View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Hans Raukas

Some Approaches for Software Defect Prediction

Bachelor's Thesis (9 ECTS)

Supervisor: Helle Hein

Tartu 2017

Some Approaches for Software Defect Prediction

Abstract:

The main idea of this thesis is to give a general overview of the processes within the software defect prediction models using machine learning classifiers and to provide analysis to some of the results of the evaluation experiments conducted in the research papers covered in this work. Additionally, a brief explanation of the algorithms used within the software defect prediction models covered in this work is given and some of the evaluation measures used to evaluate the prediction accuracy of software defect prediction models are listed and explained. Also, a general overview of the processes within a handful of specific software defect prediction models is provided.

Keywords:

Software defect prediction, machine learning, evaluation measures

CERCS: P175 informatics, systems theory

Mõningatest tarkvara vigade hindamise mudelitest

Lühikokkuvõte:

Käesoleva töö peamiseks eesmärgiks on anda üldisem ülevaade protsessidest tarkvara vigade hindamise mudelites, mis kasutavad masinõppe klassifikaatoreid, ja analüüsida mõningaid hindamiseskperimentide tulemusi, mis on läbi viidud antud töös refereeritud uurimistöödes. Lisaks on antud lühike selgitus antud töös vaadeldavates tarkvara vigade hindamise mudelites kasutatud algoritmidest ja tuuakse välja ning seletatakse lahti mõned hinnangumõõdikud, mida kasutatakse tarkvara vigade hindamise mudelite hindamistäpsuste mõõtmiseks. Tuuakse välja ka üldine ülevaade vaadeldavates tarkvara vigade hindamise mudelites toimuvatest protsessidest.

Võtmesõnad:

Tarkvara vigade hindamine, masinõpe, hinnangumõõdikud

CERCS: P175 informaatika, süsteemiteooria

Table of Contents

1 Introduction

Software defects are an inevitable coproduct of software development. Additionally, software quality assurance is complex and time-consuming. Different software projects do not usually have enough time and people available to eliminate all the faults before the release of a given product and the overall quality of the product and possibly the reputation of a company delivering the product might suffer because of it. In such a situation, the potential value of different methods that can provide alternative ways to assure software quality is huge. Software defect prediction approaches can help focus quality assurance activities on the most defect-prone code and allocate additional resources to fix critical problems. But what kind of software defect prediction models exist and which to choose for the best results?

The present thesis looks to answer the following research questions in regard to the software defect prediction models covered in this work:

RQ1: What kind of defect prediction models have been developed?

RQ2: Which is the best defect prediction model for cross-project defect prediction?

RQ3: Which is the best defect prediction model for within-project defect prediction?

The remainder of this thesis is organized as follows: Section 2 introduces the general process of a software defect prediction model and lists some definitions. Section 3 lists and explains the general idea of the algorithms used within the software defect prediction models covered in Section 5 of this work. Section 4 lists and explains the evaluation measures used in the research covering the software defect prediction models covered in Section 5 of this work. Section 5 shows the general processes within specific software defect prediction models. Section 6 presents some of the results of the evaluation experiments conducted on the software defect prediction models covered in Section 5 of this work in addition to analysis of some of theses results. Section 7 concludes this work.

2 General Process of a Software Defect Prediction Model

2.1 Definitions

First, a few key definitions need to be given to explain some of the processes in a software defect prediction model.

Software metrics are measures of a specific software property. In software defect prediction a set of software metrics are used to extract information about different properties of a software instance (e.g. a file, class, module).

Some of the software metrics are very simplistic whereas others are more complex, e.g. a simple software metric is the LOC (Lines of Code) metric, which is used to measure the total number of lines of code in the instance that it is applied to. A full list of metrics used by some of the software defect prediction models that are covered in Section 5 of this work can be found in Appendices I – IV.

Feature extraction is the process of applying software metrics to a software instance.

Labels are values used to mark whether software instances are defective (i.e. a defect is known to exist within the instance) or clean (i.e. no defects exist within the instance).

Classifiers are machine learning methods that can be used as predictors of a software defect prediction model, predicting the label (or classification) of a software instance.

2.2 General Defect Prediction Process

In the case of a software defect prediction model using a machine learning classifier, the general process contains the following steps. A dataset consisting of a set of instances with known labels (i.e. defective or clean) is used as an input for the software defect prediction model. Feature extraction is used on each instance in the dataset extracting the specified set of metrics from each instance. Next, all the sets of metrics combined with the corresponding label of the software instances are used to train a prediction model that is using a machine learning classifier. Finally, after the prediction model has been trained, the model is given new software instances without a label and the model predicts whether they should be labelled defective or clean.

3 Algorithms Used by Software Defect Prediction Models

In this section of the work, the algorithms used within the software defect prediction models covered in Section 5 of this work will be listed with the general idea behind the process of each algorithm regarding software defect prediction.

Logistic Regression (LR) – a statistical method used for classification in dataset where there are one or more independent variables that determine the outcome. The classification result is the value of one of two possible outcomes.

Naïve Bayes (NB) – a classification method based on the Bayes' rule that finds the conditional probability of an instance being labelled with a specific value from the set of labels. The label with the highest probability is chosen as the final classification.

Random Forest (RF) – a classification method consisting of a collection of tree predictors that are each used to classify an unknown instance. The final classification for the unknown instance is chosen using the majority result of the trees' predictions.

K-Nearest Neighbour (KNN) – non-parametric decision procedure which classifies an unknown instance in the category of its nearest neighbour.

Support Vector Machine (SVM) – a machine learning method that can be used for classification. Given a set of labelled data where there are two possible label classes, the algorithm builds a model mapping the data as points in a space so that the two separate classes of labelled data are divided by a clear gap as wide as possible. The model is then used to map unknown data into the previously mentioned space and predict the label class of the unknown data based on which side of the gap they are mapped.

Artificial Neural Network (ANN) – a machine learning method based on a model that can be used for classification. An ANN model consists of layers of units called neurons. The layers are typically called the input layer, hidden layer and output layer. More than one hidden layer can exist between the input and output layers. Training the ANN model with a set of data with known labels, the ANN model can learn to predict the values of unknown data.

K-Means Clustering (KM) – a method used to partition a set of data into a specified number of clusters in which each instance from the dataset belongs to the cluster with the nearest mean value.

Particle Swarm Optimization (PSO) – a method that can be used for optimizing parameter values. Particles move around in a search space trying to improve in terms of a given measure of quality. The movement of each particle is influenced by its best known position, but is also guided toward the best known positions in the search space by other particles. This is expected to move the swarm of particles toward the best solution.

Inter Quartile Range function (IQR) – a method used to detect where the bulk of the values lie in a dataset. If the range of the dataset is from the minimum value in the dataset to the maximum value, this method can be used to help detect the values ranging from 25% to 75%.

Genetic Algorithm (GA) – a method that can be used for optimizing parameter values. An initial population of candidates to a solution is randomly selected from the search space. The population is then evolved towards a better solution by mutating and altering the properties of the candidates.

Ensemble Learning (EL) – a machine learning method that can be used to improve the prediction accuracy of machine learning classifiers. Multiple classifiers are trained to solve the same problem and then combined for stronger generalization ability.

Transfer Learning (TL) – a machine learning approach that aims to transfer the knowledge learned on one dataset and use that knowledge to help solve problems in a different dataset.

Boosting (B) – a machine learning method that can be used to improve the prediction accuracy of machine learning classifiers by combining a set of weak classifiers to create a strong classifier.

4 Evaluation Measures for Software Defect Prediction Models

In this section of the work, some of the measures used to evaluate the performance of software defect prediction models covered in Section 5 of this work will be listed and explained.

In the case of software defect prediction models, there are four possible outcomes for an entity after a prediction is made about whether the entity is defective or clean. The outcomes are as follows [1]:

- A defective entity is classified as defective (true positive, TP)
- A defective entity is classified as clean (false negative, FN)
- A clean entity is classified as clean (true negative, TN)
- A clean entity is classified as defective (false positive, FP)

Based on these outcomes, measures for evaluating the accuracy of a software defect prediction model are defined. The most popular measure used for evaluating the performance of a defect prediction model in the research covering the software defect prediction models in this work is F-score, which is the harmonic mean of precision and recall. Higher F-score is an indication of a better model. Precision, recall and F-score can be defined as follows:

Precision: the proportion of entities correctly classified as defective (TP) among all entities classified as defective (TP + FP).

$$Precision = \frac{TP}{TP+FP} \tag{1}$$

Recall: the proportion of entities correctly classified as defective (TP) among all defective entities (TP + FN).

$$Recall = \frac{TP}{TP+FN} \tag{2}$$

F-score: harmonic mean of precision (1) and recall (2).

$$F\text{-}score = \frac{2*Precision*Recall}{Precision+Recall} \tag{3}$$

F-score is not the only measure used in the research covering the software defect prediction models of this work to evaluate the prediction accuracy of a defect prediction model. A few other measures that were used are accuracy and G-measure. Accuracy can be defined as follows:

Accuracy: the proportion of entities correctly classified (TP + TN) among all the entities (TP + TN + FP + FN).

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{4}$$

To define G-measure, a few additional measures are defined as follows:

Probability of False Alarm (PF): the proportion of clean entities wrongly classified as defective (FP) among all clean entities (FP+TN).

$$PF = \frac{FP}{FP+TN} \tag{5}$$

Specificity: the proportion of clean entities correctly classified as clean (TN) among all clean entities (FP + TN).

$$Specificity = 1 - PF = \frac{TN}{FP+TN} \tag{6}$$

G-measure: the harmonic mean of recall (2) and specificity (6).

$$G\text{-}measure = \frac{2*Recall*Specificity}{Recall+Specificity} \tag{7}$$

Out of 7 software defect prediction models covered in Section 5 of this work, F-score is reported in the research of 4 models (HYDRA [2], TCA+ [3], TCANN [4], extRF [5]), accuracy is reported in the research of 2 models (FixCache [6], P-SVM [7]) and G-score is reported in the research of 1 model (Peters filter [1]) for evaluating the performance of the model.

5 Examples of Software Defect Prediction Models

In this section of the work, the general process of the following software defect prediction models will be covered: FixCache [6], HYDRA [2], TCA+ [3], Peters Filter [1], extRF [5], TCANN [4], P-SVM [7].

5.1 FixCache

FixCache [6] is a software defect prediction algorithm that uses the concept of bug localities to predict future faults in a software system at the file and entity level. The algorithm uses the change history of a software project, yielding a small subset of the project's files or functions/methods that are most fault-prone and with each fix caches the location of the fixed fault itself, any locations changed together with the fault, recently added locations and recently changed locations. The cache can be used by a developer or a tester at the moment a fault is fixed to detect likely fault-prone locations that may contain additional software faults, which is useful for prioritizing verification and validation resources in a software project.

5.1.1 Bug Localities

FixCache uses the concept of bug localities to fetch files or entities into the cache. Four localities are used as an assumption about where software faults may appear.

Temporal locality is based on the intuition that faults are not introduced individually and uniformly over time, but rather appear in bursts within the same entities. "In other words, when a fault is introduced to an entity, another fault will likely be introduced to the same entity soon." An explanation for such bursts is that the changes a programmer makes might be based on a poor or incorrect understanding, thus injecting multiple faults into the software in the process of making changes.

Spatial locality relies on the intuition that when an entity has a fault, other nearby entities might also be faulty. The explanation for such intuition is that when a programmer makes changes based on incorrect or incomplete knowledge, they likely cannot assess the impact of their modifications.

Spatial locality uses the notion of nearby entities. The distance between software entities in this algorithm is measured through logical coupling which is defined as follows: "Two entities are close to each other (logically coupled) when they are frequently changed together." The distance between any two entities *e1* and *e2* is computed using formula (8)

$$distance\ (e1, e2) = \begin{cases} \frac{1}{count(\{e1,e2\})} & , \quad count(\{e1, e2\}) > 0, \\ \infty & , \quad otherwise, \end{cases} \tag{8}$$

where *count({e1, e2})* is the number of times *e1* and *e2* have been changed together.

Changed-entity locality is based on the idea that a recently changed entity is likely to contain a fault.

New-entity locality is similar to changed-entity locality, but is based on the idea that an entity added to a system most recently likely contains a fault.

5.1.2 Operation of the Cache

FixCache maintains a cache of what it has chosen as the most fault-prone entities. There are several parameters, techniques and policies that can be modified in the cache algorithm affecting the size of the cache and the content maintained by the cache. The basic process of the cache algorithm is shown in Figure 1.

FixCache waits until a fault is fixed and the cache is then updated based on the localities that existed at the time the fault was introduced. The hit rates are computed at the time of the fix.

When a fault is missed in an entity, the algorithm uses spatial locality to load nearby entities into the cache. The notion of block size from cache terminology is used to describe the upper bounds on how many entities are loaded. With block size *b*, *b-1* closest entities along with the faulty entity itself are loaded. Block size is also a configurable parameter in this algorithm.

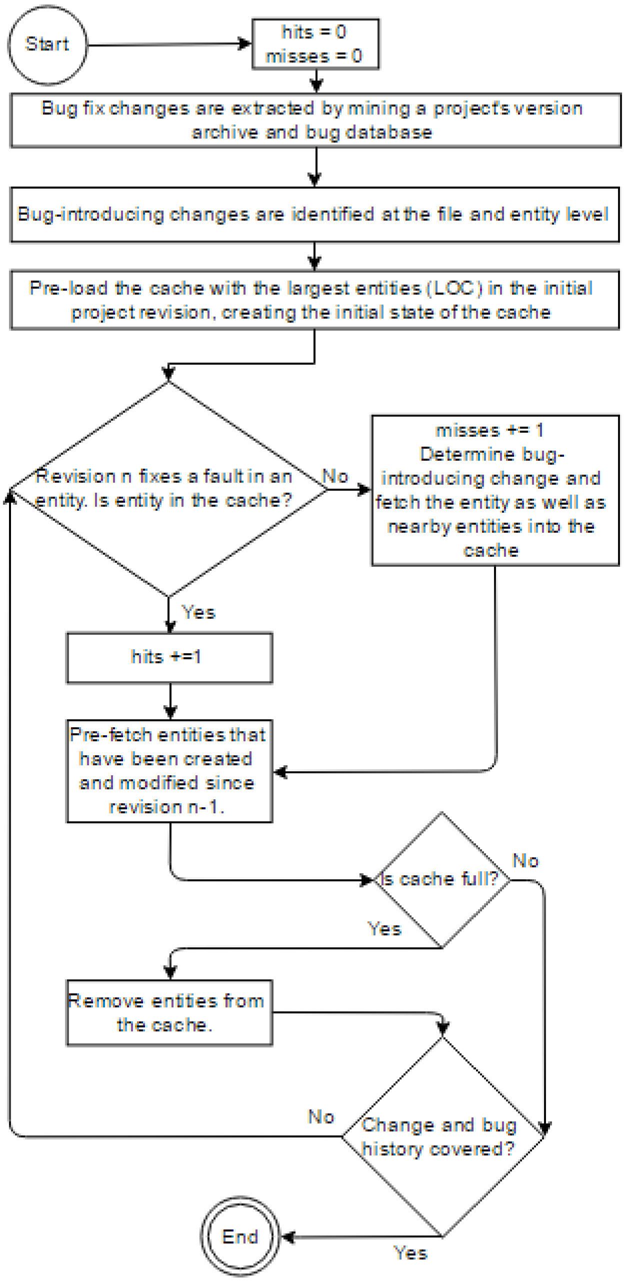
Start
hits = 0
misses = 0
Bug fix changes are extracted by mining a project's version archive and bug database
Bug-introducing changes are identified at the file and entity level
Pre-load the cache with the largest entities (LOC) in the initial project revision, creating the initial state of the cache
Revision n fixes a fault in an entity. Is entity in the cache?
No
misses += 1
Determine bug-introducing change and fetch the entity as well as nearby entities into the cache
Yes
hits +=1
Pre-fetch entities that have been created and modified since revision n-1.
Is cache full?
No
Yes
Remove entities from the cache.
No
Change and bug history covered?
End
Yes

Figure 1. Basic process of the cache algorithm.

Pre-fetching techniques are used to improve the hit rate of the cache. Entities are loaded into the cache for which a fault has not yet been encountered. This is to avoid the inevitable misses when starting with an empty cache. Also, it would otherwise be impossible to predict faults for entities with just one fault in their lifetime. Pre-fetching is done at two different parts of the algorithm: for initializing the cache with entities and per revision. Initially, entities likely to have faults as predicted by greatest lines of code (LOC) are loaded into the cache. With each revision pre-fetching is used to load entities that were modified or created between two revisions starting with entities that have the highest number of LOC. In addition, entities that were deleted between revisions are unloaded from the cache. The maximum number of pre-fetches per revision is controlled by the *pre-fetch size* parameter and can be modified.

Cache replacement policies are used by the algorithm when the cache is full to unload entities from the cache and make room for new entities. These policies describe which entities to unload first. Three policies were tested in the research [6] for the algorithm:

- least recently used (LRU) - unloads the entity that has the least recently found fault,
- LRU weighted by number of changes (CHANGE) - unloads the entity with the least number of changes,
- LRU weighted by number of previous faults (BUG) - unloads the entity with the least number of faults.

5.2 HYDRA

HYDRA (Hybrid Model Reconstruction Approach) [2] is a cross-project software defect prediction approach that contains two steps: the model building step where the defect prediction model is built and the prediction step where an entity is predicted to be defective or clean. Cross-project defect prediction approaches take training data from different projects to predict defects in a target project. This can be useful, because a new project might not have enough usable data to train a defect prediction model.

5.2.1 Overall Architecture

The overall architecture of HYDRA can be seen in Figure 2. HYDRA contains two steps which are model building and prediction. In the model building step a cross-project prediction model is built using the information learned from the instances (i.e., a class , file or module) of the multiple source projects and 5 percent instances from the target project. Both source project instances and target project instances used have to be previously labelled as either defective or clean. In the prediction step, the model is applied to predict if an unlabelled instance in the target project has defects or not.

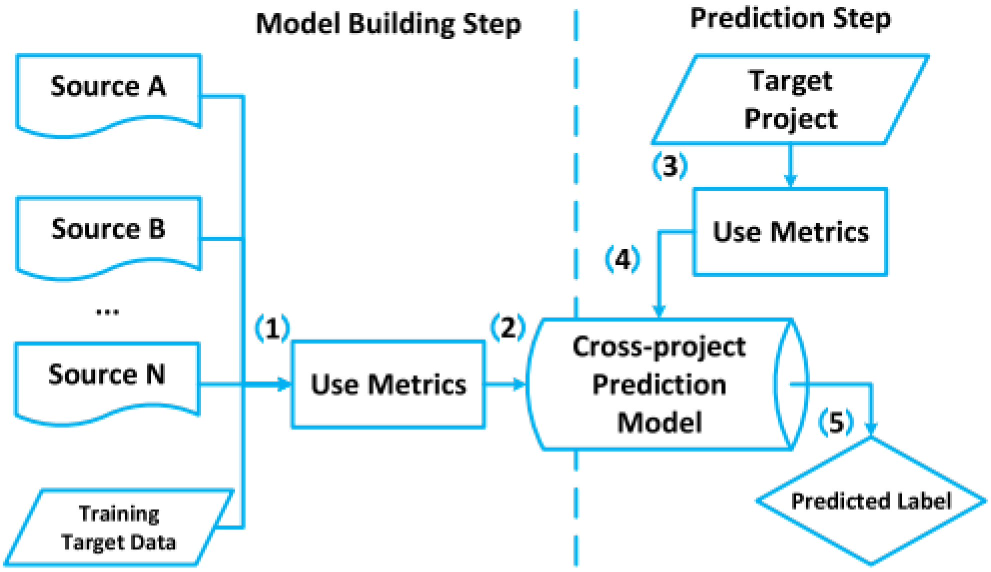

Figure 2. Overall architecture of HYDRA [2].

Various types of metrics are used from the source projects and the target project (Step 1) to build a cross-project defect prediction model based on the previously used metrics (Step 2). The built model is a machine learning classifier that labels an instance to be defective or clean based on the metrics of the instance. After the model is constructed, previously used metrics are extracted from an unlabelled instance of the target project (Step 3) and the values of these metrics are used in the prediction model (Step 4). The model will then output the prediction result for the unlabelled instance predicting it to be either defective or clean (Step 5).

5.2.2 Metrics for Defect Prediction

Different types of metrics can be used for building a software defect prediction model. A list of metrics used for building the prediction model in the evaluation experiments of HYDRA can be found in Appendix II.

5.2.3 Defect Prediction Model

The defect prediction model in HYDRA is built in two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase. The built model is shown in Figure 3.

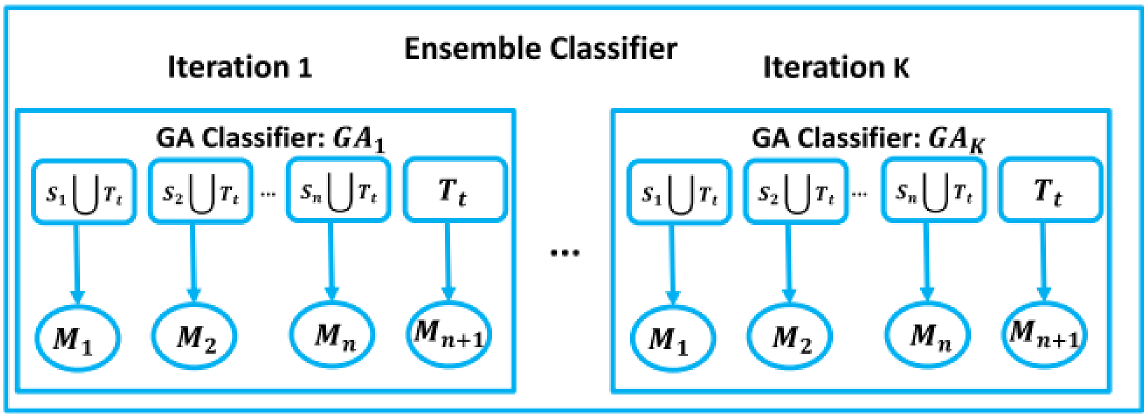

Figure 3. Model built using HYDRA [2].

In the GA phase, a classifier *M_i* is built for each source project *S_i* and training target data *T_t*. A total of *(N+1)* classifiers are built in this phase. After the classifiers are built, HYDRA finds the best composition of these classifiers using genetic algorithm (GA). The composition found is referred to as the GA classifier. In the EL phase, the GA phase is run multiple times producing multiple GA classifiers that are composed according to their training error rate.

5.3 TCA+

TCA+ [3] is an improved version of the transfer learning method TCA (Transfer Component Analysis) [8] which is used for cross-project software defect prediction. „TCA aims to find a latent feature space for the data of both the source and target projects by minimizing the distance between the data distributions while preserving the original data properties“ [3]. After finding the latent space, data of both the source and target projects are mapped onto it for the purpose of discovering a new feature representation for both of the projects. The discovered representation is used to transform the data to reduce the data distribution difference between the source and target projects. At the end, a classifier is trained on the transformed source project data and applied to the transformed target project data for prediction. TCA+ improves TCA on choosing a suitable normalization, a data preprocessing technique [9], before applying TCA for prediction.

5.4 Peters Filter

Peters filter [1] is a cross-project software defect prediction approach that creates a filtered set of training data for a defect predictor to train on which then predicts the instances of a target project to be defective or clean.

5.4.1 Filtered TDS (Training Data Set)

The idea behind the Peters filter is that carefully chosen training data can result in better cross-project defect prediction performance [10]. To create a filtered set of training data Peters filter labels instances from training data sets with their nearest instance from the test data set. Then, each of the test instances reports the closest training instance and the filtered training data set is formed by combining the reported training instances into a single set of data. Visual representation of the processes in the Peters filter are shown in Figure 4. White circles are test instances and black circles with colored borders are training instances from different projects.

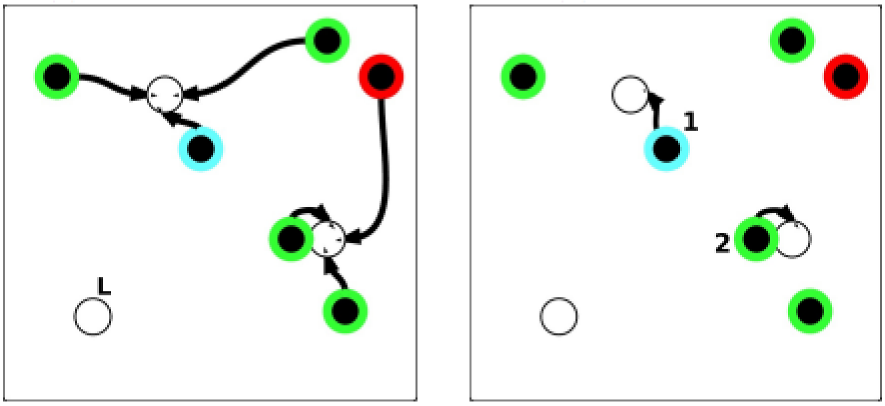

Figure 4. Illustration of the processes in the Peters filter [1].

As seen in Figure 4, having the instances from the training data set select their nearest test instance can leave some test instances without a candidate to contribute to the filtered set of training data. In this case, test instances do not contribute a candidate for the filtered set of training data. In addition, if the nearest training instance is a duplicate of the test instance, it will not be chosen for the filtered set of training data and instead the next closest instance is selected (if one exists).

5.5 ExtRF

ExtRF [5] is a within-project semi-supervised software defect prediction approach that is an extension of the Random Forest (RF) approach.

5.5.1 Overview of the extRF Model

Visual representation of the processes within the extRF approach are shown in Figure 5 and briefly explained below.

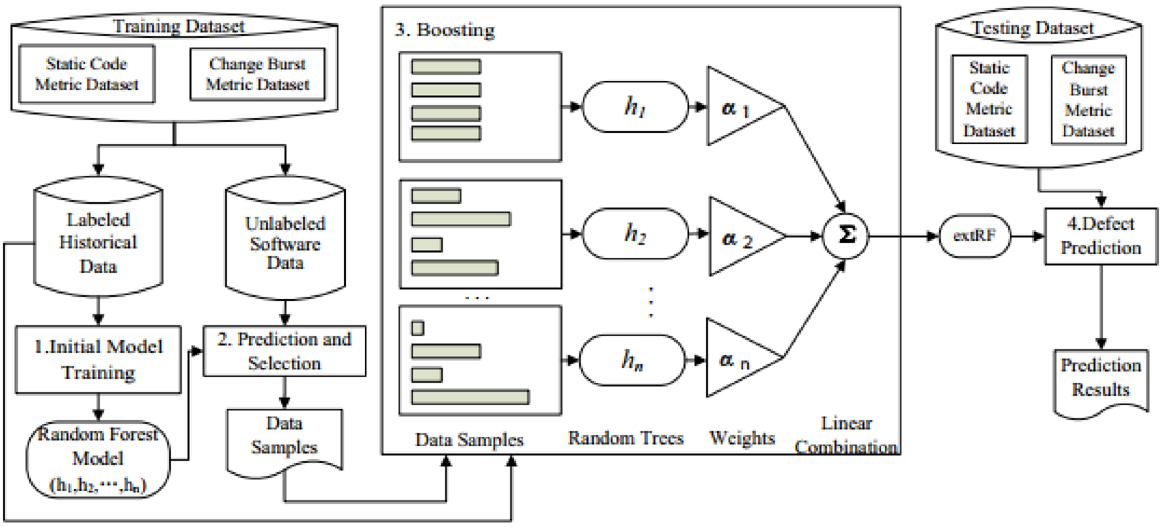

Figure 5. Overview of the extRF Approach [5].

The extRF approach contains four steps. First, a small sample of labelled data from a project's history is used to train a Random Forest (RF) prediction model (Step 1). The resulting model is then used to predict whether unlabelled data is defective or not (Step 2). Confidence of the data samples is calculated using the majority voting schema and the most confident data samples with their voting labels are combined with the initial labelled dataset to form a new dataset. Then, a boosting process is carried out assigning weights to each of the data samples in the new dataset (Step 3). Finally, the resulting classifier is used for defect prediction on new software entities (Step 4).

5.6 TCANN

TCANN (Transfer Component Analysis Neural Network) [4] is a cross-project software defect prediction approach that combines methods for noise reduction in data, transfer learning between source and target datasets and for dealing with class imbalance.

5.6.1 TCANN Model

The TCANN model consists of three components: a data preprocessing component, a data transfer component and a dynamic sampling component based on a neural network. Visual representation of the model is shown in Figure 6.

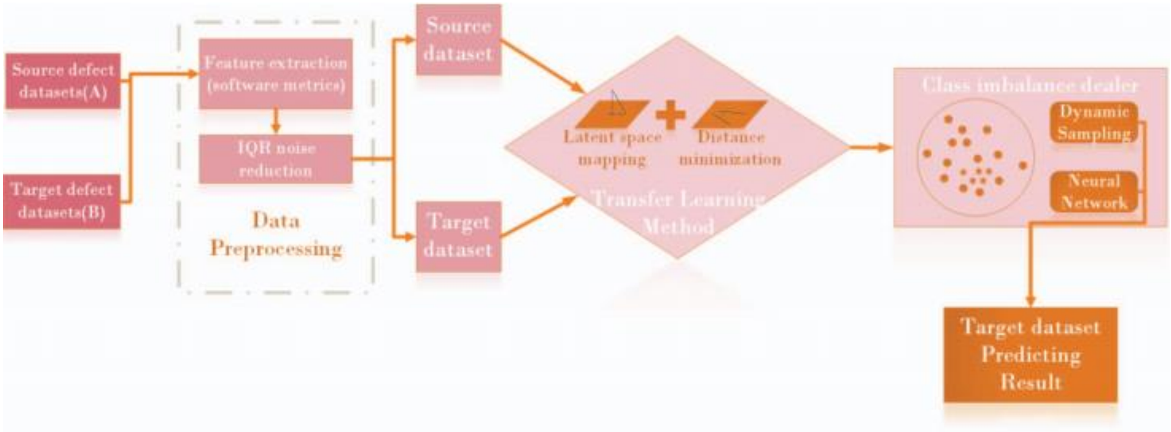

Figure 6. Processes of the TCANN model [4].

First, the source and target datasets are preprocessed to remove noise from the data. This is done using the Inter Quartile Range (IQR) function to detect and remove outliers from the data. Next, TCA (Transfer Component Analysis) [8] will be used to reduce the data distribution differences between the source and target data. Finally, to manage the problem of class imbalance, a situation where "the number of instances in one class greatly outnumbers the number of instances in the other class" [11], an artificial neural network (ANN) is used. The neural network also acts as the final classifier that labels the entities and outputs the result.

5.7 P-SVM

P-SVM [7] is a within-project software defect prediction model using Particle Swarm Optimization (PSO) and Support Vector Machine (SVM) algorithms. The P-SVM model was proposed due to the fact that the prediction accuracy of the SVM model is greatly influenced by its parameters and the SVM model usually adopts the trial-and-error method for determining its parameters, which can lead to poor prediction results. The P-SVM model uses PSO to optimize the parameters for SVM and then uses SVM with the optimized parameters to predict whether software entities are defective or not. A general overview of the processes within the P-SVM model is shown in Figure 7.

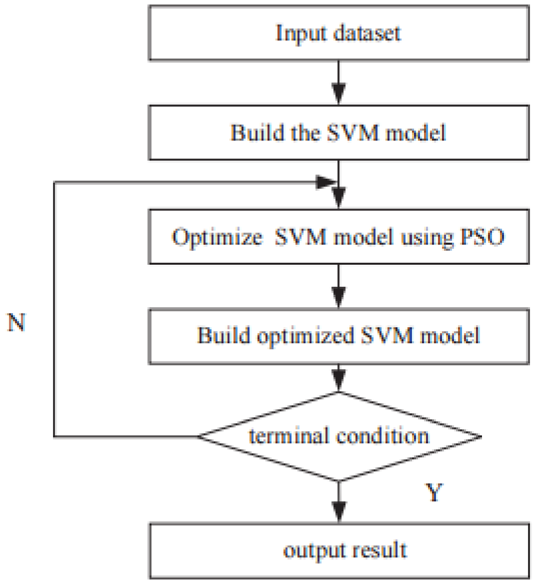

Figure 7. Overview of the processes in the P-SVM model [7].

6 Prediction Results Analysis

In this section of the work, the prediction results of the software defect prediction models will be covered and some of the results analyzed.

The software defect prediction models covered in this work have been evaluated with a number of different evaluation measures. One of the common measures used for evaluation is F-score. In addition to evaluation measures, the experiments conducted for evaluating the models have many other differences. Some of the differences between the evaluation experiments with averaged prediction results can be seen in Table 1.

Table 1. Average F-score results of evaluation experiments conducted on TCA+, HYDRA, Peters Filter, TCANN, extRF and P-SVM.

Model (Classifier)	Dataset	Within-Project / Cross-Project	Average F-score
TCA+ (LR) [3]	ReLink	Cross-Project	0.61
TCA+ (LR) [3]	AEEEM	Cross-Project	0.41
TCA+ (LR) [2]	PROMISE	Cross-Project	0.43
HYDRA (LR) [2]	PROMISE	Cross-Project	0.54
Peters Filter (LR) [2]	PROMISE	Cross-Project	0.40
TCANN (ANN) [4]	ReLink	Cross-Project	0.66
TCANN (ANN) [4]	AEEEM	Cross-Project	0.41
TCANN (ANN) [4]	ReLink	Within-Project	0.71
TCANN (ANN) [4]	AEEEM	Within-Project	0.54
extRF (extRF) [5]	Eclipse Set	Within-Project	0.53
P-SVM (SVM) [7]	JM1	Within-Project	0.82

In Table 1, the average F-score result was calculated manually for P-SVM, because the research covering the model [7] did not provide an F-score result, but presented data about the outcomes of the prediction results using the model (i.e. true positive, true negative, false positive, false negative) and characteristics of the testing data (i.e. number of total data, number of defective data, number of clean data). The classifiers used within the models are Logistic Regression (LR), Artifician Neural Network (ANN) and Support Vector Machine (SVM). The datasets that the experiments were conducted on are as follows: PROMISE refers to the dataset used in the experiments of HYDRA, that was originally collected by Jureczko et al. [12], AEEEM and ReLink refer to the datasets used in the experiments of TCANN and TCA+, that were originally collected by D'Ambros et al. [13] and Wu et al. [14] respectively, Eclipse Set refers to the dataset used in the experiments of extRF [5], which come from the Eclipse platform, JM1 refers to the dataset used in the experiments of P-SVM [7], that come from a project of NASA. The change burst software metrics [15] associated with the experiments of extRF can be found in Appendix I. The software metrics associated with the experiments performed on the datasets of PROMISE [12], AEEEM [13] and ReLink [16] can be found in Appendix II, Appendix III and Appendix IV respectively.

Each row of Table 1 shows the average F-score result of all the results obtained from experiments using a given combination of model, dataset and model training approach. For example, two of the TCANN experiments were conducted on the same dataset (ReLink), but they use different approaches to train the prediction model. In the first case, the prediction model is trained using training and testing data from the same project (within-project defect prediction) within the dataset. A dataset usually consists of data from many different projects. An evaluation experiment of the model is conducted on each project in the dataset resulting with an F-score measuring the prediction accuracy of the model. All of these F-score results are then added together and divided by the number of results to obtain the average F-score. However, in the second case the prediction model is trained using training data from one project and testing data from another project (cross-project defect prediction) within the dataset. In such cases, experiments are conducted for all possible combinations of using a project both as a source for training data (source project) and as a target for testing data (target project) and similarly to the first case, the average F-measure is calculated over all the results divided by the number of results.

The first observation from Table 1 is based on the average F-score results of the previous example (TCANN, ReLink, Within-Project compared to TCANN, ReLink, Cross-Project) which are 0.71 and 0.66 for within-project and cross-project model training approaches respectively. The within-project approach has a better performance compared to the cross-project approach. Another example of similar results in Table 1 can be seen for the TCANN experiments done on the AEEEM dataset with the average F-scores for those experiments being 0.54 and 0.41 for the within-project approach and cross-project approach respectively. These results alone are not conclusive enough to make a generalization about the performance of within-project defect prediction models being better compared to cross-project defect prediction models, but they serve as an example to support the results of the research done by Turhan et al. [10]. A possible reason for the difference in performance is that a model that is trained on one project might not generalize well to other projects [17].

The second observation from Table 1 is that the average F-score results for the same model using the same model training approach differ from each other based on the dataset that the experiments were conducted on. For example, the average F-score results for TCA+ are 0.61, 0.41 and 0.43 for experiments conducted on the ReLink, AEEEM and PROMISE datasets respectively. While the difference in the average F-score result between the AEEEM and PROMISE datasets is minor (0.02), the differences between the results of AEEEM and PROMISE compared to the result of ReLink are notable (0.2 and 0.18 respectively). Similar results can be seen for TCANN experiments conducted on ReLink and AEEEM with the average F-score result being 0.66 and 0.41 respectively for the experiments using the cross-project approach, 0.71 and 0.54 respectively for the experiments using the within-project approach. This might be caused due to the data distribution in the specific datasets. A good example to support this claim can be seen in the experiment results of the extRF model [5] where extRF achieves high F-score results in the experiments conducted on 2 out of 4 datasets used for the experiments and on the other 2 datasets where the results were lower, the overall percentage of defective modules within the datasets was under 40%.

The third observation from Table 1 is that the best defect prediction model for cross-project defect prediction based on the average F-score result achieved by a model is TCANN with an average F-score result of 0.66 achieved in the experiment conducted on the ReLink dataset. However, this result may only reflect the TCANN model’s ability to achieve high F-score results in optimal conditions as the experiments were conducted on the ReLink dataset which are shown to yield high average F-score results for all models in Table 1 that used the ReLink dataset for evaluation experiments. Alternatively, when leaving aside the results of the experiments conducted on the ReLink dataset and look at the results of the other cross-project experiments conducted on the AEEEM and PROMISE datasets, HYDRA is shown to achieve the best average F-score of 0.54 on the PROMISE dataset. Also, the results of the experiments using the AEEEM and PROMISE datasets are more generalised compared to the results of ReLink as the AEEEM and PROMISE datasets are both much larger compared to ReLink in terms of overall amount of data in the dataset and the amount of projects in the dataset.

The fourth observation from Table 1 is that the best defect prediction model for within-project defect prediction based on the average F-score result achieved by a model is P-SVM with an average F-score result of 0.82 achieved in the experiment conducted on the JM1 dataset. However, similarly to the previous observation about the best cross-project defect prediction model, this result may only reflect the P-SVM model’s ability to achieve high F-score results in optimal conditions as the experiments were conducted on a 1000 randomly chosen modules from the JM1 dataset [7]. The next best result is obtained by TCANN with the average F-score being 0.71 on the ReLink dataset, but this again might not generalise well to other projects due to ReLink being a small dataset and showing high results for all experiments conducted on it. Leaving the results of these two experiments aside, TCANN still achieves the next best average F-score of 0.54 on the AEEEM dataset, but compared to the average F-score of 0.53 achieved by the extRF model on the Eclipse Set, no clear winner of the two can be chosen based on the average F-score results achieved. Due to these reasons and the difference between the average F-scores of P-SVM and the next best TCANN result (0.11), it would still be reasonable to believe that P-SVM is the best defect prediction model for within-project defect prediction out of the models covered in this work.

6.1 Answers to Research Questions

RQ1: What kind of defect prediction models have been developed?

The specific software defect prediction models in the literature have been covered in Section 5 of this work, however due to only a small number of software defect prediction models being covered in this work, no conclusive answer can be given.

RQ2: Which is the best defect prediction model for cross-project defect prediction?

Considering the third observation from the results analysis, the best defect prediction model for cross-project defect prediction based on the average F-score alone is TCANN. However, the result achieved is questionable in terms of the model’s ability to generalise well to other projects. Additionally, HYDRA shows the next best result on a larger dataset consisting of more data and projects, which is a basis for considering HYDRA to be the best cross-project defect prediction model out of the other cross-project models covered in this work.

RQ3: Which is the best defect prediction model for within-project defect prediction?

Considering the fourth observation from the results analysis, the best defect prediction model for within-project defect prediction based on the average F-score alone is P-SVM. However, similar concerns in regards to RQ2 and the third observation arise about the P-SVM model’s ability to generalise well to other projects, but due to reasons listed in the fourth observation P-SVM can still be considered the best defect prediction model for within-project defect prediction out of the other within-project defect prediction models covered in this work.

7 Conclusions

One of the aims of this work was to give a general understanding of some of the processes of software defect prediction models. In that regard, the general process of software defect prediction models using machine learning classifiers was explained and some key definitions for understanding the processes within the software defect prediction models were given. Additionally, algorithms used within the specific software defect prediction models covered in this work were listed with a brief explanation of what the algorithm does.

Some of the evaluation measures used to evaluate the prediction accuracy of software defect prediction models covered in this work were listed and explained to better help understand how these measures are formulated and what these measures indicate.

A handful of specific software defect prediction models were presented and a brief overview of the processes within those software defect prediction models was given. In addition, some of the results of the evaluation experiments conducted in the research of the models covered in this work were listed. These results were analyzed with the purpose of providing answers to some of the research questions set in this work.

Future work can be done by studying one of the software defect prediction models more extensively to find out if a practical application of the model in the context of a software engineering company would be beneficial in terms of quality assurance.

8 References

[1] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 409-418, May 2013.

[2] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively Compositional Model for Cross-Project Defect Prediction," IEEE Transactions on Software Engineering, vol. 42, no. 10, pp. 977-998, October 2016.

[3] J. Nam, S. J. Pan, and S. Kim, "Transfer Defect Learning," 2013 35th International Conference on Software Engineering (ICSE), pp. 382-391, September 2013.

[4] Q. Cao, Q. Sun, Q. Cao, and H. Tan, "Software defect prediction via transfer learning based neural network," 2015 First International Conference on Reliability Systems Engineering (ICRSE), pp. 1-10, October 2015.

[5] Q. He, B. Shen, and Y. Chen, "Software Defect Prediction Using Semi-Supervised Learning with Change Burst Information," 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), pp. 113-122, June 2016.

[6] S. Kim, T. Zimmermann, E. J. Whithead Jr., and A. Zeller, "Predicting Faults from Cached History," 29th International Conference on Software Engineering (ICSE ’07), pp. 489–498, May 2007.

[7] H. Can, X. Jianchun, Z. Ruide, L. Juelong, Y. Qiliang, and X. Liqiang, "A new model for software defect prediction using Particle Swarm Optimization and support vector machine," 2013 25th Chinese Control and Decision Conference (CCDC '13), pp. 4106-4110, May 2013.

[8] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain Adaption via Transfer Component Analysis," IEEE Transactions on Neural Networks, vol 22, no. 2, pp. 199-210, February 2011.

[9] J. Han, M. Kamber, and J. Pei, "Data mining: concepts and techniques," 3rd ed. Waltham, Mass.: Elsevier/Morgan Kaufmann, 2012.

[10] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, vol. 14, no. 5, pp. 540–578, October 2009.

[11] B. X. Wang, and N. Japkowicz, "Boosting support vector machines for imbalanced data sets," Knowledge and Information Systems, vol. 25, no. 1, pp. 1-20, October 2010.

[12] M. Jureczko, and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE '10), pp. 1-10, September 2010.

[13] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," 2010 7th IEEE Working Conference on Mining Software Repositories (MSR '10), pp. 31-41, May 2010.

[14] R. Wu, H. Zhang, S. Kim, and S. Cheung, "ReLink: recovering links between bugs and changes," Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11), pp. 15-25, September 2011.

[15] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change Bursts as Defect Predictors," 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 309-318, November 2010.

[16] Scientific Toolworks, Understand https://scitools.com/support/metrics_list/?metricGroup=complex (11.05.2017)

[17] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE '09), pp. 91-100, August 2009.

Appendix I

Nr.	Metric	Description
1	NumberOfChanges	Number of builds in which a specific component has changed
2	NumberOfConsecutiveChanges	Number of consecutive builds for a given gap size
3	NumberOfChangeBursts	Number of change bursts for a given gap size and burst size
4	TotalBurstSize	Number of changed builds in all change bursts
5	MaximumChangeBurst	Maximum number of changed builds in all change bursts
6	NumberOfChangesEarly	Same as 1, but calculated for the first 80% of the project's lifetime
7	NumberOfConsecutiveChangesEarly	Same as 2, but calculated for the first 80% of the project's lifetime
8	NumberOfChangeBurstsEarly	Same as 3, but calculated for the first 80% of the project's lifetime
9	TotalBurstSizeEarly	Same as 4, but calculated for the first 80% of the project's lifetime
10	MaximumChangeBurstEarly	Same as 5, but calculated for the first 80% of the project's lifetime
11	NumberOfChangesLate	Same as 1, but calculated for the last 20% of the project's lifetime before release
12	NumberOfConsecutiveChangesLate	Same as 2, but calculated for the last 20% of the project's lifetime before release
13	NumberOfChangeBurstsLate	Same as 3, but calculated for the last 20% of the project's lifetime before release
14	TotalBurstSizeLate	Same as 4, but calculated for the last 20% of the project's lifetime before release

15	MaximumChangeBurstsLate	Same as 5, but calculated for the last 20% of the project’s lifetime before release
16	TimeFirstBurst	Occurrance of the first burst normalized to the total number of builds
17	TimeLastBurst	Occurrance of the last burst normalized to the total number of builds
18	TimeMaxBurst	Occurrance of the burst with the most changes normalized to the total number of builds
19	PeopleTotal	Number of people who ever committed a change to a specific component
20	TotalPeopleInBurst	Number of people involved across all bursts
21	MaxPeopleInBurst	Maximum number of people involved in a burst across all bursts
22	ChurnTotal	Total churn over the lifetime of a component
23	TotalChurnInBurst	Total churn in all change bursts
24	MaxChurnInBurst	Maximum churn across all bursts

Appendix II

Nr.	Metric	Description
1	WMC	Weighted methods per class
2	DIT	Depth of inheritance tree
3	NOC	Number of children
4	CBO	Coupling between objects
5	RFC	Response for a class
6	LCOM	Lack of cohesion in methods
7	LCOM3	Modified lack of cohesion in methods
8	NPM	Number of public methods
9	DAM	Data access metric
10	MOA	Measure of aggregation
11	MFA	Measure of functional abstraction
12	CAM	Cohesion among methods of class
13	IC	Inheritance coupling
14	CBM	Coupling between methods
15	AMC	Average method complexity
16	Ca	Afferent couplings
17	Ce	Efferent couplings
18	MaxCC	Maximum value of McCabe's cyclomatic complexity
19	AvgCC	Arithmetic mean of McCabe's cyclomatic complexity
20	LOC	Number of lines of code

Appendix III

Nr.	Software Metric	Description
1	WMC	Number of weighted methods
2	DIT	Depth of inheritance tree
3	RFC	Response for a class
4	NOC	Number of children
5	CBO	Coupling between objects
6	LCOM	Lack of cohesion in Methods
7	FanIn	Number of other classes that reference the class
8	FanOut	Number of other classes referenced by the class
9	NOA	Number of attributes
10	NOPA	Number of public attributes
11	NOPRA	Number of private attributes
12	NOAI	Number of attributes inherited
13	LOC	Number of lines of code
14	NOM	Number of methods
15	NOPM	Number of public methods
16	NOPRM	Number of private methods
17	NOMI	Number of methods inherited
18	All Bugs	Number of bugs
19	Non trivial bugs	Number of bugs with greater than trivial severity
20	Major bugs	Number of bugs with greater than major severity
21	Cricital bugs	Number of bugs with critical or blocker

		severity
22	High priority bugs	Number of bugs with greater than default priority
23	HCM	Every file modified is considered equally
24	WHCM	Modified files are considered with a weight
25	EDHCM	Earlier periods have exponentially reduced contribution
26	LDHCM	Earlier periods have linearly reduced contribution
27	LGDHCM	Earlier periods have logarithmically reduced contribution

Appendix IV

Nr.	Software Metric	Description
1	AvgCyclomatic	Average cyclomatic complexity for all nested functions or methods
2	AvgCyclomaticModified	Average modified cyclomatic complexity for all nested functions or methods
3	AvgCyclomaticStrict	Average strict cyclomatic complexity for all nested functions or methods
4	AvgEssential	Average Essential complexity for all nested functions or methods
5	AvgEssentialStrictModified	Average strict modified essential complexity for all nested functions or methods
6	CountPath	Number of possible paths, not counting abnormal exits or gotos
7	Cyclomatic	Cyclomatic complexity
8	CyclomaticModified	Modified cyclomatic complexity
9	CyclomaticStrict	Strict cyclomatic complexity
10	EssentialStrictModified	Strict Modified Essential complexity
11	Knots	Measure of overlapping jumps
12	MaxCyclomatic	Maximum cyclomatic complexity of all nested functions or methods
13	MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods
14	MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods
15	MaxEssential	Maximum essential complexity of all nested functions or methods
16	MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed
17	MaxEssentialStrictModified	Maximum strict modified essential com-

		plexity of all nested functions or methods
18	MaxInheritanceTree	Maximum depth of class in inheritance tree
19	MaxNesting	Maximum nesting level of control constructs
20	MinEssentialKnots	Minimum Knots after structured programming constructs have been removed
21	RatioCommentToCode	Ratio of comment lines to code lines
22	SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods
23	SumCyclomaticModified	Sum of modified cyclomatic complexity of all nested functions or methods
24	SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods
25	SumEssential	Sum of essential complexity of all nested functions or methods
26	SumEssentialStrictModified	Sum of strict modified essential complexity of all nested functions or methods
27	Essential	Essential complexity

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Hans Raukas,

(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Some Approaches for Software Defect Prediction,

(title of thesis)

supervised by Helle Hein,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 11.05.2017