Just-In-Time Software Defect Prediction using version control tool based software metrics and source code embeddings

CHRISTOPHER DAHLÉN
Just-In-Time Software Defect Prediction using version control tool based software metrics and source code embeddings

CHRISTOPHER DAHLEN
Abstract

Software development is a multifactorial process. Its complexity has made it challenging to study the circumstances that underlie efficient software development. However, a better understanding of these factors will reduce the long-term costs of software development. Just-In-Time Software Defect Prediction (JIT SDP) is defined as the problem of predicting whether a given piece of software is defective based on data from the development process. It is a growing research field due to its potential of increasing the development efficiency by identifying defective code at an early stage. The objective of this thesis is to test the applicability of Bayesian Neural Networks (BNN) in JIT SDP and to investigate the application of source code embeddings in JIT SDP. The data utilized are two proprietary datasets containing software metrics and source code snippets logged by a version control tool. The results show that the performance of the BNN is analogous with the state-of-the-art model. Unlike the state-of-the-art model the BNN exhibit lower uncertainty in correct predictions than in incorrect predictions. More research is however needed to establish the applicability of BNNs in JIT SDP. Furthermore, the results show that there is a significant improvement in accuracy (6.7%) and in Matthews Correlation Coefficient (13%) when using source code embeddings in addition to the software metrics. This indicates that source code embeddings are useful in JIT SDP.
Sammanfattning

Programutveckling är en multifaktoriell process. Dess komplexitet har gjort det svårt att studera de omständigheter som ligger till grund för effektiv mjukvaruutveckling. Dock kan en bättre förståelse av dessa faktorer minska de långsiktiga kostnaderna för att utveckla mjukvara. Just-In-Time Software Defect Prediction (JIT SDP) definieras som problemet att förutsäga huruvida ett givet program är defekt baserat på data från utvecklingsprocessen. Det är ett växande forskningsområde på grund av dess potential att öka utvecklingseffektiviteten genom att identifiera defekt kod på ett tidigt stadium. Syftet med denna uppsats är att testa tillämpligheten av Bayesian Neural Networks (BNN) i JIT SDP och att undersöka tillämpningen av källkodsinbäddningar i JIT SDP. Data som används är två proprietära dataset som innehåller mjukvarumätvärden och källkodsfragment som är loggade av ett versionshanteringsverktyg. Resultaten visar att BNN:s prestanda är likartad med den i litteraturen föreslagna modellen. Till skillnad från den i litteraturen föreslagna modellen uppvisar BNN lägre osäkerhet vid korrekta förutsägelser än vid felaktiga förutsägelser. Mer forskning behövs dock för att fastställa användbarheten av BNN i JIT SDP. Resultaten visar vidare att källkodsinbäddningar i tillägg till mjukvarumätvärden ger en signifikant förbättring i träffsäkerhet (6.7%) och i Matthews Correlation Coefficient (13%), vilket indikerar att källkodsinbäddningar är användbara i JIT SDP.
## Contents

1 Introduction .................................................. 1  
   1.1 Introduction .............................................. 1  
   1.2 Problem formulation ..................................... 2  
   1.3 Objective ................................................. 3  
   1.4 Research questions ...................................... 3  
   1.5 Research methodology ................................... 3  
   1.6 Outline of the thesis .................................... 4  

2 Related work .................................................. 5  
   2.1 Software metrics .......................................... 5  
      2.1.1 Static metrics ....................................... 5  
      2.1.2 Process metrics ..................................... 6  
   2.2 Software defect prediction ............................... 6  
      2.2.1 File level software defect prediction ................. 7  
      2.2.2 Just-In-Time software defect prediction .............. 8  
   2.3 Vector representation of source code .................... 9  

3 Background .................................................... 11  
   3.1 Random Forest ............................................ 11  
      3.1.1 Uncertainty in Random Forests ....................... 12  
      3.1.2 Two Layer Ensemble Learner ......................... 13  
   3.2 Bayesian Neural Networks ................................ 14  
      3.2.1 Architecture ....................................... 15  
      3.2.2 Loss function ..................................... 15  
      3.2.3 Computing gradients ................................. 16  
      3.2.4 Uncertainty in Bayesian Neural Networks .......... 17  

4 Method ........................................................ 19  
   4.1 Data ....................................................... 19  
   4.2 Process metrics ......................................... 21
Chapter 1

Introduction

1.1 Introduction

Software development is a complex process that depends on a diverse set of factors. Both skill level, time schedule and developer collaboration may affect the way software is created. This apparent complexity has made it challenging to study the factors that underlie software development. However, it is believed that a better understanding of these factors can reduce the long-term cost of software development. Due to the time it takes to put code in production and the monetary risk of having an error find its way into production, methods attempting to model these factors has caught the attention of the software industry. Hence, estimating the quality of a piece of software before deployment or testing can significantly reduce the turn-around time of software development. Moreover, it can also reduce the risk of putting defect code into production. The research in this area, called Software Defect Prediction (SDP), has gained increased attention as the request for efficient software development are increasing. A common way of approaching SDP has been to use metadata from the development process but recently progress has been made by also incorporating information from source code [1, 2]. Using natural language processing techniques to model patterns in source code is a growing research field and can provide value in SDP in terms of increased predictive performance [1]. There are numerous predictive models proposed for the purpose of SDP. Recently neural networks have shown promising results in SDP and related areas [3, 4, 5, 1]. Despite the promise of neural networks their black box nature significantly complicates interpretation. Moreover, they do not provide uncertainty estimates of predictions or parameters, which further limits their applicability. However, Bayesian Neural Networks (BNNs) are able to express uncertainty in
the parameters as well as in the output [6]. The output uncertainty can be used to understand whether the model is under-confident or falsely over-confident in its predictions. These kind of characteristics are particularly valuable in situations where an incorrect decision can have undesired consequences [7]. The importance of interpretable predictions in SDP has been pointed out [8]. In striving towards a faultless software development process without redundant testing or defective software causing serious side effect, the use of uncertainty preserving SDP models can play an important role in increasing the software development efficiency. Developers can then better assess the predictions and gain insights into what parts of the code need additional attention or extensive testing and what parts need less attention. BNNs are therefore potentially useful in developing software with high reliability standards as a tool for predicting software defects.

1.2 Problem formulation

The research is a joint collaboration between KTH and Spotify AB. Spotify provides the data necessary to carry out the experiments conducted in this thesis and expertise on the software development process used within the company. Agile software development is a widely used method for developing software in a dynamic and fast paced manner. The importance of short feedback loops, in terms of the duration from starting one task to moving on to the next, is essential to continuously adapt to a changing software developing environment. At Spotify, agile methods are extensively used. To facilitate the production of high quality software the desired way of work is that developers commit code changes using a version control tool. The developers then wait for peers to review the committed code. When accepted, the code is run through a build procedure with unit tests and integration tests where the end goal is to establish a working software module. If no defects are found during the build procedure the software is deployed. If a defect is detected the process starts over. In software development, a file change is often referred to as a \textit{diff}, which consists of textual differences, \textit{hunks}, between two versions of a source code file. In the context of this thesis a defect is defined as a failed build procedure based on the file changes in version control commits. The build procedure is both time consuming and computationally intensive. The code base is expected to continue to grow over the near future. Therefore, continued practice of this way of working will result in increased costs and decreased developer efficiency. Thus, identifying defects in the software at an early stage in the development process is highly desirable.
1.3 Objective

The objective is to test the applicability of BNNs, trained with Bayes by back-prop [6], in the domain of SDP referred to as Just-In-Time (JIT) SDP. JIT SDP aims to identify defect-inducing file changes. In addition, the application of source code embeddings in JIT SDP is evaluated. To formulate the task as a classification problem, software metrics and source code embeddings from file changes are utilized to predict the outcome of build procedures.

1.4 Research questions

R1. Does a Bayesian Neural Network secure better performance in terms of F1 score, accuracy and Mathews Correlation Coefficient (MCC) compared to the state-of-the-art model in JIT SDP?

R2. Does semantic information extracted from source code file changes improve model performance in JIT SDP?

1.5 Research methodology

In order to formalize the research questions a review of the state-of-the-art literature within SDP and JIT SDP is conducted. The research questions in Section 1.4 were chosen based on findings in the literature and the, at the time, current way of developing software at Spotify. Given the promising results of traditional neural networks in SDP related areas [3][4][5][1] and the request for intelligible predictions in SDP [8] BNNs were chosen as a suitable candidate for JIT SDP. Concurrently with the literature review and collecting the data the steps required for addressing the research questions is formulated. The method is defined by considering findings in related works and Spotify’s requisites. Measures for evaluating the research questions are identified as the method is formalized. Once the initial steps are performed the experiments are realized and evaluated. The results are then discussed in the context of related works. The process is described in Figure 1.1.
1.6 Outline of the thesis

This thesis is organized in 7 sections. Section 1 introduces the problem of JIT SDP. Research within the field is presented in Section 2. The required background is described in Section 3. The method and contribution of the thesis are presented in Section 4. In section 5 the experimental settings and the results are presented. In section 6 the method and the results are discussed. Furthermore, future work and conclusions are presented in section 7.
Chapter 2

Related work

Various software metrics are presented in Section 2.1. Previous work in SDP are outlined in Section 2.2. In Section 2.3 methods for vector representation of software are described.

2.1 Software metrics

There are multiple ways of measuring and evaluating the quality of software described in the literature. Usually these metrics are grouped into one of the two groups, static code metrics or process metrics [9].

2.1.1 Static metrics

Static code metrics describe characteristics of source code without actually executing it. These metrics are used to analyze the internal structure of a program file and serves as a proxy for the complexity of the code. A well established static code metric introduced by McCabe [10] is the cyclomatic complexity which describes the number of linearly independent paths through the source code represented as a control flow graph.

\[
M = E - N + 2P
\]  

(2.1)

where \( E \) is the number of edges of the graph, \( N \) is the number of nodes of the graph and \( P \) is the number of connected components. Simpler static metrics such as lines of code in a file, number of operators and number of operands have also been reported in the SDP literature. Such metrics are fast to compute and a simple way of assessing the intricacy of a program file.
2.1.2 Process metrics

Process code metrics entails information about the process of creating the software and has also been a popular way of evaluating software quality. The number of process metrics that can be used is large and depends on the size and type of the software project considered. Metrics related to the actual changes of the source code can be utilized in a wide range of software projects but metrics that consider for example the number of developers involved or external developer contribution are not always meaningful or applicable for obvious reasons. The debate whether static metrics or process metrics are better to use as predictors of software defects has been ongoing for years and opinions have diverged on the matter. Although previous studies by Menzies, Greenwald, and Frank [11] and Koru and Liu [12] have shown that static code metrics are useful predictors in SDP, more recent studies by Rahman and Devanbu [9], Hassan [13], and Moser, Pedrycz, and Succi [14] have shown that process metrics are better predictors than static metrics in SDP. In particular, Rahman and Devanbu [9] showed that using process metrics, the bias towards classifying large files as defective decreased. Moreover, Choudhary, Kumar, Kumar, Mishra, and Catal [15] reported an improvement in predictive performance by combining process and static metrics showing that valuable information can be drawn from both categories of metrics.

2.2 Software defect prediction

A systematic review carried out by Li, Jing, and Zhu [4] showed an increasing interest in the field of SDP over the last couple of years. The variety of predictive models used to solve the SDP problem is extensive and up until recently the most commonly used approaches in SDP have been variants of Decision Trees (DT) [16]. Their performance and adequacy for SDP have been proven on multiple open source datasets [8] [17]. However, an emerging trend in SPD is to utilize deep learning techniques. Deep learning has been used to encode semantic relationships in source code and code comments [3] [18] and for classifying defective code using process metrics [4] [5] [17]. SDP is a wide spanning concept with a plethora of variants of input variables and target variables. However, there are two major subcomponents in SDP, namely file level SDP and JIT SDP. Commonly addressed in the literature is file level SDP which aims to identify entire defective files. The closely related problem JIT-SDP which studies defect-inducing changes to a set of files is an interesting problem from an industry perspective since developers can receive immediate
feedback of the quality of their file changes. In addition, less computational power is required since only the actual file changes are considered and not entire files.

2.2.1 File level software defect prediction

The ability of neural networks to capture underlying contextual relationships in source code was reviewed by Allamanis, Barr, Devanbu, and Sutton [19]. In their report on learnable probabilistic models of source code they outline the similarities between source code and natural language. A report by Wang, Liu, and Tan [2] showed that using deep learning techniques on open source data they could improve the state-of-the-art F1 score in file-level SDP by 14.2%. They emphasized that two program files containing different semantics can have static- or process metrics with the same values, making it impossible for a model to distinguish between the files. Instead, semantic structures of the program files were encoded by using Abstract Syntax Trees (AST). An AST converts the source code in a file into a vector of type declarations, i.e. (int, for, string, while, ...). The syntax of the code is then preserved by parent/child relations between the type declarations. The vector was then passed to a Deep Belief Network (DBN) to learn a feature representation of the AST. By utilizing the learnt feature representation of the source code they could better separate files with similar static and process metrics using Naive Bayes- and Logistic regression classifiers. Furthermore, Li, He, Zhu, and Lyu [1] reported an improvement compared to Wang, Liu, and Tan [2] by combining source code semantic features from an AST with several traditional software metrics. They applied a CNN to the problem of file level SDP. Word embeddings were created and propagated through the network until just before the output layer where the traditional software metrics were concatenated to the deep semantic feature vectors. By combining both types of features the authors reported an increase in F1 score by 12% compared to Wang, Liu, and Tan [2]. The input to the CNN was padded with zeros to cope with the demands of their network architecture which required fixed size input. The relation between source code semantics and code comments to defective files were explored by Huo, Yang, Li, and Zhan [3]. Based on source code and code comments they trained two separate CNNs to retrieve deep representations of the code and the comments respectively. By combining the two outputs and feeding it into a softmax layer they showed that code comments add valuable information for predicting defective files. In a similar manner Huo, Li, and Zhou [18] showed how natural language in bug reports cold be used
in combination with source code to locate defective files by using two separate CNNs with dynamic pooling operations to be able to handle inputs with varying length.

### 2.2.2 Just-In-Time software defect prediction

In an early attempt to combine source code information from file changes with process metrics, Jiang, Tan, and Kim [20] proposed a personalized defect prediction model. A separate decision tree model was trained for each developer to predict software defects. To represent the information in the source code two bag-of-word (BOW) vectors were created. The BOW vectors contained the number of AST node types in each diff before and after a file change. By subtracting the two vectors respectively, the change of the code was expressed as the resulting difference of the two BOW vectors. The resulting vector is referred to as characteristic vectors. The results showed that characteristic vectors can be useful in JIT SDP. A F1 score between 0.55 and 0.72 on 6 datasets were reported. A similar method to extract BOW features were used by Tan, Tan, Dara, and Mayeux [8] where file changes were classified as buggy or clean based on process metrics, source code and code comments. A collection of classifiers were evaluated and averaged to evaluate different sampling techniques to cope with imbalanced data. The results show F1 scores in the range of 0.07 to 0.58 on 7 datasets. Moreover, the authors also stressed the importance of future research being focused on sensible and actionable explanations when it comes to JIT SDP. Although deep learning has pierced its way into file level SDP, several recent studies on JIT SDP have proposed variants of ensemble trees [21][8][22]. An exception is the report by Yang, Lo, Xia, Zhang, and Sun [17]. However, the authors did not make use of any source code information. Instead process metrics, originally described by Kamei, Shihab, Adams, Hassan, Mockus, Sinha, and Ubayashi [23] were input to a DBN to learn a deep representation of the metrics. The output layer with a logistic activation function was stacked on top of a 3-layer DBN. The authors showed an improvement in the number of bugs detected as well as in F1 score compared to the state-of-the-art at the time on 4 out of 6 open source datasets. In a more recent study Yang, Lo, Xia, and Sun [21], a two-step ensemble method was applied to the JIT prediction problem. The first step in order to arrive at the ensemble model consisted of training multiple Random Forests on subsets of under sampled data to cope with class imbalances. Next the classifiers were stacked and weighted equally when computing the majority vote. Using the process metrics originally proposed by Kamei, Shihab, Adams, Hassan,
Mockus, Sinha, and Ubayashi [23] the results showed that the method outperformed the method proposed by Yang, Lo, Xia, Zhang, and Sun [17]. F1 scores between 0.26 and 0.68 on 6 datasets were reported.

2.3 Vector representation of source code

Natural language is composed of non-trivial rules overhauled by many exceptions which are hard to explicitly translate into a computer program. A convenient way of dealing with this issue in natural language modelling is to encode the text as compressed numerical vectors. The benefits of such a representation is that it can preserve relations between the words in the text and that it bypasses the vast feature space obtained by a one-hot encoding. Successful attempts utilizing this method have created immense opportunities to build systems for machine translation, lexical semantics and natural language generation. Employing numerical vector representations to model semantic relations in source code has sparked the interest of researchers and progress has been made in predicting defective files, find the location of bugs and detect security vulnerabilities [3, 24, 25]. Source code, just like natural language, has an explicit syntax and affluent semantics that recently has been made available for use in SDP thanks to Deep Learning techniques applied to text data. To emulate the hierarchical structure of source code the recent migration towards using Deep Learning is natural given its prosperous results in natural language processing [19]. A well documented and extensively used method for vector representations of words is Word2Vec presented by Mikolov, Sutskever, Chen, Corrado, and Dean [26]. Given a sentence of words Word2Vec distributes each word over a real valued n-dimensional vector. This enables structural information to be captured in the sense that words that live nearby in the input text also live nearby in vector space. Although the advantages of this method has facilitated research in text modeling, a drawback is that it is unable to preserve explicit ordering of words in a text. To deal with this, models like Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) has been exploited [25, 27]. In the context of source code modeling this was exemplified by Russell, Kim, Hamilton, Lazovich, Harer, Ozdemir, Ellingwood, and McConley [25], who predicted security vulnerabilities based on source code functions. The authors utilized a CNN to extract structural information from tokenized source code. The source code were tokenized into type declarations and embedded into a compact vector representation. After stacking the token embeddings to a long vector and padding with zeros for all vectors shorter than 500 tokens the embeddings were fed to a CNN to learn the
structure of the source code. Finally, a Random Forest classifier was used in order to classifying the source code functions as vulnerable or not. The results showed that utilizing the representation of the source code learned by the CNN was favourable over simply representing the source code as a BOW vector. Furthermore, Tufano, Watson, Bavota, Di Penta, White, and Poshvyanyk [27] explored different levels of abstraction to represent source code and how these can be used in deep learning to automatically learn similarities between code components. To encode the source code as vectors they used a RNN. The embeddings of the source code were then passed to a Random Forest classifier in order to detect clones of source code. The authors showed that a stream of type declarations transformed with deep learning techniques can be useful for modelling patterns in source code.
Chapter 3

Background

3.1 Random Forest

The Decision Tree (DT) algorithm, which is a transparent and computationally efficient method, have been extensively used within the SDP field [20, 17, 21, 8]. A DT learns to fit a dataset by starting with a single root node and splitting the data based on the feature that minimizes (or maximizes) some criterion. The process is recursively repeated until a predefined depth is reached or until each leaf node only contains data points from the same class. A commonly used criterion to optimize for at each split is the \textit{Gini impurity} described in Equation (3.1).

\[ I_G(p) = 1 - \sum_{c=1}^{C} p_c^2 \]  

where \( p_c \) is the fraction of samples labeled with class \( c \). It is a measure of the ratio of misclassification when randomly choosing a sample from the data and randomly assigning it a label according to the distribution of the labels in the data. Thus, the objective is to split the data based on the feature that results in the least impure partitioning of the class labels \( C \) creating a rule for the decision tree. A commonly faced problem with DTs are overly complex models. In order to deal with this, extensions to this family of methods have been developed to increase the capability of generalizing to unseen data. The Random Forest algorithm, described in Algorithm 1 was introduced by Breiman [28] for this purpose. The Random Forest algorithm reduces the risk of overfitting by combining a set of uncorrelated decision trees trained on a random subset of the data. Predictions are obtained by taking the majority vote of the output of all trees. Further, at each feature split only a randomly selected subset of the feature space is considered. This way, the risk of including the strongest
predictors in all trees causing them to be correlated and decrease the generalization capability, is reduced [28].

**Algorithm 1 RandomForest**

1: **procedure** `TrainRandomForest(Data)`
2: \( \text{RandomForest} \leftarrow \emptyset \)
3: \( \text{for } n \in N \text{ do} \) \( \triangleright \) Number of trees
4: \( D \leftarrow \text{RandomSample(Data)} \)
5: \( DT \leftarrow \text{TrainDecisionTree}(D) \) \( \triangleright \) Random sampling features at each split
6: \( \text{RandomForest} \leftarrow \text{RandomForest} \cup DT \) \( \triangleright \) Add tree to forest
7: \( \text{end for} \)
8: \( \text{return} \text{RandomForest} \)
9: \( \text{end procedure} \)

10:

11: **procedure** `PredictRandomForest(x*)`
12: \( P \leftarrow \emptyset \)
13: \( \text{for } DT \in \text{RandomForest} \text{ do} \)
14: \( y^*_{DT} = DT\text{Predict}(x^*) \)
15: \( P \leftarrow P \cup y^*_{DT} \)
16: \( \text{end for} \)
17: \( y^* = \text{MajorityVote}(P) \)
18: \( \text{return} y^* \)
19: \( \text{end procedure} \)

### 3.1.1 Uncertainty in Random Forests

The Random forest algorithm provides for the possibility of computing the variance of the estimator since it is composed by multiple decision trees. Given a unseen data point the predictions of all trees might not be coherent. By accounting for the majority vote of all trees the standard deviation is obtained and thereby also the variance. Equation (3.2) describes the variance

\[
\sigma^2 = \frac{\sum_{b=1}^{B}(f_b(x^*) - \hat{f})^2}{B - 1}
\]

(3.2)

where \( B \) is the number of trees in the forest and \( \hat{f} = \frac{1}{B} \sum_{b=1}^{B} f_b(x^*) \) which in a classification problem corresponds to the majority vote.
3.1.2 Two Layer Ensemble Learner

The Random Forest algorithm has shown promising results in variants of JIT SDP and has been utilized by Gousios, Pinzger, and Deursen [22] to predict pull request merges and by Yang, Lo, Xia, and Sun [21] to predict defect code. In fact, the current state of the art method in JIT SDP is the Two Layer Ensemble Learner model (TLEL) model described by Yang, Lo, Xia, and Sun [21] which is based on the Random Forest algorithm. The training procedure of TLEL seen in Algorithm 2 involves randomly splitting the data in $M$ partitions using random under-sampling. Each partition of the data is then given an even class distribution. For each partition $m \ldots M$, a Random Forest classifier that consists of $N$ number of DTs is trained. By doing this a two layer ensemble model is obtained where the first layer consists of $M$ Random Forests of $N$ decision trees. The last layer combines the $M$ Random Forests by using majority voting. The procedure is described in Algorithm 2. Moreover, Yang, Lo, Xia, and Sun [21] calls for a more comprehensive hyperparameter search of the TLEL model in future research.

**Algorithm 2 TLEL**

1: procedure TrainTLEL($Data$)
2:     $Ensemble \leftarrow \emptyset$
3:     $D_M \leftarrow \text{PartitionData}(Data) \quad \triangleright \text{Under-sampling } M \text{ partitions}$
4:     for $D_m \in D_M$ do
5:         $RF_{D_m} \leftarrow \text{TrainRandomForest}(D_m)$
6:         $Ensemble \leftarrow Ensemble \cup RF_{D_m} \quad \triangleright \text{Add to ensemble}$
7:     end for
8:     return $Ensemble$
9: end procedure

10: procedure PredictTLEL($x^*$)
11:     $P \leftarrow \emptyset \quad \triangleright \text{Collection of predictions}$
12:     for $\text{RandomForest} \in Ensembl e$ do
13:         $y^*_{\text{RandomForest}} = \text{PredictRandomForest}(x^*)$
14:         $P \leftarrow P \cup y^*_{\text{RandomForest}}$
15:     end for
16:     $y^* = \text{MajorityVote}(P)$
17:     return $y^*$
18: end procedure
3.2 Bayesian Neural Networks

Deep learning techniques provide state-of-the-art performance in a diverse set of areas and have frequently been employed for image recognition, speech recognition, and object tracking, to name a few [29]. The extent to which you are certain about something can be quantified by the concept of uncertainty. In the urge for highly accurate models with striking predictive power uncertainty have been somewhat overlooked [7]. It is a fundamental aspect of prediction and key to avert undesired model behaviour and severe consequences. Traditional deep learning methods lack the ability to express uncertainty about its parameters and around the prediction of an unseendata point [6]. As Ghahramani [7] states it; "Since any sensible model will be uncertain when predicting unobserved data, uncertainty plays a fundamental part in modelling". Ideally, a SDP model which receives unseen data with characteristics completely different from what is was trained on, would classify it as defect or non-defect, but also express a noteworthy amount of uncertainty in its belief. The desire for understanding and interpreting neural networks has recently sparked research into the ability of neural networks to express uncertainty [30]. Furthermore, traditional feed forward neural networks are prone to overfitting and methods like early stopping, dropout, weight penalization have been developed to encounter this behaviour [6]. The inherent characteristics of a BNN, however, makes it resilient to overfitting by nature by learning a probability distribution over each weight. This can be thought of as an ensemble of networks where each sub-network possess weights sampled from a learnt distribution. In particular, the uncertainty in the weights reflect the uncertainty in which of the sub-networks give rise to the best fit to the data. A nice feature of this kind of ensemble network is that by using shared distributions on the weights across the ensemble, it only increases the amount of parameters of the network by a factor of the number of parameters used in the chosen variational distribution. If a Gaussian $\mathcal{N}(\mu, \sigma)$ is chosen as variational distribution the number of parameters of a traditional network is doubled, since the gaussian has two parameters, $(\mu, \sigma)$ [6]. Moreover, Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] showed that using the signal-to-noise measure $\frac{\mu}{\sigma}$ as a proxy for parameter importance they could prune up to 75% of the weights in a network resulting in a smaller and faster network without decreased performance [6].
3.2.1 Architecture

The innate abilities of BNNs makes them interesting in regards to JIT SDP. In order to train such a network some modifications to the cost function and the gradient calculations have to be made compared to a traditional neural network. Each weight in a BNN is not a scalar as in traditional neural networks but in fact a distribution parametrized by $\mu, \sigma$ in case of a Gaussian variational distribution as described in Figure 3.1.

![Figure 3.1: A Bayesian Neural Network. Each weight (arrow) is represented by a Gaussian distribution parametrized by a $\mu$ and a $\sigma$.](image)

3.2.2 Loss function

Since exact inference on the weights in large BNNs is intractable, a variational approximation to the true weight distribution is used instead [6]. The variational distribution $q(w|\theta)$ parameterized by some $\theta$ is a simplified approximate distribution to the true distribution of the weights given the data $P(w|D)$. The idea is that the weights are penalized by minimising the cost function commonly referred to as the variational free energy [6]. The variational free energy is a measure of the relative entropy between $q(w|\theta)$ and $P(w|D)$ [31]. This corresponds to finding the parameters $\theta$ of the variational distribution $q(w|\theta)$ that minimizes the Kullback–Leibler divergence between
\( q(w|\theta) \) and \( P(w|D) \), which is shown below in Equation (3.3).

\[
\begin{align*}
\theta^* &= \underset{\theta}{\text{argmin}} \left[ KL[q(w|\theta)||P(w|D)] \right] \\
&= \underset{\theta}{\text{argmin}} \left[ \int q(w|\theta) \log \frac{q(w|\theta)}{P(D|w)P(w)} \, dw \right] \\
&= \underset{\theta}{\text{argmin}} \left[ KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)} [\log P(D|w) - \log P(D|w^{(i)})] \right] \\
&\Rightarrow F(D, \theta) = KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)} [\log P(D|w)] \\
\end{align*}
\]

(3.3)

Strictly speaking, the goal is to find an approximation to the true weight distribution by minimizing the resulting cost function \( F(D, \theta) \). However, \( F(D, \theta) \) is intractable so using the work of Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] it can be approximated as follows

\[
F(D, \theta) \approx \sum_{i=1}^{n} q(w^{(i)}|\theta) - \log P(w^{(i)}) - \log P(D|w^{(i)})
\]

(3.4)

here \( w^{(i)} \) represents an observation of the weights sampled from \( q(w|\theta) \). In order to minimize Equation (3.4) and learn the approximate distributions of the weights a variant of gradient descent called Bayes by Backprop proposed by Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] can be employed.

### 3.2.3 Computing gradients

To be able to compute the derivatives of the variational distributions over the weights a reparametrization of the weights is used. The idea is to transform the parameters so that the stochastic elements of \( q \) are independent of the parameters \( \theta \). Assuming a diagonal Gaussian variational posterior, a weight \( w \) can be given by sampling \( \epsilon \) from \( \mathcal{N}(0,1) \) and shifting and scaling with \( \mu, \sigma \) respectively. To ensure a positive contribution from the variance, \( \sigma \) is parameterized by \( \sigma = \text{softplus}(\rho) = \log \exp(1 + \rho) \). The resulting transform of a posterior sample of the weights where the randomness is separated from \( q \) gives the expression in Equation (3.5).

\[
w = \mu + \log(1 + \exp(\rho)) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0,1)
\]

(3.5)
Where \( \odot \) is the element wise multiplication operation. The reparametrization procedure allows the error to backpropagate through the network without being affected too much of sampling rare cases of the weights which else would be the case when optimizing the network without reparametrization. The optimization procedure is described by Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] as follows.

1. Sample \( \epsilon \sim \mathcal{N}(0, I) \)
2. \( w = \mu + \log(1 + \exp(\rho)) \odot \epsilon \)
3. \( \theta = (\mu, \rho) \)
4. \( f(w, \theta) = \log(q(w|\theta)) - \log(P(w)P(D|w)) \)
5. Compute gradient w.r.t \( \mu \)
   \[ \delta_\mu = \frac{\partial f(w, \theta)}{\partial w} + \frac{f(w, \theta)}{\partial \mu} \]
6. Compute gradient w.r.t \( \rho \)
   \[ \delta_\rho = \frac{\partial f(w, \theta)}{\partial w} \frac{\epsilon}{1 + \exp(-\rho)} + \frac{f(w, \theta)}{\partial \rho} \]
7. Update variational parameters according to the learning rate \( \eta \)
   \[ \mu = \mu - \eta \delta_\mu \]
   \[ \rho = \rho - \eta \delta_\rho \]

By approximating the posterior distribution over the weights given the data \( P(w|D) \) a predictive distribution \( P(y^*|x^*) \) of a target \( y^* \) given an unseen data point \( x^* \) can be obtained. This is in contrast to traditional neural networks where simply a maximum likelihood point estimate prediction is obtained.

### 3.2.4 Uncertainty in Bayesian Neural Networks

The uncertainty in the prediction can be expressed as a composition of two components of the sampled predictions. The first components referred to as the aleatoric uncertainty stems from the uncertainty of the data and the second, named epistemic uncertainty, stems from model uncertainty [32] [33].
Yongchan, Won, Kim, and Paik [33] argues that an unbiased estimator of the expectation can be obtained by sampling from the variational posterior $q(w|D)$

$$
\mathbb{E}_q [p_D(y^*|x^*)] = \int q(w|D) p_w(y^*|x^*) \, dw \\
\approx \frac{1}{T} \sum_{t=1}^{T} p_{wt}(y^*|x^*)
$$

(3.6)

where $T$ is the number of samples drawn from the posterior. The predictive variance $\text{Var}_q$ can further be derived from the definition of variance and can be expressed as

$$
\text{Var}_q(p(y^*|x^*)) = \frac{1}{T} \sum_{t=1}^{T} \text{diag}(\hat{p}_t) - \hat{p}_t\hat{p}_t^T + \frac{1}{T} \sum_{t=1}^{T} (\hat{p}_t - \bar{p})(\hat{p}_t - \bar{p})^T
$$

(3.7)

where $\bar{p} = \frac{1}{T} \sum_{t=1}^{T} \hat{p}_t$ and $\hat{p}_t = \text{Softmax}(f_{wt}(x^*))$. If $T$ set of weights is sampled and used for propagating a data point through the network $T$ times, $T$ samples of the output is obtained. In general, if each of the probability vectors in the $T$ outputs are similar, the epistemic uncertainty is low. If the probabilities within each vector are dissimilar the aleatoric uncertainty is low. For a binary classification problem $\text{Var}_q$ spans from 0 to 0.25.
Chapter 4

Method

4.1 Data

Spotify provides non-public data consisting of file changes logged with a version control tool. Each entry in the database storing the file changes represents a pull request. Each pull request contains a collection of commits along with a labeled outcome of a build procedure. The outcome of the build procedure is labeled as defective if the build procedure was unsuccessful. Correspondingly it is labeled as non-defective if the build procedure is successful. The number of commits included in each pull request varies and each commit consists of changes in one or multiple files. Small and concise commits are desired but not always possible to achieve. Each database entry is subject to a static metrics analysis and a process metrics analysis. The process metrics and static metrics are calculated for each file and then aggregated into to a unified representation for each database entry. In addition, an embedding of the source code in each *hunk* is performed. Before the embedding of the source code, it is tokenized into type declarations such as `float`, `string`, `for`, `while`, etc.

As previously mentioned in section 1.2, when comparing two versions of a file the parts that differ are called *hunks*. If two identical files are compared no *hunks* are created. If two completely different files are compared one large *hunk* that contains all lines of both files are generated. If there are changes in different parts of a file multiple *hunks* will be generated. A nonsense example of a commit logged with the version control tool is shown in Figure 4.1.
Figure 4.1: Version control tool example including two files. The topmost file has one hunk and the bottommost has two hunks. Red sections are the deleted source code. Green sections are added source code.

The blue sections in Figure 4.1 characterizes a start of a new hunk. Each hunk in Figure 4.1 contains the added code (green sections), the deleted code (red sections) plus its adjacent context code (white sections). The source code used for modelling in this thesis consists of a subset of the information in each hunk. From each hunk an executable snippet of code is created from the white and green sections shown in Figure 4.1 i.e. ignoring the red and blue sections. The reason for using this procedure over only considering added code is to keep the structure of the code intact. This way small changes will result in small hunks, in contrary to file level SDP where entire files are considered.
which would increase computations significantly. Two datasets of file changes from java files are constructed. The first dataset, $D_1$, is used to answer the first research question and consists of 4800 entries containing software metrics. The second dataset, $D_2$, is used to answer the second research question and consists of the same entries as in $D_1$ but with source code snippets added. The number of defective and non-defective labels are equal in both datasets.

### 4.2 Process metrics

The process metrics used in the experiments are as far as possible adopted from Kamei, Shihab, Adams, Hassan, Mockus, Sinha, and Ubayashi [23]. The additional metrics *refactor, help others, new work* are proposed. The metric *refactor* is defined as the number of changed lines that has been untouched during the previous 3 weeks. The *help others* metric indicates the number of lines changed by another developer than the developer who created the pull request. The metric *new work* is defined as the number of lines added within the last 3 weeks.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commits</td>
<td>Number of commits</td>
</tr>
<tr>
<td>additions</td>
<td>Number of added lines</td>
</tr>
<tr>
<td>deletions</td>
<td>Number of deleted lines</td>
</tr>
<tr>
<td>changed files</td>
<td>Number of files changed</td>
</tr>
<tr>
<td>cycle time</td>
<td>Time from first commit to last commit</td>
</tr>
<tr>
<td>new work</td>
<td>Number of lines added within the last 3 weeks</td>
</tr>
<tr>
<td>refactor</td>
<td>Number of lines changed in existing code older than 3 weeks</td>
</tr>
<tr>
<td>churn</td>
<td>Number of lines removed or modified in code written within the last 3 weeks</td>
</tr>
<tr>
<td>help others</td>
<td>Number of lines modified by another developer than the one who created the pull request</td>
</tr>
<tr>
<td>total</td>
<td>$\text{refactor} + \text{churn} + \text{help others} + \text{new work}$</td>
</tr>
<tr>
<td>cyclomatic complexity delta</td>
<td>The difference between the cyclomatic complexity before and after the change</td>
</tr>
</tbody>
</table>
4.3 Source code tokenization

In order to utilize the source code as input it has to be transformed to a format that can be handled by the models. As previously proposed by Wang, Liu, and Tan [2] and Li, He, Zhu, and Lyu [1] source code can be represented as ASTs. However, the nature of the data used in this study does not naturally support the use of AST since hunks of source code are considered and not entire files. This makes the use of ASTs somewhat problematic since the version control tool stores a snapshot of the immediate neighbouring lines of the changed lines. Hence it might not comprise the entire context of a file change and its implications on code outside of the hunk. The problem arises when for example a variable is changed in the middle of a large function. The variable may have its AST parent node in the very beginning of the file, outside of the hunk, which makes it difficult to construct an AST. The information needed would be lost and the AST incomplete. To avoid this issue the method used in this thesis is the one described by Russell, Kim, Hamilton, Lazovich, Harer, Ozdemir, Ellingwood, and McConley [25] where the source code is tokenized into a stream of type declarations. The extracted type declarations is shown in Table 4.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modifiers</strong></td>
<td>public, private, protected, default, final, abstract, static, synchronized, strictfp, transient, native, volatile</td>
</tr>
<tr>
<td><strong>Keywords</strong></td>
<td>abstract, assert, boolean, break, case, catch, continue, default, do, double, else, enum, extends, final, finally, for, goto, if, instanceof, interface, new, package, private, protectedpublic, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while, native, abstract, default, final, native, private, protected, public, static, strictfp, synchronized, transient, volatile, const, class, implements, import</td>
</tr>
<tr>
<td><strong>Basic types</strong></td>
<td>byte, char, float, long, int</td>
</tr>
</tbody>
</table>

Table 4.2 Source code type declarations
4.4 Source code embedding

Using the raw format of the source code would result in a large number of dimensions since the number of variable names is large. By using the tokenization, the syntactic structure is preserved and the resulting dimensionality relatively low. RNNs have commonly been employed in natural language processing due to its ability to model sequential dependencies \[34\]. Source code exhibits, just like natural language, an explicit syntax with sequential dependencies that have been exploited in order to build predictive models \[27\]. Similar to the work by Tufano, Watson, Bavota, Di Penta, White, and Poshyvanyk \[27\], this study utilized a RNN in order to encode the source code as numerical vectors. Each type declaration is mapped to a unique integer token id. A row in the datasets contains hunks from one or multiple commits and files, therefore the source code is concatenated into a token id vector \(V_i\) per dataset row \(i\). An illustration of the method is shown in Figure 4.2.

![Figure 4.2: Conversion of source code from a pull request to a token id vector](image)

Before the creation of the embeddings of the source code the vectors \(V_i\) with significantly larger amounts of source code altered were treated as outliers and removed. By removing the outliers the dimensionality \(d\) of the data is reduced from \(d = 5000\) to \(d = 745\). The sample size \(s\) is also reduced, from \(s = 4800\) to \(s = 4614\). The remaining vectors are padded with zeros based on the longest vector \(V_i\) and then used as input to a RNN. The first layer of the RNN is an embedding layer of size \(\text{length(unique(tokens))} = 45\), and the second layer a LSTM layer with 10 nodes. The final layer is a softmax layer with two nodes. The architecture was decided by a random search over the number of units in the embedding layer and in the LSTM layer. The RNN is utilized to learn the mapping from the source code to the outcome of the build.
procedure. The output vectors from the LSTM layer is extracted and treated as the embedding of the source code. The vectors are then concatenated with the software metrics in $D_1$ accordingly. The resulting dataset, $D_2$, is used to answer the second research question. The experimental setup is illustrated in Figure 4.3.

![Figure 4.3: Description of the experimental setup](image)

### 4.5 Evaluation

Established measures for evaluating SDP-models are used to answer the research questions. Accuracy and F1 score have been frequently employed in previous works [4]. They are derived from the confusion matrix in Figure 4.3. The confusion matrix in Figure 4.3 states number of true positive, false positive, true negative and false negative classifications obtained by a classifier. The F1 score is the harmonic mean between precision and recall of a model. A F1 score closer to 1 is desirable. Further measures derived from the confusion matrix are Receiver Operating Characteristics (ROC), and MCC [35]. The former illustrates the diagnostic ability of a binary classifier [36]. The latter is a measure of the correlation between the predicted values and the target. In a binary classification problem the MCC measure ranges from -1 to 1, where 1 states prefect prediction. A MCC value of -1 states a total disagreement between the prediction and the target. A random prediction is given a MCC value of zero.
Table 4.3 Confusion Matrix

<table>
<thead>
<tr>
<th></th>
<th>Classified as non-defective</th>
<th>Classified as defective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-defective</td>
<td>True Negatives (TP)</td>
<td>False Positive (FP)</td>
</tr>
<tr>
<td>Defective</td>
<td>False Negative (FN)</td>
<td>True Positives (TN)</td>
</tr>
</tbody>
</table>

\[
F^1 = 2 \cdot \frac{TP}{(TP+FP)} \cdot \frac{TP}{(TP+FN)}
\]

(4.1)

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

(4.2)

\[
MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}
\]

(4.3)

### 4.6 Cross validation

Since data is limited a 2-fold cross-validation schema is used along with the 5x2CV paired \(t\) test proposed by Dietterich \[37\]. In order to test the null hypothesis the 5x2CV paired \(t\) test is employed. The null hypothesis states that there is no statistically significant difference between the performance of the state-of-the-art TLEL model and the BNN. The advantage of the 5x2CV paired \(t\) test over \(k\)-fold cross-validation schemas with a \(k\) larger than 2 is that it decreases the dependency problem of the data in the folds since the test sets are completely non-overlapping. Five replications of 2-fold cross-validation is performed. In each replication the data is randomly split in two equally sized partitions. The two models subject to comparison \(TLEL = M_1, BNN = M_2\) are first trained on the first partition and evaluated on the second. Then the order is switched and the models are trained on the second partition and evaluated on the first, resulting in one of five replications of the 2-fold cross-validation schema. One replication \(i\) gives an outcome of four error metrics \(E_{1M_1}^i, E_{2M_1}^i, E_{1M_2}^i, E_{2M_2}^i\) (two per model per fold, where \(i = 1 \ldots 5\)). The difference between the two models on each partition is used to compute the mean \(\bar{E}^i\) and variance \(S^{2i}\) and from there the 5x2CV \(t\) statistic can be derived \[37\].
\begin{align*}
E^i_1 &= E^{i}_{1M_1} - E^{i}_{2M_1} \\
E^i_2 &= E^{i}_{1M_2} - E^{i}_{2M_2} \\
\bar{E}^i &= \frac{E^i_1 + E^i_2}{2} \\
S^{2i} &= (E^i_1 - \bar{E}^i)^2 (E^i_2 - \bar{E}^i)^2 \\
\tilde{t} &= \frac{E^i_1}{\sqrt{\frac{1}{5} \sum_{i=1}^{5} S^{2i}}} \quad (4.4)
\end{align*}

Under the null hypothesis \( \tilde{t} \) approximately follows a \( t \) distribution with 5 degrees of freedom which is shown by Dietterich [37]. The same procedure is used to compare the null hypothesis that there is no significant difference in model performance when adding source code information to the data.

### 4.7 Experimental settings

The hyperparameter settings of the BNN are chosen by running the 5x2 cross-validation 100 times on \( D_1 \) and \( D_2 \) respectively. Each time a set of new hyperparameters are sampled from the hyperparameter search space. The hyperparameters that account for the highest sum of the scores (F1, accuracy, MCC) are the hyperparameters used in Chapter 5. The nodes of the BNN employ the ReLu activation function, except for the two nodes in the output layer which employ the Softmax activation function. An overview of the BNN model parameters and hyperparameter search space is shown in Table 4.4.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden Layers</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>Hidden Nodes</td>
<td>(5, 100)</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>(5, 100)</td>
</tr>
<tr>
<td>( \sigma_1 )</td>
<td>(0.7, 1)</td>
</tr>
<tr>
<td>( \sigma_2 )</td>
<td>(0.005, 0.1)</td>
</tr>
<tr>
<td>( \rho )</td>
<td>(−4, 0)</td>
</tr>
<tr>
<td>( \pi )</td>
<td>(0.2, 0.4)</td>
</tr>
<tr>
<td>Batch size</td>
<td>(30, 1000)</td>
</tr>
</tbody>
</table>

Following the work of Blundell, Cornebise, Kavukcuoglu, and Wierstra [6], a Gaussian distribution is chosen for the variational distribution \( q \) which is pa-
rameterized by $(\mu, \rho)$, where $\sigma = \text{softplus}(\rho)$ as earlier described in Equation (3.5). In addition, the method proposed by Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] is adopted where a scale mixture distribution is used as a prior over the weights. The prior is described in Equation (4.5) where $\sigma_1 \gg \sigma_2$ and $\pi$ is a scaling factor that sums to 1.

$$\log P(w) = \sum_i \log \left( \pi \mathcal{N}(w_i|0, \sigma_1^2) + (1 - \pi) \mathcal{N}(w_i|0, \sigma_2^2) \right)$$  \hspace{1cm} (4.5)

In order to tune the hyperparameters of the TLEL model the same procedure is used as for the BNN. The TLEL model parameters and the hyperparameter space is described in Table 4.5.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitions</td>
<td>$(10, 100)$</td>
</tr>
<tr>
<td>Decision trees</td>
<td>$(10, 500)$</td>
</tr>
<tr>
<td>Min samples in split</td>
<td>$(2, 10)$</td>
</tr>
<tr>
<td>Max depth</td>
<td>$(30, 100) \cup \text{None}$</td>
</tr>
<tr>
<td>Min samples in leaf</td>
<td>$(1, 4)$</td>
</tr>
</tbody>
</table>
Chapter 5

Results

5.1 Experiment results

The BNN and the state-of-the-art model TLEL are tested on two datasets. The first dataset $D_1$ includes exclusively software metrics. The second dataset $D_2$ includes source code embeddings in addition to the process metrics in $D_1$. The performance of the models are compared against each other on each dataset respectively. Furthermore, the performance of each model on $D_1$ is compared to its performance on $D_2$. The figures and tables in Chapter 5 considers the average performance over the 5x2 cross-validation schema described in Section 4.6. The parameter settings of the models obtained by the hyperparameter search procedures are described in Table 5.1 and Table 5.2.

Table 5.1 TLEL model settings

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>$D_1$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitions</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Decision trees</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Min samples in split</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Max depth</td>
<td>None</td>
<td>52</td>
</tr>
<tr>
<td>Min samples in leaf</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 5.2 BNN model settings

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>$D_1$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden Layers</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hidden Nodes</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.001</td>
<td>0.002</td>
</tr>
<tr>
<td>$\sigma_1$</td>
<td>0.77</td>
<td>0.7</td>
</tr>
<tr>
<td>$\sigma_2$</td>
<td>0.012</td>
<td>0.01</td>
</tr>
<tr>
<td>$\rho$</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>$\pi$</td>
<td>0.25</td>
<td>0.2</td>
</tr>
<tr>
<td>Batch size</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

5.2 Performance evaluation

Both the TLEL model and the BNN have trouble separating the classes when run on the software metrics dataset, $D_1$. The TLEL model has more evenly distributed values across the confusion matrix in Figure 5.1a. However, the BNN better identifies non-defectives which is shown in Figure 5.1b.

![Confusion matrices](image)

Figure 5.1: The confusion matrices show the distribution of the predictions over true negatives, false positives, true positives and false negatives. Positives represents defectives. Negatives represents non-defectives. Dark colors represent higher concentration of predictions. Bright colors represent less concentration of predictions. (a) Confusion matrix from running TLEL on $D_1$. (b) Confusion matrix from running BNN on $D_1$.

The performance of both models increase when run on $D_2$ which is shown in Figure 5.2. The TLEL finds defectives to a greater extent than the BNN.
does, but the BNN exhibits a greater ability of detecting non-defectives as seen in Figure 5.2.

Figure 5.2: The confusion matrices show the distribution of the predictions over true negatives, false positives, true positives and false negatives. Positives represents defectives. Negatives represents non-defectives. Dark colors represent higher concentration of predictions. Bright colors represent less concentration of predictions. (a) Confusion matrix from running TLEL on $D_2$. (b) Confusion matrix from running BNN on $D_2$.

The ROC curves in Figure 5.3b indicate a slightly greater true positive rate than false positive rate. The ROC curves of the BNN and the TLEL model are approximately equal in both Figure 5.3a and Figure 5.3b.

Figure 5.3: ROC curves of the BNN and the TLEL. Dotted black line corresponds to random performance. (a) shows the results obtained on $D_1$. (b) shows the results obtained on $D_2$. 
The scores shown in Figure 5.4 do not differ considerably between the models. However, in 5.4a the TLEL model exhibits a larger F1 score than the BNN. This is further supported by the left confusion matrix in Figure 5.1a.

![Figure 5.4: (a) shows the F1 score, accuracy and MCC of the TLEL and the BNN obtained on \( D_1 \). (b) shows the F1 score, accuracy and MCC of the TLEL and the BNN obtained on \( D_2 \).](image)

<table>
<thead>
<tr>
<th>Table 5.3 Mean and standard deviation scores on dataset 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F1</strong></td>
</tr>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>TLEL</td>
</tr>
<tr>
<td>BNN</td>
</tr>
</tbody>
</table>

A summary of the mean scores obtained from both models on \( D_1 \) are shown in Table 5.3. As previously seen in Figure 5.4a the models perform approximately equal.

<table>
<thead>
<tr>
<th>Table 5.4 5x2CV paired t test. BNN vs. TLEL on ( D_1 ). (N=10)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F1</strong></td>
</tr>
<tr>
<td>( \hat{t} )</td>
</tr>
<tr>
<td>Dataset1</td>
</tr>
</tbody>
</table>
There is no statistically significant difference between the BNN and the TLEL performance on $D_1$. Even though the TLEL mean scores are somewhat higher the standard deviation, of the F1 scores in particular, is rather high for both models. Hence, Table 5.4 shows that there is no statistically significant difference between the means.

**Table 5.5 Mean and standard deviation scores on dataset 2**

<table>
<thead>
<tr>
<th></th>
<th>F1 Mean</th>
<th>F1 Std</th>
<th>Accuracy Mean</th>
<th>Accuracy Std</th>
<th>MCC Mean</th>
<th>MCC Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLEL</td>
<td>0.588</td>
<td>0.017</td>
<td>0.567</td>
<td>0.01</td>
<td>0.135</td>
<td>0.02</td>
</tr>
<tr>
<td>BNN</td>
<td>0.558</td>
<td>0.02</td>
<td>0.567</td>
<td>0.011</td>
<td>0.134</td>
<td>0.022</td>
</tr>
</tbody>
</table>

The average performances on $D_2$ are shown in Table 5.5. The performance of the models are similar to each other and in general a bit higher compared to when run on $D_1$.

**Table 5.6 5x2CV paired $t$ test. BNN vs. TLEL on $D_2$. (N=10)**

<table>
<thead>
<tr>
<th></th>
<th>F1 $\tilde{t}$</th>
<th>F1 p-value</th>
<th>Accuracy $\tilde{t}$</th>
<th>Accuracy p-value</th>
<th>MCC $\tilde{t}$</th>
<th>MCC p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset2</td>
<td>0.474</td>
<td>0.655</td>
<td>−0.62</td>
<td>0.561</td>
<td>−0.635</td>
<td>0.55</td>
</tr>
</tbody>
</table>

There is no statistically significant difference between the means of the model scores when run on $D_2$ as seen in Table 5.6. In Figure 5.5a, the BNN mean loss over the 5x2 cross-validation schema decreases over the epochs but as previously noted in Table 5.3, the overall performance is still poor. This indicates that the model is not able to fit the data properly. Further, the BNN mean loss on $D_2$ in Figure 5.5b decreases more steadily.
The performance of the BNN on $D_2$ when removing weights from the network is shown in Figure 5.6. The pruning is done by removing the weights with the lowest signal-to-noise ratio $\frac{|\mu_i|}{\sigma_i}$. The performance is consistent up to 70% pruning of the weights, resulting in a network with reduced memory requirements and faster predictions.

Figure 5.6: Pruning of the BNN weights when run on $D_2$. The horizontal axis shows the amount of weights removed according to the signal-to-noise ratio. The vertical axis shows the score.
5.3 Uncertainty evaluation

Uncertainty measures from running the models on \( D_2 \) are shown below. Due to the relatively poor performance of both models on \( D_1 \) only the uncertainty measures from \( D_2 \) are presented.

Figure 5.7: BNN uncertainty distributions of correct and incorrect classifications obtained on \( D_2 \). (a) shows epistemic uncertainty. (b) shows aleatoric uncertainty. Purple color represents correct predictions. Light blue color represents incorrect predictions. The dotted lines represents the mean of the distributions. Solid lines represents the distribution estimates.

The low epistemic uncertainty in Figure 5.8a indicates that each sub-network in the ensemble network agrees on the predictions. However, the high aleatoric uncertainty in Figure 5.8b indicates that each sample of the probability vectors obtained from the Softmax layer has values close to the classification threshold of 0.5. Hence, the overall uncertainty is high.
Figure 5.8: Uncertainty distributions of correct and incorrect classifications obtained on $D_2$. (a) shows the TLEL uncertainty. (b) shows the BNN uncertainty, $Var_q(p(y^*|x^*))$, which is the sum of epistemic and aleatoric uncertainty. Purple color represents correct predictions. Light blue color represents incorrect predictions. The dotted lines represents the mean of the distributions. Solid lines represents the distribution estimates.

In Figure 5.8b there is a left shift of the mean uncertainty for the data points that are correctly classified compared to the data points that are misclassified. Unlike the BNN uncertainties in Figure 5.8b, the TLEL model in Figure 5.8a exhibits higher uncertainty in correct predictions than in incorrect predictions.

5.4 Evaluation of source code information contribution

To answer the second research question as to whether semantic information extracted from source code file changes improve model performance in JIT SDP the results obtained on $D_1$ and $D_2$ are compared for each model respectively. Table 5.7 shows that there is a statistically significant difference in accuracy and MCC when running the BNN on $D_2$ compared to running it on $D_1$. The results indicates that the embeddings of the source code in $D_2$ contributes to an increased performance. Likewise, there is a statistically significant difference in accuracy and MCC when running the TLEL on $D_2$ compared to running it on $D_1$. 
Table 5.7 5x2CV paired $t$ test. Comparing BNN and TLEL scores on $D_1$ vs. scores on $D_2$ respectively. (N=10)

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>Accuracy</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t$</td>
<td>p-value</td>
<td>$t$</td>
</tr>
<tr>
<td>TLEL</td>
<td>−1.4802</td>
<td>0.19</td>
<td>−5.365</td>
</tr>
<tr>
<td>BNN</td>
<td>0.4326</td>
<td>0.683</td>
<td>−5.017</td>
</tr>
</tbody>
</table>
Chapter 6

Discussion

6.1 Results discussion

The results show that there is no statistically significant difference between the average performance of the TLEL and the BNN on neither of the datasets. Moreover, the results show that adding source code data to the software metrics data increases the performance of both models which implies the usefulness of source code information in JIT-SDP. The F1 scores reported in section 5.2 are analogous to the ones shown by Yang, Lo, Xia, and Sun [21] who proposed the TLEL. However, the wide range of F1 scores reported in the literature entails that the performance of JIT-SDP models fluctuate depending on the data at hand. The performance is in general not yet good enough for practical applications, which the results in this thesis supports. The target vector in this thesis is the outcomes of software builds based on pull requests. This resembles Jiang, Tan, and Kim [20] and Tan, Tan, Dara, and Mayeux [8] but with the difference that it is created without a human in the loop. Taking this into account the result of this study might be affected by flaky tests causing unwanted randomness in the data. However, trying to adjust for this might obscure the applicability of JIT SDP models used in practice since flaky test outcomes are likely to be present in large software systems. Noteworthy is also that the performance of both the BNN and the TLEL model on the second dataset $D_2$ is related to the ability of the RNN to encode the source code into real valued vectors. A more rigorous tuning of the RNN might improve the chances of achieving better results in future studies. The results from the pruning of the network weights is in line with the results obtained by Blundell, Cornebise, Kavukcuoglu, and Wierstra [6] showing that a subset of the weights answers for the predictive performance. In practice this means a network with
less memory requirements and increased speed when performing predictions. Such characteristics are desirable in JIT SDP. Although the uncertainty measures in Figure 5.8 are derived from different calculations of uncertainty the result is interesting. The BNN is more uncertain in the incorrect predictions compared to the correct predictions, in contrast to the TLEl model which is more uncertain in the correct predictions than in the incorrect predictions. The interpretability of predictions in JIT-SDP is important. In general the desired behaviour of a JIT-SDP model is to express more uncertainty in the data which it misclassifies. The BNN illustrates this behavior in the Section 5.3 even though the total uncertainty is rather high. To further evaluate the applicability of BNN’s in JIT-SDP more studies are needed. Preferably on open source data sets in order to be able to benchmark against the state-of-the art results.

6.2 Method discussion

Source code that is not semantically connected but rather connected by belonging to the same pull request may distort the syntactical relations that source code exhibits. Hence, the concatenation of source code hunks from multiple commits and files in a pull request into a single vector can potentially create false dependencies. The explicit reason for a build failure is not entailed in the data since an instance is automatically marked as simply passed or failed. Thus, determining the reasons behind a failing build might improve the quality of the data and improve model performance.

6.3 Validity discussion

Conclusions drawn from an experiment is confined by the steps carried out to obtain the results. Delimitations and assumptions may affect the feasibility of deducing valid interpretations of the results. Hence, there is reason for discussing different types of validity [38].

6.3.1 Construct validity

Construct validity refers to the degree to which a test is measuring what it claims to measure [39]. The evaluation of the predictive performance of the models in this thesis relies on well known and established measures previously discussed in Section 4.3. In addition, the data consist of features frequently
used in the JIT SDP literature. Moreover, the presented uncertainty measures also rely on sound mathematical foundations, however, a caveat is recognized in Section 6.1 regarding direct comparison between the uncertainties of the models since they are derived from different calculations.

### 6.3.2 Internal validity

The internal validity reflects the extent to which a result supports a relation between cause and effect. Indeed, it concerns the presence of confounding factors which are a common pitfall in research. Confounding factors influence both the dependent variable and the independent variables and may be the real cause of an observed effect [40]. Reducing the risk of confounding factors strengthens the internal validity. Comparing the performance of JIT SDP models involves more than one confounding factor. For example, how the hyperparameters of the models are tuned and what data is used. In order to reduce the risk of confounding factors, both models are subject to a reasonably comprehensive hyperparameter tuning. In addition, the 5x2CV cross-validation schema is employed to as far as possible compensate for the data scarcity.

### 6.3.3 Conclusion validity

Conclusion validity concerns the ability of arriving at a correct conclusion given the data at hand. To ensure conclusion validity proper statistical tests needs to be used. Given the scarcity of data in this thesis the 5x2CV paired t-test was employed to test the null hypothesis. The advantage of the 5x2CV paired t-test over other statistical tests applied to cross validation data is the reduced dependency issue between the folds [37]. One of the largest threats to the validity in this thesis was presumably the risk of an incorrect implementation of the TLEL model described by Yang, Lo, Xia, and Sun [21] or the BNN described by Blundell, Cornebise, Kavukcuoglu, and Wierstra [6]. Since neither of the models are provided in any open source software packages, manual testing had to be carried out to attenuate the risk of software bugs.

### 6.4 Sustainability & Ethical aspects

Given a sufficiently accurate JIT SDP model there are potential gains from a sustainability perspective. The energy consumption in a software development project ought to be reduced due to removal of redundant testing. Even though the training of JIT SDP models requires substantial computational power, the
prediction phase does not. Depending on the type of project and the requirements on code quality the energy consumed while training a JIT SDP model is likely to be significantly lower than running all code through build procedures. Another benefit of a sufficiently accurate JIT SDP model is that developers would be able to spend more time on complex and challenging problems. Instead of working with time consuming but trivial tasks, more meaningful work tasks could be addressed leading to a better work environment. An important aspect of using a JIT SDP model in practice is to anonymize the identity of the developers. In this study such ids are not considered due to several reasons. Even though there already exist tools for tracking developers of specific commits, the benefit of such features in JIT SDP could be questioned from an ethical standpoint if the ids are not anonymized. A JIT SDP model might not be able to fully cover the complexity of a file change which could lead to false conclusions when analyzing the predictions. If the predictions show that some developers are more prone to write defective code than others this could be misinterpreted. In reality these developers might in fact work on highly complex problems which could be the real reason why their changes are classified as defective more often.

6.5 Contribution

Recent studies have confirmed the superior performance of deterministic neural networks for solving the file level SDP problem [1, 3]. Ensemble decision trees are still among the top contenders in JIT SDP [21]. To the best of our knowledge BNNs have not yet been used as a means of solving the JIT SDP problem. In addition, source code embeddings of hunks has to the best of our knowledge not been utilized in JIT SDP. The results show that the BNN expresses higher uncertainty in incorrect predictions than in correct predictions. Although the results are still not yet good enough for practical applications this behaviour is of potential benefit when developing JIT SDP models. Furthermore, adding source code information to the data improved the models performance which imply the usefulness of including such information in JIT SDP. Future studies are invited further establish the contribution of source code information in JIT SDP.
Chapter 7

Conclusion & Future work

7.1 Conclusion

There is no statistically significant difference between the performance of the BNN and the TLEL on neither of the two proprietary datasets. More research is needed to further evaluate the applicability of BNNs in JIT SDP. Future studies can utilize public datasets for comparing its performance against the state-of-the-art over a larger set of data sources. Using source code embeddings in combination with software metrics provides a statistically significant improvement in performance for both the TLEL model and the BNN, indicating that source code embeddings are useful in JIT SDP.

7.2 Future work

An interesting area for future work in JIT SDP is how to combine information from multiple files in a pull request in order to be used as input to a predictive model. In particular, the transformation of source code from different files in a single pull request into a unified representation is interesting since such a representation ideally would carry the information needed to determine the outcome of a build. In addition it would be interesting to investigate ways of comparing the uncertainties between BNNs and tree based models such as TLEL. Future studies can also investigate the applicability of other families of distribution in bayesian deep learning than the Gaussian as is used in this study.
Bibliography


