CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Fault Prediction in Android Systems through AI

A Method for Predicting Defects in Android Systems using Machine Learning

Master's thesis in Computer science and engineering

MURTADA AHMED AND KIRSTEN BASSUDAY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

MASTER'S THESIS 2019

Fault Prediction in Android Systems through AI

A Method for Predicting Defects in Android Systems using Machine Learning

MURTADA AHMED AND KIRSTEN BASSUDAY

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

A Method for Predicting Defects in Android Systems using Machine Learning
MURTADA AHMED AND KIRSTEN BASSUDAY

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Co-Supervisor: Daniel Strüber, Department of Computer Science and Engineering
Advisor: William Leeson, Aptiv
Examiner: Richard Johansson, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

A Method for Predicting Defects in Android Systems using Machine Learning

MURTADA AHMED AND KIRSTEN BASSUDAY
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software code defect prediction is important in improving code quality and the turnaround time of software products. In this thesis we investigate how to create and extract features, analyze existing work to create and realize a defect prediction technique that can be applied in an industrial setting. We conduct this investigation on version controlled source code from Git and Jira data. We identify and define metrics to be collected and build four Machine Learning (ML) models to predict if a file is *clean* or *defective*. We create a Cost Effectiveness (CE) evaluation technique to measure the performance of our ML models and achieve a score of 87% and an accuracy of 88 % on our best models.

Keywords: Machine Learning, Data Science, Process Metrics, Git, Defect Prediction, Repository Mining, Jira.

Acknowledgements

We would like to thank our supervisors Thorsten Berger and Daniel Strüber, and our examiner Richard Johansson, who provided us with continued support and guidance throughout our thesis. We would also like to acknowledge Aptiv and William Leesson who supplied us with the necessary resources, access and information.

Murtada Ahmed and Kirsten Bassuday, Gothenburg, June 2019

Contents

List of Figures

List of Tables

List of Acronyms

ML	Machine Learning
LR	Logistic Regression
NB	Naive Bayes
CSV	Comma Separated Value
OO	Object-Oriented
ROC	Receiver Operating Characteristic
AUC	Area Under the Curve
API	Application Programming Interface
SCM	Source Control Management
MLP	Multi-Layer Perceptron
NN	Neural Network
RBF	Radial Basis Function
RF	Random Forest
DT	Decision Tree
KNN	K-Nearest Neighbour
SVM	Support Vector Machine
BBN	Bayesian Belief Network
CE	Cost Effectiveness
CVS	Concurrent Versions System
CFS	Correlation-based Feature Selection
LOC	Lines of Code
SLOC	Source Lines of Code
VP	Voted Perceptron
LMT	Logistic model trees
CART	Classification & Regression Trees
LB	Logit Boost
BN	Bayesian Networks
DTNB	Decision Table Naive Bayes
ADT	Alternating Decision Trees
VFI	Voting Feature Intervals
Bag	Bagging
NNge	Nearest Neighbour with generalized exemplars
BAUC	Baseline Area Under the Curve
OAUC	Optimal Area Under the Curve

1 Introduction

Defect prediction in software is an important activity in software engineering [30]. Being able to correctly identify defects and error prone software can positively impact the efforts of the developers and testing resources, improve the quality of the code [30] and decrease the maintenance needed before a software release [23].

Software defect prediction can be described as grouping the code as either *defective* or *clean* [23]. There have been many studies in this field, such as Arisholm [3] and Zimmermann [39] who have used open source data for machine learning and predicting which files in their Source Control Management (SCM) systems were defective. What we can observe from these studies is that there is no single way to achieve defect prediction in software [3, 12, 23], and studies are mainly done on case studies in an academic or theoretical setting with different data sets, data pre-processing steps, validation techniques and performance measures, thus making comparisons and conclusions of the prediction results tricky [30]. This can make it difficult to correctly choose the right techniques and methods to apply software fault prediction in a practical industry setting. This thesis study tackles the challenge of selecting and applying software defect prediction techniques in a industry setting.

1.1 Problem

Aptiv is global technology company that specializes in creating and delivering software solutions in the automotive space. One of its focus areas is in Android software for vehicle infotainment systems. This system can be described as a in-dash platform in a vehicle that provides functions to entertain and interact with the driver and passengers. It has components such as media player controls and file management systems [33].

This thesis will focus on the infotainment system codebase, provided by Aptiv, where we will create a technique to detect potential harmful code changes in files. By performing early defect detection we hope to achieve a improvement on the software quality and product, minimize debugging costs and overall increase business value.

Aptiv currently performs code reviews and unit testing to their code-base to try minimize errors. These techniques help develop a healthy development cycle, but does not eliminate errors. We propose a defect detection system based on machine learning to be implemented before a code release, to help flag code changes that could be problematic before these changes detriment the platform and the business. By building a defect prediction system, we aim to analyze the logs and commits before and after a code change, in a source code file, to try identify log items caused by the changes and recognize potential harmful changes. We will also attempt to further investigate how to integrate this machine learning tool to the commit process. This will flag potentially harmful changes and recognize problematic areas and in turn trigger more detailed testing to be done on those changes.

The findings of the proposed thesis will add business value to the chain of vendors (developers) and their customers. Better testing could be implemented and time can be saved by the use of the proposed tool. This can result in higher quality code and improve the development and maintenance process [19].

1.2 Goal

Our goal is to help create and realize a defect prediction technique that can be applied within Aptiv to the infotainment system. We will create a case study to do this and find existing work and studies through a literature survey, that uses defect prediction techniques. Through our literature survey findings, we hope to develop a technique to find and extract an optimal set of features (metrics) that can be used in the defect prediction. We also aim to create a learning model and suitable evaluation measures that will have the best performance based on these features. As a result, we hope to find interesting patterns which would provide insight and feedback to Aptiv's development teams. Lastly, we also aim at integrating the defect prediction system to Aptiv's development cycle in order to predict when the changes are likely to cause conflicts and errors.

1.3 Outline

In this chapter we have introduced the reader to the concept of defect prediction in software repositories and the problem that needs to be solved.
In Chapter 2, we review some theoretical background to help establish concepts and definitions relating to this thesis work.
In Chapter 3, we explain the methodology of our work and how we conducted the thesis work and developed our approach in solving the defined problem.
In Chapter 4, we examine the literature survey results that find the best approach in terms of feature collection, machine learning models and evaluation techniques.
In Chapter 5, we analyze results from the metric selection, the machine learning model and the evaluation techniques.
In Chapter 6, we discuss and contextualize our results and design decisions. In Chapter 7, we draw conclusions from the results and the discussion and discuss future work.

2
Theory

2.1 Related work

Previous studies in this area attempted multiple combinations of techniques to predict software defects. These studies were carried out on different levels. Queiroz [27] predicted defects on a feature level, compared to Zimmermann, who was operating on lower levels of granularity such as packages and files. Some studies drilled down to the class level as in Arisholm [3] and Malhotra [24]. The norm for using machine learning in this field is to use a chronological release-based approach, meaning that in almost all studies training data was extracted from the first $n-1$ releases of a project to build the model and hence predict faults or defects in the nth release, which is regarded as the test set. This allows for tracking the histories of files and learning how previous metrics (predictors) impacted the state of the file (clean or defective).

Zimmermann et al. [39] used a release based approach to predict defects on the Eclipse software project by looking at the history of releases. Their goal was to answer why some components are more failure-prone than others. They looked for corrective commits by searching for keywords such as "fixes" in the commit message; thus, for each corrected file, its previous versions were labeled as defective. A set of code metrics were developed and used to build LR models for defect prediction. Other studies such as [24], [28] and [3] also proposed adding process metrics to code metrics to improve the performance of the models as they proved to have stronger correlations with the output class. Queiroz [27] and Moser [25] were based on the aforementioned papers and reached similar conclusions in that, process metrics outperform code metrics in almost all models and for different evaluation criteria.

Evaluating models and their corresponding metrics can differ and depend on the aim of the project, that is, a LR model using process metrics can outperform a Support Vector Machine (SVM) model using code metrics in terms of cost effectiveness and Receiver Operating Characteristic (ROC) curves, although not in terms of measures such as accuracy and recall. In projects such as the one in this thesis work, cost efficiency is an important measure since our goal is to help software testers in identifying erroneous files efficiently and minimize testing efforts.

2.2 Repository Mining

Software repositories hold a wealth of information on the source code life. It contains not only the source code, but the information around it, such as the commit information, the authors and committers who worked on it, the Application Programming Interfaces (APIs) used and the guidelines on how to use them, the bug fixes made, re-factoring changes and in-line comments on new code and functions[38]. The challenge is to extract meaningful data from these repositories through the use of repository mining. Repository Mining involves using a set of techniques and tools to efficiently extract this data from the software repositories [6].

2.3 Source Control Management

SCM systems help manage software versions and code changes. It allows code changes to be integrated without overwriting lines of code, when be merged from multiple sources [2]. It provides a certain level of accountability and trace-ability for the life cycle of the code. The most prominent example of an SCM is Github; a platform that hosts all versions of software code changes along with its metadata.

2.3.1 Git and Gerrit

Git is a open source distributed SCM tool. It can be applied in version control of source code as well as tracking changes within a file [29]. It can be locally enabled, has local branching and multiple work flows. It is mainly used for team collaboration. Gerrit is a open source software tool that helps support code and peer reviews of any changes before being committed to the repository[14]. Gerrit is a web-based tool that integrates with Git.

2.3.2 Jira

Jira is a software platform that provides tools to help software development teams to plan, track and manage their work [4]. It can be used for bug and issue tracking, task management and helps support agile software development projects[4].

2.4 Metrics

The term *metrics* in this thesis refers to the features of the ML models. It is a commonly used term in the field of defect prediction. The terms *feature* and *metric* are used interchangeably. Three main type of metrics are considered when dealing with defect prediction problems. These are *Code, Process* and *Delta.* Code metrics (also called complexity, static or Object-Oriented (OO) metrics) are properties that measure software or a piece of code. Examples of code metrics can vary from the Lines of Code (LOC), number of methods or classes in the source code, to a value to measure the maintainability (Maintainability Index) or how tightly coupled the code is (Class Coupling Index). Process metrics define how developers log information about their work. Number of developers, developer experience and commit count are all process metrics. Lastly, delta metrics are attributes that evaluates code churn.

2.5 Supervised Learning

Supervised Learning is a technique in ML where, given input *(X)* and an output variable *(Y)*, an algorithm is applied to map the function from the input to the output *Y = f(X)* [17]. The goal of supervised learning is to build a model of the distribution of class labels, in terms of the predictor features, so that prediction of class labels can be done on the output variables (Y) for new data given in (X) [22]. The type of supervised learning used and discussed in this thesis is classification as the aim is to predict if a source code file is defective or clean. Logistic Regression, Multi-Layer Perception, Gaussian Naive Bayes and Random Forrest Classifier are the four classification algorithms that are used in this thesis.

2.5.1 Logistic Regression

Logistic Regression predicts the probability of an outcome, that can have one of two values. The nominal variable is the dependent variable (outcome), and the measurement variable is the independent variable [1]. A goal of LR is to find if the probability of attaining a particular value of the nominal variable is associated with the measurement variable [1] . The second goal is to predict the probability of getting a particular value of the nominal variable, given the measurement variable [1].

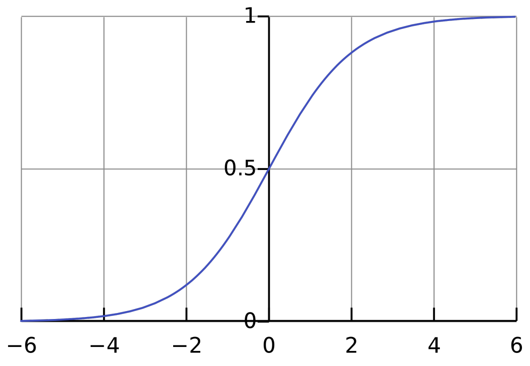


Figure 2.1: Logistic Sigmoid Function [36]

Logistic Regression produces a *S -shaped* curve seen in figure 2.1, which is limited to values between 0 and 1.

$$p(X) = \frac{e^{(b_0+b_1x)}}{1+e^{(b_0+b_1x)}}$$

In the formula for logistic regression, *P(X)* will give values of 0 or 1 for all values of X. The coefficients b_0b_1 represent the intercept and the slope of the model and are estimated using the training data. This estimation is done using the *maximum likelihood*. Hyper-parameters available for tuning are the solver (type of optimization algorithm used), penalty (L1 or L2) and the class weights [9].

2.5.2 Multi-Layer Perceptron

The MLP algorithm is an implementation of a Neural Network (NN), that predicts a set of outputs from a set of inputs [34]. MLP's "train on a set of input-output pairs and learns to model the correlation (or dependencies) between those inputs and outputs" [31]. They also may have many layers in-between the input and output layer which perform the computations of the MLP.

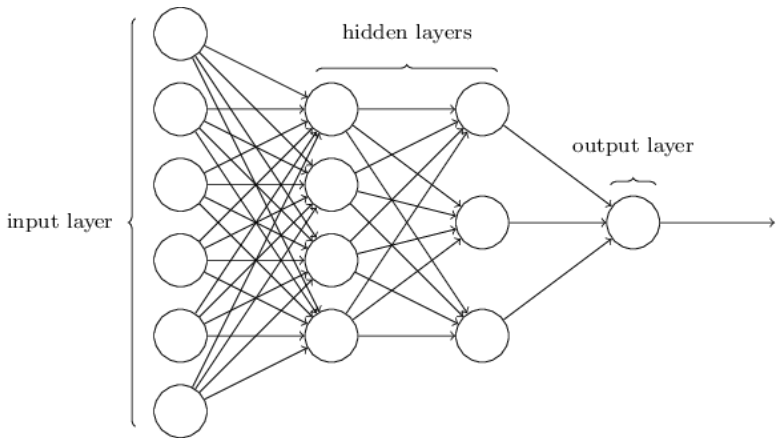


Figure 2.2: Multi-layer Neural Network [10]

A MLP can be seen as a LR classifier, where the input is first transformed using a trained non-linear transformation [35]. This transformation projects the input data into a space where it becomes linearly separable [35]. This in-between layer is called a hidden layer as seen in figure 2.2.

$$f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x)))$$

The above matrix notation represents a one-hidden-layer MLP, with bias vectors $b^{(1)}$, $b^{(2)}$; weight matrices $W^{(1)}$,$W^{(2)}$ and activation functions G and s. A popular choice of activation function for s is the *sigmoid* activation function. Some of the hyper-parameters for this ML model that can be optimized are the activation function, numbers of hidden layer units, regularization parameters, the learning rate, the weight initialization and the solver used [9].

2.5.3 Gaussian Naive Bayes

NB is a probabilistic classification algorithm for binary (two-class) and multi-class classification problems. As mentioned, in the context of this thesis binary classification will be used. NB uses conditional probability to calculate the probability of each feature. It assumes that each feature is independent of one another. Gaussian NB uses a Gaussian (normal) distribution, where the mean and the standard deviation are estimated from the training data [17]. Predictions are from a continuous value and are not discreet. The formula for this classifier is based on Bayes theorem. Here we assume the features are independent of each other and then find the probability of x_i given that y has occurred.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2})$$

2.5.4 Random Forest Classifier

The RF algorithm is part of the ensemble Decision Tree ML models. Bagging is a technique to help reduce the variance in statistical learning model [16], and used on decision trees to help improve predictions. In bagging, successive trees do not depend on earlier trees, each tree is independently built using a bootstrap sample of the dataset [21]. Majority vote is then used for prediction. This improves the accuracy on decision trees but also means the bagged trees could look similar to each other and the predictions could be highly correlated [16]. RF overcomes this problem by introducing a layer of randomness to the construction of the trees. RF builds each tree using a different sample of the data and each node is split using the best solution from the set of predictors (features) randomly chosen at that node [16]. This means other features will have more of a chance and the averaging of the resulting trees will be more accurate and reliable and there is a less chance of over-fitting.

Hyper-parameters from [9] that can be optimized for the RF algorithm are:
Criterion: The function to measure the quality of a split.
Number of estimators/trees: number of uncorrelated trees to create the random forest.
Number of features: the number of features to sample and pass onto each tree, this is where feature bagging happens.
Maximum depth: The maximum depth of the tree.
Minimum samples split: The minimum number of samples required to split an internal node.
Minimum samples leaf: The minimum number of samples required to be at a leaf node.
Maximum features: The number of features to consider when looking for the best split.

2.6 Evaluation

2.6.1 Confusion Matrix Measures

Confusion matrices are commonly used when evaluating binary class predictions. Measures taken from this type of evaluation are: Accuracy, Precision, Recall and F-measure scores. We cite the definitions of the measures from [28] as follows.
Accuracy of the model is the proportion of correct predictions.
Precision measures the percentage of model declared defective entities that are actually defective.

Recall is the proportion of actually defective entities that the model can successfully identify.
F-Measure is the harmonic mean of the Precision and Recall.
ROC is a curve that maps the benefits (true positives) against the cost (false positives) of the model against thresholds from 0 to 1, [28, 3]. Using ROC allows model comparison to be done against different thresholds.
AUC is a measure that is derived from ROC analysis to calculate the accuracy of the model [24]. The idea is that the higher the AUC, the better the model is. We transfer this concept to the context of CE (section 2.6.2), so that a model with a higher area under the CE curve indicates that it has a better performance.

2.6.2 Cost Effectiveness

The aforementioned confusion matrix measures are adequate for most machine learning models, where only the quality of the prediction matters. Nonetheless, none of them help in identifying the most probable defective files with the minimum effort. This is the problem that we are trying to solve through our thesis work; helping developers to focus their testing and debugging efforts on a subset of files that are most likely to be faulty. Thus, Cost Effectiveness, inspired from [3] is used. To illustrate, if we try to find defective files randomly (represented by the baseline model in the figure 2.3) we get a linear curve, meaning that the time to find defects increases linearly with the number of files sampled.
This can be improved; the idea of this CE method is to rank the files from highest to lowest probability of being defective and then plotting the percentage of sampled files versus the percentage of defects found. Notice that the optimal model finds almost all defective files by sampling only a small subset of the files, meaning its classifications results had very high quality. A good model should have a curve and an AUC value between the optimal and baseline models. The higher this value, the higher the performance of the model.

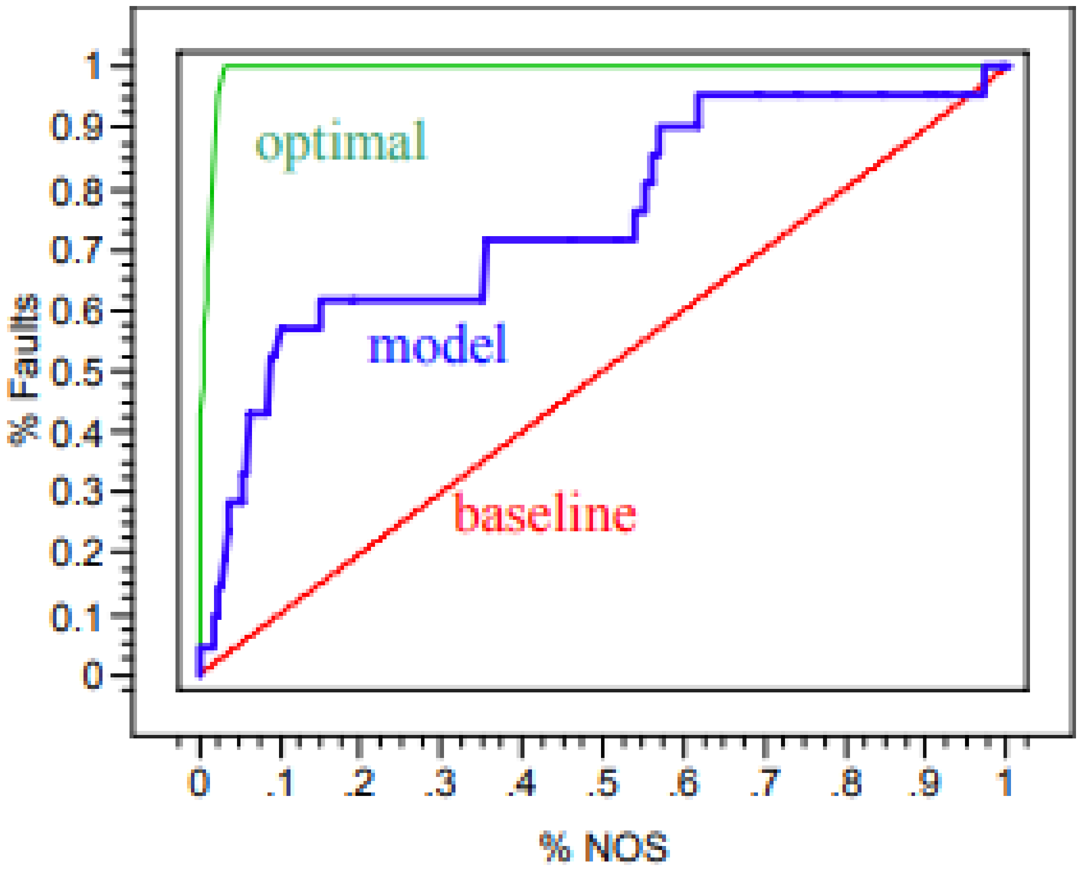


Figure 2.3: Model CE Curve [3]

3
Methodology

In this section we outline our approach and steps taken to conduct a literature survey, gather and clean the data, build the ML models and the evaluation techniques chosen.

3.1 Literature Survey

In our research towards designing models for predicting software defects and fault-prone files, we will compare and analyze the methodologies, results and conclusions of previous related work through a literature survey. The aspects of interest will revolve around four main topics: data collection, the choice of explanatory variables (code metrics, process metrics or delta measures [3]), learning models (SVM, LR, NN) and the evaluation criteria used.

In order to find related work a qualitative investigation method is used to search for the relevant sources for analysis. Articles such as [39, 28, 27] were provided by our supervisor as a starting point. We use a "snowballing" technique from the bibliography and citations in the papers, to find other relevant literature sources. This study also uses the internet to collect the data and resources needed. The phrases such as *"machine learning detecting faults in code"* and *"Machine Learning in software defect prediction"* were used in searching for research papers and academic journals. Each article was briefly examined using techniques from [18], on *"How to Read a Paper"* . If the abstract, key headings and conclusion had mentions of *machine learning*, *fault prediction*, *software defects* or known machine learning techniques such as SVM or NB, these articles were recorded as a potential data source. These selected papers are then read and analyzed in regards to the context of the text and appropriateness to the questions posed in this literature survey.

3.2 Data Gathering

3.2.1 Application Programming Interface

We use two different APIs to automatically perform repository mining and extract data from the Issue Tracking system, Jira. To retrieve Git data from the repositories we use the Pydriller API [32]. Commit-related data such as committer, author, files and other metrics are extracted on request. After extraction, all the desired metrics are loaded into a pandas dataframe where each record is uniquely defined by the commit hash and the file name to indicate that version of the file and its associated attributes. We use this dataframe as a staging table in order to calculate more metrics and perform aggregations.

The second API used is the Jira-Python [26] library. This connects directly to the Jira Atlassian web application and accesses the issues of the specified projects. As there is a limit of the number issues that can be extracted via the API, a pagination technique was used to extract and iterate the issues. A subset of the columns that matched our defined metrics were selected via the API methods and stored in a pandas data-frame as well.

3.2.2 Metric Collection

The data used to collect metric information was held in Aptiv's infotainment system's Git repositories (using Gerrit), in the form of the Git stat logs and in their Jira projects. We used Git to pull the repositories from Gerrit's web platform into local environments as the Infotainment system code was spread over several repositories. Some repositories held the original code base, while others were branches of the original that were modified or new code to create new functionality of the infotainment system. Each repository contained a commit history ranging from 10 commits to 200 commits each. The challenge was to find and link the repositories with most commits that stemmed from the same code base.

The Jira data was spread across four large projects which was stored in Jira Issues containing a combination of predefined and project customized columns. Each Jira issue had over 300 columns when extracted from the Jira web application. Manual extraction was not feasible as the Jira web application only allows a maximum of 1000 issues to be extracted to a csv file at a time. We used the Jira API in python to extract the relevant columns to aid in generating our metrics; as many of those 300 columns were used differently by different projects and development teams.

As Aptiv uses a general 2 week sprint cycle, the data set is grouped accordingly. As there is no recorded release dates to indicate where the releases start and end, a grouping of 2 week cycles is done from the first recorded commit date. By splitting the data into releases, we achieve a level of granularity that is required by the defined metrics.
Both of theses data sources require dynamic extraction methods. This thesis uses open source python libraries to achieve this.

3.2.3 Data Cleaning

The gathered data needed to be cleaned and stored in accessible formats. Empty string values were replaced with zeros.

3.2.4 Data Labelling

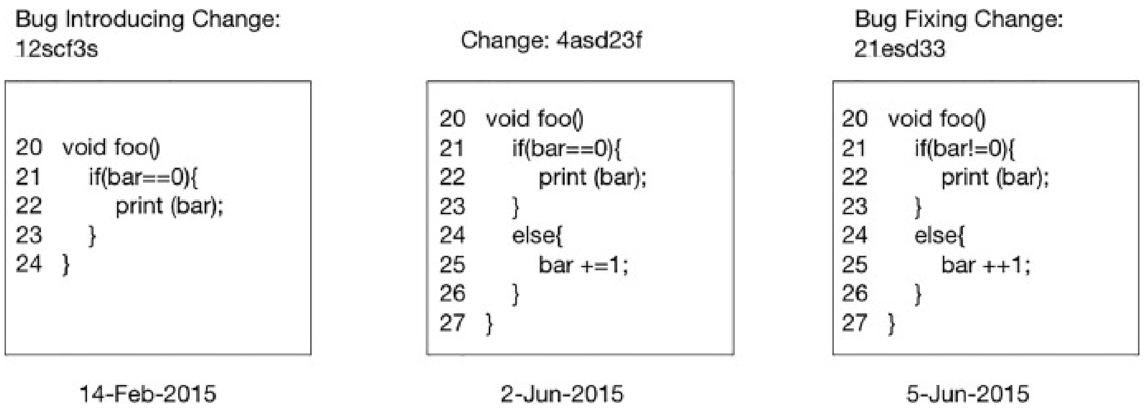
Due to the nature of the Git and Jira systems, the data was unlabeled. The challenge is to identify a defective instance of a file or a commit. Manual labeling of our data was not feasible due to the number of data points. Two approaches are used to label the data. The first approach takes inspiration from Daniel Alencar da Costa et al. [11], where the SZZ framework is used to identify corrective commits based on previous commits, to indicate defective files. The method used to do this, is from the Pydriller API. A *getModifiedLines* function implements the SZZ algorithm [11] to retrieve the set of commits that last influenced the lines of the current commits file. For every file in the commit, we apply the SZZ algorithm to obtain the diff, blame the file and obtain the previous commits were those lines were added and label them with the underlying file as defective. This approach labels a version of the file as "clean" and traces back to previous versions of that file to label them as defective. Although all previous instances of the corrected file should not be labelled as defective, but only the files that have the changes that are likely to introduce the associated bug.

The second approach takes inspiration from the literature survey [27], where keyword matching is used to identify corrective commits that indicate fixed versions of files. After some investigation and analysis of the data, the full keyword list: *fix*; *bug*; *fixing* ; *Buggy*. The keyword list search was applied on the Git comments data and Jira issues data. Both data sources (Git and Jira), were combined to provide adequate information about whether the file was fixed or not, although not all commits had accompanying Jira issues. We used regular expressions to detect keywords, as this will capture more words without the need for exact matching or stemming the words. The matching was not case sensitive.

An example of data labelling is illustrated in figure 3.1 [8], where we have three versions of the same file, due to three changes (commits). The first change introduced a bug into the file that was not caught until the third change (right-most code snippet). We know that the third commit is a corrective one since it has keywords "Bug" and "Fixing". Therefore, we apply the SZZ algorithm to detect which file introduced that bug. Since this bug-fixing commit edits line "21" in the code, then this is assumed to be the line causing the defect. SZZ can trace back and detect which commit introduced or last "touched" that line, which is in this case, the first commit (change), associated with the first version of the file, so we label that version as defective and the following two versions are clean by default. This process is repeated to label all file instances.

3.2.5 Data Loading and Staging

As mentioned before, the raw data resides in the Git repositories. In order to make our solution and flow of data more robust, we load data into Comma Separated Value (CSV) files. Each CSV file holds data for one repository. These files serve as a staging phase, so if the raw data is needed, we can track back to the CSV files



```
void foo()
    if(bar==0){
        print (bar);
    }
}
```

```
void foo()
    if(bar==0){
        print (bar);
    }
    else{
        bar +=1;
    }
}
```

```
void foo()
    if(bar!=0){
        print (bar);
    }
    else{
        bar ++1;
    }
}
```

Figure 3.1: Data Labeling through SZZ and Keyword Matching [8]

since they have more data than we need. It is also efficient to save the data onto the disk as reading directly from the repositories takes a very long time (up to 28 hours), which is expected to increase as the repositories grow in size.

3.3 Building the models

After data preparation we pre-processed the data to make it compatible with the learning model. The aim is to normalize all numerical metrics and encode string metrics as numbers.

3.3.1 Pre-processing Data

As most of our metrics were numeric, we scaled them using the MinMax[1] normalizer from the SKlearn[2] library [9]. We chose this scalar because it was the most robust. Scaling was done to make all metrics (features) equally important so that features with higher numbers do not dominate other features while training the learning model. All numbers after this step were scaled down to a range between "0" and "1".

Next, we encoded the string metrics like "Authors", as categorical variables. Machine learning model accept only numbers to train the data, hence we apply this transformation. To do that we used one-hot vector encoding which means that each value in a string-type metric is pivoted so it has its own metric column. This process is the reason our metrics have increased to 208 metrics instead of 14. The same technique is applied to "Committers" and "FileType" metrics.

Boolean metrics like "Refactor", "Merge" and "Modify" did not need to be encoded as categorical variables since their values of "True/False" are implicitly transformed to "1/0" in while fitting the learning model.

[1]An function from the SKlearn library which transforms features by scaling each feature to a given range

[2]The scikit-learn library for machine learning in python.

3.3.2 Types of Models Used

After preparing the data, we split it into training and testing data (20%) and run RandomGridSearch[3] function introduced by [7], which is also implementated in the Sklearn library [9] to tune the hyper-parameters of our models. We use 10-fold cross-validation for training. As aforementioned, we use the CE AUC here as the function to optimize the model on. We then get the best estimator from the RandomGridSearch and use it as the output model. We used the default number of iterations for RandomGridSearch, that is 10 iterations. This process was repeated for the four classifiers that we used:

- Logistic Regression
- Naive Bayes
- Random Forest
- Multi-Layer Perceptron

It is worth mentioning that RandomGridSearch is not deterministic; it yields different results every time, though not far apart from each other. This is due to RandomGridSearch's innate algorithm to find a local optimum in a timely manner. We decided to use it instead of using the other alternative, GridSearch, because it gives better results while the latter takes longer to converge to the global optimum. In applying the NB model, RandomGridSearch could not be used as it has no hyper-parameters and can not be optimized. Here the default parameters are used.

3.4 Evaluation

3.4.1 Model of Choice

We chose the model with the highest AUC since this is our evaluation criterion that defines the best model as it serves well the objective of our thesis work at Aptiv.AUC is also the best evaluation measure when dealing with a unbalanced data set[24].
We configured our program to re-train the four models we selected periodically. Each time the models are trained, they will be measured against each other and the model with the highest AUC is selected and stored for that period to be the model of choice, until the next re-training phase. For instance if for this Quarter of the year, the best performing model was the MLP, that doesn't not mean that it will use it the next time we perform a re-training process.
We chose this evaluation criterion because it fits the purpose of the thesis towards helping Aptiv in flagging commits (files) that are likely to be error prone in a time efficient manner. When we optimized the hyper-parameters of the model to achieve, for instance, the highest precision, the accuracy of the model would drop, and vice versa. It was hard to develop a model that kept all scoring criteria balanced and high at the same time. Finally, we apply Correlation-based Feature Selection (CFS) to each final model in order to check if the model can be reduced to avoid overfitting.

[3]A function that automatically tunes the hyper-parameters of a machine learning model and returns the best estimator that yields the best scoring function.

3.4.2 Limitations

Data labeling, an important step in our methodology, depends on the quality of documentation and commit messages that developers make in their daily development activities. Due to the large mismatch of linked Jira issue to Git commits, there are only 8,921 Git commits matched to Jira issues out a possible 23,926; therefore we could not be fully reliant on labelling on keywords and used a SZZ approach as well to compensate. Acquiring of metrics was a challenge since the Git tool can not provide all possible metrics that we intended to use. Metric collection steps were continuously revised and applied due to their availability.

3.5 Integration into Production

We deployed our working model to the production environment at Aptiv, this helps them to preemptively focus their testing efforts on files that are likely to be "defective", based on our model predictions. The best performing model was serialized to an external file. When source code is committed and thereafter build files and unit tests are executed, a customized script is triggered to run the prediction model. We use this model to predict both the class of the file (defective or clean) as well as the confidence of our prediction (probability estimate). For example, each file in a commit will have its name and timestamp as a unique identifier; and the class and confidence as the output values. Finally, these results are projected in a web interfaces (Gerrit), which alerts the developers of the flagged files to revise.

4

Literature Survey

4.1 Data Gathering and Preparation

Most studies in literature implemented data collection tasks through extracting file-level change data from SCM repositories using the appropriate scripts. The aim of this metadata is to provide all historical versions of all files in order to track the changes and the propagation of defects with time and detect events that fixed previous bugs and the relevant files associated with those events. The complementary tool used in most studies are issue trackers to log all the bugs associated with the file-changes in SCMs.

Rahman et al. [28], conducted defect prediction studies at the file-level as they extracted process metrics from a sample of 12 Java-based projects by using the commit history from their Git repositories. The average number of releases per project was 7 and the total number of files across all projects amounted to 27,912 files. The command *git blame* was used on every file at each release to get the detailed information. More data was extracted from the issue tracking system, Jira, to find more details about defects and fixes commited. Where features like changed lines and author were extracted from those commits in the Git repository, in an attempt to find interesting patterns and correlations between features, fixes and bugs. Rahman et al. stated that "Any files modified in these defect-fixing commits are considered as defective". Code metrics were divided between the file-level and class level. Since all projects are Java-based, most files contain a single class; so class level metrics were aggregated to file level. Similarly some metrics were available only at method level, (such as fan-in, fan-out), which were also aggregated at file level.

Arisholm et al. [3] stated that OO (or code) metrics can be collected from code snapshots. Delta metrics need various versions of the system to be available (in order to calculate the differences); while Process metrics are for instance, changes developers did, the descriptions of commit messages , and fault corrections, developer information and time of changes. In this study Arisholm et al. used a tool called JHawk to extract OO and Delta metrics, while process metrics were extracted from the MKS (Configuration Management System). The dataset had a size of 22 releases and over 2,600 Java classes but only the most recent 13 releases were used in their study.

Zimmermann et al. [39] used Concurrent Versions System (CVS) as their SCM system along with the bug tracking system Bugzilla to identify corrective commits by searching for keywords in commit messages such as "Fixed" or "Bug #<number>", where each bug number is linked to a bug report in the bug tracking system. The idea is that bug reports have a *version* field which holds the release in which the bug was first reported. Defects were predicted on both file and package levels. This study included 3 releases, with a total of of 25,210 files.

Inspired from the Eclipse project in [39], Moser et al. [25] used a public dataset to predict defects on a file-level. This dataset included a large number of static code metrics (198 attributes) and pre- and post-release defects for the Eclipse releases 2.0, 2.1, and 3.0 extracted from the PROMISE[1] repository. These releases contained a total of 14,539 files.

Berger et al. [27], performed a case study on a project named BusyBox. They extracted the code history and change data from a Git repository having over 13 releases. The dataset attributes included a total of 3,860 commits, performed by 244 different authors, contributing to the development of 821 unique features across all releases. The analysis was performed on a feature level instead of files, as in the aforementioned studies. The dataset was in the form of vectors, meaning that for each feature in a specific release, it was associated with its feature name, the release number, values of five process metrics aggregated over all commits associated to the release, and the classification as *defective* or *clean*. Each commit contained details of the file changes which were extracted including attributes like: lines of code, author, and commit message. Commit messages were then linked to the corresponding features to construct the data model. While implementing this, they also mention that "If a commit contained code changes within conditional compilation directives that belong to one or more features, then they attributed the commit to each feature. This contributes to establishing relations between the authors of the commits and the feature, and to prepare for the labeling process." [27]

The study in [24] was made on seven features (applications) on six Android releases. The total number of classes in this study was 9,190 (an average of 250 classes per feature per release). Open-source repositories were used to extract the source code with OO metrics using Chidamber and Kemerer Java Metrics (CKJM) tool. Defect Collection and Reporting System (DCRS) tool was used for collection of the defects. Defects were made on the class-level in this study.

Based on the study in [13], Aversano et al. [5] used the time-window heuristic to extract data from two open-source systems; JHotDraw (489 files) and DNS-Java (179 files) while ignoring changes that involved more than 30 source code files in order to exclude large CVS maintenance activities. This CVS system allowed for extracting detailed changes on the file level and even on lower levels such as detecting where each line of any file to was introduced for the first time (release number). Accordingly, this system has the ability to log administrative data for each line in any file such as: date, time , the author (developer) and the release number. Thus,

[1]http://promise.site.uottawa.ca/SERepository/

time sequence analysis is possible since *time* attributes are present, which can be combined with other attributes.

Karim et al. in [12] used different levels of granularity, that is, they performed their study on the basis of modules across four projects; two were written in procedural languages and the other two in object oriented languages. The total number of modules was 4,168.

4.2 Metrics

Paper References	**Code Metrics**	**Process Metrics**
[24]	18	
[12]	21	
[39]	14	
[28]		14
[3]	18	26
[27]		5
[25]	31	19

Table 4.1: Types and Number of Features used in Previous Studies

Previous studies in [28, 25, 27] concluded that process metrics are more effective for predicting defects on the file-level, although the latest paper argues that it may not be known how superior they are when it comes to predicting defective features.

Rahman et al. [28] evaluated four combinations to compare process and code metrics for building the learning models. One: using only process metrics, two: using code metrics, three: combining process metrics and size; since size alone is an important metric for defect prediction and to also separate the impact of size from the whole combination of process and code metrics, four: building the model with the entire collection of metrics. The log transformation of metrics significantly improved prediction performance. They used 14 process metrics across a large number of releases of diverse projects on different types of models. The conclusion stated that process metrics were superior to code metrics when evaluated using CE AUC in terms of stability, portability and performance. Moreover, they stated that code metrics may not evolve with the changing distribution of defects which leads to *stagnation* which causes the model to focus on files which are recurringly defective.

In the Android project studied in [24], only code metrics (18 metrics) were extracted and filtered based on the method in [15], that is, CFS which selects relevant attributes (features) to the dependent variable and removes variables that are highly correlated with each other (redundant). The LOC, CAM, WMC, Ce and LCOM3 metrics were found to be significant predictors over the various releases of seven application packages of the Android software using the CFS method.

To compute their 14 code metrics, Zimmermann et al. [39] used the Java parser for Eclipse. They developed algorithms to compute standard code metrics for methods, classes and files. The metrics were aggregated into single values (for files and packages) as maximum, total and average. They concluded that code complexity is proportional to defects found and stated that the results were far from perfect and raised questions like are there any metrics that are better than complexity metrics and whether these other methods can be carried out to predict defects across projects.

The work in Arisholm [3], made a comprehensive study on different kinds of metrics including 18 OO metrics, 25 process metrics and one delta metric (which evaluates code churn between files in successive releases). To assess the significance of these variables they made seven combinations, grouping 2 types of variables at a time and finally using all variables to build their learning models. They also used the full set of metrics and a reduced set using CFS for building models. Surprisingly, models built using the reduced set of metrics were marginally poorer than the complete metric set across most evaluation criteria. Similar to the conclusion of [28], there is little benefit by including code metrics to a model that is using process metrics. In some evaluation methods like CE, the values are almost zero and some are negative, meaning that some models using code metrics perform worse than the baseline models. Arisholm et al. also noted that adding the OO metrics consistently degrades the CE of a model. Further, although the deltas have the smallest average ROC area, these metrics are consistently more cost-effective than the OO metrics. The low CE of the OO metrics may be due to their correlation with size measures, which has been reported previous studies. On the other hand there is so much gain when using a model based on process metrics, the best model is approximately 50% of the optimal model in terms of CE. When it comes to ROC area OO metrics are great to use.

In Karim's paper of Predicting Software models using SVMs, [12], each dataset used began with 21 static metrics of independent variables and a dependant boolean variable that indicated whether or not the module is defective or not. As it is likely that the 21 variables may be correlated, [12], CFS as also done in [3] and [15], was also applied to narrow down the best combination of metrics. Each dataset was left with the best subset of independent variable metrics, suggesting that the reduced metric subset performed better than the complete set of metrics.

Moser et al. [25] studied the impact of adding 18 process metrics both alone and in combination with the 31 code metrics used in [39]. Again process metrics outperformed code metrics. The findings in [39] agree with studies, like [28], in that change data, and more in general process-related metrics, are better at describing and capturing more meaningful information about the defect distribution than the source code itself (code metrics). They provide an explanation that while complexity metrics are related with the cognitive effort for understanding the source code they are not necessarily sound indicators for software defects. They state the following example that summarizes why process metrics are preferable: "A source file may be very complex and still defect free, because the developer who coded it was very

skilled and did a prudent job. However, a prediction model based on complexity metrics would classify it as defective. On the other hand, if a file is involved in many changes throughout its life cycle there is a high probability that at least one of those changes introduces a defect, regardless of its complexity" [25]. That does not mean however that there is no correlation between code complexity and defects.
Berger et al. [27] agreed with Rahman et al. [28] and used 5 process metrics as they performed better than code metrics at the file-level, yet they stated that it is not clear whether that holds for the feature level.

4.3 Learning Models

A variety of machine learning models are used in fault-detection. In combination with the environment, problem, metrics used and evaluation criteria, the performance of the same learning models differ in each case. This section aims to evaluate each paper and the machine learning models used in respect to their context. Below is a list of learning models evaluated in the literature.

Study	Statistical classifiers	Neural Network	SVM-based	Decision Tree Methods	Ensemble Learning	Rule-based Learning	Other
[24]	LR, NB, BN	MLP, RBF	SVM, VP	CART, J48, ADT	Bag, RF, LMT, LB, AdaBoost	NNge, DTNB	VFI
[12]	LR, NB, BBN	MLP, RBF	SVM	DT	RF	KNN	
[39]	LR						
[28]	LR, NB		SVM	J48			
[3]	LR	NN	SVM	C4.5	AdaBoost, Decorate	PART	
[27]	NB			J48	RF		
[25]	LR, NB		J48				

Table 4.2: ML Models from Previous Studies

In [39], LR was selected to classify the Eclipse bug dataset. The purpose of the study was to predict which file or package contained the post-release defects. Here the choice of LR, which predicts the likelihood estimate between 0 and 1, is a good fit. If the likelihood estimate was above 0.5, the file or package was classified as defect prone, as the estimate was close to the value of 1 [39]. Any estimate below 0.5 and close to the value 0, resulted in the file being flagged as defect free [39]. Six models were built to be used on both a file and package level. Models were tested across all the releases, and on the related file or package level they were built on [39]. As only a LR model was used, there is no comparison to be made in regards to it's performance. For this we look at the work of [25] who builds on [39], who contrasts the existing LR model used with Naive Bayes NB and a Decision Tree (DT) (J48). The results show that the DT outperforms the two other models for each release tested. Another finding is that NB is a strong performer in terms of code metrics, [25]

Karim's [12], objective is to analyze the efficiency of the SVM model in defect-prone software over four datasets, therefore SVM is compared to statistical classifiers (K-Nearest Neighbour (KNN) & LR), NN (MLP & Radial Basis Function (RBF)), Bayesian techniques (Bayesian Belief Networks & Naive Bayes) and tree-structure models (Random Forests & Decision Trees).
In terms of the evaluation criteria used (accuracy, recall, precision, F-measure) the overall results from the four datasets and the eight different models, show that SVM had a higher accuracy than most although, MLP achieves a significantly higher accuracy when used in a dataset (KC1) [12] that is object oriented and in C++.
Top performing learning models in terms of accuracy mean values across all data sets are: SVM, LR, MLP, RBF, Decision Tree (DT) and RF. In precision mean values, models such as Bayesian Belief Network (BBN), NB, KNN and RF are overall top achievers. SVM only does marginally better than 3 models over 2 datasets. Making SVM a weak choice in precision-driven results. When comparing recall, SVM all models tested against all the datasets, and shows a significant difference against 4 other models [12]. SVM clearly is a strong learning model where recall is important. Lastly F-measure scores were contrasted and it was found that they were not largely outperformed by any model in all datasets used [12]. SVM results in a higher F-measure score than at least 5 other models across all datasets [12]. We can conclude that in terms of recall, the SVM model is an excellent choice as it has scores within 99.4–100%. It also ranks third in terms of F-measure scores overall. Although it does not have such high accuracy and precision scores, it is still a good practical choice given its performance over the datasets in comparison to the other 8 models. Other noteworthy models are that gave an overall high performance in all the evaluation metrics are LR, MLP and RBF.

Three different classifiers are explored in [27], these are DTs, RFs and NB. The DT chosen was J48, which is an implementation of the C4.5 algorithm used in [3]. The accuracy of all three classifiers were similar in score and it was discovered that imbalanced data has a impact on the accuracy score [27]. After analyzing the confusion matrix, NB was the top performer in regards to overall accuracy while the decision tress classifier proved the best for predicting defective features in terms of the F-measure and ROC [27].

Arisholm et al. [3] contrasted and compared 5 different machine learning techniques to understand which one was the best performing in their given context. Their approach in choosing the classifiers varied. A DT classifier (C4.5), was picked as it is one of the most popular in this context [3]. Other classifiers chosen were a coverage rule algorithm (PART) due to its performance at the time of writing, backpropogation NN classification, SVM and LR, which is used as a statistical standard for comparison [3]. The DT classifier (C4.5) is improved by applying Adaboost [37] and Decorate to help improve its performance on the training sets [3]. Another decision made to help evaluate the classifiers was to use the rule or leaf with the lowest entropy for the fault probability distribution for C4.5 and PART as rules and leaves can be compared on the same level [3]. The overall top performing machine learning technique was C4.5 combined with Adaboost.

In [24], the study uses 18 machine learning techniques which cover NN, SVM, ensemble learners and DTs [24]. The full list is LR, NB, Bayesian Networks, MLP, RBF, SVM, Voting Perceptron (VP) , CART, Alternate Decision Trees and J48 DTs, Bagging, RF, AdaBoost, LoGit Boost, Logistic Model Trees (Ensemble Learning), Nearest neighbour with generalized examples, Decision table Naive Bayes (Rule based learning), Voting Feature Intervals (VFI). After testing each learning model it was the NN, MLP that performed best, followed by NB and LR. The worst performers were SVM, VP, CART and J48. MLP and LR being the top performers echoes the results of [12]

NB and LR were also top performers in Rahman's paper [28]. They use LR, J48, SVM, and NB, on different combinations of process and code metrics. The result was that while NB was better performing on code metrics, also seen in [25], it is LR that is best performing on all metric combinations and produces the same p-values as NB in regards to its results in code metrics [28].

We find that LR is used in all the papers and can be a good statistical standard to compare machine learning models. It is also frequently a top performer from the reviewed literature. NB is also a model worth considering based on this analysis. It is used 70% of the time as a selected model from the reviewed papers and also has a good performance rate in [24, 28, 27]. The last two machine learning models note mentioning are: DT and MLP. Both have shown good performances in [27, 25, 3] and [12, 24] respectively.

4.4 Evaluation Criteria

Choosing the appropriate evaluation criteria plays a vital role in determining the best model [3]. The evaluation criteria chosen depends on the type of problem being solved, [3].

Confusion matrix measures for instance, help quantify the comparison of predication models, [12], but do little in terms of analyzing the CE of the models in the context of validation and verification [3]. Karim, Zimmermann, Moser and Arisholm [12, 39, 3, 25] use these metrics in predicting the performance measures of the models used. It is noted that a shortfall in using the measures from confusion matrices, is that all of them require a predefined cut-off value for the predicated probabilities, [3].

In [3], the accuracy measure comparison showed that SVM and LR yield the highest values, although the accuracy alters depending on the metrics chosen [3]. It is also noted in the paper that accuracy may not be the best way to evaluate how efficient fault type prediction models are [3]. Recall and Precision measures showed that the

Boost C4.5 modeling technique was the best performer in paper [3]. The findings also show that some methods (tree and rule based models) give lower precision and higher recall than others (SVM, NN and LR) [3]. Type I and II misclassification rates are used in [3], it was found that both types were inversely correlated and decision trees or rule-based techniques (C4.5 with or without boosting) give better prediction models in terms of the type II misclassification rates [3]. Another evaluation measure used by Arisholm, [3] is the ROC curve.

AUC serves as a good measure when handling unbalanced and noisy datasets as it is insensitive to changes in the distribution class and therefore was suitable to use in the study by Malhotra [24].

CE of a model can not be measured using the above methods. CE can measure how well different parts of the model rank and perform, as seen in [3]. Arisholm et al. [3], calculate CE by calculating a normalizing it over a baseline, optimal and model CE values. Cost effectiveness can be defined differently depending on what needs to be evaluated. In [25], a unique cost effective function is defined to measure the costs associated with different prediction errors made by a model [25].

In the paper [28], the metrics of accuracy, precision, recall, F-measure at 0.5 threshold, ROC and CE were examined [28]. Rahman et al. mentions the use of Source Lines of Code (SLOC) to measure the CE by "plotting the proportion of defects against proportion of SLOC coming from the ordered (using predicted defect proneness) set of files" [28]. Although SLOC and ROC are seen as similar measures, SLOC uses a smaller area under the CE curve (AUCEC), fit for the resource limits [28]. Raham uses AUC, CE AUC at 10% (CE AUC 10) and 20% (CE AUC 20) SLOC for his evaluation criteria for each model's performance [28].

The quality of ranking of defects can be measured using the *Spearman Correleation*, as used in the papers by Rahman and Zimmermann [28, 39]. In the paper by Zimmermann, the Spearman Correlation was calculated between the pre- and post-release defects and their metrics used. It was found that a directly proportional relationship exists between the number of defects in the pre- and post-release files [39]. The Spearman correlation coefficient is also able to indicate which group of metrics has the most influence on defective files on both the file and package level [39]. The *Pearson correlation coefficient* is also calculated in the paper by Zimmerman [39]. This measure assumes a linear relationship between the metrics and is calculated to ensure completeness when comparing against Spearman [39].

4.5 Validity

Validation is performed across the results from the evaluation criteria by some of the papers. The main validity checks and tests used were cross validation, inter-release validation and significant tests.

Kairm, [12] , uses *cross validation* on each dataset used by the models. A 10-fold cross validation is used to split the data in 10 equal bins. 9 bins are used to train the model and the last bin is used as a test bin. This is done for a total of 10 times, rotating the test bin in each round [12]. To ensure a low bias and validity of the test trials,the trials are run 100 times and shuffled as well as a randomized seed value is used [12]. The mean and standard deviation are then calculated for each evaluation criteria measure used over the 100 test runs [12]. In the papers by [24, 25],10- fold cross validation is also applied in order to validate their results from several machine learning techniques. This seems a popular tool in performing validation on unbalanced data.

Malholtra also performed an *inter-release validation* [24]. This is when defined releases are used to train the model and it is then validated against its latest release [24]. Statistical tests such as Friedman test and a post-hoc Nemenyi test was also carried out in this study [24], to analyze how different the results from the different models are from one another. The Friedman test calculates and ranks the mean performance values based on the AUC values [24]. The lowest ranking value indicates the best performing model that was used. If the Friedman test showed significant results, a post-hoc Nemenyi test was done to compare the pairwise differences from the different models, based on AUC values [24].

Another way in which a *Significance test* was carried out, is illustrated by Karim [12], to establish if there was a meaningful difference between the results of the SVM model used and other predication models [12]. A corrected re-sampled t-test was used as it is better suited when used with a x-fold cross validation as used by Karim [12]. The t-test was run at a significance level 0.05 (95% confidence level). By performing these tests, Karim was easily able to display which model outperformed SVM and vice versa.

Rahman used Wilcoxon tests and the corrected the p-values using BH correction in their study to determine if there was a significant difference of code vs process metrics [28]. Arisholm analysed the p-values from Wilcoxon tests to compare the different models [3] and set the significant level for this test to 0.001.

5 Results

By following the processes outlined in chapter 3, we obtained the metrics we aimed to use as our machine learning features (predictors). The models trained had similar results in terms of CE AUC with varying statistics for the f1-score, accuracy, precision and recall.

5.1 Data Set

Three main repositories were identified to have the most commits and its codebase to be similar to each other. The three main repositories each hold 908, 231, 720 commits respectively (at the time of collection). As the metrics are based on a file level, the commits were extracted per file for each commit.
As the metrics are based on a file level, the commits were extracted per file for each commit.

Repositories	**Total Commits**	**Commits on a File level**
Repository 1	908	15,653
Repository 2	231	5,176
Repository 3	720	3,097

Table 5.1: Data points per Repository

These commits from the different repositories were merged to form a complete dataset.

5.2 Metric Results

The metrics chosen are process metrics, which proved superior to other metrics based on previous studies as in [28]. We also created our own metrics that we thought were best for prediction. We also included a few code metrics. Defining our metrics became a iterative process in the beginning due to the availability of data. We established in the beginning that our metrics will be on the file level and per release. Table 1 shows the resulting metrics we selected and calculated from our raw data. We created initially 14 metrics, which later expanded to 209 metrics. This large number is due using one hot vectors for encoding string metrics such as "Author" or "FileType" as categorical dummy variables.

The following metrics are per file instance aggregated per release. A release is launched every two weeks. The value *Calculated* in the *Source* column indicates new metrics introduced by this thesis.

Metric	**Description**	**Source**
Distinct Developers Count	Number of distinct developers that worked on the file per release	[28]
Active Developers Count	Number of developers in current release (not distinct)	[28]
Commit Count	Number of times a file was committed within a release.	[28]
Added Lines	Number of total lines added to the file.	[3]
Deleted Lines	Number of total lines deleted from the file.	[3]
Cyclomatic Complexity	The complexity of the code in the file; a measure of the number of linearly independent paths through a program's source code.	[3](from API)
Merge	A boolean value indicating whether the code was merged with another file or not due to conflicts when committing codes by different Authors.	Calculated
Refactor	Whether the change was a refactoring change for the code.	Calculated
Modify	A boolean value to indicate whether the lines of code where modified instead of merely adding or deleting. Modifications can be just extending one line of code or renaming variables for example.	Calculated
Author Experience	The number of files the Author has changed (across all projects).	[3]
Committer Experience	The number of commits done by the committer (across all projects)	Calculated
Committer *	The name of the Committer	Calculated
Author **	The name of the Author	Calculated
File Type	The file extension of the file (or its type e.g. .java or .html).	Calculated

* *The committer is assumed to be the person who committed the code on behalf of the original author.*
** *The author is the person who originally wrote the code.*

Table 5.2: Metric Collection

After aggregation we have around 10K data points from 3 repositories. The Jira data had to be refined further as some of the selected data that matched our metrics had missing and incomplete data. This data included fields such as priority, votes and time to complete task. These metrics will later be reduced when using feature selection in some of the machine learning models.

5.3 Models

We developed the models and optimized them as outlined in chapter 3, using the metrics we collected in section 5.2. Following are the results and evaluations of each model based on our main criteria, CE, and the other conventional confusion matrix

Hyper-Parameter	**Value**
C	1000
class_weight	None
dual	False
fit_intercept	False
intercept_scaling	1
max_iter	100
multi_class	ovr
n_jobs	None
penalty	l1
random_state	None
solver	liblinear
tol	0.001
verbose	0
warm_start	False

Table 5.3: Hyper-parameters for Optimized LR Model

measures. Tables 5.3, 5.4 and 5.5 show the resulting hyper-parameters for each of the optimized models.

Hyper-Parameter	**Value**
activation	relu
alpha	0.0001
batch_size	auto
beta_1	0.9
beta_2	0.999
early_stopping	False
epsilon	1.00E-08
hidden_layer_sizes	(50 50 50)
learning_rate	adaptive
learning_rate_init	0.001
max_iter	300
momentum	0.9
nesterovs_momentum	True
power_t	0.5
random_state	None
shuffle	True
solver	adam
tol	0.1
validation_fraction	0.1
verbose	False
warm_start	False

Table 5.4: Hyper-parameters for Optimized MLP Model

Hyper-Parameter	**Value**
bootstrap	False
class_weight	None
criterion	gini
max_depth	100
max_features	log2
max_leaf_nodes	None
min_impurity_decrease	0
min_impurity_split	None
min_samples_leaf	2
min_samples_split	5
min_weight_fraction_leaf	0
n_estimators	400
n_jobs	None
oob_score	False
random_state	None
verbose	0
warm_start	False

Table 5.5: Hyper-parameters for Optimized RF Model

5.3.1 Cost Effectiveness

Figures 5.1 to 5.4 show the evaluation method used for each of the four models, that is, the CE curve which expresses the performance of our models. It depicts how fast a model can find defective files by looking at the most likely ones to be defective first. See section 2.6.2.
It can be seen that the NB classifier has the worst performance of the four learning models; with an AUC close to that of the baseline model. On the other hand LR performs fairly well, but is outperformed by the Neural Network MLP model. Finally is the RF model which had the best performance of them all, with an AUC closest to the optimal model. The statistics of each model are shown in table 5.6. We refer to the baseline and optimal models in the table by their AUCs as baseline AUC or Baseline Area Under the Curve (BAUC) and optimal AUC Optimal Area Under the Curve (OAUC).
For all classifiers, the full model was the best performing; when we tried to use less features for each model, the CE AUC degraded.

	AUC	**Diff from BAUC**	**Gain over BAUC**	**Ratio over OAUC**
NB	5,778	1,090	23%	61%
LR	7,590	2,901	62%	80%
MLP	8,090	3,401	73%	85%
RF	8,250	3,562	76%	87%

Table 5.6: AUC values for all four models compared to baseline and optimal models' AUCs which respectively are 4,688 and 9500.

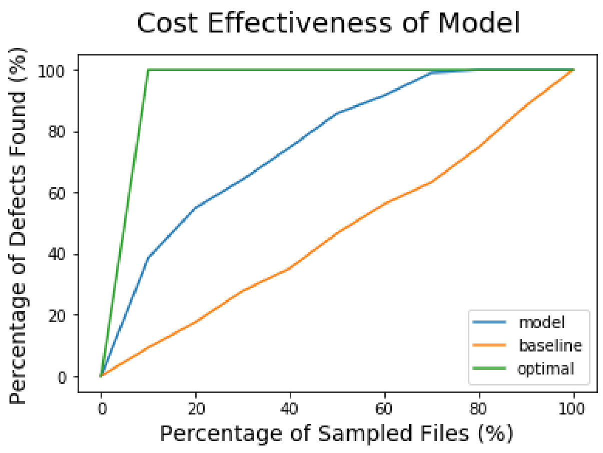


Figure 5.1: CE Curve for LR

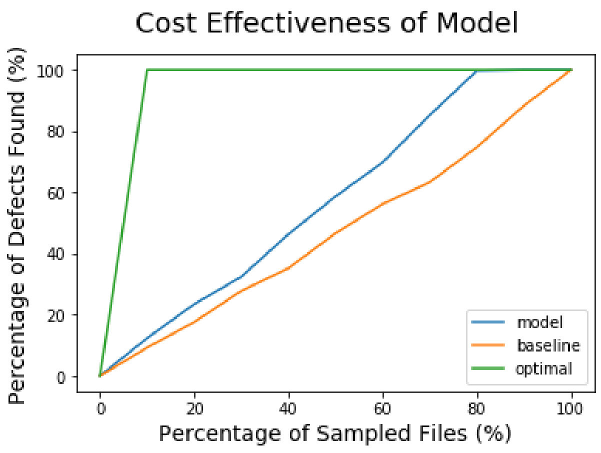


Figure 5.2: CE Curve for NB

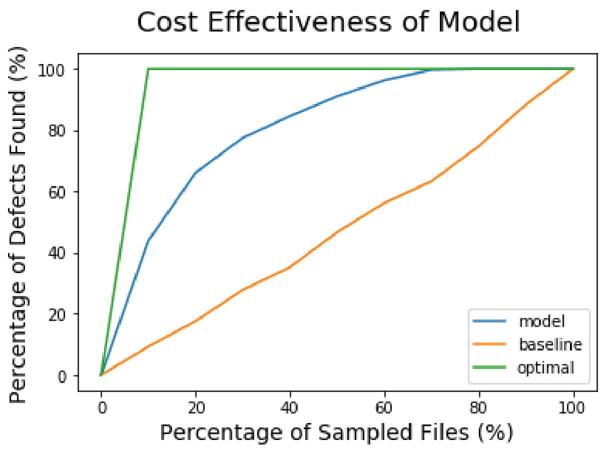


Figure 5.3: CE Curve for MLP

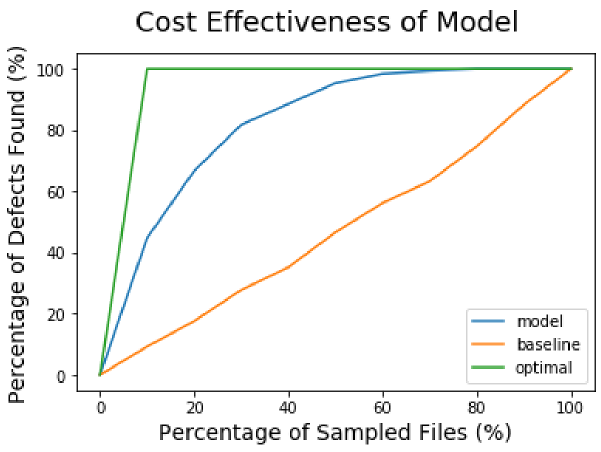


Figure 5.4: CE Curve for RF

5.3.2 Confusion Matrix Measures

The reason for optimizing the model using CE was based on the project needs, nonetheless we also attempted to take the other confusion matrix statistics such as accuracy, precision, recall and f1-score into account. The main problem when optimizing the model's hyper-parameters; optimizing the accuracy would sacrifice another measure such as recall. It was almost impossible to obtain a model of balanced performances on all these aspects. This made our choice for CE seem even more sensible.

	Precision	**Recall**	**F1-score**	**Support**
False	1.00	0.25	0.40	1776
True	0.19	1.00	0.33	324
micro avg	0.36	0.36	0.36	2100
macro avg	0.60	0.62	0.36	2100
weighted avg	0.87	0.36	0.38	2100
Accuracy: 36%				

Table 5.7: Statistical measures for NB

	Precision	**Recall**	**F1-score**	**Support**
False	0.89	0.97	0.92	1776
True	0.64	0.32	0.42	324
micro avg	0.87	0.87	0.87	2100
macro avg	0.76	0.64	0.67	2100
weighted avg	0.85	0.87	0.85	2100
Accuracy: 87%				

Table 5.8: Statistical measures for LR

	Precision	**Recall**	**F1-score**	**Support**
False	0.90	0.97	0.93	1776
True	0.69	0.38	0.49	324
micro avg	0.88	0.88	0.88	2100
macro avg	0.79	0.67	0.71	2100
weighted avg	0.86	0.88	0.86	2100
Accuracy: 88%				

Table 5.9: Statistical measures for MLP

	Precision	**Recall**	**F1-score**	**Support**
False	0.87	1.00	0.93	1776
True	0.98	0.15	0.27	324
micro avg	0.87	0.87	0.87	2100
macro avg	0.92	0.58	0.60	2100
weighted avg	0.88	0.87	0.83	2100
Accuracy: 87%				

Table 5.10: Statistical measures for RF

As shown throughout tables 4.2 to 4.5, although the RF model has the highest AUC, it has the lowest recall and f1-score values of the four models. It can also be noticed that it has the highest precision value. The NB model is poor in all aspects, while the other 3 are comparable in terms of CE and accuracy measures.

6 Discussion

All four models used in this thesis were also used in [12, 24]. It is interesting to see that in Malhotra [24], MLP was the best performing ML model, followed by NB and LR. Comparing this to our results we find both MLP and LR at the top, but not NB. NB performs well when its attributes are independent and unrelated, as shown in [24]. Although when we built the models with feature selection, the reduced set of metrics were marginally poorer than the complete metric set across most evaluation criteria, this result correlates with that of Arisholm [3], who used a smaller set of features than us. Perhaps if we had more features, the feature selection process would have been more impactful and reduced set of features would have had a low correlation with each other and perhaps lead to a better performance of the NB model. Another factor that possibly influenced the NB performance is the choice in metrics. Choosing code metrics as in [24, 25, 28] lead to the NB model being a top performer.

We also observed that the evaluation technique can compliment the type of metric chosen. In Rahman [28] a finding was that process metrics were superior to code metrics when evaluated using CE AUC in terms of stability, portability and performance, which can correlate to the low CE scores in the NB model. A contrasting result to our NB model was in the study [27]. Here NB was the top performer with process metrics being selected. The methodology in [27] is similar to the one used in this thesis. It is interesting to try see why NB is a top performer here. One main difference is that the data used in [27] is slightly more balanced than our own dataset. Another reason could be that we are using different evaluation criteria to measure the models.

Using a LR model was a calculated decision on our part as we observed it was a standard ML model used in almost all studies and previous work we examined. From the findings through our literature survey in [24, 12, 39] we found the LR as a good performer but generally does not outperform other models in terms of accuracy and AUC scores. Similarly in our own results we see a correlation of the performance of LR. The respectable results of the LR could hint that the type of ML model selected may not be of too much importance and the results could largely depend of the metrics selected.

MLP was the second best performer. We expected MLP to be a top performer from the the findings in [24, 12, 20]. There is a very small marginal difference between the performance of the RF model and MLP. Another observation of our results to that of [24], show RF as a good choice for defect prediction on various types of

	RF		**LR**	
#	**Feature Name**	**Coefficient**	**Feature Name**	**Coefficient**
0	Added Lines	0.18	Added Lines	42.8
1	Cyclomatic Complexity	0.11	Author_3979a0e268d70f98d642b20	8.32
2	Deleted Lines	0.06	Author_f4ba57e3501b0f2fa9bb945	5.32
3	Commiter Experience	0.04	Active Developers Count	4.23
4	Author Experience	0.04	AddedLines_count	3.79
5	FileType_java	0.04	Commiter_472e6c8e1b90776044192	3.35
6	Refactor	0.03	Author_f3dea260bc67ed18adf64b0	2.53
7	Modify	0.02	Commiter_c2329d6acdff7139f97eb	2.53
8	Author_968f949c41383aa08d8cadd	0.02	Author_e6e18d41a6e318fd1024ca6	2.46
9	Commmit Count	0.02	Commiter_8f36e4c3b1d8a36fb0f09	2.29
10	AddedLines_count	0.02	Commiter_ad94204efa89fe869b22b	2.27
11	Active Developers Count	0.01	Author_279399fcb100d1abcfe6f29	2.00
12	Commiter_490674fe6e6708ccdbae3	0.01	Author_ae71f5f8cfb2c961ab7e68d	1.97
13	Author_baf95f7ee8bd67e30e6c2b7	0.01	Author_f34cc4b6d61fec27c9d819b	1.85
14	FileType_json	0.01	Author_e3757d1be3d2341e734d99a	1.76
15	Commiter_8f36e4c3b1d8a36fb0f09	0.01	Author_b0277f3a1b52a0d5da7fdbc	1.62
16	Author_84020f066432a8181486e6a	0.01	FileType_crc	1.36
17	Author_4a9cdb0293a6da6c2dee9d4	0.01	Author_079690db340d72406cf61ba	1.33
18	Commiter_1622bcdb0f25f23bafaa0	0.01	Commiter_096d524d5c1caef4fd85e	1.33
19	Merge	0.01	Commiter_75a4d5b3179961faff45b	1.25
20	Distinct Developers Count	0.01	Author_8a3501c8caaeb8b77e4d78a	1.16

Table 6.1: Top 20 Predictive Features of Defective Classes for RF and LR

data sets used. This is due to it being an ensemble learner algorithm, where many learning techniques are used together to achieve better predictive performance than constituent learning [24]. As our data set is a merge of different software applications in the Infotainment system, this could explain why RF is the top model of choice in our results. Our RF model gives us a CE score of 87%. This is a significantly good score as Arisholm [3], mentions that the good model is approximately 50% of the optimal model in terms of CE.

6.1 Feature Importance

We analyzed the features by extracting them from the classifiers along with their associated weights (coefficients) as shown in table 6.1. The higher the weight, the more important the feature in contributing to a decision of the defectiveness of a file.
We noticed in table 6.1 that some features had the same importance in both classifiers, such as *Added Lines*, which was the strongest predictor. It seems also that RF is the most "objective" classifier as most of its top features are non-related to the developers while LR seems to rely heavily on which developer authored or committed a file to make a prediction on whether it was *defective*.

Interestingly, the cyclomatic complexity was a top feature in RF. Therefore code metrics are sometimes superior predictors to process metrics.
We also observed that the 20 bottom features in both classifiers, represented in table 6.2, are interpreted differently. RF seems to assign zeros to all features, indicating that they are meaningless and do not contribute in the classification decision. On

	RF		**LR**	
#	**Feature Name**	**Coefficient**	**Feature Name**	**Coefficient**
188	FileType_flowconfig	0	FileType_aidl	-9.6
189	FileType_frag	0	FileType_apk	-9.6
190	FileType_gitattributes	0	Commiter_84194d6d42f27cc38b619	-9.7
191	FileType_lib	0	FileType_ttf	-9.8
192	FileType_lib_boost	0	FileType_pump	-9.9
193	FileType_lib_openssl	0	FileType_md	-10.
194	FileType_ogv	0	FileType_mp3	-10.
195	FileType_package-list	0	FileType_gradlew	-10.
196	FileType_rar	0	FileType_bat	-10.
197	FileType_ser	0	FileType_mtl	-10.
198	FileType_sln	0	FileType_docx	-10.
199	FileType_store	0	FileType_pro	-10.
200	FileType_suo	0	FileType_obj	-10.
201	FileType_vert	0	FileType_mp4	-10.
202	FileType_wktree	0	FileType_css	-10.
203	FileType_xcf	0	FileType_rawproto	-11.
204	FileType_zip	0	FileType_jar	-11.
205	Commiter_a875100b57d5e24e18ff3	0	FileType_jpg	-12.
206	Commiter_d93dfa0a6fd63f8a931a0	0	FileType_class	-12.
207	Commiter_0e891b8eace486a2fa4f2	0	FileType_png	-13.
20	Distinct Developers Count	0	Author_8a3501c8caaeb8b77e4d78a	1.16

Table 6.2: Uninformative Features in RF and Predictive Features of Negative Class in LR

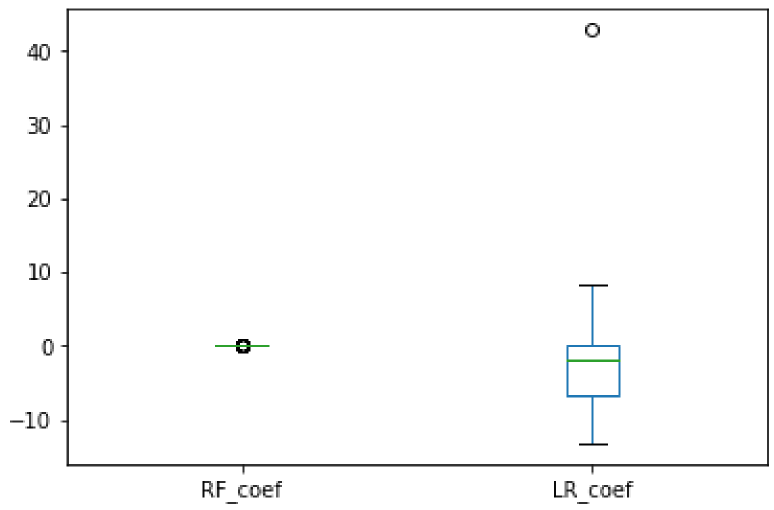


Figure 6.1: Feature Coefficient Ranges for RF vs. LR

the other hand, LR appears to associate its bottom features with the negative class, that is, they tend to predict the file as *clean*.
By inspecting the boxplot in figure 6.1, we notice that LR has a larger inter-quartile range and an outlier (Added Lines), regarding the weights assigned to the features. RF, in contrast, has a much smaller range. By comparing these findings with the CE results for model evaluation, we think that the smaller the range the more robust and the higher performing the classifier. RF has all of its weights ranging from 0 to 0.18. We also notice that the *Added Lines* metric, a top feature in both classifiers, LR assigned a very high weight to it, 43, while RF only assigned a value of 0.18. To illustrate what this means, think about a scenario of a file having many *Added Lines* then the LR would most likely label it as *defective* even though it might not be. Since RF assigns much lower weights to that same feature, it is still comparable in terms of weight to other features and hence *Added Lines* will not dominate and the features will equally contribute to the prediction result, so the result will not necessarily be *defective* for a file with many *Added Lines*. Perhaps this was one of the reasons RF outperformed LR. Other reasons may be due to the fact that RF is inherently robust and minimizes internal bias; this is because it implements multiple decision trees and implements bagging of features and instances, then uses a majority vote for its final result.
It was not feasible to extract the feature coefficients for NB and it was irrelevant to use features from MLP since it creates it's own features internally within the hidden layers. One common characteristic between MLP and RF is that they are nonlinear classifiers while LR is linear. This could also be a factor in why the former two were superior to LR. It is generally easier for nonlinear classifiers to separate classes if data points have many features that are hard to separate through more basic linear classifiers.

6.2 Machine Learning vs Traditional Algorithms

In this thesis work we decided to use machine learning instead of algorithms based on predefined rules and logic; because we are interested in a prediction problem of many dimensions for predicting defective files. Of course traditional algorithms would have worked, had we had clear, predefined rules to define which files were defective, that would have been a valid option. Another reason is that we have over 200 features (metrics), we cannot input the logic manually into an algorithm to determine whether files are defective or not. On the other hand, machine learning algorithms develop the logic and rules implicitly, merely by getting exposed to enough examples of the data. Another reason is that we have enough data for machine learning, which is usually one of the main hindrances when using such models. Looking back at our results, it would have been virtually impossible to define rules that would be accurately correct. We have false positives and false negatives in our classifiers; an expected outcome in machine learning. This indicates that even if a metric was an excellent predictor of the output class, it won't necessarily mean it is always correct. This is something that usually holds true in solutions that rely on traditional algorithms.

6.3 Threats to Validity

This section describes the various threats to validity of the study.

6.3.1 Conclusion Validity

We based this thesis on the finding that ML models that use process metrics outperform code metrics from the literature survey findings. If this finding changes, the collection of metrics will need to be revised and the outcome of the models will potentially be different.

6.3.2 Internal Validity

The quality of the data is measured in terms of the amount of data available and the missing values linked to each commit. The life span of the projects and commit history dictates the amount of data that can be collected. Too small a data set will lead to insignificant results. To overcome this challenge, similar source repositories were identified and combined to make a larger, more complete data set. This was possible as many of the source code repositories stemmed from the same original code base. These branches were modified to become separate modules in the infotainment system.

Even with a larger dataset, missing values will affect the data quality. Missing values include Jira comments, priority level, type of change (bug/fix/feature), Jira ID's in the Git message and change size. This issue can be attributed to team and work culture. If standards are set by the team, such as writing descriptive commit messages, linking Jira issue reports to Git commits and making some Jira issue fields mandatory, then missing data would be less irregular. By analyzing the data, we found that many Jira issues are not connected to Git commits. The Git integration was not set up properly in the Jira tool, this was a missed opportunity for the linking of Git commits to their respective Jira issues.

Another issue was the availability of metrics. Extracting and gathering metric information could not be attained from the open source APIs. Therefore these metrics needed to be excluded. A list of excluded metrics are found in the Appendix. New metrics were also added that were not originally included due to their availability from the API and data available. As these data quality issues lead to a imbalanced data set, 10-fold cross validation is performed to help overcome these threats.

6.3.3 Construct Validity

Through conducting a literature survey and comparing our results to that of similar studies, we can reduce the threat of construct validity. We also apply the same evaluation technique across all the ML models.

6.3.4 External Validity

The metrics and features used to train the model were generic in regards to the dataset and can be extracted from most projects that are integrated with both Git and Jira. The model works on many different file types and extensions. The best selected model may have features that are good predictors for certain projects and domains and therefore may possess internal biases and will need to be retrained for other Android projects.

6.3.5 Granularity

Datasets from similar studies are split into chunks or releases, as releases are generally considered to be a better logical unit instead of using a time shifting window. The rationale is that each release represents its own logical iteration which is a well-defined granularity. Our data set is organized and divided to match the sprint cycle of Aptiv.

6.3.6 Ethical Considerations

Throughout this thesis, the privacy of Aptiv's developers, testers and reviewers was preserved. Their names were encrypted, so that features related to those employees would by anonymous.

7
Conclusion

7.1 Summary

Our thesis set out to: discover and extract features, find existing work that supports our approach, create and realize a defect prediction technique and evaluation methods, and apply it to an industry case study. We were able to achieve this and developed models that achieved up to 88 % accuracy score and a CE score of 87% of compared to the OAUC, on predicting if a file is *clean* or *defective*, which surpassed similar attempts in the supporting literature studies. Our findings show that developing a model to perform defect prediction is more reliant on the features chosen than the type of model. We also discovered that choosing process metrics is a difficult and iterative process and is dependent on the industry data availability. The process metrics we created can be generalized and specialized to the industry case study it is applied to. We also confirmed that features such as code metrics, utilize ML models such as NB more effectively in terms of accuracy [24, 25, 28]. From our results we can see that models such as RF and DT [27] perform better on predicting defective features in terms of the F-measure and ROC. The full models were better performing than the reduced ones, indicating that all the features we used were important for the classifying the testing data. This is not something that happens often in ML as overfitting is a common problem; though the same results were obtained by Arisholm et al. [3]. Another point worth mentioning is that non-linear models are perhaps a better choice in these types of problems as our top two classifiers were nonlinear and their difference in performance was not statistically significant. With a more balanced dataset and more features, maybe the models' performance could be improved. The work outlined in this thesis can provide a repeatable and pragmatic approach in developing and building a defect prediction model in an industry setting. The main contributions are: (i) how to collect process metrics, (ii) pre-processing and labelling the data using word token matching and SZZ, (iii) create a defect prediction model using various ML models, (iv) provided a evaluation method to analyze and validate the results and, (v) deployment of the best model in a production environment in the software industry each time it automatically re-trains.

7.2 Future Work

Improving the F-scores of the models would be part of the future work of this thesis, as well as apply the model to other software repositories within this same domain and industry. It would also be interesting to perform the prediction on the changeset (diff) data using a Neural network. This can be done on projects that have a longer commit history. The changeset (diff) data could also be added as a metric to the existing metrics. At the time of the thesis investigation there was difficulty in attaining all the diff data from Pydriller API. This was a reported bug and will be fixed in a later patch. Another area that will be interesting is to perform defective prediction with process metrics on features instead of files. We know that studies in [28], [25] and [27] concluded that process metrics are more effective for predicting defects on the file-level, but it is not known how superior they will be when it comes to predicting defective features.

Bibliography

[1] John H. McDonald. Handbook of biological statistics (3rd ed.). Sparky House Publishing, Baltimore, Maryland. pg 238-246. `http://www.biostathandbook.com/simplelogistic.html`, 2014.

[2] Amazon. What is source control? Amazon Web Services. `https://aws.amazon.com/devops/source-control/`, 2019. [Online; accessed 19-May-2019].

[3] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[4] Atlassian. Jira overview | products, projects and hosting. `https://www.atlassian.com/software/jira/guides/getting-started/overview`, 2019. [Online; accessed 19-May-2019].

[5] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26. ACM, 2007.

[6] Vangie Beal. mining software repositories - msr.

[7] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[8] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. SZZ unleashed: An open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. *CoRR*, abs/1903.01742, 2019.

[9] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. *API design for machine learning software: experiences from the scikit-learn project*, pages 108–122. 2013.

[10] Chatbots Life. How neural networks work. `https://chatbotslife.com/how-neural-networks-work-ff4c7ad371f7`, 2019. [Online; accessed 19-May-2019].

[11] Daniel A. da Costa, Shane McIntosh, Kulesza Uirá Shang, Weiyi, , Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *Transactions on Software Engineering (TSE)*, 2016.

[12] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649 – 660, 2008. Software Process and Product Measurement.

[13] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23. IEEE, 2003.

[14] Gerrit. Gerrit's history. `https://www.gerritcodereview.com/about.html`, 2019. [Online; accessed 19-May-2019].

[15] Mark A Hall. Correlation-based feature selection of discrete and numeric class machine learning. 2000.

[16] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning – with Applications in R*, volume 103 of *Springer Texts in Statistics*. Springer, New York, 2013.

[17] Jason Brownlee. Supervised and unsupervised machine learning algorithms in understand machine learning algorithms. `https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/`, 2016. [Online; accessed 19-May-2019].

[18] S. Keshav. How to read a paper. *SIGCOMM Comput. Commun. Rev.*, 37(3):83–84, July 2007.

[19] F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314, Aug 2009.

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, July 2008.

[21] Andy Liaw and Matthew Wiener. Classification and Regression by randomForest. *R News*, 2(3):18–22, 2002.

[22] Ilias G. Maglogiannis. *Emerging artificial intelligence applications in computer engineering: real word AI systems with applications in eHealth, HCI, information retrieval and pervasive technologies*. IOS Press, 2007.

[23] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.*, 27(C):504–518, February 2015.

[24] Ruchika Malhotra. An empirical framework for defect prediction using machine learning techniques with android software. *Applied Soft Computing*, 49:1034–1050, 2016.

[25] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[26] Python JIRA. Python jira api documentation. `https://jira.readthedocs.io/en/master/api.html`, 2019. [Online; accessed 19-May-2019].

[27] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. Towards predicting feature defects in software product lines. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, pages 58–62. ACM, 2016.

[28] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.

[29] Scott Chacon. Git documentation. `https://git-scm.com/`, 2019. [Online; accessed 19-May-2019].

[30] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, June 2014.

[31] Skymind. A beginner's guide to multilayer perceptrons (mlp). `https://skymind.ai/wiki/multilayer-perceptron`, 2019. [Online; accessed 19-May-2019].

[32] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories.* 2018.

[33] Steven Symes. What is a car infotainment system? we explain that big touchscreen in your car, Apr 2019.

[34] Techopedia. What is a multilayer perceptron (mlp)? - definition from techopedia. `https://www.techopedia.com/definition/20879/multilayer-perceptron-mlp`, 2019. [Online; accessed 19-May-2019].

[35] Theano Development Team. Multilayer perceptron. `http://deeplearning.net/tutorial/mlp.html`, 2015. [Online; accessed 8-June-2019].

[36] Wikipedia contributors. Adaboost — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Logistic_regression`, 2019. [Online; accessed 19-May-2019].

[37] Wikipedia contributors. Adaboost — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=AdaBoost&oldid=889372345`, 2019. [Online; accessed 19-May-2019].

[38] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[39] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.

A

Appendix 1

Metric	**Description**
LOC	Lines of code
Distinct Developer Count Cumulative	Cumulative number of distinct developers that worked on the file from 1st release to current
Major Contributor to file	Highest contributor (percentage of lines per file)
Minor Contributor Count	Lowest contributor (percentage of lines per file)
Changed Code Scattering	Number of related files that changed with this one (logically coupled)
Neighbor's Active Developer Count	Number of developers that co-committed the file at the same time in the current release (not distinct)
Neighbor's Distinct Developer Count	Number of distinct developers that worked on the file per release that co-committed at the same time.
Neighbor's Commit Count	Number of times a file was committed within a release by a co-committer.
Neighbor's Change Scattering	Number of related files that changed with this one by the co-committer
High priority level commits	The number commits grouped by high priority level.
Medium priority level commits	The number commits grouped by medium priority level.
Low priority level commits	The number commits grouped by low priority level.
Number of Change Requests	The number of change requests in this release.
Total number of tests	Number of system tests/unit tests/ failures for each change request in the release

Table A.1: Excluded Metrics