_train_uncorr=X_train.drop(labels=corr_features, axis=1)
X_test_uncorr=X_test.drop(labels=corr_features, axis=1)
```

In [33]:
```
X_train_uncorr.shape, X_test_uncorr.shape
```

Out[33]: ((797, 12), (200, 12))

In [34]:
```
run_Random_Forest(X_train_uncorr,X_test_uncorr,y_train,y_test)
```

```
Accuracy on test set
0.855
```

In [35]:
```
X_new=X.drop(labels=corr_features,axis=1)
```

In [36]:
```
X_new.shape, y.shape
```

Out[36]: ((997, 12), (997,))

In [37]:
```
X_new
```

Out[37]:

	cbo	dit	fanOut	lcom	noc	numberOfAttributes	numberOfAttributesInherited	numberOfLinesOf(
0	9	2	9	15	0	1	8	
1	1	1	0	0	0	2	0	
2	114	1	18	190	6	131	249	
3	5	6	4	10	0	0	61	
4	23	2	22	820	0	7	416	
...	...	...	...	...	...	...	...	
992	0	1	0	1	2	3	0	
993	9	6	7	15	1	3	386	
994	35	3	10	153	9	11	52	
995	7	2	7	190	0	2	6	
996	8	2	4	780	0	49	20	

997 rows × 12 columns

In [38]:
```
#X_new
```

In [39]:
```
#Step forward feature Selection
sfs=SFS(RandomForestClassifier(n_estimators=100, random_state=0, n_jobs=-1
),
        k_features=12,
        forward=True,
        floating=False,
        verbose=2,
        scoring='accuracy',
       cv=4,
       n_jobs=-1).fit(X_new,y)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers
.
[Parallel(n_jobs=-1)]: Done   4 out of  12 | elapsed:    3.1s remaining:
  6.4s
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:    4.5s finished

[2021-04-12 10:55:26] Features: 1/12 -- score: 0.8385140562248996[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of  11 | elapsed:    1.5s remaining:
  7.1s
[Parallel(n_jobs=-1)]: Done   8 out of  11 | elapsed:    1.7s remaining:
  0.6s
[Parallel(n_jobs=-1)]: Done  11 out of  11 | elapsed:    3.1s finished

[2021-04-12 10:55:30] Features: 2/12 -- score: 0.8264618473895583[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   7 out of  10 | elapsed:    1.8s remaining:
  0.7s
[Parallel(n_jobs=-1)]: Done  10 out of  10 | elapsed:    3.1s finished

[2021-04-12 10:55:33] Features: 3/12 -- score: 0.8304979919678714[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   4 out of   9 | elapsed:    1.7s remaining:
  2.1s
[Parallel(n_jobs=-1)]: Done   9 out of   9 | elapsed:    2.8s remaining:
  0.0s
[Parallel(n_jobs=-1)]: Done   9 out of   9 | elapsed:    2.8s finished

[2021-04-12 10:55:36] Features: 4/12 -- score: 0.8254578313253013[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   3 out of   8 | elapsed:    1.9s remaining:
  3.3s
[Parallel(n_jobs=-1)]: Done   8 out of   8 | elapsed:    2.3s remaining:
  0.0s
[Parallel(n_jobs=-1)]: Done   8 out of   8 | elapsed:    2.3s finished

[2021-04-12 10:55:38] Features: 5/12 -- score: 0.8334939759036145[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   4 out of   7 | elapsed:    2.1s remaining:
  1.6s
```

```
[Parallel(n_jobs=-1)]: Done   7 out of   7 | elapsed:    2.3s finished

[2021-04-12 10:55:41] Features: 6/12 -- score: 0.8525502008032129[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   3 out of   6 | elapsed:    1.9s remaining:
  1.9s
[Parallel(n_jobs=-1)]: Done   6 out of   6 | elapsed:    2.1s finished

[2021-04-12 10:55:43] Features: 7/12 -- score: 0.855562248995984[Parallel(
n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    1.8s remaining:
  2.8s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    1.8s remaining:
  0.0s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    1.8s finished

[2021-04-12 10:55:45] Features: 8/12 -- score: 0.8555502008032129[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   4 out of   4 | elapsed:    1.6s remaining:
  0.0s
[Parallel(n_jobs=-1)]: Done   4 out of   4 | elapsed:    1.6s finished

[2021-04-12 10:55:46] Features: 9/12 -- score: 0.8545542168674698[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   3 out of   3 | elapsed:    1.5s finished

[2021-04-12 10:55:48] Features: 10/12 -- score: 0.854574297188755[Parallel
(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   2 | elapsed:    1.4s finished

[2021-04-12 10:55:50] Features: 11/12 -- score: 0.8535702811244981[Paralle
l(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    1.3s finished

[2021-04-12 10:55:51] Features: 12/12 -- score: 0.8585823293172691
```

In [40]:
```
sfs.k_feature_names_
```

Out[40]:
```
('cbo',
 'dit',
 'fanOut',
 'lcom',
 'noc',
 'numberOfAttributes',
 'numberOfAttributesInherited',
 'numberOfLinesOfCode',
 'numberOfMethodsInherited',
 'numberOfPrivateAttributes',
 'numberOfPrivateMethods',
 'numberOfPublicAttributes')
```

In [41]:
```
sfs.k_score_
```

Out[41]:
```
0.8585823293172691
```

In [42]:
```
feat_cols=list(sfs.k_feature_idx_)
```

In [43]:
```
feat_cols
```

Out[43]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [44]:
```
pd.DataFrame.from_dict(sfs.get_metric_dict()).T
```

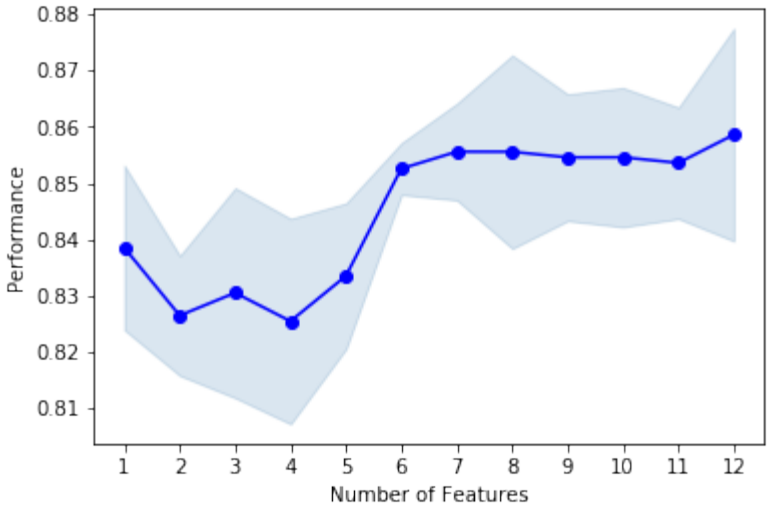
Out[44]:

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev
1	(2,)	[0.84, 0.8554216867469879, 0.8152610441767069,...	0.838514	(fanOut,)	0.0234001	0.0145979
2	(2, 11)	[0.844, 0.8152610441767069, 0.8232931726907631...	0.826462	(fanOut, numberOfPublicAttributes)	0.0170611	0.0106434
3	(2, 4, 11)	[0.824, 0.8514056224899599, 0.8433734939759037...	0.830498	(fanOut, noc, numberOfPublicAttributes)	0.0298776	0.0186387
4	(0, 2, 4, 11)	[0.844, 0.8313253012048193, 0.8313253012048193...	0.825458	(cbo, fanOut, noc, numberOfPublicAttributes)	0.0292228	0.0182303
5	(0, 2, 4, 6, 11)	[0.84, 0.8514056224899599, 0.8232931726907631,...	0.833494	(cbo, fanOut, noc, numberOfAttributesInherited...	0.0207357	0.0129357
6	(0, 2, 3, 4, 6, 11)	[0.86, 0.8514056224899599, 0.8473895582329317,...	0.85255	(cbo, fanOut, lcom, noc, numberOfAttributesInh...	0.00737858	0.00460304
7	(0, 1, 2, 3, 4, 6, 11)	[0.86, 0.8473895582329317, 0.8473895582329317,...	0.855562	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0137677	0.0085888
8	(0, 1, 2, 3, 4, 6, 10, 11)	[0.872, 0.8714859437751004, 0.8313253012048193...	0.85555	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0275086	0.0171609
9	(0, 1, 2, 3, 4, 6, 7, 10, 11)	[0.864, 0.8674698795180723, 0.8433734939759037...	0.854554	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0180301	0.0112478
10	(0, 1, 2, 3, 4, 5, 6, 7, 10, 11)	[0.844, 0.8755020080321285, 0.8514056224899599...	0.854574	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0198188	0.0123637
11	(0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11)	[0.844, 0.8634538152610441, 0.8433734939759037...	0.85357	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.0158471	0.0098860
12	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)	[0.852, 0.8875502008032129, 0.8353413654618473...	0.858582	(cbo, dit, fanOut, lcom, noc, numberOfAttribut...	0.030238	0.0188636

In [45]:
```
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
```

In [46]:
```
plot_sfs(sfs.get_metric_dict(),kind='std_dev')
```

Out[46]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1c517155708>)
```



In [47]:
```
X_best=X_new.iloc[:,feat_cols]
```

In [48]:
```
#X_best
X_best
```

Out[48]:

	cbo	dit	fanOut	lcom	noc	numberOfAttributes	numberOfAttributesInherited	numberOfLinesOf(
0	9	2	9	15	0	1	8	
1	1	1	0	0	0	2	0	
2	114	1	18	190	6	131	249	
3	5	6	4	10	0	0	61	
4	23	2	22	820	0	7	416	
...	...	...	...	...	...	...	...	
992	0	1	0	1	2	3	0	
993	9	6	7	15	1	3	386	
994	35	3	10	153	9	11	52	
995	7	2	7	190	0	2	6	
996	8	2	4	780	0	49	20	

997 rows × 12 columns

In [49]:
```
X_best.shape, y.shape
```

Out[49]:
```
((997, 12), (997,))
```

In [55]:
```
#10 fold cross Validation and scoring
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
y = np.array([number[0] for number in lb.fit_transform(y)])

rfc=RandomForestClassifier(n_estimators=100, max_depth=10, random_state=1)

accuracy=cross_val_score(rfc,X_best,y,cv=10,scoring='accuracy')
a_float = np.mean(accuracy)
formatted_float = "{:.2f}".format(a_float)
print('Accuracy', formatted_float, accuracy)

recall = cross_val_score(rfc, X_best, y, cv=10, scoring='recall')
a_float = np.mean(recall)
formatted_float = "{:.2f}".format(a_float)
print('Recall', formatted_float, recall)

precision = cross_val_score(rfc, X_best, y, cv=10, scoring='precision')
a_float = np.mean(precision)
formatted_float = "{:.2f}".format(a_float)
print('Precision', formatted_float, precision)

f1 = cross_val_score(rfc, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)999
```

```
Accuracy 0.85 [0.89       0.81       0.86       0.89       0.83       0.86
 0.88       0.78787879 0.8989899  0.83838384]
Recall 0.96 [0.975      0.92405063 0.94936709 0.97468354 0.93670886 0.9873
4177
 0.96202532 0.92405063 0.97468354 0.94936709]
Precision 0.87 [0.89655172 0.84883721 0.88235294 0.89534884 0.86046512 0.8
5714286
 0.89411765 0.82954545 0.90588235 0.86206897]
F1 Score 0.91 [0.93413174 0.88484848 0.91463415 0.93333333 0.8969697  0.91
764706
 0.92682927 0.8742515  0.93902439 0.90361446]
```

In [58]:

```
from sklearn import svm
sv=svm.SVC(probability=True)

accuracy=cross_val_score(sv,X_best,y,cv=10,scoring='accuracy')
print('Accuracy', np.mean(accuracy), accuracy)


recall = cross_val_score(sv, X_best, y, cv=10, scoring='recall')
print('Recall', np.mean(recall), recall)

precision = cross_val_score(sv, X_best, y, cv=10, scoring='precision')
print('Precision', np.mean(precision), precision)

f1 = cross_val_score(sv, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)
```

```
Accuracy 0.8194949494949496 [0.81       0.82       0.79       0.83       0
```

```
.8        0.83
 0.82       0.80808081 0.87878788 0.80808081]
Recall 0.9848101265822784 [1.         0.98734177 0.96202532 1.         0.9
7468354 0.98734177
 1.         0.97468354 0.98734177 0.97468354]
Precision 0.8230009650869492 [0.80808081 0.82105263 0.80851064 0.82291667
0.81052632 0.82978723
 0.81443299 0.81914894 0.87640449 0.81914894]
F1 Score 0.90 [0.89385475 0.89655172 0.87861272 0.90285714 0.88505747 0.90
17341
 0.89772727 0.89017341 0.92857143 0.89017341]
```

In [59]:
```
from sklearn.neural_network import MLPClassifier
nn=MLPClassifier()

accuracy=cross_val_score(nn,X_best,y,cv=10,scoring='accuracy')
print('Accuracy', np.mean(accuracy), accuracy)


recall = cross_val_score(nn, X_best, y, cv=10, scoring='recall')
print('Recall', np.mean(recall), recall)

precision = cross_val_score(nn, X_best, y, cv=10, scoring='precision')
print('Precision', np.mean(precision), precision)

f1 = cross_val_score(nn, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)
```

```
Accuracy 0.8014242424242426 [0.76       0.79       0.82       0.88       0
.83       0.73
 0.78       0.77777778 0.82828283 0.81818182]
Recall 0.9103006329113923 [0.8625     0.88607595 0.96202532 0.96202532 0.9
7468354 0.97468354
 0.87341772 0.83544304 0.92405063 0.84810127]
Precision 0.8607060816290526 [0.88888889 0.83146067 0.86206897 0.81521739
0.82352941 0.87804878
 0.8625     0.87837838 0.88235294 0.88461538]
F1 Score 0.89 [0.8974359  0.84615385 0.87209302 0.90909091 0.90909091 0.88
095238
 0.87654321 0.88484848 0.90361446 0.90243902]
```

In [60]:
```
from sklearn.naive_bayes import GaussianNB
nbc=GaussianNB()

accuracy=cross_val_score(nbc,X_best,y,cv=10,scoring='accuracy')
print('Accuracy', np.mean(accuracy), accuracy)


recall = cross_val_score(nbc, X_best, y, cv=10, scoring='recall')
print('Recall', np.mean(recall), recall)

precision = cross_val_score(nbc, X_best, y, cv=10, scoring='precision')
print('Precision', np.mean(precision), precision)
```

```
f1 = cross_val_score(nbc, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)
```

```
Accuracy 0.8254949494949495 [0.85       0.8        0.85       0.81       0
.82       0.81
 0.82       0.81818182 0.85858586 0.81818182]
Recall 0.9506487341772152 [0.9875     0.94936709 0.96202532 0.94936709 0.9
6202532 0.93670886
 0.92405063 0.96202532 0.94936709 0.92405063]
Precision 0.8481846647975682 [0.84946237 0.82417582 0.86363636 0.83333333
0.83516484 0.84090909
 0.85882353 0.83516484 0.88235294 0.85882353]
F1 Score 0.90 [0.9132948  0.88235294 0.91017964 0.88757396 0.89411765 0.88
622754
 0.8902439  0.89411765 0.91463415 0.8902439 ]
```

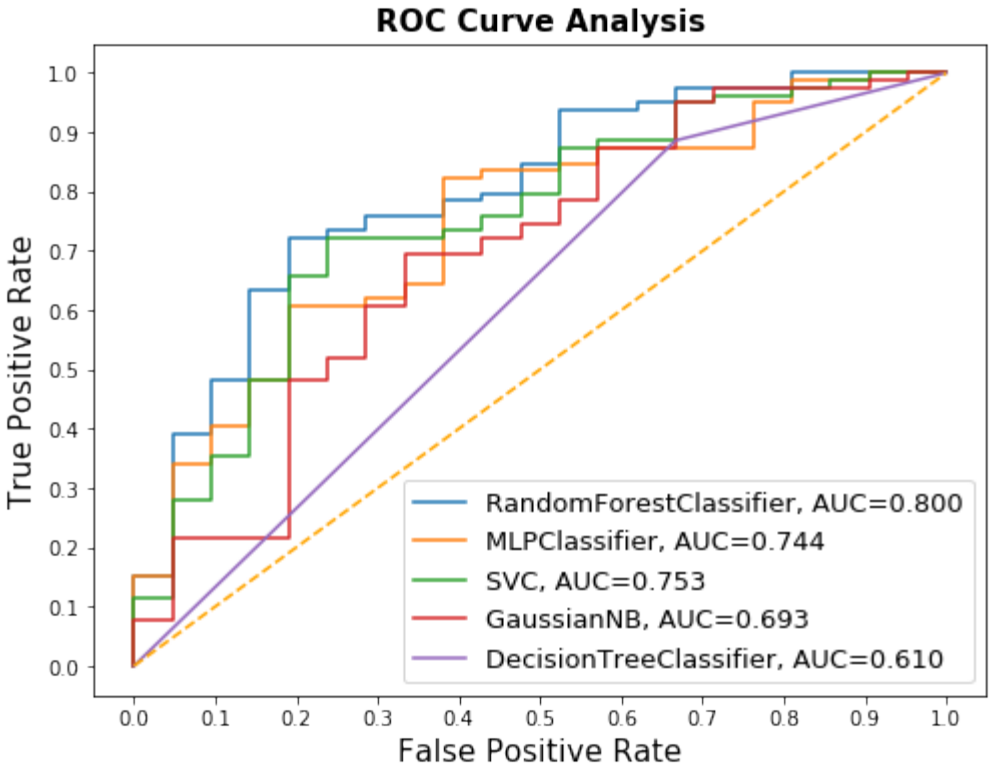
In [61]:
```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()

accuracy=cross_val_score(dtc,X_best,y,cv=10,scoring='accuracy')
print('Accuracy', np.mean(accuracy), accuracy)


recall = cross_val_score(dtc, X_best, y, cv=10, scoring='recall')
print('Recall', np.mean(recall), recall)

precision = cross_val_score(dtc, X_best, y, cv=10, scoring='precision')
print('Precision', np.mean(precision), precision)

f1 = cross_val_score(dtc, X_best, y, cv=10, scoring='f1')
a_float = np.mean(f1)
formatted_float = "{:.2f}".format(a_float)
print('F1 Score', formatted_float, f1)



#roc_auc = cross_val_score(dtc, X_best, y, cv=5, scoring='roc_auc')
#print('ROC_AUC', np.mean(roc_auc), roc_auc)

### Visualize accuracy for each iteration

#scores = pd.DataFrame(roc_auc,columns=['Scores'])

#sns.set(style="white", rc={"lines.linewidth": 3})
#sns.barplot(x=['Iter1','Iter2','Iter3','Iter4','Iter5'],y="Scores",data=s
cores)
#plt.show()
#sns.set()
```

```
Accuracy 0.8074848484848485 [0.89       0.73       0.8        0.83       0
.78       0.8
 0.76       0.78787879 0.86868687 0.82828283]
Recall 0.8634335443037975 [0.8875     0.82278481 0.89873418 0.89873418 0.8
6075949 0.83544304
 0.78481013 0.88607595 0.87341772 0.88607595]
```

```
Precision 0.8967232668485237 [0.97260274 0.86486486 0.87804878 0.89873418
0.87012987 0.85526316
 0.88157895 0.86585366 0.94594595 0.93421053]
F1 Score 0.88 [0.92207792 0.83018868 0.88198758 0.89308176 0.83660131 0.87
179487
 0.84415584 0.88198758 0.9044586  0.8961039 ]
```

In [89]:

```
# Instantiate the classfiers and make a list
classifiers = [rfc,nn,sv,nbc,dtc]

# Define a result table as a DataFrame
result_table = pd.DataFrame(columns=['classifiers', 'fpr','tpr','auc'])

X_train, X_test, y_train, y_test=train_test_split(X_best,y,test_size=0.1,r
andom_state=0, stratify=y)

# Train the models and record the results
for cls in classifiers:
    model = cls.fit(X_train, y_train)
    yproba = model.predict_proba(X_test)[::,1]

    fpr, tpr, _ = roc_curve(y_test,  yproba)
    auc = roc_auc_score(y_test, yproba)

    result_table = result_table.append({'classifiers':cls.__class__.__name
__,
                                        'fpr':fpr,
                                        'tpr':tpr,
                                        'auc':auc}, ignore_index=True)

# Set name of the classifiers as index labels
result_table.set_index('classifiers', inplace=True)

#Plot the results

fig = plt.figure(figsize=(8,6))

for i in result_table.index:
    plt.plot(result_table.loc[i]['fpr'],
             result_table.loc[i]['tpr'],
             label="{}, AUC={:.3f}".format(i, result_table.loc[i]['auc']))

plt.plot([0,1], [0,1], color='orange', linestyle='--')

plt.xticks(np.arange(0.0, 1.1, step=0.1))
plt.xlabel("False Positive Rate", fontsize=15)

plt.yticks(np.arange(0.0, 1.1, step=0.1))
plt.ylabel("True Positive Rate", fontsize=15)

plt.title('ROC Curve Analysis', fontweight='bold', fontsize=15)
plt.legend(prop={'size':13}, loc='lower right')

plt.show()
```


ROC Curve Analysis
True Positive Rate
False Positive Rate
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
RandomForestClassifier, AUC=0.800
MLPClassifier, AUC=0.744
SVC, AUC=0.753
GaussianNB, AUC=0.693
DecisionTreeClassifier, AUC=0.610