Investigating the Practicality of Just-in-time Defect Prediction with Semi-supervised Learning on Industrial Commit Data

ARSALAN SYED
Investigating the Practicality of Just-in-time Defect Prediction with Semi-supervised Learning on Industrial Commit Data

ARSALAN SYED
arsalans@kth.se

Master in Computer Science
Date: June 27, 2019
Examiner: Benoit Baudry
Supervisor: Martin Monperrus
Supervisor at principal: Alexander Andelkovic
Host company: King
School of Electrical Engineering and Computer Science
Abstract

Some of the challenges faced with Just-in-time defect (JIT) prediction are achieving high performing models and obtaining large quantities of labelled data. There is also a limited number of studies that actually test the effectiveness of software defect prediction models in practice.

In this thesis, the performance of five notable classification algorithms is investigated when applied to Just-in-time defect prediction. The utility of semi-supervised techniques such as the self-training algorithm is also explored. In order to test the viability of JIT defect prediction models in practice, a case study was set up at King, a game development company. Finally, to have a better understanding of how software developers at King identify and resolve bugs, a series of interviews were conducted.

The investigation found that ensemble learning models such as XGBoost can outperform deep learning approaches such as Deeper. The self-training algorithm can be used to train on labelled and unlabelled data and still achieve similar performance to purely supervised approaches. The case study found that although a JIT defect prediction model based on random forests could achieve better performance than a random model, there is still a large discrepancy between the cross validation performance and the performance in practice. Finally, the interviews found that developers rely on inspecting builds, manual debugging and version control tools to identify bugs. Additionally, the interviews found that risky code tends to have high dependency on other code, is difficult to comprehend and does not follow proper coding practices.
Sammanfattning

Några utmaningar med Just-in-time defect prediction är att uppnå modeller med hög förmåga att identifiera riskabla commits och att ta fram stora kvantiteter av etiketterad data. Ytterligare en utmaning är att det finns ett begränsat antal studier som testar effektiviteten av modeller för programvarufel i praktiken.

I detta examensarbete undersöks resultatet av fem olika klassificeringsalgoritmer som tillämpas på Just-in-time (JIT) defect prediction. Även användningen av halvövervakade metoder så som self-training-algoritmen utforskas. För att testa lönsamheten av JIT-modeller i praktiken har en fallstudie genomförts hos King, ett spelutvecklingsföretag. För att få en bättre förståelse för hur programmerare hos King identifierar och löser buggar genomfördes flera intervjuer.

Acknowledgements

I would like to thank Alexander Andelkovic for his immense support and for creating opportunities that allowed me to further develop myself throughout my time at King. I would like to thank Sahar Asadi for providing continuous feedback and dedicating her time to help guide this thesis. I would like to thank Nacho Fernandez, Jonathan Beck and Tommy William Brooks II for assisting me with the deployment of the Incoming tool as well as Daniel Lindkvist and the developers that participated in the case study. Additionally, I would like to thank Alice Reinaudo, Sara Karimi, Alex Nodet, Jessica Lee, Philip Eisen, Marcus Adamsson, Sami Purmonen, Alice Karnsund, Max Fischer and Rachael Ahn for their support and friendship throughout my time at King.

I would also like to thank Martin Monperrus and Benoit Baudry at the School of Electrical Engineering and Computer Science (EECS) at KTH for supervising and examining my thesis, I am very grateful for their guidance and enthusiasm in this project.

Finally, I would like to thank my friends and family, here in Stockholm and abroad for their continuous support and encouragement over the years.
# Contents

1 Introduction ........................................ 1
   1.1 Research Questions .............................. 2
   1.2 Scope ......................................... 3
   1.3 Outline of Report ............................... 3

2 Background ........................................ 4
   2.1 Machine Learning ................................ 4
      2.1.1 Logistic Regression ......................... 4
      2.1.2 Decision Trees .............................. 5
      2.1.3 Ensemble Methods ........................... 5
      2.1.4 K-Nearest Neighbours ....................... 5
      2.1.5 Semi-Supervised Learning .................... 6
      2.1.6 The Self-Training Algorithm .................. 6
   2.2 Evaluating Classification Models ................. 7
      2.2.1 Accuracy, Precision, Recall and F1 score ...... 8
      2.2.2 K-fold Cross Validation ....................... 8
   2.3 Related Work .................................... 9
      2.3.1 Software Defect Prediction .................... 9
      2.3.2 SZZ Algorithm ................................ 9
      2.3.3 File Level Defect Prediction .................. 10
      2.3.4 Just-in-time Defect Prediction ................. 10
      2.3.5 Practical Implementations of Just-in-time Defect Prediction Models 12

3 Technical Contribution ............................. 13
   3.1 The Incoming Tool ................................ 13
   3.2 Data Collection .................................. 13
   3.3 Training ....................................... 14
   3.4 Sending Predictions .............................. 15
   3.5 Deployment ..................................... 15

4 Methodology ........................................ 16
   4.1 Data .......................................... 16
      4.1.1 Data From Previous Research ............... 16
      4.1.2 Newly Collected Data ....................... 17
      4.1.3 Features Used ................................ 17
      4.1.4 Data Preprocessing .......................... 17
   4.2 Experimental Setup ................................ 18
   4.3 Experiments .................................... 18
      4.3.1 Experiments for RQ1 ......................... 19
4.3.2 Experiments for RQ2 ........................................... 19
4.3.3 Experiments for RQ3 ........................................... 21
4.3.4 Experiments for RQ4 ........................................... 23

5 Results ............................................................... 24
5.1 Research Question 1 Results .................................... 24
5.2 Research Question 2 Results .................................... 25
  5.2.1 Experimenting with Five Models ......................... 25
  5.2.2 Varying the Percentage of Labelled Data ............. 28
  5.2.3 Varying the Iterations for Self-Training ............... 30
5.3 Research Question 3 Results .................................... 31
5.4 Research Question 4 Results .................................... 32
  5.4.1 Interview topic 1: Identifying the Origin of Bugs .......... 32
  5.4.2 Interview topic 2: Tools Used to Identify and Reduce Bugs .................. 33
  5.4.3 Interview topic 3: Differences Between Clean and Faulty Code .......... 35
  5.4.4 Interview topic 4: Top Causes for Bugs .................. 36

6 Discussion .......................................................... 39
6.1 Interpreting Results of Conducted Experiments ............... 39
  6.1.1 RQ 1: Performance across multiple models ............... 39
  6.1.2 RQ 2: Practicality of the Self-Training Algorithm ....... 39
  6.1.3 RQ 3: Just-in-time Defect Prediction in Practice .......... 40
  6.1.4 RQ 4: Interpreting the Results of the Interviews .......... 41
6.2 General Considerations for Defect Prediction Models ........ 42
  6.2.1 Prioritising Precision or Recall ......................... 42
  6.2.2 Binary Labelling of the Dataset ......................... 43
6.3 Ethical Concerns .................................................. 43
  6.3.1 Discrimination Against Developers ....................... 43
  6.3.2 De-anonymising Datasets .................................. 44
  6.3.3 Handling Commercial Datasets .............................. 44
6.4 Threats to Validity .............................................. 44
  6.4.1 Limitations of the SZZ Algorithm ....................... 44
  6.4.2 Observed Performance in Case Study ..................... 45
  6.4.3 Differences in Results due to Randomness ............... 45

7 Conclusion ........................................................ 46
7.1 Future Work ..................................................... 47

Bibliography ........................................................ 47
Chapter 1

Introduction

A universal issue that affects software developers is the presence of software bugs. Bugs in software can be defined as unintentional errors that cause software to perform incorrectly. Defects in software can have a serious financial impact on businesses and reduce the time available for developers to create new features. Bugs in software can also introduce security vulnerabilities that could lead to the leakage of user data as well a loss of reputation [3] for an organisation. There have been several large-scale incidents in the past involving defective software. For example, the Mars Climate Orbiter, a space probe launched by NASA in 1998 crashed on Mars due to a bug which failed to use metric measurements instead of imperial units [25]. Another example is the Heartbleed bug, a security vulnerability in the OpenSSL protocol which had the ability to expose confidential information such as passwords and credit card numbers [9]. In 1994, the Pentium FDIV bug incident caused Intel processors to produce faulty values when dividing floating point values which lead to a mass recall of defective processors [22]. These examples highlight some of the large scale impacts that bugs can have and it can be very challenging to identify these vulnerabilities in the first place.

A study by IBM found that the associated cost with resolving a bug in software increases depending on what phase in the development lifecycle the bug was identified [3]. Moreover, software bugs that are discovered in the production phase are more costly because they can directly affect customers as opposed to bugs that occur early on in the development lifecycle. Therefore there is a justification for being able to detect bugs as early as possible.

Software developers identify bugs by performing frequent testing and they reduce the likelihood of new bugs appearing by improving development practices. For example, unit testing involves verifying the validity of individual methods within code and these tests can be run every time a new build is created. Development practices have also shifted from traditional waterfall methods to Agile methodologies in order to keep up with the fast pace of development cycles. Agile methodologies allow for more frequent testing as well as code reviews before integrating new features to the main product [10]. However, a challenge with identifying bugs is that as a project grows in size, it becomes more complex to maintain. This complexity makes it impractical to try out all possible executions of a program and running these tests can become time consuming for developers.

To assist developers with identifying bugs, recent research has suggested the usage of software defect prediction models [17]. These models aim to identify regions in code that are most vulnerable to defects. Using these models, developers would be able to
prioritise their testing efforts, spend less time reviewing code and receive continuous feedback in an automated fashion.

In order to create such a defect prediction model, one could leverage the capabilities of machine learning. Machine learning is the field of study concerned with algorithms that can learn from historical samples in order to predict values for unseen samples. Although machine learning has been around since the 60s, its viability has grown immensely in recent years due to advances in hardware, the open sourcing of machine learning frameworks and a large availability of data. Machine learning algorithms are able to spot complex patterns in data to solve various tasks such as facial recognition or translation of text and are now being tested on the task of identifying bugs [21]. In the context of identifying risky portions of code, one could use a supervised learning approach which involves showing a machine learning algorithm examples of risky and clean source files to allow the algorithm to predict whether a new file contains a defect.

A proposed method for identifying risky code is Just-in-time (JIT) defect prediction. Unlike previous research which has focused on predictions at the file-level, JIT defect prediction aims to predict individual commits as being risky or not [17]. The justification for these models is that commits tend to be smaller to review than files which can make the reviewing process simpler. Also, a problem with identifying risky files is that files can have multiple authors so it is unclear as to who should be assigned to resolve that file. Commits on the other hand only have one author so one could assign the commit’s author to resolve any potential issues. Finally, JIT prediction models can make predictions rapidly because one can extract the features required for a prediction as soon as the commit is made.

When applying supervised learning techniques for this particular task, labelled data is required. The problem is that the majority of commits do not have any labels that indicate whether or not they caused a defect. In order to automatically label commits as risky or not risky, the Śliwerski Zimmermann Zeller (SZZ) algorithm can be used to obtain large datasets for building defect prediction models [27]. However, a limitation of the original algorithm is that it requires the project being analysed to have a bug tracking database that confirms if certain commits resolve bugs. Some projects might not have this database and even if they do, the information it stores might not contain data on a commit by commit basis. Instead, one could rely on the Approximate SZZ (ASZZ) algorithm which removes this restriction.

Since labelling commits can be quite time consuming, it would be useful to have a technique that could leverage the information contained in unlabelled commits. Semi-supervised learning is an example of a subset of machine learning algorithms that can train on both labelled and unlabelled data [36]. The self-training algorithm is a simple semi-supervised technique that can be applied to any classification method that outputs probabilities for its labels [35].

1.1 Research Questions

The purpose of this thesis is to investigate the performance of supervised classification models at the task of JIT defect prediction as well as to explore the viability of the self-training algorithm. A case study is also performed in order to understand how practical JIT defect prediction models are in an industrial environment such as King ¹, a game

¹https://king.com/
development company. This thesis also performs a qualitative study to understand how software developers at King identify and resolve bugs. The research questions that will be answered are:

- **RQ1:** What is the effectiveness of five notable machine learning classification algorithms at Just-in-time defect predictions?
- **RQ2:** When comparing the semi-supervised method of self-training with supervised learning techniques, how well does the semi-supervised approach perform when applied to Just-in-time defect predictions?
- **RQ3:** When provided with a Just-in-time defect prediction model, how accurate are the model’s predictions according to software developers?
- **RQ4:** How do software developers identify bugs and what are some characteristics of code that contains a bug?

### 1.2 Scope

This project is focused on Just-in-time defect prediction models which predict at the commit level and only rely on commit metadata rather than analysing the source code. The collection of new datasets is done using the approximate SZZ algorithm and using a smaller set of features than those typically found in datasets for JIT defect prediction.

### 1.3 Outline of Report

Chapter 2 provides relevant background theory on the field of software defect prediction as well as achievements in related work. Chapter 3 describes the technical contribution that was created for this thesis project. Chapter 4 covers the methodology utilised for answering the specified research questions. Chapter 5 presents the results of these experiments. Chapter 6 analyses and discusses the key findings and chapter 7 summarises the findings and suggests possibilities for future work in the field.
Chapter 2

Background

This chapter describes the relevant theory behind software bug predictions and what the state of the art has been able to achieve.

2.1 Machine Learning

Machine learning can be defined as the study of algorithms that learn from a set of observed samples to predict values for unseen samples [19]. These algorithms have recently gained traction and are applied more in products and services due to the increase in computing power, availability of open source machine learning frameworks and vast amounts of data. Machine learning algorithms have become a popular approach for software defect prediction models as they eliminate the need for a programmer to explicitly set rules that identify a commit or file as being risky or not. Machine learning approaches can automatically spot patterns in data that may have been difficult for people to find, especially if the dimensionality of the input space is large. Some of the main challenges with these algorithms are to obtain substantial quantities of labelled data and to ensure that the model’s predictions are reliable.

In the context of software defect prediction, machine learning models are used to solve a binary classification problem. Classification involves taking samples of data and identifying their correct category (in this case, risky or not risky) when given their features. There are various algorithms available for classification which differ by their mathematical formulations.

2.1.1 Logistic Regression

Logistic regression is a binary classification model that takes real valued inputs and returns their probability of belonging to one of two classes, 0 or 1. The logistic regression model makes predictions by applying the sigmoid function \( \sigma(z) = \frac{1}{1+e^{-z}} \) upon a linear model \( f(x) = x^T \theta \) (where \( x \) is our input and \( \theta \) is the model’s parameters) [1]. When provided an input vector \( x \), the function \( \sigma(f(x)) \) will predict the probability that \( x \) belongs to the class 1 whereas \( 1 - \sigma(f(x)) \) computes the probability that the class of \( x \) is 0. The final label that is predicted is based on the most likely class, for example if the output of \( \sigma(f(x)) > 0.5 \), the predicted class is 1.

The optimal parameter for \( \theta \) can be found using maximum likelihood estimation which aims to find the parameter \( \theta \) that maximizes the likelihood \( p(t|\theta) \) (where \( t \) is a vector containing output labels). Typically, this maximization problem is transformed
into a minimization problem by taking the negative log of the likelihood. Then, iterative methods such as gradient descent can be used to obtain the optimal parameters.

### 2.1.2 Decision Trees

Decision trees are a type of tree-based model which classify inputs by using a set of conditional rules [12]. Every node in the decision tree represents a single conditional rule applied to an input feature, for example a dataset containing information about people could be split into two subsets by using the rule $\text{Age} < 30$. After a decision rule has been evaluated, the input is passed further down the tree along one of two possible branches. One heuristic algorithm used for creating a decision tree is ID3 which constructs a tree using a greedy, top-down approach [23]. Given a dataset $D$ and some features, ID3 chooses a feature that has not been included in the tree yet such that it provides the largest information gain. A decision rule is created upon this feature and this allows a dataset to be partitioned. The algorithm terminates when all features have been used or when all elements of a partition have the same label.

### 2.1.3 Ensemble Methods

An issue with decision trees is that an individual tree can be very prone to overfitting the training data, resulting in a high variance model. In order to combat this, one can rely on ensemble techniques which involve combining multiple base models when training. One example of an ensemble learning technique is bagging which trains multiple classifiers independently and aggregates their outputs [2]. Given that we want to train $n$ models, bagging will randomly sample the training set and train a model on this sample so that each individual model has a different view of the data. A random forest model applies bagging by training $B$ decision trees. When a prediction has to be made for a classification task it aggregates the predictions by taking the majority value from each tree.

Boosting methods on the other hand involve sequential training of an ensemble of classifiers to convert weak classifiers (a model that does slightly better than random) into stronger ones [26]. Unlike in bagging methods such as random forest, the classifiers are dependently trained on each other. When trained sequentially, subsequent models will attempt to correct upon previous learners by giving an increased weight to data points that the last model misclassified. Some examples of commonly used boosting methods applied upon decision trees are AdaBoost and XGBoost.

### 2.1.4 K-Nearest Neighbours

The K-nearest neighbours algorithm is another example of a non-parametric algorithm. To make a prediction for a new data point $x$, the algorithm must find the $k$ elements of the training set that are closest to $x$ [6]. The proximity of points can be measured using distance metrics such as Euclidean distance or similarity metrics like the Pearson correlation [13]. The predicted label for $x$ will then be the most frequent label among its k-nearest neighbours. In the case of a binary classification problem, $k$ should be chosen to be odd to guarantee that a majority label exists. When the labels of the dataset are imbalanced, the algorithm will tend to predict the dominant class more often because of how frequently it occurs rather than because it is the correct label [5]. To address
this, one can apply a weighted aggregation of the \( k \) labels so that closer data points contribute more to the final prediction.

### 2.1.5 Semi-Supervised Learning

The majority of software defect prediction models are implemented using supervised learning which trains exclusively on labelled data. Semi-supervised learning on the other hand relies on labelled and unlabelled data to train models. The motivation for semi-supervised approaches is that labelling data can be expensive in terms of resources so one could instead compensate by using unlabelled data as much as possible alongside the limited labelled data.

According to [4], in order for semi-supervised learning to be effective, there are several assumptions that have to be made. First of all, data points that are close to each other or form clusters should have a high probability of having the same label. This is because semi-supervised methods propagate labels from labelled instances to unlabelled instances. Also, it is required that points in higher dimensions need to be representable in lower dimensions in order for points to be able to be considered close to each other.

### 2.1.6 The Self-Training Algorithm

The self-training algorithm is a type of semi-supervised algorithm that can act as a wrapper method for any classifier [29]. When provided labelled and unlabelled data, the algorithm begins by training a classifier on the labelled data. It then makes predictions for the unlabelled data and if the classifier’s prediction has a high probability, these predicted samples will be included in the training dataset. Then a model trains on this new dataset and these steps are repeated until the maximum number of iterations are reached or if the predictions for the unlabelled data is identical to the previous iteration’s predictions.

For the self-training algorithm, let us define the following. Let \( X_L, y_L \) be our labelled inputs and outputs. Then \( X_U \) consists of unlabelled inputs and the model is defined as the function \( f : X \rightarrow Y \). The parameters for the algorithm are \( I \in \mathbb{N} \), the max number of iterations and \( C \in \{ c \in \mathbb{R} | 0 \leq c \leq 1 \} \), the minimum confidence for a prediction on unlabelled data. The variable \( \text{iter} \) represents the current iteration and \( y_{U, \text{old}} \) is a local variable for storing the previous predictions. The variable \( \text{idx} \) contains the indices of all unlabelled samples such that their probability of being either class is greater than the threshold \( C \).

Some variations of this algorithm will permanently move samples from \( X_U \) and their predicted labels to the sets \( X_L, y_L \). This implementation on the other hand does not remove entries from the unlabelled dataset \( X_U \).
Algorithm 1 Self-training algorithm [35]

```python
procedure SELF-TRAIN($X_L, y_L, X_U, I, C_t', f$)
    $\text{iter} = 0$
    $f = f.fit(X_L, y_L)$
    $y_U = f.predict(X_U)$
    $y_{U,old} = \text{an empty array with same length as } y_U$
    while $\text{iter} < I$ and $|y_{U,old}| == 0$ or $y_{U} \neq y_{U,old}$ do
        $y_{U,old} = y_U$
        $idx = \{1 \leq i \leq \text{len}(y_U) | P(y_i = 1) > C \lor P(y_i = 0) > C\}$
        $X_{\text{new}} = \{x_i \in X_U | i \in idx\}$
        $y_{\text{new}} = \{y_i \in y_U | i \in idx\}$
        $f = f.fit(X_L.concat(X_{\text{new}}), y_L.concat(y_{\text{new}}))$
        $y_U = f.predict(X_U)$
        $\text{iter} = \text{iter} + 1$
    return $f$
```

One variation of this algorithm involves using no thresholds when adding the unlabelled data to the labelled data. One can also make predictions on a smaller batch of unlabelled data rather than on all possible unlabelled data. As the algorithm works as a wrapper method to any base classifier, it is possible to make a direct comparison between a supervised learning algorithm and its self-trained counterpart. A risk with using this algorithm is that its performance could degrade if it spots an incorrect pattern early on and reinforces this on the unlabelled data.

### 2.2 Evaluating Classification Models

In order to verify that our models have managed to perform well at a classification task, the models are evaluated on a test set and various performance measures are recorded. Below are metrics that describe the possible outcomes that can occur when making a prediction:

- **TP** = The number of true positives, instances belonging to the target class that were correctly identified

- **TN** = The number of true negatives, instances belonging to the non-target class that were correctly identified

- **FP** = The number of false positives, instances belonging to the non-target class that were misclassified as the target class

- **FN** = The number of false negatives, instances belonging to the target class that were misclassified as the non-target class

In the context of JIT defect prediction, a false positive (FP) would describe a commit that was actually clean but predicted to be risky. A false negative (FN) would be a commit that was risky but classified as not risky. These four values can be used to reason about a model’s performance and define further performance metrics.
2.2.1 Accuracy, Precision, Recall and F1 score

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}
\]

Accuracy represents the fraction of total data samples our model predicted correctly [8]. However, if a dataset has a class imbalance (which software defect prediction datasets typically do), accuracy can be a misleading metric for performance. Since one of the classes would be underrepresented, the machine learning classifiers learn to predict the majority class label for every sample it sees. Take the following example, with a dataset of emails where 99% are not spam and 1% are spam, a classifier could achieve 99% accuracy by labelling each email as not containing spam. However, this would lead to a very high number of false negatives and the classifier would not have actually learned what a spam email is. Therefore we rely on additional metrics such as precision and recall.

\[
\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}
\]

Precision in the context of software defect prediction is the fraction of instances that were correctly classified as defective relative to all instances that were deemed defective by the classifier [7]. Recall is the fraction of correctly identified defects relative to the actual number of defective instances. Precision and recall are inversely proportional to each other, therefore as soon as a classifier aims to optimize recall, its precision will decrease and vice-versa.

\[
F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.4}
\]

The F1 score aims to combine precision and recall into one metric using the harmonic mean so that one can optimize models towards having a good combination of both.

2.2.2 K-fold Cross Validation

K-fold cross validation is a statistical method for generalizing a model’s performance to unseen data [15]. Cross validation has two main roles, the first being for model selection during training and the second is for evaluating a model’s final performance. K-fold cross validation involves splitting a dataset into \(k\) partitions such that a classifier trains on \(k-1\) of the partitions and is evaluated on the remaining set (designated as the test set). When performing model selection, a dataset will typically be split into a training, validation and test set. The validation set is used to tune the hyperparameters of a model whereas the test set is used for a final evaluation.

A benefit of using cross validation is that it reduces bias and error when estimating the classifier’s true performance. However, there is criticism for using cross validation as an evaluation method for software defect predictions. This is because a model may
end up using future knowledge by training on samples that occurred later in time rather
than on those in the test set [28].

2.3 Related Work

This section outlines some of the past achievements in the field of software defect
prediction and the various techniques that were used.

2.3.1 Software Defect Prediction

The purpose of software defect prediction is to predict the likelihood of a defect within
software. The proposed benefit of these models would be to assist software developers
with bug detection and to prioritize testing efforts [3]. The general process for construct-
ing software defect prediction models begins with obtaining a dataset that labels files or
commits as risky or not risky (a risky commit is defined as a commit that will produce
a bug). Then for each instance in the dataset (a commit or file), various features such
as McCabe complexity or lines of code are computed [34]. The features and labels are
then used to train a supervised machine learning classifier and finally, the model can be
used to make predictions for new instances.

When building defect prediction models, there are various input features that can
be used. One could provide metadata concerning files or commits such as lines of code
added or the number of developers working on a particular file. One could also use text
based features such as commit messages or comments which can be encoded using a
bag-of-words representation. One could also parse the source code itself with abstract
syntax trees which can be encoded as vectors [28].

2.3.2 SZZ Algorithm

In order to build defect prediction models, one requires labelled data that indicates
whether or not a file or commit contains a bug or not. However in practice, this infor-
mation may not always be available and manually labelling commits is time consuming.
Research in defect prediction models have benefitted greatly from the Sliwerski Zimmer-
mann Zeller (SZZ) algorithm, originally described in the paper "When do changes induce
fixes?" [27]. The focus of the paper was limited to identifying bug inducing commits
rather than creating defect prediction models but their labelling algorithm is valuable for
automatically labelling a large number of commits.

The algorithm works by initially locating commits that fix a bug. This can be done
by filtering commits by their message or comments for keywords such as "bug" or "fix".
Those commits are then cross referenced with a bug tracking database in order to confirm
that these commits do resolve a bug. For each bug fixing commit, the files this commit
touched as well as the exact lines changed within those files are computed using the
diff command in a version control system (such as CVS or Git). One can then use
the blame or annotate command to find the previous commit responsible for causing a
fix to be created. This previous commit becomes labelled as a bug causing commit. A
variant of this algorithm denoted Approximate SZZ (ASZZ) follows the same essential
steps but does not rely on a bug tracking database for identifying bug fixing commits.
Since not all projects use a bug tracking database, the ASZZ algorithm would allow more
training and testing data to be collected.
Algorithm 2 SZZ: labels commits as *risky* or *not risky* [27]

1: procedure SZZ()
2:   keywords = {"bug", "fix"}
3:   commits = a list of all commits in a repository
4:   bugFixCommits = {c : c ∈ commits and containsKeyword(c.message)}
5:   result = ∅
6:   for bugFixCommit in bugFixCommits do
7:     filesTouched = getFilesTouched(bugFixCommit)
8:     linesTouched = getLinesTouched(bugFixCommit, filesTouched)
9:     result = result ∪ getBlame(filesTouched, linesTouched)
10:  return result

2.3.3 File Level Defect Prediction

In their work, Meng Yan et al. investigate the claim that unsupervised models may outperform supervised models for providing effort-aware defect predictions on the file level [31]. They replicate a previous study upon the PROMISE dataset and their main finding was that unsupervised models can outperform supervised models for cross-project prediction but not for within-project predictions.

In *Personalized Defect Prediction*, the authors propose creating defect prediction models that are tailored to individual developers [14]. The motivation for this is the fact that developers exhibit varying coding patterns and styles and the number of bugs introduced can depend upon the experience the developer has. Results found that this method can improve a model’s F1 score by 0.01-0.06.

Song Wang et al. apply a Deep Belief Network upon the file-level changes of software projects [30]. Their deep learning models are trained upon vector representations of the programs’ abstract syntax trees (AST) and this approach was shown to improve within-project as well as cross-project defect prediction. The issue with non-semantic metrics (such as lines of code) is that programs performing different tasks may have the same values for these metrics. Their models were evaluated upon 10 open source Java projects from the PROMISE datasets. Their evaluation compares to two baselines, the first being non-semantic features that were available in the PROMISE datasets and the second being AST nodes that they provided to their models.

2.3.4 Just-in-time Defect Prediction

A method for providing software defect predictions that has gained traction recently is Just-in-time defect predictions. These models make predictions at the commit level rather than at the file level and utilize commit metadata for features rather than product metrics or by analyzing the source code [17]. The motivation for this approach is the fact that commits tend to be smaller and more manageable to review than files. Also, there is the fact that files can be edited by multiple developers which makes it challenging to assign an individual developer to resolve the bug whereas commits only have one author [17]. A proposed benefit of these models is having the ability to provide immediate feedback to a developer as soon as a change is made.

One of the earlier works on change-level defect prediction, "*Classifying Software Changes: Clean or Buggy?*” suggested that one should train models on file changes rather than files or functions themselves [18]. This paper differs from previous work in
the sense that it used the SZZ algorithm to find the exact changes that cause a bug whereas previous work relied on bug-fix data which could only show roughly where a bug occurred. Also, due to the fact that they represent their features using a bag-of-words encoding, their technique could work independently of programming languages.

In their paper, Kamei et al. [17] introduce the idea of Just-in-time prediction models where models attempt to predict risky commits rather than risky files. They also ensure that the bug-fixing commits that the SZZ finds are present in a bug-tracking database for most of the projects they investigate. They evaluated their models on six open source projects as well as five commercial projects to obtain an average accuracy of 68% and average recall of 64%. They also evaluate if the model serves a useful purpose practically, i.e. will prioritising commits by risk level reduce the amount of work required when reviewing code. This paper is considered to be the most related to the research undertaken within this thesis as the same datasets and similar modelling techniques are used.

A follow-up work to the previous paper stated that predictions need to be accurate for projects that have little historical data available [11]. This can be a challenge for defect prediction models where the model only trains on data from a single repository. Instead, they investigate the feasibility of cross-project models which train on data from multiple projects. They found that within-project models did not perform well on cross-project performance. They also found that within-project models can improve if the project they intend to predict on has similar correlations between predictor and dependent variables. It was shown that file level predictions seem to do better at cross-project performance than commit level predictions. Finally, they also show that ensemble learning techniques provide a benefit to cross-project performance.

In Deep learning for just-in-time defect prediction, the authors propose a deep-learning based approach to detect risky commits [33]. They evaluate performance on the same six open source projects that Kamei et al. used and showed that their approach outperformed previous models. They used a deep belief network algorithm for feature selection and then utilised deep neural networks to build the classifier. Their model named Deeper was able to find 32.22% more bugs and had statistically significant higher F1 scores. This paper was one of the first examples of applying deep learning to Just-in-time defect predictions and they also investigate the effect that the quantity of data has on their model.

Yang et al. attempt a different approach to Just-In-Time predictions that involves a 2-layer ensemble learning classifier named TLEL that combines stacking and bagging techniques with decision trees [32]. The first layer consists of applying bagging to a decision tree to produce a random forest model. The second layer uses stacking and random undersampling to train multiple random forest models and combine them together. The motivation behind this two-layer approach according to them is that they observed that decision trees perform particularly well at this task as well as the fact that ensemble methods tend to produce better results over single models. They compare their approach to three previous models, Deeper, DNC and MKEL. For evaluating their models, they rely on F1 score and PofB20 which measures how well the model can find bugs by only reviewing 20% of all lines of code.

In the paper Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction, the authors state that one challenge in this field is obtaining labelled training data so they investigate the possibility of using unsupervised models. Their contribution is the CBS algorithm (Classify Before Sorting) which makes
predictions and then sorts the output by the size of the commit in ascending order. They focus on making their predictions effort-aware because it takes time to inspect a commit and they want to minimise the time a developer would need to do a review after they receive predictions. They also observed that the distribution of lines of codes in commits is skewed, the majority of commits are small and very few are large, thus the need for applying logarithmic transformations to their features.

In his paper Online Defect Prediction for Imbalanced Data, Ming Tan addresses the challenge of class imbalance in defect prediction [28]. Usually, the minority of entries in datasets will have a label corresponding to a bug. The author tried out four re-sampling methods which alter the training set in order to deal with the imbalance in the dataset. One issue that is brought up in this paper is that previous studies rely on cross validation to evaluate model performance which does not assess models in a realistic manner. The reason being that cross validation splits up the training and test sets in a way such that models can use future knowledge to make predictions. Instead, the author suggests that a time-sensitive classification should be used so that the test set only contains commits that occur later in time than the commits provided in the training set.

2.3.5 Practical Implementations of Just-in-time Defect Prediction Models

A recent open source implementation of a JIT risk prediction model is Commit Guru which builds directly on top of the work presented in A large-scale empirical study of just-in-time quality assurance [24]. Commit Guru is able to mine a GitHub repository and returns statistics on which commits may have high risk. The associated paper for Commit Guru outlines the architecture of the tool as well as the usage of the SZZ algorithm and the classification model they use. As [21] points out, the limitation of this tool is that it cannot point out which exact code block caused a bug. Also, there has not been a formal study that investigates how useful the predictions from Commit Guru are for developers and what performance is required in terms of precision and recall for developers to trust the tool.

The video games developer Ubisoft performed a study in collaboration with Concordia University in which they developed a tool named CLEVER which builds upon Commit Guru when performing predictions of risky commits [21]. They rely on a two stage process, the first is similar to Commit Guru which involves predicting commits that are risky. The second step then does a deeper analysis of those potentially risky commits by comparing code snippets with a database of known bugs to confirm that the commit does cause some sort of issue and is not a false positive.
Chapter 3

Technical Contribution

This chapter describes the implementation of a tool named Incoming that was implem-
ented with the purpose of predicting risky commits using Just-in-time defect prediction

3.1 The Incoming Tool

The tool has two main modes of operation that can be used as independent processes.
The first is the training mode which collects new commits for a repository and trains
models on them. The second mode involves providing predictions using trained models
and new commits. All of the scripts for the data collection as well as for training the
model and making predictions were implemented in Python 3 using the pandas and
scikit-learn libraries and could be executed on Windows 10 as well as Linux operating
systems.

3.2 Data Collection

This tool was designed to collect data from any Git repository hosted on GitHub, publicly
or internally at an organisation. When provided the HTTPS URL for a GitHub repository,
it would clone the project to the local disk. It then collects a list of commits and extracts
the features for each commit.

Extracting commit data is done using Python scripts that create a subprocess that
calls the command "git log --numstat". The output of this command is parsed to
obtain information regarding the commit such as the author, commit message, number
of files changed and more. Then for all of these new commits, the ASZZ algorithm
is run in order to identify potential commits that cause a bug. When assigning labels,
any commit that was found to be causing a bug by ASZZ is given a label of 1 and all
remaining commits are assigned a label of 0.

The data collection tool can be used repeatedly on the same repository in order to
fetch the latest commits. However, for larger projects it is wasteful to repeat this process
on commits that have already been scraped. So whenever new commits are mined, their
data is saved remotely in CSV files. Then in subsequent runs of the data collection, the
tool avoids mining data for old commits by comparing the commit’s hash.

An alternative method of collecting data that was considered was utilising the GitHub
API. Although the API could be used to obtain features and label commits, it was not
practical to use for extracting large datasets due to its rate limits. The GitHub API was better suited for fetching the names or URLs of repositories in this case.

3.3 Training

The classification algorithm used for this tool was a random forest model with 100 estimators using the Gini impurity metric for creating decision rules. Random forest was used because it is easier for developers to interpret the most important features and is simple to train due to a small number of hyperparameters. Also, past research has shown that ensemble learning methods tend to outperform single models at the task of JIT defect prediction [32]. For each project that the tool would run for, a separate random forest model was created. This was because cross defect prediction models tend to suffer in terms of performance as opposed to within-project models [16]. For preprocessing the data, missing values where replaced using the mean. The datetime value extracted was parsed and converted into new features such as the hour and the day of the week the commit was made. Categorical features with more than one category were represented using a one-hot encoding. All features that were not categorical were standardised by transforming the mean to 0 and the standard deviation to 1. In order to deal with the class imbalance in the dataset the SMOTE sampling technique was used because it has been shown to be useful for defect prediction models [28]. Similar to the commit datasets for each repository, the models are also stored in a common storage such that any instance of the tool has access to them. Once trained, the models are evaluated using 10-fold cross validation to provide live statistics using measures such as accuracy, precision, recall and F1 score.

![Figure 3.1: Training mode](image-url)
3.4 Sending Predictions

For the experiments, the tool was set up to send out predictions periodically to individual developers through Slack, a messaging tool. When making predictions, the tool would obtain a list of recent commits from several branches made by developers who registered to use the tool. Branches other than the master branch were included because development branches contain commits that are not reviewed during a pull request, hence the possibility of them being more risky. Once a prediction was made, it was sent to the author of the commit by using a Slack bot.

Each prediction message sent on Slack would contain the link to the commit that was predicted on, the model’s prediction for this commit (risky or not risky) as well as how confident the model was in that prediction. When machine learning models perform classification, they will output a probability for each class that an instance belongs to this class. The confidence is just the probability our commit is risky according to the model. The purpose of adding confidence to the message was because if a prediction had a 51% confidence, this prediction would not be reliable as the chance of it being a false positive is very high.

![Figure 3.2: Prediction mode](image)

3.5 Deployment

In order to have data collection, training of models and predictions be performed frequently, the tool was deployed on King’s continuous integration servers. The data collection and training was bundled into one Jenkins job whereas creating and sending predictions ran in a separate job. These jobs were set up so that there were separate jobs for each project for which the tool was deployed to. Unlike the tool Commit Guru which analyses commits once a user requests it, this tool runs autonomously. A proposed benefit of this would be that developers might be preoccupied with other tasks and forget to run the tool manually. Having it run in the background means that the tool can naturally be integrated into the developers’ workflow.
Chapter 4

Methodology

This chapter highlights the specific algorithms and datasets used as well as the setup for each performed experiment.

4.1 Data

In order to answer some of the specified research questions, a total of nine labelled datasets were obtained. Six of these were obtained from previous research [17] and the other three were collected using software projects at King.

4.1.1 Data From Previous Research

In their work *A Large-Scale Empirical Study of Just-in-Time Quality Assurance*, Kamei et al. provided 6 labelled datasets ¹ which contained commit metadata from open source projects [17]. These six datasets correspond to the first six rows of Table 4.1.

<table>
<thead>
<tr>
<th>Project</th>
<th>Time</th>
<th># Commits</th>
<th>% Buggy</th>
<th>Approximate SZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>08/1998-12/2006</td>
<td>4,260</td>
<td>36%</td>
<td>No</td>
</tr>
<tr>
<td>Columba</td>
<td>11/2002-07/2006</td>
<td>4,455</td>
<td>31%</td>
<td>Yes</td>
</tr>
<tr>
<td>JDT</td>
<td>05/2001-12/2007</td>
<td>35,386</td>
<td>14%</td>
<td>No</td>
</tr>
<tr>
<td>Platform</td>
<td>05/2001-12/2007</td>
<td>64,250</td>
<td>14%</td>
<td>No</td>
</tr>
<tr>
<td>Mozilla</td>
<td>01/2000-12/2006</td>
<td>98,275</td>
<td>5%</td>
<td>No</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>07/1996-05/2010</td>
<td>20,431</td>
<td>25%</td>
<td>Yes</td>
</tr>
<tr>
<td>C1</td>
<td>-</td>
<td>32,661</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>C2</td>
<td>-</td>
<td>121,636</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>C3</td>
<td>-</td>
<td>57,802</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>439,156</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

¹The six previous datasets can be found here http://research.cs.queensu.ca/kamei/jittse/jit.zip
4.1.2 Newly Collected Data

The implemented tool described in chapter 3 of the report was used to collect data on several proprietary projects at King. The data that was collected included all commits from these projects except those that represented merges between branches in Git. Merge commits were excluded because they represent a collection of several commits as one large change, hence distorting the dataset as an outlier. Also, merge commits are created automatically by Git rather than a developer and defect prediction tools intend to predict on commits made by developers. The names and percent of buggy commits for the commercial projects in table 4.1 are anonymous for confidentiality reasons. Most of the features in the new datasets were computed in an identical manner to those in the Kamei datasets. The reason for not including the exact same set of features as the old datasets was due to limitations in time when implementing the tool.

4.1.3 Features Used

Table 4.2 describes the individual features used in all provided datasets. These features can be directly extracted using Git commands such as `git log`. The column Datasets used in in Table 4.2 indicates if a feature is included in the old datasets, new datasets or in both. The features FIX, DAY and HOUR are categorical features which can only take a finite number of values whereas the remaining features are all integers or real valued.

4.1.4 Data Preprocessing

First, the features with a high positive Pearson correlation coefficient were removed in order to simplify the model. Next, due to the fact that features associated with commits are not normally distributed, log transforms were applied to all continuous valued features using the function \( f(x) = \ln(x + 1) \). This function was used because some features could take a value of 0 and the standard log function would make these values undefined. The data was then standardized which ensured that the mean and standard deviation of each feature was 0 and 1 respectively. To deal with multilabel categorical features such as "DAY", these feature were represented using a one-hot encoding. In order to deal with the class imbalance, random undersampling is performed on the training set in order to produce a dataset with an equal number of risky and not risky commits. The steps undertaken in this stage were similar to what Kamei et al. performed [17] in order to ensure results would be comparable under similar conditions.
### Table 4.2: Features used in Table 4.1 datasets

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Datasets used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>Number of modified subsystems</td>
<td>Both</td>
</tr>
<tr>
<td>ND</td>
<td>Number of modified directories</td>
<td>Both</td>
</tr>
<tr>
<td>NF</td>
<td>Number of modified files</td>
<td>Both</td>
</tr>
<tr>
<td>Entropy</td>
<td>Distribution of modified code across each file</td>
<td>Both</td>
</tr>
<tr>
<td>LA</td>
<td>Lines of code added</td>
<td>Both</td>
</tr>
<tr>
<td>LD</td>
<td>Lines of code deleted</td>
<td>Both</td>
</tr>
<tr>
<td>LT</td>
<td>Lines of code in a file before the change</td>
<td>Old</td>
</tr>
<tr>
<td>FIX</td>
<td>Whether or not the change is a defect fix</td>
<td>Both</td>
</tr>
<tr>
<td>NDEV</td>
<td>The number of developers that changed the modified files</td>
<td>Old</td>
</tr>
<tr>
<td>AGE</td>
<td>The average time interval between the last and the current change</td>
<td>Old</td>
</tr>
<tr>
<td>NUC</td>
<td>The number of unique changes to the modified files</td>
<td>Old</td>
</tr>
<tr>
<td>EXP</td>
<td>Developer experience</td>
<td>Old</td>
</tr>
<tr>
<td>REXP</td>
<td>Recent developer experience</td>
<td>Old</td>
</tr>
<tr>
<td>SEXP</td>
<td>Developer experience on a subsystem</td>
<td>Old</td>
</tr>
<tr>
<td>DAY</td>
<td>The day of the week</td>
<td>New</td>
</tr>
<tr>
<td>HOUR</td>
<td>An integer between 0 and 23</td>
<td>New</td>
</tr>
<tr>
<td>WORDS</td>
<td>The number of words in a commit message</td>
<td>New</td>
</tr>
</tbody>
</table>

#### 4.2 Experimental Setup

All scripts for the experiments were implemented in Python 3. The Scikit-learn library was used to create, train and evaluate machine learning models such as logistic regression, random forest, Adaboost and K-nearest neighbours. The XGBoost model was provided under a separate python package \(^2\). The Pandas library was used for preprocessing the datasets and Matplotlib was used for all graphs and visualisations. An open-source implementation of the self-training algorithm was used in experiments for the second research question \(^3\). When generating random floating point numbers, the random library was used with a seed of 42. All scripts were executed on high-end build servers provided at King.

#### 4.3 Experiments

This section describes the exact methods used to answer the four research questions mentioned in the introduction.


\(^3\)https://github.com/tmadl/semisup-learn/blob/master/frameworks/SelfLearning.py
4.3.1 Experiments for RQ1

For this question, the performance of five classifiers was evaluated using 10-fold cross validation. The datasets used in the evaluation are the six open source projects from table 4.1. Due to limitations in time, it was not possible to reimplement models from previous research and apply them on the new datasets.

Classification Models Used

The chosen models were logistic regression (LR), random forest (RF), AdaBoost (ADA), XGBoost (XGB) and K-nearest neighbours (KNN). Logistic regression was chosen in order to observe how linear classification models perform at this task. Random forest was chosen as tree based methods tend to outperform linear models for software defect prediction. AdaBoost and XGBoost were chosen to compare boosting methods with bagging methods such as random forest. Finally, K-nearest neighbours was selected to have another non-parametric model that did not use decision trees. No approaches based on artificial neural networks (ANN) were used because they require a lot of data to perform well. If the amount of data supplied to an ANN is not sufficient, one would get a false impression that these models perform poorly overall at JIT defect predictions.

Parameters for Classification Models

Logistic regression used the \texttt{lbfgs} solver and 5000 maximum iterations. Random forest used 100 estimators and K-nearest neighbours used $k = 5$ for the number of neighbours. All other parameters for these models used the default values provided by the scikit-learn package in Python. All classifiers had their classification threshold set to 0.5. As binary classifiers output a probability between 0 and 1, a threshold of 0.5 indicates that any value above 0.5 becomes classified as \textit{risky} and any value below it is labelled as \textit{not risky}.

Performance Metrics Used

When evaluated on the test set, the performance metrics that were measured were precision, recall and F1-score. As a baseline, all models are compared to a random model which outputs each possible label with equal probability.

4.3.2 Experiments for RQ2

In order to answer this question, the performance of various classifiers must be compared when they are trained in a supervised manner as opposed to when using the self training algorithm (see 2.1.6). The specified null hypothesis for this experiment would be:

$H_0$: "The self-training algorithm does not lead to a change in performance compared with the base performance of five specified supervised algorithms when applied to Just-In-Time defect predictions."

Evaluating Self-Training Using Cross Validation

The evaluation procedure involves providing supervised models and their self-trained counterpart with training data and evaluating their performance on test data. Every
commit in each dataset is labelled so in order to simulate a setting where a model has access to labelled and unlabelled data, a portion of labels must be removed. Although the supervised classifier will only be able to train on the labelled instances, the semi-supervised classifier can make use of labelled and unlabelled instances. Both classifiers are ensured to train on the exact same set of labelled training data and are evaluated.

Algorithm 3 is used repeatedly when measuring performance in each of the experiments for this research question. The models used in the experiments for this research question are identical to those used for research question 1. In the algorithm below, $D$ is a dataset, $M \in \{LR, RF, ADA, XGB, KNN\}$ is the model type, $I$ is the maximum number of iterations the self-training algorithm runs for, $C$ is the confidence threshold for the self-training algorithm and $U$ is the fraction of training data that will be made unlabelled. In order to verify the specified null hypothesis above, three experiments were performed. This involved a broad comparison of various models as well as a more in-depth analysis by using the following experiments.
Algorithm 3 Evaluates the performance of semi-supervised vs supervised classifiers

1: procedure Evaluation(D, M, I, C, U)
2: MetricSet = {Accuracy, Precision, Recall, F1-score}
3: Preprocess dataset D
4: Randomly shuffle and partition the dataset D into k partitions
5: for i = 1 . . . k do
6: \( D_{test} \) := the \( i \)th partition of \( D \).
7: \( D_{train} \) := the union of all other partitions
8: Instantiate untrained models \( f, f' \) of type \( M \)
9: Train \( f \) on \( D_{train} \) using supervised learning
10: Partition \( D_{train} \) into \( D_{train}^{L}, D_{train}^{U} \)
11: \( X_L, y_L \) is obtained from \( D_{train}^{L} \)
12: \( X_U \) is obtained from \( D_{train}^{U} \)
13: Run Self-Train(\( X_L, y_L, X_U, I, C, f' \)) to train \( f' \) (see algorithm 1)
14: Evaluate \( f, f' \) on \( D_{test} \) and record various performance metrics
15: Discard the models \( f, f' \)
16: for metric in MetricSet do
17: Compute mean, standard deviation of metric

Evaluation for several models

This experiment involves running algorithm 3 for each of the five classification models and for each of the nine datasets. The number of iterations for self-training was set to 20, the percentage of labelled data was 20% and the confidence threshold for the self-training algorithm was 0.8.

Varying the percentage of labelled data

Here, algorithm 3 ran on the C1 dataset using a random forest model with 100 estimators. The percentage of labelled data was varied from 10% to 100% with a step size of 10%. The confidence thresholds used were 0.80, 0.85 and 0.90 and the number of max iterations for self-training was set to 20. The purpose of this experiment was to explore the point at which unlabelled data becomes useful and to see the trade-off between performance and reduction in labelled data.

Varying the number of iterations for self-training

For this experiment, algorithm 3 ran on the C1 dataset. The percentage of labelled data was set to 20% and the confidence threshold at 0.9. The number of max iterations was varied from 5 to 100 with a step size of 5. This experiment was of interest because the self-training algorithm may possibly converge and the time spent using the algorithm can be reduced if its limits are known.

4.3.3 Experiments for RQ3

In order to evaluate this question, a case study was set up in order to evaluate the usefulness of these models on developers at King. Below is the proposed methodology for answering such a question objectively. One could define the null hypothesis as
\( H_0 : \) "An online machine learning tool that has trained on data that was labelled using ASZZ does not have an increased ability at spotting risky commits".

Seven developers which were working on the C3 project (see table 4.1) were recruited and all of their latest commits on this project were recorded. The tool described in chapter 3 sent out predictions periodically to each participant so that they would only see predictions for commits that they authored and committed. Each participant was then asked to state if they agreed or disagreed with the prediction. In order to compare the model to a baseline, a random model was used to make predictions on the exact same set of commits that the trained model predicted on. The metrics that were recorded for both the trained and random model were accuracy, precision, recall and F1 score. As the commits produced by developers all happened to have a label of not risky, manual labelling and the SZZ algorithm were used in order to find risky commits for the trained model to predict on.

The theoretical values for the performance metrics can be derived for the random model. Given \( n \) commits where \( r \) have a true label of risky and \( p \) being the probability that the random model outputs the label risky, the random model would then have the following metrics:

\[
TP = p \cdot r \\
TN = (1 - p) \cdot (n - r) \\
FP = p \cdot (n - r) \\
FN = (1 - p) \cdot r
\]

Using the quantities defined above, the accuracy, precision, recall and F1-score for the random model can be determined.

\[
Accuracy = \frac{pr + (1 - p)(n - r)}{pr + p(n - r) + (1 - p)(n - r) + (1 - p)r} = (1 - p) + \frac{r}{n}(2p - 1) \tag{4.5}
\]

\[
Precision = \frac{pr}{pr + p(n - r)} = \frac{r}{n} \tag{4.6}
\]

\[
Recall = \frac{pr}{pr + (1 - p)r} = \frac{pr}{r} = p \tag{4.7}
\]

\[
F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 2 \cdot \frac{pr}{pm + r} \tag{4.8}
\]

Since our random model produced an equal probability for either label (\( p=0.5 \)), it had an accuracy, precision, recall and F1-score of 0.5, \( \frac{r}{n} \), 0.5 and \( \frac{2r}{n + 2r} \) respectively. The value of \( \frac{r}{n} \) is the ratio of buggy commits to total commits and depends upon the dataset. If a trained model has learned what a defective commit is properly, it should be able to have a significant improvement over these values.
4.3.4 Experiments for RQ4

In order to answer this question, qualitative interviews were performed at King which asked engineers from various backgrounds the following questions:

1. Once you realize something is faulty with your software, how do you go about locating the origin of the bug?

2. What existing tools do you use that help you mitigate the risk of introducing bugs or to help detect bugs?

3. What differences do you notice between faulty code and clean code?

4. What do you believe are the top causes of bugs?

The motivation for asking these questions is that it may give an insight as to how software developers can identify code that contains defects. It could potentially also lead to new ideas for features that defect prediction models could rely on. Additionally, it can help to build defect prediction systems so that they benefit developers as much as possible.
Chapter 5

Results

This chapter presents the results of each experiment proposed in chapter 4. In all tables and graphs shown below, the precision, recall and F1 score metrics are expressed as real numbers between 0 and 1 with 3 decimal places.

5.1 Research Question 1 Results

Tables 5.1, 5.2 and 5.3 provide a comparison between the performance of the random forest and XGBoost models with previous benchmarks. The values for the Kamei and Deeper baselines were obtained from the paper Deep Learning for Just-In-Time Defect Prediction [33] and the values for TLEL were obtained from TLEL: A two-layer ensemble learning approach for just-in-time defect prediction [32].

Table 5.1: Comparing precision to past results

<table>
<thead>
<tr>
<th>Project</th>
<th>Kamei</th>
<th>Deeper</th>
<th>TLEL</th>
<th>LR</th>
<th>RF</th>
<th>ADA</th>
<th>XGB</th>
<th>KNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.548</td>
<td>0.556</td>
<td>0.624</td>
<td>0.539</td>
<td>0.583</td>
<td>0.568</td>
<td>0.586</td>
<td>0.497</td>
</tr>
<tr>
<td>Columba</td>
<td>0.487</td>
<td>0.469</td>
<td>0.512</td>
<td>0.490</td>
<td>0.495</td>
<td>0.475</td>
<td>0.493</td>
<td>0.421</td>
</tr>
<tr>
<td>JDT</td>
<td>0.249</td>
<td>0.260</td>
<td>0.293</td>
<td>0.260</td>
<td>0.265</td>
<td>0.272</td>
<td>0.281</td>
<td>0.224</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.124</td>
<td>0.132</td>
<td>0.158</td>
<td>0.129</td>
<td>0.148</td>
<td>0.136</td>
<td>0.149</td>
<td>0.095</td>
</tr>
<tr>
<td>Platform</td>
<td>0.232</td>
<td>0.264</td>
<td>0.314</td>
<td>0.263</td>
<td>0.294</td>
<td>0.284</td>
<td>0.296</td>
<td>0.236</td>
</tr>
<tr>
<td>Postgres</td>
<td>0.504</td>
<td>0.457</td>
<td>0.499</td>
<td>0.497</td>
<td>0.478</td>
<td>0.485</td>
<td>0.503</td>
<td>0.399</td>
</tr>
<tr>
<td>Average</td>
<td>0.357</td>
<td>0.356</td>
<td>0.400</td>
<td>0.363</td>
<td>0.377</td>
<td>0.370</td>
<td>0.385</td>
<td>0.312</td>
</tr>
</tbody>
</table>

Table 5.2: Comparing recall to past results

<table>
<thead>
<tr>
<th>Project</th>
<th>Kamei</th>
<th>Deeper</th>
<th>TLEL</th>
<th>LR</th>
<th>RF</th>
<th>ADA</th>
<th>XGB</th>
<th>KNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.702</td>
<td>0.721</td>
<td>0.759</td>
<td>0.659</td>
<td>0.690</td>
<td>0.695</td>
<td>0.697</td>
<td>0.656</td>
</tr>
<tr>
<td>Columba</td>
<td>0.649</td>
<td>0.670</td>
<td>0.743</td>
<td>0.644</td>
<td>0.682</td>
<td>0.648</td>
<td>0.694</td>
<td>0.636</td>
</tr>
<tr>
<td>JDT</td>
<td>0.661</td>
<td>0.688</td>
<td>0.735</td>
<td>0.667</td>
<td>0.695</td>
<td>0.677</td>
<td>0.702</td>
<td>0.616</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.608</td>
<td>0.682</td>
<td>0.778</td>
<td>0.639</td>
<td>0.738</td>
<td>0.702</td>
<td>0.740</td>
<td>0.685</td>
</tr>
<tr>
<td>Platform</td>
<td>0.709</td>
<td>0.700</td>
<td>0.775</td>
<td>0.691</td>
<td>0.718</td>
<td>0.713</td>
<td>0.738</td>
<td>0.665</td>
</tr>
<tr>
<td>Postgres</td>
<td>0.602</td>
<td>0.680</td>
<td>0.770</td>
<td>0.646</td>
<td>0.711</td>
<td>0.702</td>
<td>0.707</td>
<td>0.629</td>
</tr>
<tr>
<td>Average</td>
<td>0.655</td>
<td>0.690</td>
<td>0.760</td>
<td>0.658</td>
<td>0.706</td>
<td>0.690</td>
<td>0.713</td>
<td>0.648</td>
</tr>
</tbody>
</table>
Tables 5.1 to 5.3 show that the choice of classification algorithm does make a significant impact on precision, recall and F1 score. Out of the five models chosen to experiment (LR to KNN), XGBoost performs the best overall. K-nearest neighbours on the other hand consistently underperforms when compared to the other models. Tables 5.1 and 5.2 show that in most cases, models tend to achieve higher values for the recall score as opposed to the precision score.

When comparing to the previous benchmarks, decision tree based models such as random forest, Adaboost and XGBoost all outperform the deep learning model Deeper as well as the results obtained by Kamei et al. The benchmark model TLEL on the other hand performs the best out of all models. TLEL consists of two layers of random forest which can explain the higher performance as opposed to a single random forest model.

The hardest project to achieve a high performance on is Mozilla which has the lowest F1 scores. This is because the project’s rate of buggy commits is only 5%. Due to the severe imbalance, sampling techniques like undersampling have to be applied so that these models see roughly the same number of risky commits as not risky commits. However, undersampling deletes samples for the training set which could have been of value.

## 5.2 Research Question 2 Results

For this section, the result of running the evaluation algorithm (see algorithm 3) is summarised in tables and graphs. For some of the experiments, the performance of a random model is provided as an additional baseline in the column RAND. The values for a random model’s performance are obtained by creating and running a model that returns randomised predictions as opposed to using the theoretical value of each metric. This is done to see if the theoretical value matches what is obtained in practice. For the purpose of confidentiality, the precision and F1 score of random models are not included for commercial projects as the percentage of risky commits can be derived using these values.

### 5.2.1 Experimenting with Five Models

Tables 5.4, 5.5 and 5.6 show the precision, recall and F1 score for each model when applied on all nine datasets. Each table cell has two values, one for the supervised model (top row in cell) and another for its self-trained counterpart (bottom row in cell). The values in bold are those where the self-training algorithm made a statistically significant
improvement over its respective base classifier. In all cases, these models manage to achieve higher than random performances which indicate that the models have found some connection between the provided features and whether or not a commit is risky.

**Precision**

When it comes to self-training performance, the self-training algorithm does not appear to improve performance when both classifiers are provided the exact same amount of labelled data. As seen in table 5.4, K-nearest neighbours has the most benefit from self-training as seen for the JDT, Platform, Postgres, C2 and C3 projects. However, even with this increase, KNN is performing worse than the tree based models (RF, ADA, XGB). In most cases, the precision due to self-training is similar or slightly lower than the base performance. Also, the precision on the commercial projects happens to be higher, possibly due to a smaller class imbalance.

**Table 5.4:** Precision for various models and datasets, bold value represents cases where self-training provides a statistically significant improvement over the base classifier

<table>
<thead>
<tr>
<th>Project</th>
<th>LR</th>
<th>RF</th>
<th>ADA</th>
<th>XGB</th>
<th>KNN</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.539</td>
<td>0.583</td>
<td>0.568</td>
<td>0.586</td>
<td>0.497</td>
<td>0.372</td>
</tr>
<tr>
<td></td>
<td>0.531</td>
<td>0.602</td>
<td>0.565</td>
<td>0.592</td>
<td>0.485</td>
<td></td>
</tr>
<tr>
<td>Columba</td>
<td>0.490</td>
<td>0.495</td>
<td>0.475</td>
<td>0.493</td>
<td>0.421</td>
<td>0.312</td>
</tr>
<tr>
<td></td>
<td>0.496</td>
<td>0.491</td>
<td>0.471</td>
<td>0.495</td>
<td>0.434</td>
<td></td>
</tr>
<tr>
<td>Jdt</td>
<td>0.260</td>
<td>0.265</td>
<td>0.272</td>
<td>0.281</td>
<td>0.224</td>
<td>0.143</td>
</tr>
<tr>
<td></td>
<td>0.261</td>
<td>0.258</td>
<td>0.272</td>
<td>0.266</td>
<td>0.239</td>
<td></td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.129</td>
<td>0.148</td>
<td>0.136</td>
<td>0.149</td>
<td>0.095</td>
<td>0.052</td>
</tr>
<tr>
<td></td>
<td>0.137</td>
<td>0.146</td>
<td>0.136</td>
<td>0.137</td>
<td>0.097</td>
<td></td>
</tr>
<tr>
<td>Platform</td>
<td>0.263</td>
<td>0.294</td>
<td>0.284</td>
<td>0.296</td>
<td>0.236</td>
<td>0.147</td>
</tr>
<tr>
<td></td>
<td>0.256</td>
<td>0.290</td>
<td>0.284</td>
<td>0.269</td>
<td>0.243</td>
<td></td>
</tr>
<tr>
<td>Postgres</td>
<td>0.497</td>
<td>0.478</td>
<td>0.485</td>
<td>0.503</td>
<td>0.399</td>
<td>0.252</td>
</tr>
<tr>
<td></td>
<td>0.503</td>
<td>0.491</td>
<td>0.486</td>
<td>0.495</td>
<td>0.429</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>0.532</td>
<td>0.639</td>
<td>0.651</td>
<td>0.657</td>
<td>0.566</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td><strong>0.563</strong></td>
<td>0.641</td>
<td>0.651</td>
<td>0.651</td>
<td>0.579</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>0.495</td>
<td>0.605</td>
<td>0.614</td>
<td>0.623</td>
<td>0.533</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.499</td>
<td>0.613</td>
<td>0.614</td>
<td>0.623</td>
<td><strong>0.548</strong></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>0.565</td>
<td>0.644</td>
<td>0.663</td>
<td>0.667</td>
<td>0.583</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td><strong>0.579</strong></td>
<td><strong>0.653</strong></td>
<td>0.663</td>
<td>0.666</td>
<td><strong>0.594</strong></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.419</td>
<td>0.461</td>
<td>0.461</td>
<td>0.473</td>
<td>0.395</td>
<td>0.247</td>
</tr>
<tr>
<td></td>
<td>0.425</td>
<td>0.465</td>
<td>0.460</td>
<td>0.466</td>
<td>0.405</td>
<td></td>
</tr>
</tbody>
</table>

**Recall**

For the commercial projects, there is a large difference between logistic regression’s performance and all other classifiers. In table 5.5, logistic regression only has a recall of 0.485 on the C1 project whereas the recall for tree based methods (RF, ADA and XGB) varies from 0.778 to 0.804. Furthermore, the recall score achieved by logistic regression for the C1 and C2 datasets is worse than a random model. This phenomenon does not
appear to be occurring for the six open source projects (Bugzilla to Postgres). This could be due to the fact that the set of features supplied in the commercial datasets are not the same as those in the open source datasets.

**Table 5.5:** Recall for various models and datasets, bold value represents cases where self-training provides a statistically significant improvement over the base classifier

<table>
<thead>
<tr>
<th>Project</th>
<th>LR</th>
<th>RF</th>
<th>ADA</th>
<th>XGB</th>
<th>KNN</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.659</td>
<td>0.690</td>
<td>0.695</td>
<td>0.697</td>
<td>0.656</td>
<td>0.508</td>
</tr>
<tr>
<td></td>
<td>0.652</td>
<td>0.653</td>
<td>0.679</td>
<td>0.667</td>
<td>0.686</td>
<td></td>
</tr>
<tr>
<td>Columba</td>
<td>0.644</td>
<td>0.682</td>
<td>0.648</td>
<td>0.694</td>
<td>0.636</td>
<td>0.530</td>
</tr>
<tr>
<td></td>
<td>0.645</td>
<td>0.696</td>
<td>0.647</td>
<td>0.680</td>
<td>0.624</td>
<td></td>
</tr>
<tr>
<td>Jdt</td>
<td>0.667</td>
<td>0.695</td>
<td>0.677</td>
<td>0.702</td>
<td>0.616</td>
<td>0.508</td>
</tr>
<tr>
<td></td>
<td>0.656</td>
<td>0.690</td>
<td>0.677</td>
<td>0.703</td>
<td>0.613</td>
<td></td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.639</td>
<td>0.738</td>
<td>0.702</td>
<td>0.740</td>
<td>0.685</td>
<td>0.504</td>
</tr>
<tr>
<td></td>
<td>0.610</td>
<td>0.703</td>
<td>0.702</td>
<td>0.715</td>
<td>0.692</td>
<td></td>
</tr>
<tr>
<td>Platform</td>
<td>0.691</td>
<td>0.718</td>
<td>0.713</td>
<td>0.738</td>
<td>0.665</td>
<td>0.500</td>
</tr>
<tr>
<td></td>
<td><strong>0.707</strong></td>
<td>0.713</td>
<td>0.715</td>
<td>0.740</td>
<td><strong>0.680</strong></td>
<td></td>
</tr>
<tr>
<td>Postgres</td>
<td>0.646</td>
<td>0.711</td>
<td>0.702</td>
<td>0.707</td>
<td>0.629</td>
<td>0.512</td>
</tr>
<tr>
<td></td>
<td>0.618</td>
<td>0.684</td>
<td>0.703</td>
<td>0.685</td>
<td>0.590</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>0.485</td>
<td>0.778</td>
<td>0.804</td>
<td>0.789</td>
<td>0.675</td>
<td>0.499</td>
</tr>
<tr>
<td></td>
<td><strong>0.571</strong></td>
<td>0.771</td>
<td>0.804</td>
<td>0.783</td>
<td><strong>0.701</strong></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>0.477</td>
<td>0.741</td>
<td>0.752</td>
<td>0.758</td>
<td>0.652</td>
<td>0.500</td>
</tr>
<tr>
<td></td>
<td>0.480</td>
<td>0.746</td>
<td>0.752</td>
<td>0.767</td>
<td><strong>0.677</strong></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>0.569</td>
<td>0.771</td>
<td>0.806</td>
<td>0.813</td>
<td>0.703</td>
<td>0.497</td>
</tr>
<tr>
<td></td>
<td><strong>0.669</strong></td>
<td><strong>0.797</strong></td>
<td>0.806</td>
<td>0.821</td>
<td><strong>0.725</strong></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.609</td>
<td>0.725</td>
<td>0.722</td>
<td>0.738</td>
<td>0.657</td>
<td>0.507</td>
</tr>
<tr>
<td></td>
<td><strong>0.623</strong></td>
<td>0.717</td>
<td>0.721</td>
<td>0.729</td>
<td><strong>0.665</strong></td>
<td></td>
</tr>
</tbody>
</table>

**F1 Score**

Table 5.6 shows that that the commercial projects have much higher F1 scores, for the random forest, AdaBoost and XGBoost these scores are close to 0.70. For K-nearest neighbours, the self-training algorithm provides an improvement for all three commercial projects.
Table 5.6: F1 score for various models and datasets, bold value represents cases where self-training provides a statistically significant improvement over the base classifier.

<table>
<thead>
<tr>
<th>Project</th>
<th>LR</th>
<th>RF</th>
<th>ADA</th>
<th>XGB</th>
<th>KNN</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.592</td>
<td>0.631</td>
<td>0.624</td>
<td>0.635</td>
<td>0.564</td>
<td>0.423</td>
</tr>
<tr>
<td></td>
<td>0.584</td>
<td>0.625</td>
<td>0.615</td>
<td>0.624</td>
<td>0.566</td>
<td></td>
</tr>
<tr>
<td>Columba</td>
<td>0.554</td>
<td>0.572</td>
<td>0.547</td>
<td>0.573</td>
<td>0.505</td>
<td>0.385</td>
</tr>
<tr>
<td></td>
<td>0.557</td>
<td>0.573</td>
<td>0.544</td>
<td>0.567</td>
<td>0.509</td>
<td></td>
</tr>
<tr>
<td>Jdt</td>
<td>0.373</td>
<td>0.383</td>
<td>0.387</td>
<td>0.401</td>
<td>0.328</td>
<td>0.221</td>
</tr>
<tr>
<td></td>
<td>0.372</td>
<td>0.375</td>
<td>0.387</td>
<td>0.385</td>
<td>0.343</td>
<td></td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.215</td>
<td>0.246</td>
<td>0.228</td>
<td>0.248</td>
<td>0.168</td>
<td>0.094</td>
</tr>
<tr>
<td></td>
<td>0.223</td>
<td>0.242</td>
<td>0.227</td>
<td>0.230</td>
<td>0.169</td>
<td></td>
</tr>
<tr>
<td>Platform</td>
<td>0.381</td>
<td>0.417</td>
<td>0.406</td>
<td>0.422</td>
<td>0.348</td>
<td>0.226</td>
</tr>
<tr>
<td></td>
<td>0.376</td>
<td>0.412</td>
<td>0.406</td>
<td>0.395</td>
<td>0.358</td>
<td></td>
</tr>
<tr>
<td>Postgres</td>
<td>0.561</td>
<td>0.571</td>
<td>0.573</td>
<td>0.587</td>
<td>0.488</td>
<td>0.337</td>
</tr>
<tr>
<td></td>
<td>0.554</td>
<td>0.571</td>
<td>0.574</td>
<td>0.574</td>
<td>0.496</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>0.507</td>
<td>0.702</td>
<td>0.720</td>
<td>0.717</td>
<td>0.615</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.567</td>
<td>0.700</td>
<td>0.720</td>
<td>0.711</td>
<td>0.634</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>0.486</td>
<td>0.666</td>
<td>0.676</td>
<td>0.684</td>
<td>0.587</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.489</td>
<td>0.673</td>
<td>0.676</td>
<td>0.688</td>
<td>0.606</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>0.567</td>
<td>0.702</td>
<td>0.727</td>
<td>0.733</td>
<td>0.637</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.621</td>
<td>0.718</td>
<td>0.727</td>
<td>0.735</td>
<td>0.653</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.471</td>
<td>0.543</td>
<td>0.543</td>
<td>0.556</td>
<td>0.471</td>
<td>0.318</td>
</tr>
<tr>
<td></td>
<td>0.483</td>
<td>0.543</td>
<td>0.542</td>
<td>0.545</td>
<td>0.482</td>
<td></td>
</tr>
</tbody>
</table>

5.2.2 Varying the Percentage of Labelled Data

This experiment was carried out on two datasets, C1 and JDT to investigate the generalisability of the results. The C1 and JDT datasets were both chosen because they have nearly the same number of commits. Additionally, the C1 dataset is a commercial project whereas the JDT project is open source. There are two graphs for each metric and each graph shows the performance of a base model and three models trained using the self-training algorithm but with different confidence thresholds. When the fraction of labelled data is 1.0, all four models are identical as there is no unlabelled data to train on.
Figure 5.1: Self-training precision on JDT (top) and C1 (bottom) when varying fraction of labelled data

Figure 5.1 shows that the precision of all four models is rather similar and it was found that the models do not statistically outperform each other. What is of interest is that the self-trained classifiers achieve similar accuracy when the portion of labelled data is 10% (hence 90% being unlabelled) when compared to the accuracy at 100%. The chosen confidence values do not appear to make a significant difference over one another. The precision values tend to be more similar in the JDT project but experience more fluctuations for the C1 project.

Figure 5.2: Self-training recall on JDT (top) and C1 (bottom) when varying fraction of labelled data
For the C1 project, figure 5.2 shows that the recall of the self-trained classifiers is much lower than that of the base classifier when the fraction of labelled data is between 0.1 and 0.3. However after 0.3, the classifiers recover and achieve similar performance.

**5.2.3 Varying the Iterations for Self-Training**

![Figure 5.3: Self-training F1 score on JDT (top) and C1 (bottom) when varying fraction of labelled data](image)

![Figure 5.4: Self-training precision and recall on JDT (top) and C1 (bottom) when varying iterations](image)
As seen in figure 5.4, the precision and recall of the self-trained model does not appear to be affected by the number of iterations. This could indicate that the algorithm stops adding new samples to its training set early on.

5.3 Research Question 3 Results

This section shows the results of the case study that was performed (see section 4.3.3) using seven members of the C3 project. The model used in this experiment was a random forest model with 100 estimators. This model trained entirely on historical commits for the C3 project and did not use the self-training algorithm. A total of 27 predictions with a label of not risky were sent to the developers such that each developer receives a prediction for a commit they created. Then, 20 commits with a label of risky were also found using manual inspection of commits within the C3 project. The confusion matrix (see table 5.7) highlights the performance of this model.

Table 5.7: Confusion matrix for trained classifier making live predictions

| Predicted label | True label | | | |
|-----------------|------------|-------------|-----------|
| Risky           | Risky      | TP = 6      | FP = 5    | \(P' = 11\) |
|                 | Not Risky  | FN = 14     | TN = 22   | \(N' = 36\) |
|                 | Total      | \(P = 20\) | \(N = 27\) |

Table 5.8: Performance of trained vs random model

<table>
<thead>
<tr>
<th>Metric</th>
<th>Trained Model</th>
<th>Random Model</th>
<th>Cross validation performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.596</td>
<td>0.500</td>
<td>0.835</td>
</tr>
<tr>
<td>Precision</td>
<td>0.545</td>
<td>&lt;0.545</td>
<td>0.639</td>
</tr>
<tr>
<td>Recall</td>
<td>0.300</td>
<td>0.500</td>
<td>0.778</td>
</tr>
<tr>
<td>F1</td>
<td>0.387</td>
<td>&lt;0.387</td>
<td>0.702</td>
</tr>
</tbody>
</table>

Table 5.8 shows the trained model’s performance metrics which are derived from the confusion matrix above. It also shows the performance of a random model as a worst case baseline. Finally, the column Cross validation performance contains the results obtained from the tables 5.1, 5.2 and 5.3 for the exact same project and model. There is a large discrepancy between the performance achieved when evaluating with datasets labelled by the approximate SZZ as opposed evaluating on manually labelled commits. A reason for this could be that the definition of a risky commit according to developers is different to the definition that the SZZ algorithm considers.
5.4 Research Question 4 Results

For this research question, the conducted interviews asked 10 participants a total of 4 questions. All responses are presented in tables such that there is a table for each question asked and each table row contains the response provided by each developer. For each question asked, the results are also summarized as bar plots where the independent axis contains categories for the responses and the dependent axis is the number of responses that associate with this category. Note that a response can fall into more than one category and that the available categories cover all of the responses. Any table row which contains the value N/A indicates no applicable response was available for this question. The $i^{th}$ interviewee’s response is recorded in the $i^{th}$ row of each table.

5.4.1 Interview topic 1: Identifying the Origin of Bugs

Once you realise something is faulty with your software, how do you go about locating the origin of the bug?

Table 5.9: Responses to interview question 1

<table>
<thead>
<tr>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>I approach the developers working on the particular project with bugs and see if they know what is wrong. Alternatively, I try and guess what the cause is and to inspect the code manually.</td>
</tr>
<tr>
<td>Once a fault occurs, I open up Jenkins and investigate the log which shows any detected failures with the code.</td>
</tr>
<tr>
<td>When a bug occurs in a live product, I rely on assistance from the quality assurance team.</td>
</tr>
<tr>
<td>I usually look at the crash logs to see what caused an issue or reach out to other people who may know what’s gone wrong. It is difficult to find bugs when the software runs without crashing but certain bugs cause unexpected behaviours. It’s more challenging when you don’t know which exact system was affected and need a way to narrow it down.</td>
</tr>
<tr>
<td>I use automation testing and try to see when a buggy commit first appeared and I do this by testing every commit with unit tests.</td>
</tr>
<tr>
<td>I try to reproduce the bug and see what part of the code it belongs to. This is usually done using breakpoints and debugging to see why the error occurs.</td>
</tr>
<tr>
<td>I use lots of log statements when debugging before and after the location of where a potential error or crash occurs.</td>
</tr>
<tr>
<td>I try to reproduce the bug using test cases. Also, I use debugging to step through the code.</td>
</tr>
<tr>
<td>I go through logs to see if the software is behaving normally. I also look at the stack trace as well as previous commits and compare the difference using git diff. It’s harder to detect a bug if no crash occurs and the software appears to be running normally.</td>
</tr>
<tr>
<td>I tend to use the debugger and look at stack traces to see what’s wrong. If the program has incorrect behaviour but doesn’t crash, I check that individual variables have the correct values.</td>
</tr>
</tbody>
</table>

The four categories of responses are reach out to other members, inspect logs, detect bug in unit testing and manually inspect code with debugger. The category reach out...
to other team members considers responses where verbal communication was required between colleagues, this can include talking to people who do not work with software development. The category inspect logs includes looking at stack traces other types of logs. The category detect bug in unit testing is for responses which involved finding bugs through test cases. The category manually inspect code with debugger is for explicitly setting breakpoints within code and using a debugging tool.

Half of the developers agreed that manually inspecting code is a good technique to identify bugs.

![Responses to Question 1](image)

**Figure 5.5**: Responses for interview question 1 categorized

### 5.4.2 Interview topic 2: Tools Used to Identify and Reduce Bugs

*What existing tools do you use that help you mitigate the risk of introducing bugs or to help detect bugs?*
### Table 5.10: Responses to interview question 2

<table>
<thead>
<tr>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>I use Trello for planning and communication to ensure people are coordinated.</td>
</tr>
<tr>
<td>I use test environments which try to replicate the environment that a live system uses without having to make changes to the live system. It allows you to see if given software is working well for various releases and dependency versions.</td>
</tr>
<tr>
<td>I rely on automated unit testing and crash tools that show a stack trace of function calls that failed. I also rely on analytic tools which show the modules that failed. Finally, I use continuous integration servers for regular testing.</td>
</tr>
<tr>
<td>I use GitHub to look at differences between commits and understand how they altered the behaviour of code. I also use CppCheck to perform a static analysis of C++ code. Most of my development is done in an integrated development environment (IDE) that provides warnings for common errors. I also use tools that can visualise memory usage to spot memory leaks and linting tools to see if methods are deprecated.</td>
</tr>
<tr>
<td>I use tools that perform static checks, for example verifying that JSON and XML files are valid. I also use GitHub to do pull requests when reviewing code as well as relying on unit tests and manual testing.</td>
</tr>
<tr>
<td>I do frequent unit testing for the game logic and automatic testing that plays the games to ensure that features work during execution.</td>
</tr>
<tr>
<td>N/A</td>
</tr>
<tr>
<td>I rely on unit testing.</td>
</tr>
<tr>
<td>I use the Git merge tool and IDE hints.</td>
</tr>
<tr>
<td>I use tools within the IDE that highlight potential errors like SonarLint.</td>
</tr>
</tbody>
</table>

![Responses to Question 2](image)

**Figure 5.6**: Responses for interview question 2 categorized

The four specified categories are communication tools, version control tools, automated testing and static analysis of code.
### 5.4.3 Interview topic 3: Differences Between Clean and Faulty Code

*What differences do you notice between faulty code and clean code?*

**Table 5.11:** Responses to interview question 3

<table>
<thead>
<tr>
<th>Responses</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy code tends to be confusing for other developers to understand, its files are too large and has a lot of dependencies on other files. Also, it’s very likely for code to have bugs if there are a lot of nested loops because it makes it more complex for the developer to comprehend.</td>
<td></td>
</tr>
<tr>
<td>In risky code, it is easy to spot if a certain coding practice has been misused such as pointers in C++. It’s difficult to tell if the code will function correctly if the function calls become very deep.</td>
<td></td>
</tr>
<tr>
<td>High code complexity such as nested loops and lots of if statements make it hard to understand the code and leads to more bugs.</td>
<td></td>
</tr>
<tr>
<td>Code that is messy such as having nested loops and being hard to understand tends to have more bugs. Sometimes intuition tells us if certain code is written in an improper manner.</td>
<td></td>
</tr>
<tr>
<td>Buggy code tends to have hardcoded values and contains methods that fail to generalise for all possibilities.</td>
<td></td>
</tr>
<tr>
<td>Code tends to be more buggy when the files are long and when function and variable names are difficult to understand. When code is not easily readable, it becomes harder to debug. Also, buggy code tends to overuse the nesting of loops and if statements.</td>
<td></td>
</tr>
<tr>
<td>If the code is too complex with lots of if else statements and for loops, it becomes more complex to understand and harder for programmers to spot bugs.</td>
<td></td>
</tr>
<tr>
<td>Buggy code is difficult to understand and is poorly documented or the documentation is out of date. It also has a lot of nested loops or nested if statements.</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.7: Responses for interview question 3 categorized

The five categories for this response are high complexity, failing to be modular, poor coding standards, solutions not general and too specific and files too large. High complexity corresponds to code where it is difficult for a developer to understand what the code does. Poor coding standards can involve practices such as having little to no comments.

5.4.4 Interview topic 4: Top Causes for Bugs

What do you believe are the top causes of bugs?
Table 5.12: Responses to interview question 4

<table>
<thead>
<tr>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A lack of resources such as time and money causes development to get rushed.</td>
</tr>
<tr>
<td>Files that are too large because it’s easier for defects to go unnoticed. Also having code that is poorly organised due to too many dependencies and non modular code.</td>
</tr>
<tr>
<td>Legacy code that is difficult to change as the fix could lead to more problems than before. Some developers create new code with the assumption that legacy code works properly because it has been given lots of time to fix its bugs.</td>
</tr>
<tr>
<td>Tight deadlines which caused work to be rushed and puts pressure on developers. Sometimes developers put a lot of pressure on themselves, especially when trying to get a new feature released before a holiday. Refactoring code can also cause a lot of bugs because when refactoring, there is a constraint to ensure the changes don’t break the current system.</td>
</tr>
<tr>
<td>When you get frequently interrupted when doing a task and forget crucial details about the task you were working on.</td>
</tr>
<tr>
<td>Logical errors where the developer hasn’t thought of all the possible cases.</td>
</tr>
<tr>
<td>Tight deadlines which cause developers to write quick solutions that work for now but not in the long term. High complexity within code is also a large contributing factor to causing bugs as well as having developers write code for tasks they are inexperienced with.</td>
</tr>
<tr>
<td>When code becomes too complex and it’s hard to keep track of everything.</td>
</tr>
<tr>
<td>Temporary solutions that will work for now but not in long term. These solutions fail to generalise and causes technical debt to build up. Also, bugs can arise when files are too long and when code is not modular.</td>
</tr>
<tr>
<td>When there’s a lack of frequent testing and when you have incorrect assumptions about the expected behaviour of a new feature.</td>
</tr>
</tbody>
</table>

Poorly organized code is the category contains causes such as failing to be modular, having poor documentation or having too many dependencies for example. The category cognitive state of developers indicates factors such as stress, external pressure to meet deadlines as well as losing focus as causes of bugs. Temporary solutions are for responses where the solution fails to work in the long term.
Figure 5.8: Responses for Question 4 categorized
Chapter 6

Discussion

6.1 Interpreting Results of Conducted Experiments

In this chapter, the results of the previous chapter are analysed with respect to the individual research questions.

6.1.1 RQ 1: Performance across multiple models

Table 5.3 shows that the random forest and XGBoost models achieve higher F1 scores than the deep learning model Deeper. The benefits of decision tree based models is that they are interpretable, the model is able to tell a user which features are most responsible for risky commits. Deep learning models on the other hand have difficulty with this due to their complexity. While random forest and XGBoost did outperform Deeper, they do not manage to outperform TLEL which is a model that consists of stacking multiple random forests together. What this may suggest is that models that rely on more ensemble learning techniques would be best for approaching defect prediction.

6.1.2 RQ 2: Practicality of the Self-Training Algorithm

When performing a broad comparison of the self-training algorithm's performance across various datasets and models (see section 5.2.1), it is evident that the self-training algorithm does not outperform a purely supervised approach. Tables 5.4 to 5.6 show that in only a minority of cases does self-training outperform the base classifiers. Self-training does provide a boost in performance for K-nearest neighbours, however there is little justification for choosing that algorithm in practice because the tree based methods (random forest, Adaboost and XGBoost) outperform it in all situations. In this experiment, both the supervised and semi-supervised classifiers had the exact same labelled data. Despite the fact that the semi-supervised model has access to more samples in the form of unlabelled data, the additional data did not provide a benefit in this respect.

Using Self-training for Reducing Labelled Data

Despite the previous observation, the self-training algorithm does have the benefit of reducing the required labelled data. Take figure 5.1 for example, even as the percentage of labelled data decreases, the precision does not vary greatly. This graph seems to indicate that instead of using only labelled samples, one could achieve a similar accuracy
if only 10% of this labelled data is used and complimenting it with unlabelled data. For example, if one had the goal of collecting 100,000 labelled samples, one could instead collect 10% of that labelled amount as well as 90,000 unlabelled instances and only experience a slight decrease in precision. One should still be cautious about the chosen size of for the percentage of labelled data. Figure 5.2 shows that using 10% of labelled data can reduce the recall from 0.78 to 0.72. However after the fraction of labelled data is 30%, the recall of all four classifiers is rather close.

Given that the self-training algorithm is viable for reducing the required labelled data, one would be interested in how it is affected by its parameters. When observing the graphs for precision, recall and F1 score (see figures 5.1, 5.2 and 5.3), there does not appear to be a clear confidence threshold that is better. The reason for this could be that these thresholds are rather close so a drop in performance may be more noticeable when the threshold is much lower (like 0.5).

Effect of Varying Iterations

When observing how iterations affect the self-training algorithm, it appears that running the algorithm for more iterations does not generally lead to improvements in performance. This can be seen in figure 5.4 where there is little change across iterations. This is probably due to the fact that the implementation of the self-training algorithm that was used does not remove unlabelled instances once their predicted label is high enough. Using the algorithm this way can be ideal because the self-training algorithm can be quite time consuming and one can make use of its benefits while only using a few iterations.

6.1.3 RQ 3: Just-in-time Defect Prediction in Practice

The results in table 5.8 show that the trained model has increased performance over a random model for accuracy and precision. It also shows that the trained model does not outperform the random model for recall and only has a slightly higher F1 score. However, the main concern is the large discrepancy between the theoretical cross validation performance and the performance seen in practice. The percentage of bugs spotted (recall) is 159% higher on cross validation than when evaluated in practice. One reason may be that cross validation is insensitive to the temporal order of commits. Cross validation can train on commits that occurred after commits in the test set and this has been shown to exaggerate performance metrics as opposed to a time sensitive evaluation [28].

Another explanation could be that the labels that the ASZZ algorithm provides could differ greatly from the labels that a developer would suggest when manually labelling the commits. Another possibility is that the model has correctly identified risky commits but it is not apparent to the developer that the commit actually contains a defect.

An alternative methodology for evaluating the third research question was initially proposed but could not be carried out due to lack of resources. This experiment would involve setting up a case study where a large group of software developers would be recruited and be randomly assigned to a control and a treatment group. The treatment group would have access to a properly trained classifier whereas the control group would be receiving predictions from a random model that simply has a 50% chance of producing the labels risky or not risky. The purpose of having a control group would be to investigate the possibility of a placebo effect. There may be a risk that developers have a positive or negative bias towards a trained model and a treatment group could show evidence of this.
6.1.4 RQ 4: Interpreting the Results of the Interviews

How Developers Find Bugs

The first question asked in the interviews can provide some useful insights into the methods that developers use to find bugs. The responses showed that one source of information developers use is logs for particular errors as well as how well a software build did with respect to its unit tests. This is a source of information that many JIT defect prediction models lack as their features are solely from commit metadata. Defect prediction models could include metrics from these other sources of information such as if the commit caused a build to fail, the types of error thrown as well as the percentage of code test coverage. Although code test coverage is computed at a file level, one could find all files that a commit touched, compute the coverage for each file and provide the mean value as a feature for the model.

One common response from the developers was that they manually inspect code by using a debugging tool and setting up breakpoints. Stepping through the code allows one to investigate if the program is running as expected. However, it can be difficult at times for a developer to know what part of the code to investigate, especially if the software is producing bugs without crashing (see table 5.9, row 4). Assuming a software defect prediction model can accurately identify risky commits, their predictions could be used so that a developer would have to spend less time searching for the source of the defect.

Tools Developers Use to Deal With Bugs

One standard practice used for mitigating the risk of bugs is using GitHub for frequent code reviews and for merging new feature branches to the master branch. Version control systems are also used for the developer to understand the changes made using the `diff` command. The two other prominent methods used are tools that can automate unit testing (such as continuous integration servers) as well as tools that perform a static analysis of code when being developed.

Static Analysis of Risky Code

As seen from figure 5.6, tools that perform a static analysis of code were frequently mentioned in responses. What may be of interest is if it is possible to incorporate the results of these analyses into a defect prediction model. Although it may be theoretically possible to do so, there are some challenges associated with this. First of all, a commit-level defect prediction model requires that each commit includes the same information. This implies that one would have to run these tools for every revision of a project and this would be very intensive on resources. Another issue is how to extract the information of the static analysis with respect to the lines changed by a commit rather than for an entire file or submodule. For example, if a commit changed only the condition in an `if`-statement, should the static analysis tool consider just that condition or the entire `if`-clause?

Instead of including static code analysis into a defect prediction model itself, one could use it as a complementary tool. For example, a defect prediction model could flag risky commits and then automatically run the analysis on files that the commit touched. Some static analysis tools such as linting tools even suggest fixes that would resolve the issue. This would allow one to create an automated system that continuously repairs
risky code without developers having to intervene. Typically, software defect prediction models are designed to provide predictions to developers. However, developers may find a prediction such as risky or not risky as too difficult to interpret because they usually do not highlight what is incorrect within a risky commit [21]. Developers may also be less tolerant to mistakes made by a defect prediction model as false positives would take up time with no reward. Automated systems on the other hand do not care about how much time they spend analysing risky commits, especially if there is a large amount of false positives. A similar example that uses defect prediction models this way is CLEVER, the defect prediction tool created at Ubisoft. CLEVER does not send the initial predictions to the developers and performs an additional analysis of the code added by the commit.

Clean Code Compared to Risky Code

When attempting to tell the difference between clean and risky code, a common response among developers was that risky code has a high complexity within the code’s structure. This complexity is typically not included in datasets for JIT prediction models. This is because these metrics can only be computed at a method or file level whereas a commit might only change one line within a method. Large files and lots of dependencies on other files are also mentioned as characteristics of risky code. Poor coding standards were another quality of risky code. Bugs occurring due to poor standards are not due to logical errors but due to misuse of standard programming constructs (for example pointers in the C language).

Top Causes of Bugs

As seen from the responses in table 5.12, buggy commits tend to occur often when development is being rushed and due to a lack of testing. This motivates the need for having autonomous testing that can flag risky code accurately as it could reduce the time developers need for resolving bugs and instead have more time to focus on new features. Poorly organised code is also another contributing factor to introducing bugs as making a change in one portion of the codebase could cause another to break.

6.2 General Considerations for Defect Prediction Models

6.2.1 Prioritising Precision or Recall

When creating software defect prediction models in practice, precision and recall are two metrics that are important to consider because of the imbalance of labels. It is important to understand how much precision and recall is required in order for a defect prediction model to be viable in practice. For some developers, receiving too many false positive could deter them from wanting to use the tool because they can spend too much time reviewing commits with no issues and would instead prefer a high precision model. Other developers may want a high recall model so that there is little risk of missing a buggy commit and do not mind spending more time on reviewing their code. Typically, the classification threshold has been set to 0.5 but it should be investigated as to what threshold is most ideal for developers.
6.2.2 Binary Labelling of the Dataset

This report as well as past research has formulated the problem of identifying risky commits as a binary classification problem. Although software developers may be able to identify multiple risk levels for implemented features (such as high, medium, low, none), only two labels are used in our defect prediction datasets. One may ask as to why numerical values are not used as opposed to categorical values. The use of integers or real values for a commit’s risk level is generally avoided because it is difficult to define objectively. Integers could be used when measuring the number of bugs caused and be normalized to a value between 0 and 1. However, if two commits have the same number of bugs caused, this does not necessarily mean that both have the same level of risk. For example, one commit could cause a bug that causes an application to fail whereas another commit only forgets to refresh the user interface with new data in that same application. Both of these bugs would not be considered to have the same risk level but it would be difficult to distinguish this with a numerical value.

Instead, one could use multiple categorical labels such that a partial order over these labels exists. However, even if one could define multiple labels like high, medium, low, the challenge would be to label our datasets in such a way. The SZZ algorithm would not be applicable because it cannot determine the how risky a change is, only if the change resulted in a bug or not. Therefore we resort to only using two labels instead of multiple. However, using only two labels may make individual predictions less easy to interpret for developers. Instead, one could convey multiple risk levels when several commits are predicted together. Given that a build has 100 new commits, one could highlight the percentage of risky commits in that build and assign a discrete category (for example above 80% could have the label very high risk).

6.3 Ethical Concerns

When creating defect prediction models, there are various ethical concerns that need to be addressed to ensure that developers will feel comfortable using these systems and to ensure that datasets are handled carefully.

6.3.1 Discrimination Against Developers

One feature that could have been included in the defect prediction model was the developer that was responsible for writing a commit. This information is readily available within Git commits and one could see how developers produce risky commits across various projects. However, this provides the opportunity for particular developers to be discriminated against when being considered for new projects or responsibilities within an organisation. Some organisations may even use it as an indicator of a developer’s performance and in the worst case, be used to justify a layoff. Even if the author of a commit is not included in the model, anyone observing the prediction would know that a risky commit was created by a certain individual. This is one reason why it may be best for only the author of the commit to see the prediction. Also, the prediction model does not take into account the fact that not all projects are made equally. For example, a very high quality developer may take on more challenging projects and thus create bugs more often because of the nature of their work but that does not imply that they have poor performance.
6.3.2 De-anonymising Datasets

Another concern is when creating datasets for software defect prediction models that may be available publicly. Although one could anonymise each developer by replacing their name with a hash value, it could still be possible to identify the author of a given commit. One related example where de-anonymization has been achieved is with the Netflix challenge. This challenge involved designing recommender systems to recommend movies based on past ratings. The dataset for this challenged contained real ratings from users, however their usernames were anonymised [20]. The issue arose when it was discovered that one could cross reference the ratings in the Netflix dataset with a public dataset such as IMDB. Although the de-anonymization of movie ratings may not necessarily pose a large threat to users, this example shows that the technique can be exploited because of additional public information. So in the context of our datasets, one could use information such as the project worked on, the time of the commit and other features to deduce the original author of a commit.

6.3.3 Handling Commercial Datasets

For commercial datasets, even more care must be taken when presenting various statistics. For example in tables 5.4 and 5.6, the performance of the random classifier can reveal the percentage of risky commits within a project. When referring to equation 4.6, the precision of a random model directly reveals this value. For the F1 score, one could rearrange equation 4.8 and use the value of a random model's recall to retrieve the rate of buggy commits. Even if the information does not accurately represent the true rate of buggy commits (due to false positive introduced by SZZ), future documents presenting this information might use it without context. Therefore anonymising this data is important in order to avoid any potential harm to a company partaking in these studies as well as to establish trust in future collaborations between academia and the industry.

6.4 Threats to Validity

6.4.1 Limitations of the SZZ Algorithm

The ability of a defect prediction model to identify risky commits depends heavily on the correctness of the labels provided by the SZZ algorithm. A limitation with the SZZ algorithm is that it is not exhaustive i.e. there is no formal guarantee that it will identify all bugs in a version control system. This is because the bugs it discovers depend upon the initial bug-fixing commits that were identified. If a developer fails to report these bug-fixing commits in a bug tracking database or if their commit messages do not contain the necessary keywords, SZZ will not regard these commits as bug-fixing. Therefore any commits containing a bug that would have been linked to this bug-fixing commit are not added to the final set.

Another issue with the algorithm is that it produces false positives because it does not understand the different types of changes that can be made within a commit. For example, if a commit includes a comment, the algorithm would consider this commit as a bug given that a bug-fixing commit removes the comment or moves it somewhere else. Even if a comment had incorrect information that could indirectly cause a programmer
to write faulty code, it still does not constitute as a bug as it does not directly affect the program’s execution. The same problem applies when making refactoring changes to a program such as renaming methods. To make the model aware of the type of change, one would have to parse the source code when running the SZZ algorithm to ensure that a change to the source code actually causes a different execution of the program.

6.4.2 Observed Performance in Case Study

When evaluating the performance of a trained model in practice as seen in section 5.3, one risk is that the number of commits for the evaluation were too few. The true observed performance could have become much higher or lower if given many more commits to evaluate on. Additionally, the case study was only done over a period of two weeks and these two weeks could have been a less busy period for the developers where less bugs were produced. Obtaining more commits to predict on is necessary but can be challenging if developers are reluctant to manually label the commits once they receive a prediction.

Another threat to validity is the fact that risky and not risky commits were not obtained using the exact same methods. When obtaining new commits to predict on for the experiments, only non risky commits occurred according to the developers evaluation of each commit. Therefore some risky commits had to be obtained manually by following the steps of the SZZ algorithm and using sound judgment to determine if the change was actually fixing a bug. Finally, the performance of the model could be lower due to the fact that the ASZZ algorithm was used and there is a risk that it generates more false positives in its labelling as opposed to the vanilla SZZ algorithm.

6.4.3 Differences in Results due to Randomness

When comparing current results to previous works, one risk is that the training and test sets used are not identical. Cross validation involves randomly partitioning a dataset and unless the seed used for performing that partition is provided, one must trust that cross validation is providing an accurate description of performance in general.
Chapter 7

Conclusion

In this work, the performance of various well known classifiers was investigated in order to have a better understanding of which models are best suited to Just-in-time defect predictions. It was found that models based on ensemble techniques and decision trees such as TLEL, random forest and XGBoost perform well overall. Additionally, models such as random forest and XGBoost can outperform deep neural networks. The K-nearest neighbours model significantly underperforms compared to the other models.

Next, in order to see if it was possible to reduce the amount of labelled training data, the performance and applications of the self-training algorithm was investigated. It was found that that the self-training algorithm does not have an improved performance over its corresponding supervised model when supplied the same amount of labelled data. It does however have the ability to achieve similar performance to supervised models that use more labelled data. Also, there is no observed benefit of running the self-training algorithm for a large number of iterations. Out of the confidence thresholds 0.8, 0.85 and 0.9, none of them appear to significantly outperform the others.

A defect prediction tool was created and deployed at King in order to evaluate the performance of defect prediction models when their predictions are sent and reviewed by software developers. Although this model could achieve higher accuracy and precision than a random model, there is still a large discrepancy between the cross validation performance and its performance when evaluated by developers.

Finally, to grasp a better understanding of how developers at King identified and dealt with bugs, ten interviews were conducted. The interviews provide some responses that could be of interest for researchers experimenting with defect prediction models as well as anyone who is interested in how bugs are identified and resolved. The interviews found that developers tend to identify bugs through inspecting logs, relying on frequent unit testing, manually inspecting code as well as communicating with other team members. The types of tools used are typically static analysis tools, version control tools such as Git as well as tools that automate testing. According to developers, risky code tends to be difficult to read and comprehend, has too many dependencies on other code and does not follow proper coding practices. The top causes of bugs are suggested to be a lack of time and resources, incorrect assumptions on how code was meant to function as well as code being poorly organized.
7.1 Future Work

This section suggests potential topics that could be explored in the future for the field of software defect prediction.

Case studies

Some more emphasis could be put on evaluating defect prediction models in practice as it is important to understand if these models have a clear benefit for developers. It needs to be understood if it is sufficient for developers to simply know if a commit is risky or if they need exact suggestions for what the commit has done wrong.

Machine learning

New features for defect prediction models could be incorporated using information from sources other than GitHub such as the build status or success rate of tests. A challenge would be to optimize the collection of this data so that it can be available for every commit. One could also delve deeper into semi-supervised learning techniques. Although the self-training algorithm does not lead to improvements in performance, its utility is in the fact that it reduces the need for labelled data. One could investigate more complex techniques such as generative models or SV3M.

ASZZ vs SZZ

A major benefit of the ASZZ algorithm is the fact that it can be used for any project that is structured using Git. However, if the algorithm degrades the quality of data too much, it may not be viable in practice one would be limited to defect prediction models for projects with a bug tracking database. It is worth investigating the limitations of this algorithm, possibly through evaluating its performance on a manually verified dataset.
Bibliography


