DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2018

Investigation into predicting unit test failure using syntactic source code features

ALEX SUNDSTRÖM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Investigation into predicting unit test failure using syntactic source code features

ALEX SUNDSTRÖM

Master in Machine Learning
Date: July 13, 2018
Supervisor: Hamid Reza Faragardi
Examiner: Mårten Björkman
Swedish title: Undersöking om förutsägelse av enhetstestfel med användande av syntaktiska källkodssärdrag
School of Electrical Engineering and Computer Science

Abstract

In this thesis the application of software defect prediction to predict unit test failure is investigated. Data for this purpose was collected from a Continuous Integration development environment.

Experiments were performed using semantic features from the source code. As the data was imbalanced with defective samples being in minority different degrees of oversampling were also evaluated.

The data collection process revealed that even though several different code commits were available few ever failed a unit test. Difficulties with linking a failure to a specific file were also encountered. The machine learning model used in the project produced poor results when compared against related work, from which it was based on. In F-measure, it on average achieve 53% of the mean performance of state-of-the-art for software defect prediction on bugs in Java source files.

Specifically, it would appear that very little information was available for the model to learn defects in files not present in training data.

Sammanfattning

I denna avhandling undersöks applikationen av prognos för mjukvarudefekter för att förutse enhetstestfel. Data för detta syfte samlades in från en utvecklingsmiljö med kontinuerlig integration.

Experimenten utfördes med användning av semantiska särdrag samlade från källkod. Då data var obalanserat med defekta exempel i minoritet evaluerades olika grader av översampling.

Datainsamlingsprocessen visade att även om det fanns många kodinlämningar så misslyckades få någonsin ett enhetstest. Svårigheter med att länka testmisslyckanden till en specifik fil påträffades också. Den använda maskininlärningsmodellen uppvisade också dåliga resultat i jämförelse med relaterade värk. Mätt i F-measure uppnåddes i genomsnitt 53% av genomsnittlig prestandan av bästa möjliga prognos av mjukvarudefekter av buggar i Java källkod.

Specifikt så framträdde det att väldigt lite information verkar finnas för modellen att lära sig defekter i filer som ej fanns med i träningsdata.

Acknowledgments

I would like to start by thanking my friends and family who have supported me during all of my university studies.

Thank you to the team which I was doing my thesis with for making me feel very welcomed and helping me setting up a development environment for conducting the research.

Thank you to my supervisor Hamid Reza Faragardi for his input on the thesis.

Thank you to Mårten Björkman for acting as my examiner.

Contents

List of Figures

Chapter 1

Introduction

Software Defect Prediction (SDP) is a research area centered around using machine learning or statistical models for predicting the presence of defects, like bugs, in software. One way to collect data of defects is to look through bug reports and linking them to a specific component. Another method is to crawl commits messages in a version control system to identify bug fixes. Once located, the file history can be examined to find the defect [22] [45] [48]. These datapoints can then be linked to specific software versions and/ or components (files, classes etc.).

A common method for developers to catch defects in their code is to employ unit testing. Continues Integration (CI) [8] is a software development practice which employs frequent testing of new code, not only limited to unit testing. In very large software projects where CI is used the process of building the project, such as compiling and testing, can be more efficiently done on remote agents. Agents can receive their own specific task to accomplish, such as performing the unit tests. These tasks are often refereed to as builds, regardless of their purpose.

However, in CI the latency for developers to get test results can still be quite large depending on the test and number of available agents. If a machine learning model could simply look at a component and classify it as clean or defective it could speed up the feedback loop. In this thesis project, the aim is to investigate if SDP could be applied to predict the failure of unit test builds in a CI setup. The project was

conducted at a digital rights management company.

1.1 Thesis Objective

The objective of this thesis is to investigate the use of machine learning on data gathered from a CI environment. Specifically the aim is to use syntactic features in the source code to investigate if they can be used to predict unit test failures. Related works have found that this can be used to detect general bug inducing patterns in code.

When developers submit their code for testing it will more often then not pass the tests[33], resulting is less failed unit test builds then passed ones. Such a situation would lead to imbalanced data, which is a common problem in machine learning. A countermeasure to this is to rebalance the data. This project will also look at how applying this rebalancing would affect the results.

These objectives can be summarized with the following two Research Questions (RQ):

RQ1: *How does a SDP method with syntactic features perform when learning and predicting test case failures in an industrial continuous integration environment?*

RQ2: *How does the balance of the defective and non-defective samples in the dataset impact on the performance of a SDP method when learning and prediction test case failures in an industrial continuous integration environment?*

1.2 Delimitation

The following settings are applied in this thesis:

- Use data from a Java project
- Focus on builds that run unit tests
- Use syntactic features from source code
- Predict defects on file level

- Binary target, defective or not defective

By recommendation of the principal this thesis focused on a project coded in the Java programming language.

Several builds can be run to test code for integration, however unit tests builds are chosen for two reasons. First since they specifically test source code logic, they are closely related to RQ1. Second they are considered the most stable internally by the principal, reducing potential noise in the data.

When collecting features for the final model the syntax of the source code will be what is considered. This is firstly related to the previous constraint. Furthermore, this has also shown good results in related works, see Chapter 3. The choice of granularity (file level) is also motivated by choices made in related works.

The target for the model is a simple binary case for whether a file would fail or pass a unit test build. This is adopted to reduce the complexity of the models.

1.3 Social & Ethical Impact

The work in this thesis can have some possible social impact on society at large. As touched upon, by speeding up the feedback cycle for developers this thesis could speed up development in a CI environment. This follows from the developers potentially having to run their builds fewer times, as code errors could be caught earlier. Putting less stress on the CI agents can also have an economic and ecological impact. By lowering the amount of processing spent on a single new code change, both time and energy could be saved.

From an ethical standpoint, the data used by the machine learning model will have been generated by real developers. This could potentially lead to the model discriminating towards code from certain developers over others. However, the focus of this thesis is to use the syntax of the source code itself and no metadata, such as developer names, surrounding it. This anonymises the data somewhat, lessening ethical dilemmas related to using the model in a production environment. As such, while the possible ethical impact of this thesis should

not be completely ignored it can be considered fairly low.

1.4 Research Methodology

The projected begins with establishing a topic and goals for the research. In the next step, potential challenges related to accomplishing the goals are identified. Afterwards a literature study is started, focused on researching related works.

An iterative process is then commenced, starting with presenting a solution. The solution is implemented and then evaluated. After implementation or evaluation, the solution may need to be changed. If so, the new solution is presented and the process starts over from there. Further study of related works can be motivated based on the evaluation of the solution. A flowchart of this Methodology is shown in Figure 1.1.

1.5 Thesis Structure

Chapter 2 covers the theoretical background relevant to the thesis. Related works are covered in Chapter 3. Topics covered in these two chapter are of relevance to the method, which is presented in Chapter 4. In Chapter 5 the results of the experiments generated using the method are presented. Finally these results are discussed and conclusions are drawn in Chapter 6.

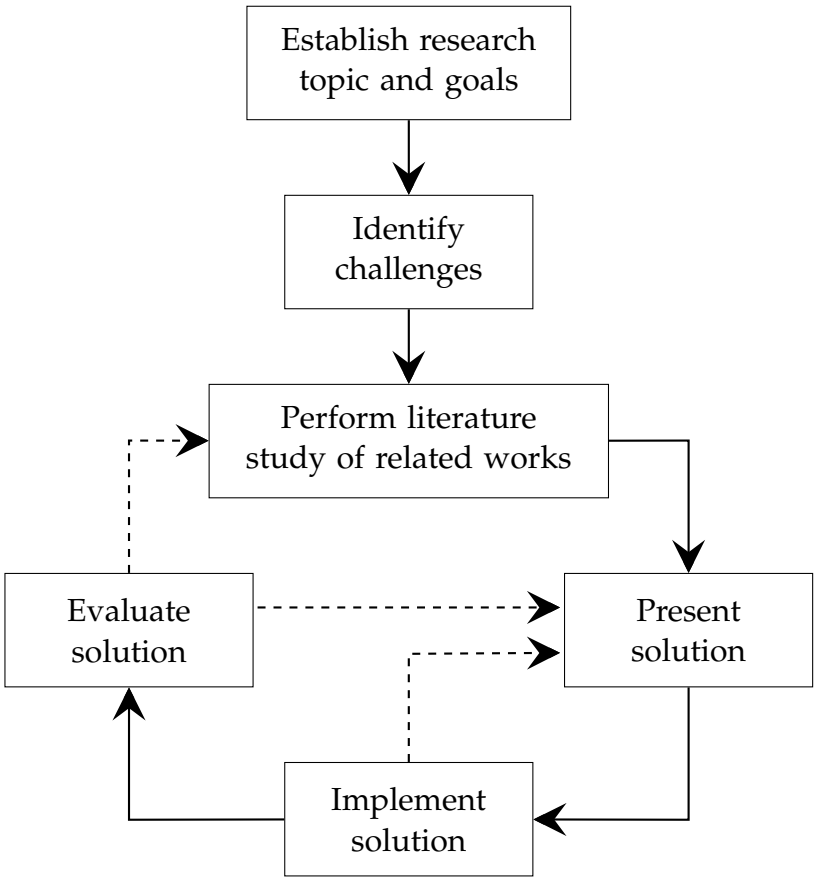

Figure 1.1: *Flowchart depicting the research methodology. Dashed lines indicate that the step is optional and is part of an iterative process.*

Chapter 2

Background

This chapter presents an overview of theoretical background and related works that is relevant to this thesis. In Section 2.1 the concept of CI is introduced. This is relevant as data for the experiments are gathered from such an environment. Section 2.2 explains the theory behind the machine learning components used in this thesis. Background regarding SDP is covered in Section 2.3, including commonly used evaluation metrics.

2.1 Continuous Integration

CI is a concept to encourage faster development and integration of new code [8]. Code changes are automatically tested, to make certain that they can be integrated to the main repository safely. This testing procedure can be handled by a central CI server controlling a set of remote build agents. The purpose is to promote fast and safe integration. This allows new features and changes to be added constantly to the project instead of at set points in development.

One of the possible problem that is associated with CI is that of flaky tests [27]. A test is flaky if it can fail randomly. This could be because of issues which are not related to the actual code being tested. There could be issues with infrastructure causing abnormal behavior or agents not responding, which in turn lead to failures. The tests

themselves can also be a source of flakiness, for example not accounting for race conditions.

2.2 Machine Learning

2.2.1 Neural Networks

A neural network [14] is a machine learning model inspired by brain cells. The purpose of such a network is to model the relationship $y = f(X)$ for some input vector X. For this purpose a cost function is used, which is a function describing how far the model is from the desired solution. Neural nets are made up of several units, or neurons, which are stacked in different layers. Layers are divided into input layers, zero or more hidden layers and one output layer. As seen in Figure 2.1, neurons in two consecutive layers are heavily interconnected. This allows the model to learn relationships between features.

Connections between neurons represents weights. A neurons input is the sum of the preceding layers outputs multiplied by their respective weights. The output is then calculated by running this sum through a non-linear activation function. Once an input has propagated all the way through the network (a forward pass) the cost function is applied.

To make the network learn, the weights in the connections are updated after an input has completed it's forward pass and a prediction has been made. This is done via the backpropagation algorithm, which propagates the error of the networks output to the different layers. When determining how much to update a weight, the partial derivative of the cost function given the weight is calculated. The update is often scaled by a constant, called the learning rate. This is applied to avoid learning being either too sporadic or too slow.

2.2.1.1 Convolutional Layers

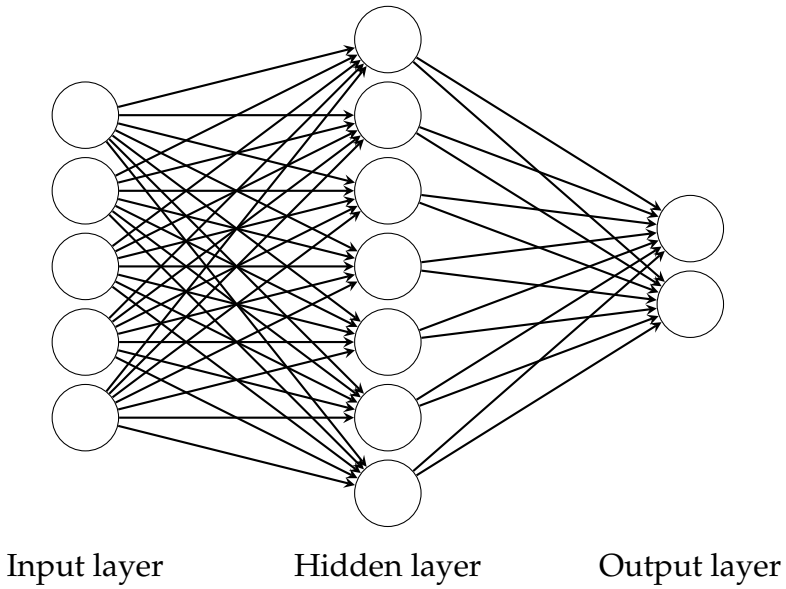

Figure 2.1: *Example of a neural network*

A Convolutional Neural Network (CNN) [14] [10] is a type of neural network which uses convolution. Convolutional layers have a set of filters, sometimes called kernels, which are applied to the entire input. It can be visualized as having a sliding window over the input, see Figure 2.2. Because of this property, a CNN makes use of the same weights for different parts of the input which cuts down on memory requirements. Filters search for patterns in the input and can be trained using the backpropagation algorithm. The output of a convolutional layer is commonly referred to as feature maps.

After applying convolution, it is common to also apply a pooling operation [10]. Pooling looks at a region of the feature map and summarizes it in some way. A popular example is max pooling, which will output the maximum value at the currently considered region. This helps further reduce the dimension of the input for later layers and helps the network become resistant to small translations in the input.

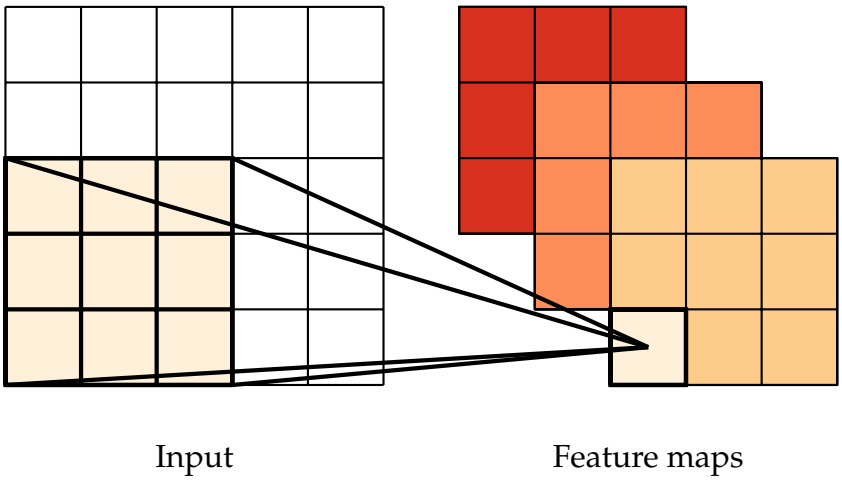

Figure 2.2: *Example of 2-dimensional convolution with a* 3×3 *filter. In this example there are three feature maps, all which have their own filter weights.*

2.2.1.2 ReLU Activation Function

As was described in Section 2.2.1 the output of a neuron is determined in part by an activation function. A popular example of such a function is the Rectified Linear Unit [37], most often referred to as *ReLU*. The ReLU activation function is of the form $\max(0, X)$ where X is the input, or sums of input, to the neuron. It has been shown to be superior choice compared to other activation functions [9], most notably in deep networks. A plot of the function is shown in Figure 2.3.

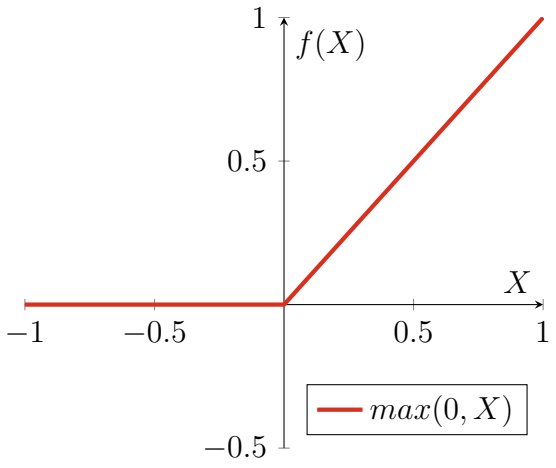

Figure 2.3: *Plot of the ReLU activation function*

2.2.1.3 Adam Optimizer

During training of a neural network, the weights will be updates according to the gradient of the loss given the weights. A learning rate will often be multiplied with this gradient to adjust how sensitive the model is to change. *Adam* [25] is a parameter optimizer which extends this idea by having this sensitivity be dynamic instead of static.

While the Adam algorithm is still supplied with a fixed learning rate, it also relies on a dynamic component to provide adaptive learning of parameters. This is accomplished firstly by estimating the first and second moment of the gradients. These estimations are then used to update two exponential moving averages. The moving averages and the learning rate are then what is used to help update the parameters.

2.2.2 Word Embedding

To apply machine learning to learn from text, words can be represented by continuous vectors [17]. When words are embedded into vectors space representation, relationship between words are automatically learned [34]. Computing these representations and relationships have proven to be both computationally efficient and to aid in building effective models [35].

These vector representations can be used together with the CNNs described in Section 2.2.1.1 to achieve good results in natural language processing tasks [24]. In this task instead of having the sliding window applied in two dimensions of the input like in images, it is only applied in one. The width of the convolutional filters is the same as the width of the word embedding vectors and convolution is applied over a number of full word representations. It can be seen as applying one dimensional convolution on all different N-grams in the input, where a single "*gram*" is a complete word embedding instead of a word.

2.2.3 K-Fold Cross Validation

After a ML model has been trained, it has to be evaluated. Since the interest is to find out how well the model generalizes to new data, some datapoints can be withheld during training and be used to test the model afterwards. One such way of partitioning the data into test and training sets is K-Fold Cross Validation [26]. This method splits the data into K "*folds*". The model is exposed to $K - 1$ of these folds during its training phase and the one left out is used for testing. An advantage of this model is that every sample is used for both training and testing. In total K models are trained, each with a different set used for testing. Once all models have been evaluated, the final result is the average performance from all K runs to reduce variance.

2.3 Software Defect Prediction

2.3.1 Features

Features are the inputs to a machine learning model. In SDP several different kinds of features have been used. Examples include features based on source code complexity metrics [13] or properties of object oriented languages [5]. In more recent years several studies have been conducted on using the semantic information in source code as features [47] [19] [40] [41] [19] [18] [6].

2.3.2 Dataset Balance

When gathering data for machine learning classification, it is possible that the different classes present in the dataset are not balanced. This can negatively impact the training of machine learning models as they can become biased towards the majority class [4]. Some previous studies of SDP for binary classification uses such imbalanced datasets [12]. This issue of imbalance can yield false conclusions about the predictive power the trained model [49].

2.3.3 Common Evaluation Metrics

When measuring the strength of a defect predictor, it is recommended to make use of Precision, Recall and their harmonic mean F-measure [11]. *Recall* is the ratio of how many of the positive samples present in the dataset that were correctly predicted. *Precision* instead gives a value of how often a prediction is correct when it classifies a sample as positive.

To calculate these measures a confusion matrix can be used. An example of such a matrix is presented in Figure 2.4. Precision, Recall and F-measure can then be calculated with Equations (2.1), (2.2) and (2.3), respectively. All of these measurements are in the range $[0, 1]$.

		Predicted value	
		True	False
Actual value	True	True Positive (TP)	False Negative (FN)
	False	False Positive (FP)	True Negative (FN)

Figure 2.4: *Confusion matrix*

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

$$\text{F-measure} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.3}$$

Another measure that can be used is Area Under the receiver operator characteristic Curve (AUROC) [3]. It can sometimes be referred to as *'AUC'* The AUROC metric is computed by first calculating the true positive rate against the false positive rate for some different thresholds. These values are then plotted against each other and the area under the resulting graph is the AUROC value. A model with a AUROC value greater then 0.5 is considered to be better then randomly guessing.

Matthews Correlation Coefficient (MCC) [31] is another measure that can be used specifically when classifying binary classes. The metric is useful in classification problems with imbalanced datasets. It is defined as shown in Equation (2.4), using definitions from the confusion matrix shown in Figure 2.4. Models evaluated with a MCC value around zero can be considered no better then simply randomly guessing. Positive values indicates that performance is better then random while negative values indicate they perform worse.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.4}$$

Chapter 3

Related Work

This chapter covers related work found in literature.

Kim et al. [23] generated very large corpora from different datasets of code changes, for the purpose of predicting if a change would introduce a bug. A bag of words approach was done used to create feature vectors from a corpus, thereby incorporating some semantic information. Shivaji et al. [43] [44] also used this approach. They however applied feature reduction methods to reduce the size of the input.

Wang et al. [47] used Abstract Syntax Tree nodes to create input vectors from source code files. This was then feed into a deep belief network to learn from these semantic features. Li et al. [29] achieved better results by using word embedding on the AST nodes and applying a shallow convolution network.

Dam et al. [6] also parsed source files into vectors of AST nodes and used word embedding to represent nodes as real valued vectors. They however used a Tree-LSTM [46] structure for their model as opposed to a feed forward network.

Huo et al. [19] aimed to use both bug reports and source files as input to link the two. This was done by applying convolution on both inputs separately and then combining features from the convolution in fully connected layers. Source code input was formatted as a single one hot vector. The approach was further improved by the addition of LSTMs to the network in [18].

Phan et al. [40] employed the use of assembly code instead of ASTs to predict software defects. Inputs tokens were converted to real value vectors before applying convolution to find patterns. The authors further used convolutional networks to analyze software control flow graphs [41]. These graphs were extracted from assembly code and feed into their network.

Anderson et al. [1] trained classifiers to predict test case failures. Syntactic features from the source code was not considered. Instead features were gathered from test history, code churn and static code analysis.

Rausch et al. [42] conducted analysis of build failures in CI builds in 14 open source project. Failures here were not limited to unit tests and considered other types of integration testing as well. The authors found that process metrics can be good indicators of failure. However the overall best metric was to consider historical build stability.

Herzig [15] used test execution metrics to try and predict pre- or post-release defects. Specifically test history was evaluated to see if several test failures in succession for a specific component could act as a good indicator of defect. This technique of considering bursts was based on the work of Nagappan et al. [36]. They used the rate of change in components as input to a prediction model to successfully predict defects in software.

Chapter 4

Method

In this chapter the methods employed in the thesis are presented. Section 4.1 covers how data was handled. This includes how it was collected, labeled, processed and finally split into training and test partitions. In Section4.2 an explanation of the machine learning model and it's implementation is given. Finally Section 4.3 gives an overview of hardware and software used in this research work.

4.1 Data Collection & Processing

4.1.1 Source Code Extraction

The process described in this subsection is shown as a flowchart in Figure 4.1.

Older revisions of the codebase were stored in a version control system. Data concerning build results for these different revisions was in turn stored in a database. These revisions represented a Pull Request (PR) made by a developer which contain modifications to one or more files. To evaluate if a PR could be integrated safely, several different builds are triggered.

In this thesis only unit test builds were considered when looking for failures, for two reasons. One reason was that unit test results should be closely related to the source code. The other reason was due to the

principal company's higher trust in these tests stability, i.e. they are considered less flaky than others.

To then gather initial datapoints, the database was queried to find successful and failed unit test builds for a Java project. Failed builds contain a text field in which would indicate if the failure caused by failing tests or other factors. As such the query was extended to take into consideration the contents of the mentioned field when querying for failed builds. The result returned by the query contained an identifier for which PR was associated with the build. This way lists of failed and successful PRs could be gathered. Developers have the ability to re-trigger builds, if they believe the result to be wrong due to flakiness. If any PR revision had been run several times and flipped from failed to successful, it would be counted as successful.

The collected commit identifiers were used to find the files which were changed by a PR in the version control system. A PR could contain changes to several different file types. For this project, only Java source files were downloaded from the version control system. Some of these source files could however be test cases. To indicate this the convention at the principal was to either append or prepend the word "*Test*" to the filename. By checking for the occurrence of this term test case source files could be filtered out.

4.1.2 Labeling Data

Since one PR could contain modifications to several files locating which of these actually failed a test was of great importance. Determining the culprit file or files is not trivial. However, the heuristic described in Section 4.1.1 above for locating test names could be used to again to link failed tests to modified files.

This approach is very conservative and may result in less data but should result in less noisy data as well. Gathering a list of failed test case names for this comparison was doable by querying another database used exclusively for storing test results. While gathering defective samples a list of filenames was kept in memory. The list was used in conjunction with the test-filename matching method to gather samples of normal (non-defective) files. This was done with the purpose of assuring that files labeled as normal were covered by unit tests,

to limit possible noise.

4.1.3 Syntactic Features

After file candidates had been selected they had to be converted to a suitable input format. One way to represent the syntax of source code files is with Abstract Syntax Trees (AST). This representation is helpful as it can assist in filtering out characters such as parenthesis, which are instead implicitly considered in the tree structure itself. One application of parsing source code to ASTs is in compilers, such as the Java compilers [7].

Using ASTs as input for SDP models has already seen success in related works [29] [47] [6]. This project based the method of extracting AST nodes on the work by Li et al. [29], which in turn took inspiration from [47]. As in the research of Li et al. an open source Python library called Javalang [1] was used to parse the source files. Only certain AST nodes were selected and are listed in Table 4.1. The selected nodes were then saved as vectors of strings.

To create input suited for the neural network model the ASTs were tokenized into vectors of integers. Again the method described in [29] and [47] was used. Node values were only included if they appeared at least 3 times in the dataset, to sort out values that might be specific to a single file. To have all of the input vectors be similar length smaller vectors were appended with zeros until they matched the longest vector in size.

4.1.4 Cross Validation

When training and evaluating the model K-Fold cross validation as described in Section 2.2.3 was applied. The number of folds were selected to be 10, as it has shown to be an optimal number of folds [26]. This evaluation method gives better insight into the average performance of the model. To reduce variance further this 10–Fold split was performed ten times in total, resulting in a total of 100 different splits. These ten sets were precomputed and saved before any training was

[1] https://github.com/c2nes/javalang

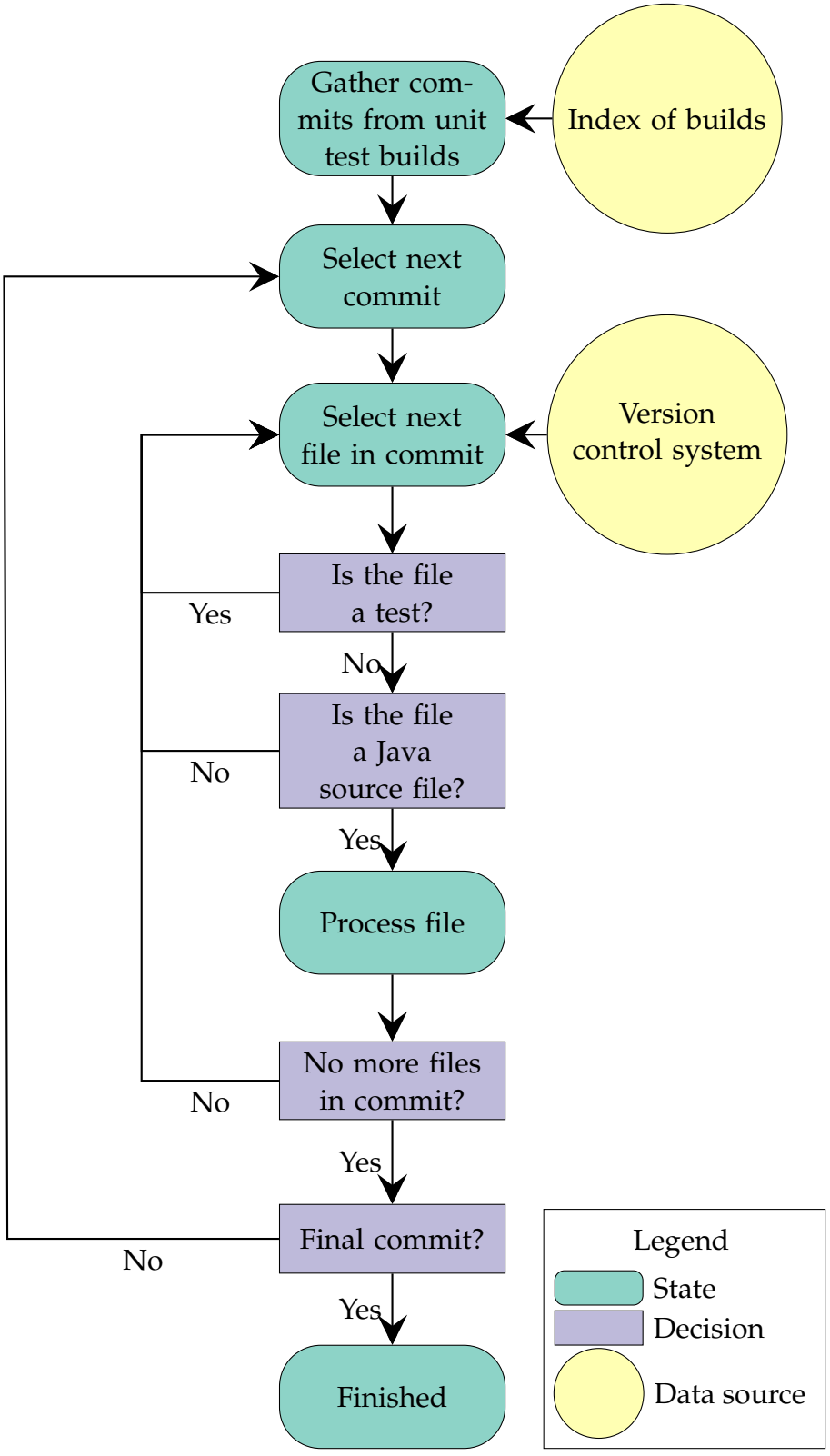

Figure 4.1: *Flowchart of the data collection process*

FormalParameter	BasicType
CatchClauseParameter	MethodInvocation
SuperMethodInvocation	MemberReference
SuperMemberReference	ReferenceType
TryResource	PackageDeclaration
InterfaceDeclaration	ClassDeclaration
ConstructorDeclaration	MethodDeclaration
VariableDeclarator	EnumDeclaration
IfStatement	WhileStatement
DoStatement	ForStatement
AssertStatement	BreakStatement
ContinueStatement	ReturnStatement
ThrowStatement	SynchronizedStatement
TryStatement	SwitchStatement
BlockStatement	CatchClause
SwitchStatementCase	ForControl
EnchancedForControl	

Table 4.1: *The AST nodes which were extracted using the Javalang Python package. Choice of nodes inspired by [47] and [29].*

done. Reusing the same sets between runs with different balance ratios was done to give a fairer comparison.

Two different paradigms were used when splitting the data into folds. The first was to split the data in a stratified manner, which preserve the ratio of defective and normal files. This means every fold has the same distribution of the two classes as the original dataset. With this approach, different revisions of files which was modified several times by the developers can appear in both training and test data. This is in line with the underlying distribution which is found when developing with CI.

The other way of creating folds was to apply a specific constraint wherein datapoints from the same filename were constricted to only appear in the same fold. With this constraint, different samples all drawn from different revisions of any file X would all appear only in one fold. This was done to test the models ability to find defects in previously unseen files. When similar datapoints can appear in both test and training set, a models performance might be overestimated for predicting the more different datapoints. This issue was highlighted in relation to SDP by Boetticher el al. [2].

4.2 The Model

4.2.1 Neural Network Structure

The machine learning model used in the thesis was the model Li et al. [29] used to learn syntactic source code features. Li et al. also experimented with adding traditional SDP features, yielding an increase of approximately 2% in average F-measure performance. These additional features were not used in our work since the focus was on syntactic features only.

The motivation behind the choice of model was to allow for comparison with state-of-the-art SDP performance in a Java project, with regards to syntactic features only. There are some additional differences, related to hyperparameter choices and another regarding the construction of the output layer.

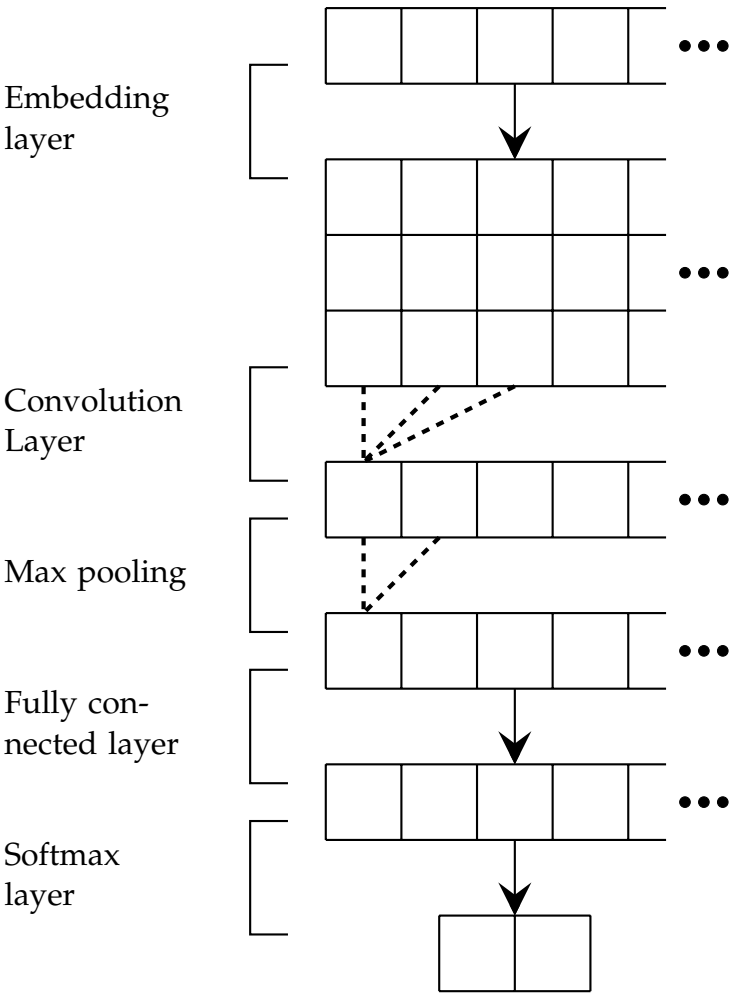

Figure 4.2: *Abstract structure of the neural network used in this thesis. To save on space certain elements in the figure are not accurate to the acutally trained model. Specificaly the deminsion of the word embedding is lower and only one filter in both convolution and max pooling layers are shown.*

Hyperparameter	This project	Li et al. [29]
Training epochs	15	15
Batch size	16	32
Word embed length	30	30
Number of Convolutional filters	16	10
Convolutional layer filter length	3	5
Units in fully connected layer	100	100

Table 4.2: *Choices of hyperparameters. Bold text indicates a difference between out work and the work of Li et al. [29].*

The choice of hyperparameters is shown in Table 4.2, where the bold text indicates a difference in choice. Stride for the convolution layer was not explicitly stated in [29] but was set to one in this thesis. Choices were motivated by conducting test runs and manually tuning values. For the output layer a binary softmax layer is used instead of a logistic regression classifier. However using softmax for two classes means the predicted probabilities are equivalent to using logistic regression.

Figure 4.2 shows the network structure. The first layer of the network is a word embedding layer which turns a word token, an integer, into a real value vector. Just as in [29] these embeddings are trained together with the rest of the network. Note that the embedding step was performed on a CPU while the steps that follow were all executed on a GPU.

The embeddings were then feed into a one dimensional convolutional layer, which performs feature extraction with ReLU as an activation function. Following that is a one dimensional max pooling layer of size two. Afterwards, the features were feed into a fully connected layer with ReLU activation function and then finally to the softmax layer for classification. The network is trained with cross entropy loss using the Adam optimizer [25] with a learning rate of 0.001. This value is recommended by the authors of Adam and is also the default setting in the TensorFlow framework.

4.2.2 Evaluation

The model was trained and evaluated using 10–10–Fold Cross Validation as described in Section 4.1.4. Both of the methods of creating the folds were employed, stratified folds and grouping samples from the same filename. The evaluation metrics chosen were precision, recall, F-measure, AUROC and MCC, see Section 2.3.3. The raw accuracy of the models was also considered to observe it's behavior.

The choice of these metrics was motivated by their usage in related works to allow for easier comparisons. When applicable, a positive result related to defective files while negative results were non-defective files. The metrics were evaluated for all 100 models and from there their average values could be calculated. To showcase the spread in values boxplots were plotted for all metrics.

Furthermore, to answer RQ2 the minority class (defective samples) were randomly oversampled. This random oversampling was only applied on the training data. Targeted balance ratios for normal to defective samples were $1 : 0.25$, $1 : 0.50$, $1 : 0.75$ and $1 : 1$.

4.3 Experimental Setup

All experiments were run on Google Compute Engine [2] VM instances. Data collection and processing, the parts documented in Section 4.1, were run on a *n1-highmem-8* instance. It was equipped with a 2.5GHz Intel Xeon E5 v2 (Ivy Bridge) processor with 8 cores and 52GB of RAM.

For training and evaluating the model the same kind of CPU was used with an added Nvidia Tesla K80 GPU. The large memory instance was motivated to decrease memory load on the GPU from the word embedding step. A GPU was added to decrease training time. The choices of GPU model was limit and the K80 was chosen due to being the cheapest available alternative.

To implement the method the Python [3] libraries NumPy [38], SciPy [20], imbalanced-learn [28], scikit-learn [39], Pandas [32] and TensorFlow [30]

[2] https://cloud.google.com/compute/
[3] https://www.python.org/

(version *1.4.1*) were used extensively. To calculate the AUROC metric, TensorFlow's *tf.metric.auc* [4] function was used with default parameters.

[4] https://www.tensorflow.org/versions/r1.4/api_docs/python/tf/metrics/auc

Chapter 5

Results

This chapter presents the experimental results. Section 5.1 contains results from the data collection process. The actual performance of the trained models is then discussed in Section 5.2.

5.1 Resulting Data

Table 5.1 showcases how much data was collected. For confidentiality reasons, the time frame in which the values were collected can not be disclosed. The first row denotes how many pull requests were collected and the second how many files were extracted using the chosen method. Note that defective pull requests here refer to ones which failed builds running unit tests and indicates nothing about how well they integrated in other regards.

As can be seen in the table the amount of failed revisions is in the vast minority, with only about 4.8% of unit test builds failing. A similar pattern was observed by Memon et al. [33] at Google where few of

Data type	# Passed	# Defective	% Defective
Pull Request	17712	890	4.8
Files (total)	5025	443	8.1

Table 5.1: *Overview of the collected data*

their tests ever fail. Furthermore, as raised in Section 4.1.2 difficulties with identifying which file exactly was responsible for the failure lead to not every failed PR corresponding to a defective file. Another possibly contributing aspect is that some tests cases (not limited to unit tests) sometimes fail to be ingested to the queried index. All in all this resulted in an defective rate of 8.1% in the final dataset.

5.2 Experiment Results

This section includes a series of boxplots in Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8. They follow the following conventions:

- The median is the line inside of the boxes
- The horizontal edges of the boxes indicates the upper and lower quartiles
- Whiskers are the maximum and minimum values (excluding outliers)
- Outliers are visualized as circles

When the models were trained on the stratified K-fold data the average precision, recall and F-measure were poor, see Figure 5.1. The precision-recall trade-off is visible, with recall averages increasing from 0.245 up to 0.412 as the minority class gets oversampled. This comes at the expense of precision, which starts at an average of 0.363 with no oversampling and goes down to 0.262 at max oversampling. The same result can be seen plotted as boxplots in Figures 5.4 and 5.3.

F-measure is observed to, on average, be slightly higher at lower oversampling ratios. It reaches a maximum average of 0.315 when the ratio of normal and defective samples are $1 : 0.50$. A comparison with the average F-measure achieved by Li et al. [29] with syntactic features, 0.596, is shown in Figure 5.5. The lowest F-measure Li et al. reports to achieve is 0.311, on the *xerces* dataset [21].

Both AUROC and MCC metrics reported shows that the models perform better than random (Figure 5.6 and Figure 5.7). AUROC values are in general higher as more oversampling is applied, with median values going from 0.598 to 0.650. MCC, while having a higher median

of around 0.254 at the $1 : 0.50$ balance ratio, does have lower spread in values at the $1 : 1$ ratio.

Accuracy values dip from a median of 0.90 to 0.85 as oversampling is applied, see Figure 5.8. Larger spread is also observed as the balance ratio approaches $1 : 1$.

When trained and tested on splits grouped on filenames, performance is very poor across all metrics. Precision only averages 0.071 on unchanged training data, increasing to a maximum average of 0.137 when oversampling is applied. Both recall and F-measure sees increases, up to averages of 0.303 and 0.144 respectively at a $1 : 1$ balance ratio, see Figure 5.2. However precision, recall and accuracy values are very spread out as seen in Figures 5.3, 5.4 and 5.8. Results for AUROC and MCC, which are close to 0.5 and 0 respectively in Figures 5.6 and 5.7, indicates that the model performs very close to random guessing on average on grouped K-fold.

Tables presenting average and standard deviation values of the six metrics for both K-fold paradigms are available in Appendix A.

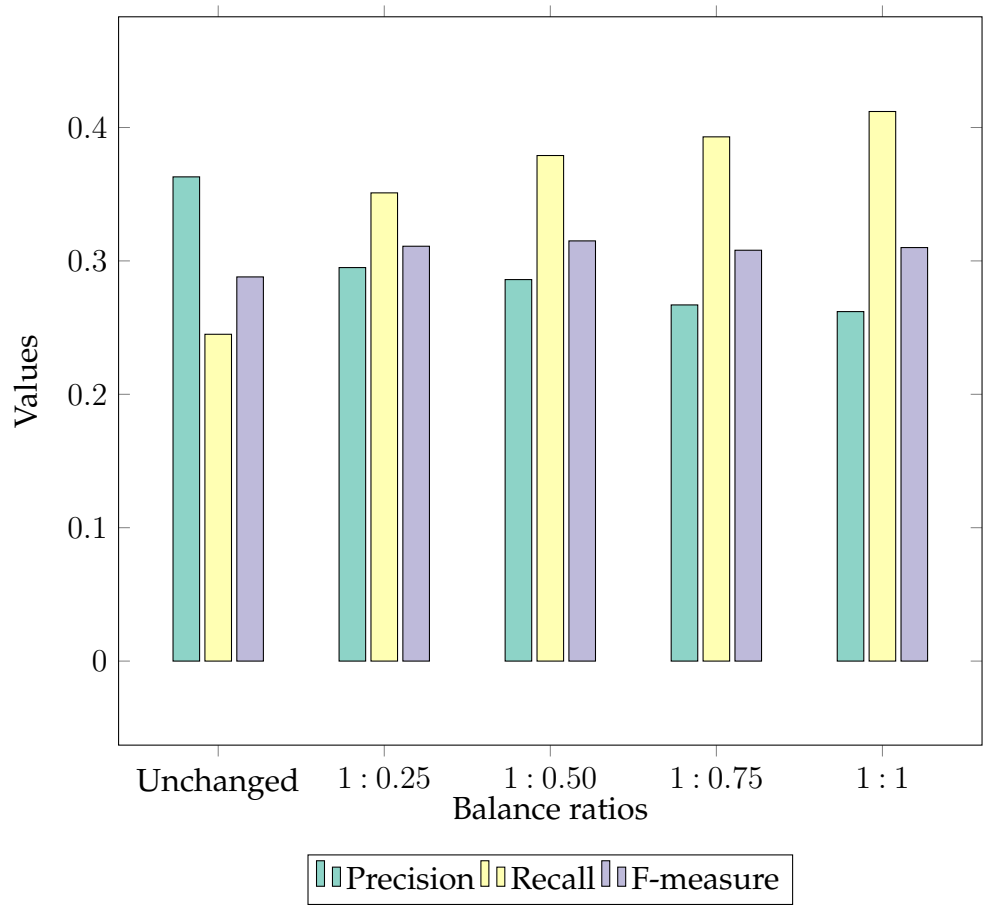

Figure 5.1: *Barplot of average precision, recall and F-measure values for stratified K-fold, higher values are better.*

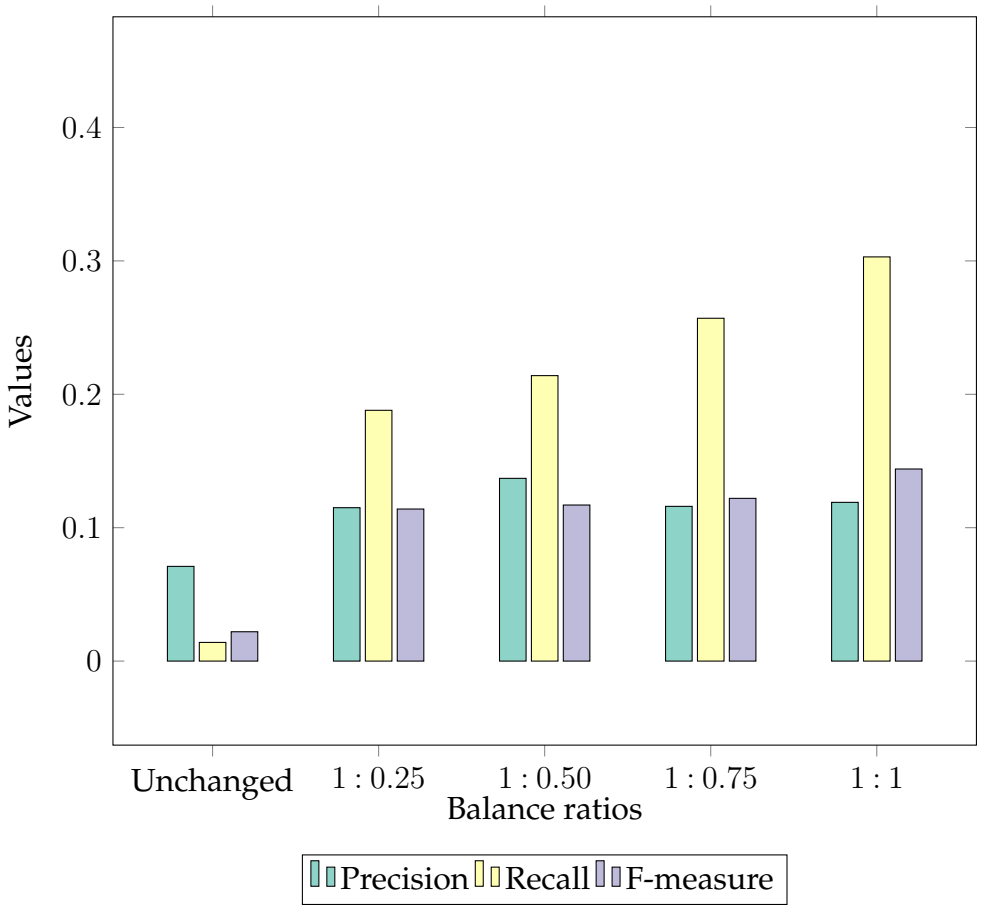

Figure 5.2: *Barplot of average precision, recall and F-measure values for grouped K-fold, higher values are better.*

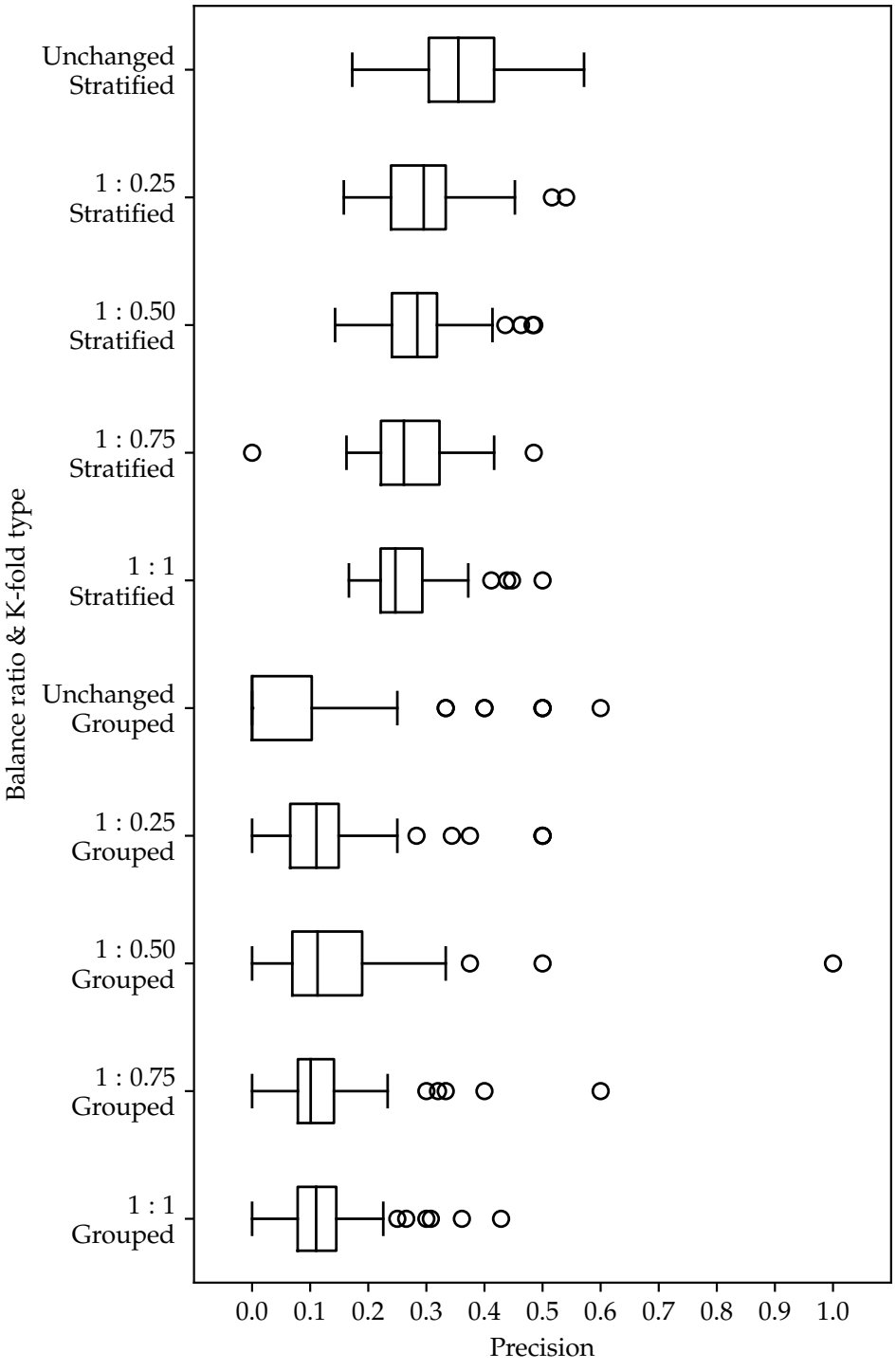

Figure 5.3: *Boxplot of Precision values, higher values are better*

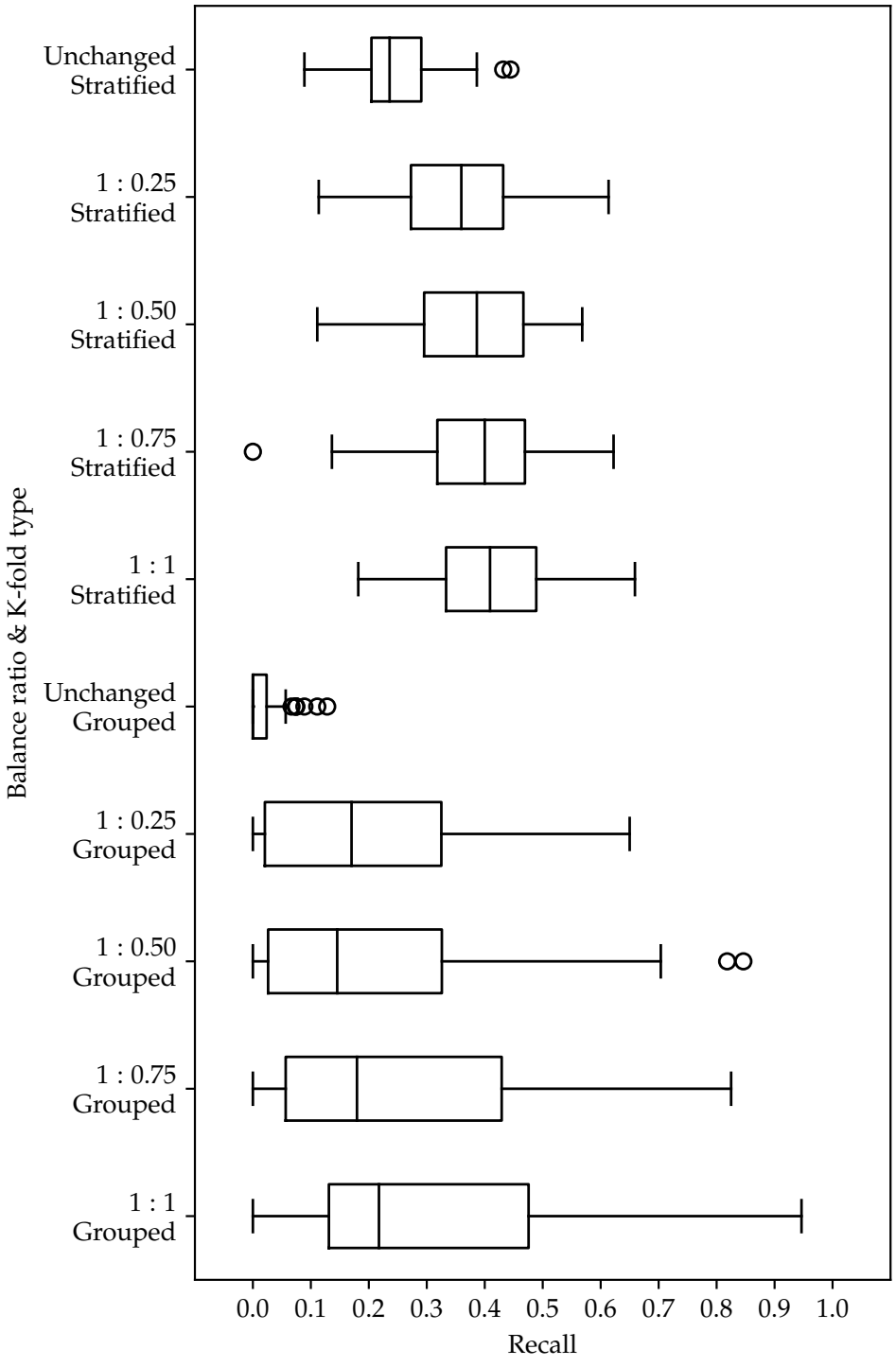

Figure 5.4: *Boxplot of Recall values, higher values are better*

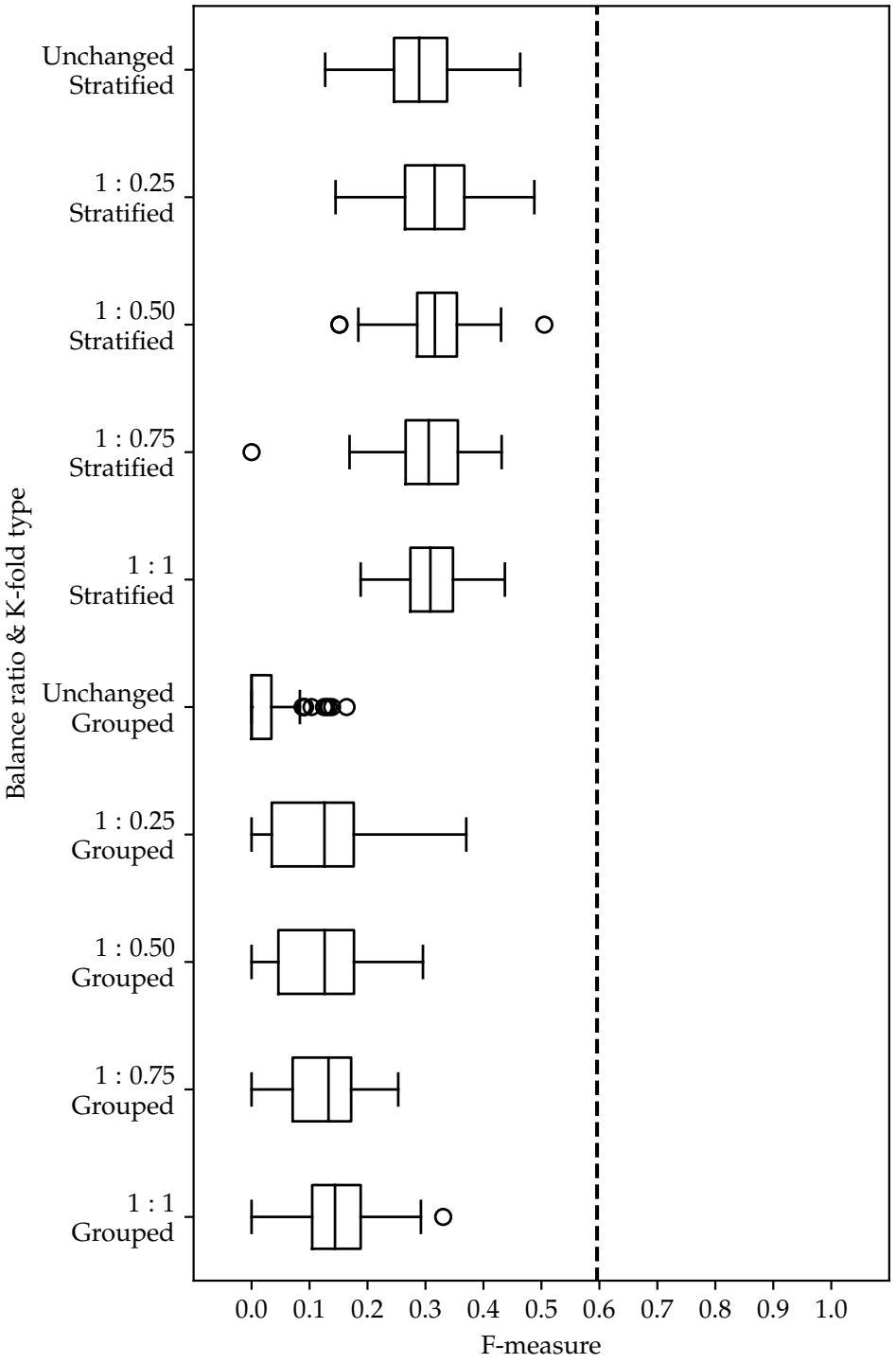

Figure 5.5: *Boxplot of F-measure values, higher is better. The dashed vertical line represents the average value over seven datasets Li et al. [29] achieved using semantic features.*

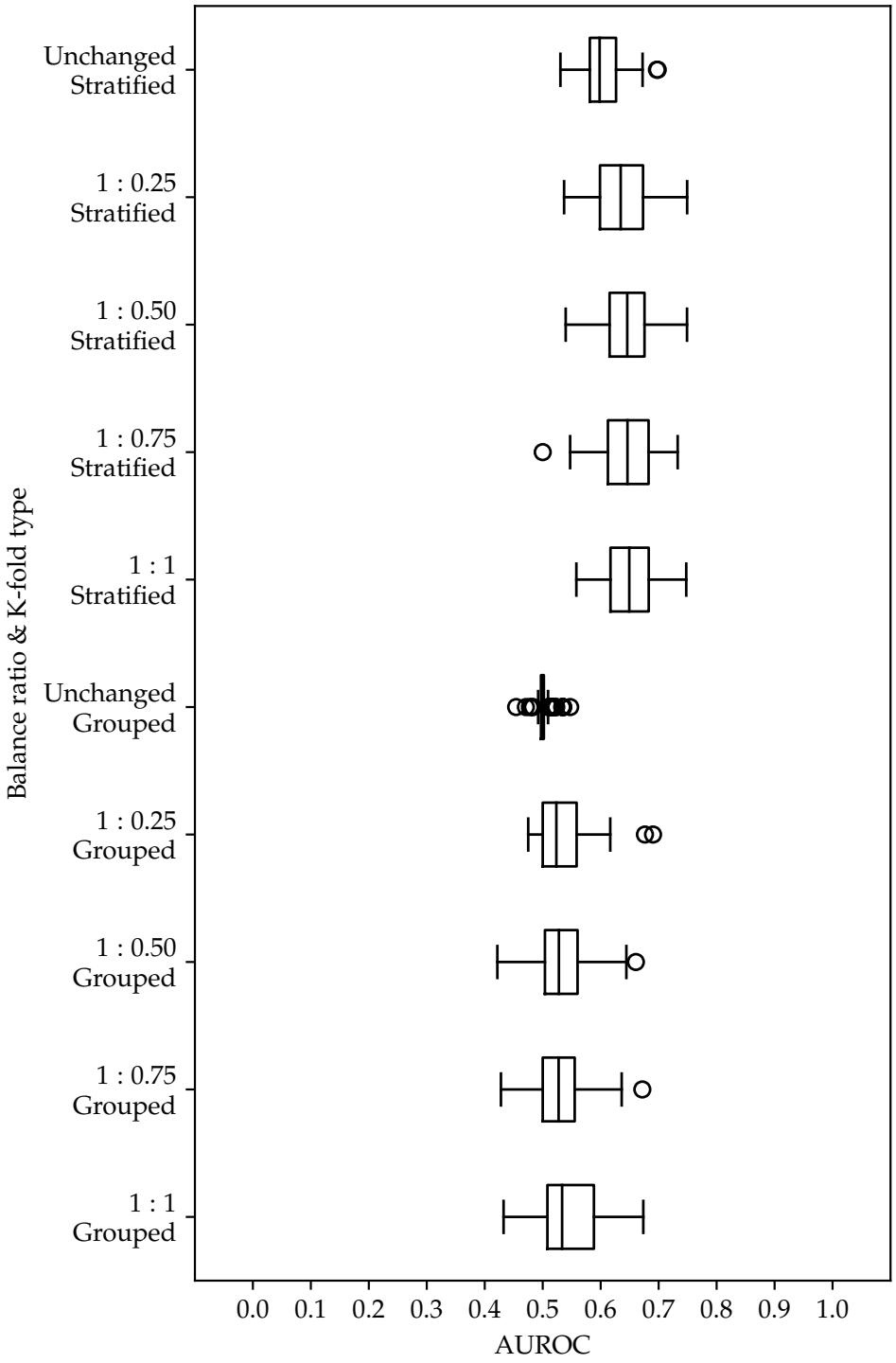

Figure 5.6: *Boxplot of AUROC values,* 0.5 *is considered no better then chance. Higher values are better.*

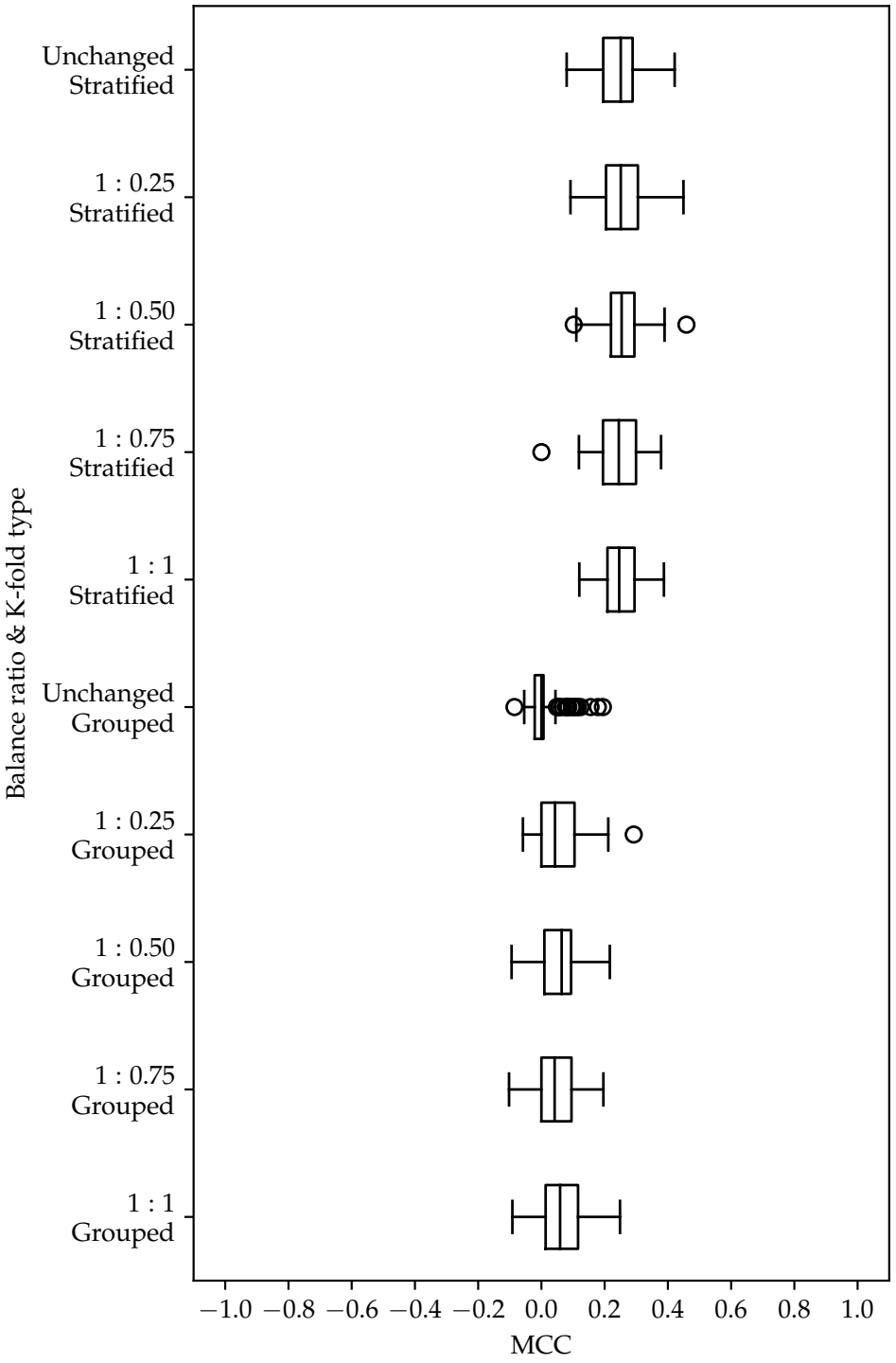

Figure 5.7: *Boxplot of MCC values,* 0.0 *is conisdered no better then chance. Higher values are better*

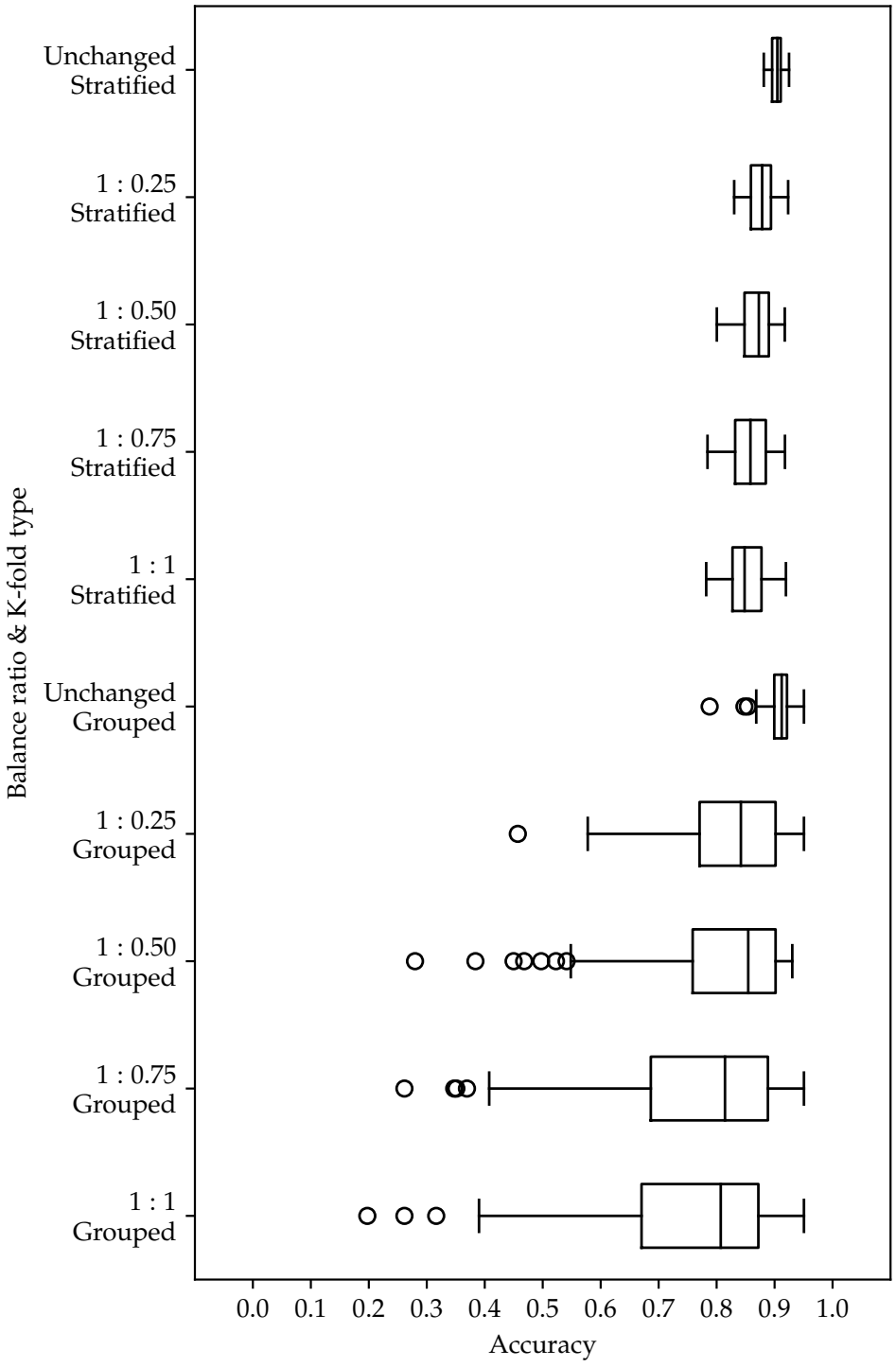

Figure 5.8: *Boxplot of Accuracy values, higher values are better*

Chapter 6

Discussion

In this chapter the results are discussed and conclusions about them are drawn. The internal and external validity of the thesis project is also touched upon, as well as future work.

6.1 Discussion Of Results

6.1.1 Stratified K-fold

For the results when using the stratified K-fold method the overall performance is in in general quite low. When comparing to the finding of Li et al. [29] the f-measure values are only comparable to their lowest result when using only semantic features: 0.311. Said result came from the *xerces* dataset which they reported as the having the worst balance between normal and defective files.

A trend that appears as oversampling is applied is that precision, Figure 5.3, decreases making way for higher recall, Figure 5.4. The trade off between recall and precision is normal, but such low precision indicates very high chance of false positives. Such behavior may not be desirable for developers. However as seen in Figure 5.1 the average F-measure does not change much after it has been balanced. Oversampling could as such be tweaked depending on what is valued higher, precision or recall.

6.1.2 Grouped K-fold

When evaluating the models on the grouped K-fold data with no class rebalance applied, we observe very poor performance across all metrics. After applying random oversampling of the defective files, slight improvements can be seen in precision, f-measure, AUROC and MCC. Recall get's a fairly large boost, however this appears to have more to do with the random oversampling biasing the classifier to predict defective behavior more often. This can be seen in Figure 5.8, where the models' overall accuracy is seen to decrease after applying rebalancing. In contrast, the accuracy of the stratified data does take a hit with oversampling applied but the spread in values is much smaller.

This points towards the conclusion that there is not enough information in the data to model defects in completely new files. Tests may be very file specific and with limited quantity of defective samples there may only be very specific patterns that can be learned.

6.2 Validity

6.2.1 Internal Validity

A source of possible uncertainty is the test cases themselves. Since the collection of data relies on the tests results (pass or fail), only defects which are tested for can be found. Some files marked as passing can still have what the model considers defective patterns in them, they are just not tested for them. Similar validity issues exists in other SDP datasets using bug reports, since they also relies on human decided labels which can be noisy [16].

The heuristic used when selecting what files failed or passed a unit test is another possible source of issues. However, in effort to combat this the most conservative method was chosen. This did result in a low number of samples. A more elaborate heuristic could perhaps have caught more files. Had more resources (time, personal etc.) been available, manual selection defective files could have been possible given the relatively low number of failed revision.

A correct implementation of the method in Li et al. [29] in this project is also of importance to validity. However, the authors describe their method in clear detail, which strengthens the confidence in the comparisons made in performance.

6.2.2 External Validity

The findings in this project is of course dependent on the specific application the dataset was gathered from. Collecting data from other projects may well yield different amounts of data and more defective samples.

6.3 Conclusions

Applying machine learning on semantic features to predict unit test failure appear to result in quite poor performance. When data is split in a stratified manner the model does perform 29.8% better then random, according to the highest median AUROC value. That is with no regard to if a different version of a file is present in test data, which could be seen as a realistic distribution in CI. However when forced to try and generalize to unseen files the model does not perform well at all. As there appears to be a lack of sufficient information in trained samples, increasing the amount of data may be a solution to this issue. This could be investigated in future research. Since this project also only focused on one dataset further validation on other projects is recommended.

6.4 Possible Extensions / Future Work

For this project it was decided to investigate if semantic features could be found on the file level to enable comparisons to related work. A finer level of granularity, such as commit additions or subtractions, could be worthwhile to explore. Further extensions could include also using the unit test file itself as additional input. Huo et al. [19] [18]

have done similar work using bug reports and source files as simultaneous input to locate buggy files.

Other possible extensions could aim to improve on the method of data collection. For this project a naive and conservative approach was applied. One advantage of this was that it could minimized the noise in the dataset. However, having a more advanced heuristic for locating the file responsible for triggering the failed test could yield more data without increasing noise.

Bibliography

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. "Striving for failure: an industrial case study about test failure prediction". In: *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press. 2015, pp. 49–58.

[2] Gary D Boetticher. "Improving credibility of machine learner models in software engineering". In: *Advanced Machine Learner Applications in Software Engineering (Series on Software Engineering and Knowledge Engineering)* (2006), pp. 52–72.

[3] Andrew P Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[4] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. "Special issue on learning from imbalanced data sets". In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 1–6.

[5] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[6] Hoa Khanh Dam et al. "A deep tree-based model for software defect prediction". In: *arXiv preprint arXiv:1802.00921* (2018).

[7] David Erni and Adrian Kuhn. "The Hacker's Guide to javac". In: *University of Bern, Bachelor's thesis, supplementary documentation* (2008).

[8] Martin Fowler and Matthew Foemmel. "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), p. 14.

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] David Gray et al. "Further thoughts on precision". In: *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*. IET. 2011, pp. 129–133.

[12] Tracy Hall et al. "A systematic literature review on fault prediction performance in software engineering". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304.

[13] Maurice H Halstead. "Elements of software science". In: (1977).

[14] Simon Haykin and Neural Network. "A comprehensive foundation". In: *Neural networks* 2.2004 (2004), p. 41.

[15] Kim Herzig. "Using pre-release test failures to build early post-release defect prediction models". In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE. 2014, pp. 300–311.

[16] Kim Herzig, Sascha Just, and Andreas Zeller. "It's not a bug, it's a feature: how misclassification impacts bug prediction". In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 392–401.

[17] Geoffrey E Hinton, James L McClelland, David E Rumelhart, et al. "Distributed representations". In: *Parallel distributed processing: Explorations in the microstructure of cognition* 1.3 (1986), pp. 77–109.

[18] Xuan Huo and Ming Li. "Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press. 2017, pp. 1909–1915.

[19] Xuan Huo, Ming Li, and Zhi-Hua Zhou. "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code." In: *IJCAI*. 2016, pp. 1606–1612.

[20] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 23-04-2018]. 2001– . URL: http://www.scipy.org/.

[21] Marian Jureczko and Lech Madeyski. "Towards identifying software project clusters with regard to defect prediction". In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM. 2010, p. 9.

[22] Sunghun Kim et al. "Automatic identification of bug-introducing changes". In: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE. 2006, pp. 81–90.

[23] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. "Classifying software changes: Clean or buggy?" In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196.

[24] Yoon Kim. "Convolutional neural networks for sentence classification". In: *arXiv preprint arXiv:1408.5882* (2014).

[25] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[26] Ron Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Ijcai*. Vol. 14. 2. Montreal, Canada. 1995, pp. 1137–1145.

[27] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. "Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (2017), pp. 55–79.

[28] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: http://jmlr.org/papers/v18/16-365.

[29] Jian Li et al. "Software defect prediction via convolutional neural network". In: *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE. 2017, pp. 318–328.

[30] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[31] Brian W Matthews. "Comparison of the predicted and observed secondary structure of T4 phage lysozyme". In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975), pp. 442–451.

[32] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[33] Atif Memon et al. "Taming google-scale continuous testing". In: *Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on*. IEEE. 2017, pp. 233–242.

[34] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. "Linguistic regularities in continuous space word representations". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2013, pp. 746–751.

[35] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[36] Nachiappan Nagappan et al. "Change bursts as defect predictors". In: *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE. 2010, pp. 309–318.

[37] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[38] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[39] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[40] Anh Viet Phan and Minh Le Nguyen. "Convolutional neural networks on assembly code for predicting software defects". In: *Intelligent and Evolutionary Systems (IES), 2017 21st Asia Pacific Symposium on*. IEEE. 2017, pp. 37–42.

[41] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. "Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction". In: *arXiv preprint arXiv:1802.04986* (2018).

[42] Thomas Rausch et al. “An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software”. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press. 2017, pp. 345–355.

[43] Shivkumar Shivaji et al. “Reducing features to improve bug prediction”. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 600–604.

[44] Shivkumar Shivaji et al. “Reducing features to improve code change-based bug prediction”. In: *IEEE Transactions on Software Engineering* 39.4 (2013), pp. 552–569.

[45] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When do changes induce fixes?” In: *ACM sigsoft software engineering notes*. Vol. 30. 4. ACM. 2005, pp. 1–5.

[46] Kai Sheng Tai, Richard Socher, and Christopher D Manning. “Improved semantic representations from tree-structured long short-term memory networks”. In: *arXiv preprint arXiv:1503.00075* (2015).

[47] Song Wang, Taiyue Liu, and Lin Tan. “Automatically learning semantic features for defect prediction”. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 297–308.

[48] Chadd Williams and Jaime Spacco. “Szz revisited: verifying when changes induce fixes”. In: *Proceedings of the 2008 workshop on Defects in large software systems*. ACM. 2008, pp. 32–36.

[49] Hongyu Zhang and Xiuzhen Zhang. “Comments on" data mining static code attributes to learn defect predictors"”. In: *IEEE Transactions on Software Engineering* 33.9 (2007).

Appendix A

Tables

This appendix includes tables of averages and standard deviations of the six metrics which were used for evaluation of the machine learning models. Every table includes values for both methods of splitting the data into folds and all tested balance ratios.

Precision values are presented in Table A.1, recall in Table A.2, F-measure in Table A.3, AUROC in Table A.4, MCC in Table A.5 and finally accuracy in Table A.6.

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.363	0.078
Stratified	1 : 0.25	0.295	0.072
Stratified	1 : 0.50	0.286	0.063
Stratified	1 : 0.75	0.267	0.068
Stratified	1 : 1	0.262	0.064
Grouped	Unchanged	0.071	0.133
Grouped	1 : 0.25	0.115	0.101
Grouped	1 : 0.50	0.137	0.128
Grouped	1 : 0.75	0.116	0.086
Grouped	1 : 1	0.119	0.075

Table A.1: *Precision values for the different sets of data*

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.245	0.072
Stratified	1 : 0.25	0.351	0.104
Stratified	1 : 0.50	0.379	0.109
Stratified	1 : 0.75	0.393	0.115
Stratified	1 : 1	0.412	0.111
Grouped	Unchanged	0.014	0.026
Grouped	1 : 0.25	0.188	0.174
Grouped	1 : 0.50	0.214	0.215
Grouped	1 : 0.75	0.257	0.232
Grouped	1 : 1	0.303	0.250

Table A.2: *Recall values for the different sets of data*

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.288	0.067
Stratified	1 : 0.25	0.311	0.066
Stratified	1 : 0.50	0.315	0.060
Stratified	1 : 0.75	0.308	0.064
Stratified	1 : 1	0.310	0.054
Grouped	Unchanged	0.022	0.038
Grouped	1 : 0.25	0.114	0.084
Grouped	1 : 0.50	0.117	0.078
Grouped	1 : 0.75	0.122	0.069
Grouped	1 : 1	0.144	0.077

Table A.3: *F-measure values for the different sets of data*

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.603	0.034
Stratified	1 : 0.25	0.637	0.045
Stratified	1 : 0.50	0.645	0.044
Stratified	1 : 0.75	0.646	0.046
Stratified	1 : 1	0.651	0.044
Grouped	Unchanged	0.502	0.012
Grouped	1 : 0.25	0.535	0.042
Grouped	1 : 0.50	0.535	0.041
Grouped	1 : 0.75	0.533	0.042
Grouped	1 : 1	0.548	0.053

Table A.4: *AUROC values for the different sets of data*

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.246	0.069
Stratified	1 : 0.25	0.251	0.071
Stratified	1 : 0.50	0.255	0.063
Stratified	1 : 0.75	0.246	0.067
Stratified	1 : 1	0.247	0.062
Grouped	Unchanged	0.009	0.048
Grouped	1 : 0.25	0.056	0.064
Grouped	1 : 0.50	0.060	0.061
Grouped	1 : 0.75	0.050	0.058
Grouped	1 : 1	0.069	0.072

Table A.5: *MCC values for the different sets of data*

Data	Balance ratio (normal:defective)	Average	std.
Stratified	Unchanged	0.903	0.010
Stratified	1 : 0.25	0.876	0.023
Stratified	1 : 0.50	0.868	0.028
Stratified	1 : 0.75	0.858	0.032
Stratified	1 : 1	0.851	0.031
Grouped	Unchanged	0.910	0.023
Grouped	1 : 0.25	0.826	0.094
Grouped	1 : 0.50	0.804	0.138
Grouped	1 : 0.75	0.765	0.160
Grouped	1 : 1	0.753	0.161

Table A.6: *Accuracy values for the different sets of data*