b
UNIVERSITÄT
BERN

Bug Prediction with Neural Nets

Using regression- and classification-based approaches

Bachelor Thesis

Sébastien O. Broggi
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

26. Januar 2018

Prof. Dr. Oscar Nierstrasz
Haidar Osman

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Bugs can often be hard to identify and developers spend a large amount of time locating and fixing them. Bug prediction strives to identify code defects using machine learning and statistical analysis and therefore decreasing time spent on bug localization. With bug prediction, awareness of bugs can be increased and software quality can be improved in significantly less time. Machine learning models are used in most modern bug prediction tools and with recent advances in machine learning, new models and possibilities have arisen that further improve the possibilities and performance in bug prediction.

In our studies, we test the performance of "Doc2Vec" – a current model that is used to vectorize plain text – on source code to perform classification. Instead of relying on code metrics, we analyze and vectorize plain-text source code and try to identify bugs based on similarity to learned paragraph vectors.

Testing two different implementations of the Doc2Vec model, we find that no usable results can be achieved by using plain text classification models for bug prediction. Even after abstracting the code and applying parameter tuning on our model, all experiments deliver a constant 50% accuracy, so no learning can be achieved by any of the models. The experiments clearly show that code should not be treated as plain text and should instead contain more code-specific information like metrics about the code.

Our second setup of experiments consists of a 3-layer feed forward neural network that performs classification- and regression- based approaches on code metrics, using datasets that contain a discrete number of bugs as a response variable. There have already been many successful experiments using metrics to perform classification based on code metrics. In our studies we compare the performance of a standard regression and standard multi-class classification model to the models "classification by regression" (CbR) and "regression by classification" (RbC).

In the RbC model, we use the output from classification to predict an accurate number of bugs and then calculate the root-mean-square error (RMSE). In the CbR model we use the output of regression to perform binary classification and calculate area under the receiver operating characteristic curve (ROC AUC) to compare the results.

In our experiments we find that a neural network delivers better results when using the CbR model on an estimated defect count compared to the results using standard multi-class classification. We also suggest that the RMSE can significantly be decreased by using the RbC model compared to standard regression.

Contents

1 Introduction **1**

2 Empirical Study **4**
2.1 Text-Based Bug Prediction 4
2.1.1 Machine Learning Models 4
2.1.1.1 Doc2Vec / Word2Vec 4
2.1.1.2 Doc2Vec Implementations 5
2.1.2 Datasets 6
2.1.3 Experiments 6
2.1.4 Conclusions 7
2.2 Metrics-Based Bug Prediction 9
2.2.1 Machine Learning Models 9
2.2.1.1 Network layout 9
2.2.1.2 Regression 10
2.2.1.3 Classification 10
2.2.1.4 Regression by Classification 11
2.2.1.5 Classification by Regression 11
2.2.2 Datasets 11
2.2.3 Experiments 13
2.2.4 Results 14
2.2.5 Conclusions 20

3 Related Work **21**
3.1 Text-Based Bug Prediction 21
3.2 Metrics-Based Bug Prediction 22

4 Conclusions and Future Work **23**
4.1 Doc2Vec 23
4.2 Feedforward neural net 23
4.3 Future Work 24

5 Anleitung zu wissenschaftlichen Arbeiten **27**
5.1 Prerequisites 27
5.2 Doc2Vec 28
5.2.1 Setup 28
5.2.2 Training and Inference 29
5.3 Feedforward Neural Network 30
5.3.1 Setup 30
5.3.2 Training and Inference 32

1 Introduction

Machine learning algorithms can be used for a wide array of applications, such as classifying images and videos and providing intelligent assistants, and they are increasingly starting to outperform humans in new tasks. Recent advances in machine learning – especially in the topic of neural nets / deep learning – are leading to a variety of new models and libraries, offering new approaches for a lot of different problems.

Bug prediction is one application of machine learning that aims to identify critical parts in source code likely to contain defects. This practice can be used in software projects to gain insights into how and where bugs occur to improve software quality in significantly less time. Using bug prediction, development time and costs of a software project can be decreased, while augmenting the code quality. The construction and analysis of machine learning algorithms for bug prediction tools has already been well researched [12].

One of the applications of machine learning that has gained more attention recently, has been the vectorization of text and its use in linguistic analysis. With models like Word2Vec [10] and Doc2Vec [8] it is possible, to represent single words or whole paragraphs as vectors and as such find similarities and relations to each other. Knowing that bugs often occur in similar contexts, a text-based model might be able to predict bugs in a vectorized form of content.

Motivated by this hypothesis, we decided to explore the model "Doc2Vec" and test whether it can classify vectorized plain text source code into "bugs" and "fixes". In the experiments we use two different implementations of the Doc2Vec model: The distributed memory implementation of paragraph vectors (PV-DM) and the distributed bag of words (PV-DBOW). While PV-DM uses a continuous input of words and trains

word vector representations as well as paragraph vectors, the PV-DBOW implementation uses a random sampling approach of words and is only training paragraph vectors in the process. We train our two models on a dataset containing bug fix commits before and after a fix was applied, labeling the text before the commit as "bug" and after the commit as "fix". Using the trained paragraph vectors model, we can compare the similarity of previously unseen code chunks to a representation of buggy or fixed chunks to perform classification.

Using different parameter configurations for both implementations, the model keeps performing at a constant 50% accuracy with the used dataset. Trying to improve the results, we introduce anonymization as a measure to reduce complexity of the data. By anonymizing context specific names and declarations, we hope to achieve a more keyword focused training of the model, as some variable names or type declarations are used only very seldom in the dataset and might negatively interfere with the training process. As before, we test both implementations with the full dataset.

Since the experiments with anonymization do not yield any better results than without it, we additionally introduce scoping. By only selecting specific bug fix patterns for training and testing, we hope to achieve a more bug specific classification with Doc2Vec. We test scoping with and without anonymization on both implementations, but the results indicate a performance of a random guesser for all experiments, as they did before.

The Doc2Vec model has been shown to perform well in general classification of an article and semantic analysis, but our experiments conclude it is not able to accomplish bug prediction with either implementation of the model. Even after introducing anonymization and scoping, we are not able to confidently classify code chunks into "bugs" and "fixes". While a more complex model together with some more specific data might be able to achieve plain text classification of bugs, we learn that code should not be treated as simple text.

Using plain text source code may not provide enough information for a neural network to achieve successful bug prediction, but previous experiments with machine learning have already shown success in bug prediction using metrics multiple times[12] [20] [1] [15] [18] [6]. While using machine learning algorithms on code metrics provides decent results, in most cases only simple binary classification is performed or discrete defect counts are predicted. Knowing this, these experiments do not make use of the full data range available when dealing with discrete defect counts in the dataset. More efficient approaches could be found by using the information generated by regression to perform binary classification. Considering that multi-class classification also performs well in many situations, using the output of classification as a regressor might also be able to reduce the root-mean-square error (RMSE) compared to standard regression, when often classifying correctly. Using this approach, the RMSE can also be further artificially improved, by reducing the number of possible classes and thus reducing the potential error.

In our studies we examine if using a regression based algorithm can improve binary classification and if a multi-class classifier can be used as viable regressor in bug prediction. This is accomplished by constructing a 3-layer neural network, taking independent source code metrics as input and delivering an estimated number of bugs, either as continuous output value in regression or as a class – an integer defect count – in classification. To achieve classification by regression (CbR), we use the output of regression to perform binary classification based on a selected threshold with respect to the RMSE. For regression by classification (RbC) we calculate the RMSE using the predictions of our classification model. We use the software metrics of 9 projects from Tera-PROMISE [19] and 5 projects from a study by D'Ambros *et al.* [9] to train our model.

The results show that our models RbC and CbR are able to outperform standard regression and classification if used right. The RbC model manages to deliver a smaller RMSE than standard regression in most cases and the RMSE of the model can even further be decreased by reducing the available classes. The CbR model also generally performs at a higher AUC than classification, showing an overall improvement in the ability to do binary classification. Based on our results, we conclude that using a regression based classifier makes sense in bug prediction when dealing with discrete defect counts. The same can also be applied to other domains when trying to perform binary classification on data with discrete response variables. Furthermore, we suggest to use a RbC model rather than standard regression to reduce the RMSE. Although RbC delivers better results than regression, this model might only be useful in some cases, as the predictions will only consist of integer numbers rather than a precise approximation of the defect count.

The rest of the thesis is organized as follows:

Chapter 2 shows our empirical study using Doc2Vector for text-based bug prediction and a feedforward neural network for metrics-based bug prediction.

Chapter 3 lists related work to our studies.

Chapter 4 concludes our thesis and shows potential improvements for future studies.

Appendix contains a tutorial on how to implement and apply the models used in this thesis.

2 Empirical Study

2.1 Text-Based Bug Prediction

In these experiments we try to achieve bug prediction on plain text source code. Using the Doc2Vec model we vectorize previously unseen paragraphs of code and compare their similarity to either defected or bug-free paragraphs to perform binary classification.

2.1.1 Machine Learning Models

2.1.1.1 Doc2Vec / Word2Vec

For our experiments in plain text classification, we are using the model "Doc2Vec" [8] (paragraph vectors). Doc2Vec is a derived version of the Word2Vec model by Mikolov *et al.* [10]. The Word2Vec model is a two-layer neural net that processes text and generates a vocabulary of words, each of them mapped to a vector in a matrix W. The Word2Vec model is then trained to predict the next word based on previous words or context. The model constantly adapts the word vectors in W to better match the predictions. In this process, words often used in a similar context end up having similar vectors.

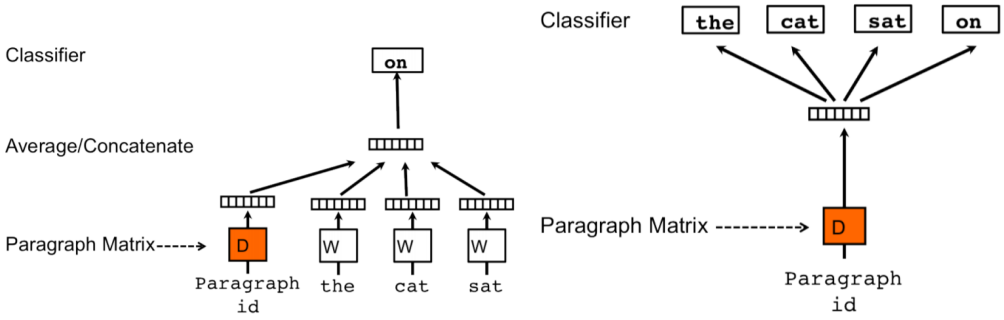
The trained vectors in the model thus represent a learned semantic of the words, based on the context they are used in. Similarity of words can then be calculated by measuring the cosine similarity of their vectors. A value of 1 implies a 0° degree angle and thus the words are used identical. A value of 0 indicates a 90° angle between the vectors, so no similarity between the words at all.

The Doc2Vec model essentially uses the same approach, but instead of just single words, the vectors are calculated for whole paragraphs. The model can then be used to classify texts into classes or perform other tasks like sentiment analysis or measuring similarity of articles.

2.1.1.2 Doc2Vec Implementations

Doc2Vec supports two different types of implementations: In the "Distributed Memory" implementation of Paragraph Vectors (PV-DM), each paragraph is represented by a column in a matrix D and every word is mapped to a column in a matrix W. The model then works like Word2Vec and trains the word vectors in W and tries to predict the next word. But in addition, it uses the paragraph vector from D as additional input and trains it as well as can be seen in Figure 2.1(a).

The other implementation of Doc2Vec is the "Distributed Bag of Words" version of Paragraph Vector (PV-DBOW). In this model, the paragraph vectors are trained to predict words randomly sampled from a paragraph. The input consists of a single input vector from D, which makes the model conceptually simple and uses less storage, but in general is less performant. This implementation can be seen in seen in Figure 2.1(b)



(a) PV-DM Implementation

(b) PV-DBOW Implementation

Figure 2.1: The two implementations of Doc2Vec. PV-DM uses a continuous input of words and trains word vector representations as well as paragraph vectors. The PV-DBOW implementation uses a random sampling approach and is only training paragraph vectors in the process.

2.1.2 Datasets

Origin

In the first experiment, where we aim at classifying plain text code as buggy or clean, we use the dataset from the study by Osman *et al.* on mining bug-fix code changes [14]. In this study, the authors collected two corpora: the Java popular corpus and the Eclipse ecosystem. Osman *et al.* identify bug-fix commits and extract the buggy code hunks and the fixed code. We use this dataset as a labeled dataset to train the Doc2Vec model.

Structure

The dataset contains 91'644 commits with the code before and after a bug was fixed, which we label accordingly and use to train and test the model. Furthermore, the dataset contains the applied bug fix pattern as an anonymized transition. The transition describes the changed code by replacing words in the modified snippet by the letter "T", switching all numbers to the value 0 and merging multiple white spaces into a single space. This is very useful to filter the dataset for specific bug fix patterns (*e.g.,* adding missing null checks).

2.1.3 Experiments

In the experiments using Doc2Vec we use three different approaches: First we try to perform classification on the raw data as is. The second experiment introduces anonymization to the data. In the third approach, we filter the dataset to only include missing null check patterns.

The experiments are run five times with the PV-DBOW implementation, and another 5 times using the PV-DM model. Each run uses a different 80% of the data as training set and 20% as the testing set. The models are then trained to perform classification based on the previously chosen labels.

Raw Data

As described previously, the Doc2Vec model creates close vectors for similar paragraphs, based on how often words are used and in which context they occur. Assuming bugs are more likely to occur in certain contexts (*e.g.,* missing null check), the model might be able to perform bug prediction based on learned paragraph vectors. Using this assumption as motivation, the paragraph vectors are trained to perform classification on plain text source code from the dataset. We label the pre-commit paragraph (method body) as "bug" and the post-commit one as "fix".

As can be seen in Table 2.1 and Table 2.2, we get a constant 50% accuracy for both implementations, so no knowledge about bugs is achieved by Doc2Vec when using the raw data. To decrease the complexity of the problem, we start to abstract and filter the data.

Anonymization

In source code, certain words (*e.g.,* variables and type definitions) only occur very seldom, if not only once in a project. Due to this, we introduce anonymization of the code to decrease the number of different words to get a more general representation of code and focus more on keyword usage. Using a Java parser library [5], the code is parsed and some of the text was replaced with placeholders. All variables are replaced by the letter "v", type definitions are changed to "T" and methods to "m" and strings to a generic placeholder "s". Some examples:

A variable initialization: "T v = new T (v) ;"

A null check: "if (v != null) { v . m (v , v) ; }"

Using the anonymized data, the experiments are run again on both models. But even with abstracted data, the Doc2Vec model does not deliver any better results with either implementation, PV-DBOW or PV-DM. When using anonymization, the model also seems to constantly classify everything as "fix", leading to a 0% recall and 0% precision for bugs or a 100% recall and 50% precision for fixes.

Scoping

To further decrease complexity and variance of the data, we choose to reduce the number of different bug fix patterns. Instead of using all examples available in the dataset, we only select the examples adding a null check, using the transition data mentioned previously. The experiments are run again with and without abstraction of the data, testing both implementations. Even after introducing scoping to the dataset, classification of bugs with Doc2Vec performs at a constant 50% accuracy and no successful bug prediction is achieved with this model.

2.1.4 Conclusions

At a constant 50% accuracy, we can safely state that a standard Doc2Vec model is not able to perform bug predication on plain text source code. Neither of the implementations is able to gain knowledge about bug fix patterns, even after abstracting and filtering the data. While the two Doc2Vec implementations are working fine for classification on general texts about a topic or to compare similarity of paragraphs, the vector representation of paragraphs fails to represent bugs or fixes accordingly and should not be used for bug prediction with plain text.

Experiments with PV-DM	Accuracy	Precision	Recall
Raw data	0.5	0.24	0.34
Anonymization	0.5	0	0
Scoping with anonymization	0.5	0	0
Scoping without anonymization	0.5	0.25	0.38

Table 2.1: The results of the different experiments using the PV-DM implementation of Doc2Vec for bug prediction.

Experiments with PV-DBOW	Accuracy	Precision	Recall
Raw data	0.5	0.25	0.12
Anonymization	0.5	0	0
Scoping with anonymization	0.5	0	0
Scoping without anonymization	0.5	0.25	0.53

Table 2.2: The results of the different experiments using the PV-DBOW implementation of Doc2Vec for bug prediction.

2.2 Metrics-Based Bug Prediction

In these experiments we try to achieve bug prediction on code metrics. We use a feedforward neural network to study a classification-based regressor and a regression-based classifier for bug prediction and compare their performance to simple classification and regression.

2.2.1 Machine Learning Models

2.2.1.1 Network layout

In a our approach using metrics to perform bug prediction with a neural net, we use a 3-layer feedforward network. The neural net uses any number of features about source code as input and delivers an estimate of the contained bugs, either as a labeled class (in classification) or as floating point value (in regression). An illustration of the model is shown in Figure 2.2.

The network's first layer is a simple input layer, taking the number of features as input and forwarding it to the next layer. The second layer is a hidden layer. Its size is defined by the average between number of features and number of classes. The last layer – the output layer – is a softmax layer, which distributes the total probability of 1 over all possible outputs. In practice, the softmax function is used in tandem with the negative log-likelihood loss function, defined as:

L(y) = -log(y)

To optimize predictions, a neural net usually tries to minimize the output of a loss function. Most other loss functions use the error of the predictions as input and assign a high input value to a high cost and low input value to a low cost. This forces the network to adapt more drastically when the error is big and change less when the error is small. The negative log-likelihood function in contrast results in a bigger cost with smaller inputs. This is used to an advantage in softmax, by using the predicted probability of the correct class as input of the function. If the confidence of the correct answer is low, the loss function returns high values and the network changes more drastically. If the confidence of the correct answer is high, the network only adapts slightly.

The model described is used to perform four different types of experiments, by calculating the expected value of bugs based on the probability of each possible output.

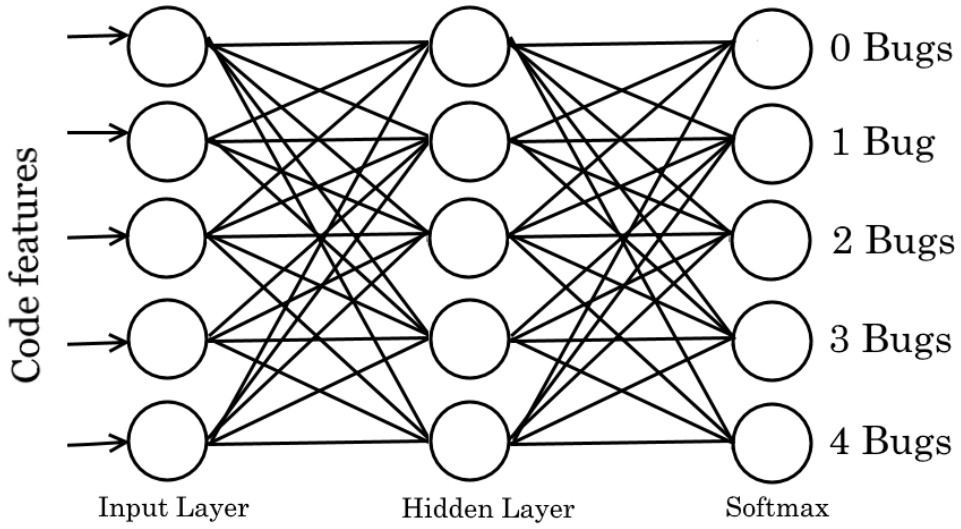


Figure 2.2: Illustration of a feedforward neural network used in the experiments. The actual number of features and maximum number of classes is determined by the dataset used.

2.2.1.2 Regression

Using this model, regression is achieved by calculating the mathematical expected number of bugs using the predicted probability of the softmax layer. The value is generally defined as the sum of all possible number of bugs multiplied by their probability of occurring:

Predicted Bugs = $\sum_{X=1}^{MaxBugs} P(X) * X$

X=Number of bugs and P(X)=Predicted confidence of X bugs

2.2.1.3 Classification

Standard classification is similarly achieved with the same model by selecting the most probable output class, whereas the class label describes a discrete number of bugs. If the estimated number of bugs is 1 or bigger, we classify the input as a "bug".

Predicted Bugs =$\arg\max_X P(X)$

X=Number of bugs and P(X)=Predicted confidence of X bugs

2.2.1.4 Regression by Classification

Regression by classification (RbC) is created as a construct on top of standard classification. By using the predicted number of bugs in classification, we calculate the RMSE to test its performance as a regressor.

Since there are few examples in the datasets with a very high defect count, they are very unlikely to be classified correctly. Due to this we also choose to cap the number of possible bugs in RbC. The "RbC10" model for example limits the output to a maximum of 10 classes (0-9 bugs). So if more than 9 bugs are contained in an example of the training set, we set it down to 9. Using this approach, we set up the different models "RbC2", "RbC5","RbC10" and "RbC100". As there are no examples in the dataset with more than 100 bugs, the RbC100 always uses the full range of possible classes.

2.2.1.5 Classification by Regression

The classification by regression (CbR) model is constructed by using the estimated number of bugs from standard regression to perform classification. Binary classification is achieved by simply applying a specific threshold. If the estimated number of bugs is smaller than the threshold, we classify the input as "fixed". If the value is bigger than the threshold, it is classified as "buggy". In our experiments the threshold to perform binary classification is selected based on the RMSE of standard regression.

2.2.2 Datasets

Origin

The dataset we use for our experiments consists of 14 software metric sets of different projects. They originate from two different sources. The first 9 sets used to train the model stem from the Tera-PROMISE repository [19]. The remaining 5 of 14 projects originate from the bug prediction dataset provided by D'Ambros *et al.* [9]. This dataset has already been used in various studies concerning bug prediction and serves as a good reference in the topic [12] [13].

Structure

To be able to perform regression and classification, it is important for the response variable of the data to consist of numeric values (1, 2, ..., n bugs) and not just a binary class (buggy or clean). The used software metric sets provide exactly the data needed for the experiments to work.

The metrics of the projects from the Tera-PROMISE database contain 20 code features and 1 defect count as response variable. They use the CK metrics as specified by Chidamber and Kemerer [3], designed specifically for code developed in Object Oriented languages. The metrics of the 5 projects in the bug prediction dataset by D'Ambros *et al.* come with a full 32 features, which consist of 17 source code metrics – shown in Table 2.3 – and 15 change metrics – listed in Table 2.4.

As described by the Tera-PROMISE repository, CK Metrics can be grouped under three stages of OO design processes: Identification of classes, semantics of classes and relationship between classes. The datasets used in the experiments contain a varying percentage of buggy classes from 9% up to 74% as shown in Table 2.5.

Metric Name	Description
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
FanIn	Number of classes that reference the class
FanOut	Number of classes referenced by the class
LCOM	Lack of Cohesion in Methods
NOC	Number Of Children
NOA	Number Of Attributes in the class
NOIA	Number Of Inherited Attributes in the class
LOC	Number of lines of code
NOM	Number Of Methods
NOIM	Number of Inherited Methods
NOPRA	Number Of PRivate Atributes
NOPRM	Number Of PRivate Methods
NOPA	Number Of Public Atributes
NOPM	Number Of Public Methods
RFC	Response For Class
WMC	Weighted Method Count

Table 2.3: Source code metrics included in the bug prediction dataset, containing the CK metrics suite and other object-oriented metrics [9].

Metric Name	Description
REVISIONS	Number of reversions
BUGFIXES	Number of bug fixes
REFACTORINGS	Number Of Refactorings
AUTHORS	Number of distinct authors that checked a file into the repository
LOC ADDED	Sum over all revisions of the lines of code added to a file
MAX LOC ADDED	Maximum number of lines of code added for all revisions
AVE LOC ADDED	Average lines of code added per revision
LOC DELETED	Sum over all revisions of the lines ofcode deleted from a file
MAX LOC DELETED	Maximum number of lines of code deleted for all revisions
AVE LOC DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code - deleted lines of code) over all revisions
MAX CODECHURN	Maximum CODECHURN for all revisions
AVE CODECHURN	Average CODECHURN for all revisions
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED AGE	Sum over age of a file in weeks times number of lines added during that week normalized by the total number of lines added to that file

Table 2.4: Change metrics included in the bug prediction dataset, as proposed by Moser *et al.* [11]

2.2.3 Experiments

In the experiments using a feedforward neural net, we first train the model previously explained to perform regression and classification based on source code metrics. This approach quickly leads to promising results. To achieve decent results, some parameter tuning is done in the beginning and is then gradually adapted to achieve better results. We use the same parameters for all experiments, so the performance for different projects may vary and could be improved significantly by customizing them.

After the initial setup of the experiments, some unwanted effects like heavily varying results, very low recall and missing values for precision (caused by zero positive predictions) keep occurring. 5-fold cross-validation is introduced to partition the test- and training data and to get more consistent results. To improve recall and receive even more consistent results, stratification is additionally used on the data. Stratification ensures that each fold is a good representative of the whole, in terms of the distribution of the response variable.

System	Release	Features	Classes	% Buggy
Eclipse JDT Core	3.4	32	997	20%
Eclipse PDE UI	3.4.1	32	1,497	14%
Equinox	3.4	32	324	40%
Mylyn	3.41	32	1,862	13%
Lucene	2.4.0	32	691	9%
xalan	2.5	20	803	48%
xalan	2.6	20	885	46%
Camel	1.2	20	608	36%
Prop	1	20	18'471	15%
Prop	2	20	23'014	11%
Prop	3	20	10'274	11%
Prop	4	20	8'718	10%
Prop	5	20	8'516	18%
Xerces	1.4	20	588	74%

Table 2.5: Used metric sets in the experiments and their percentage of buggy classses

Using 5-fold cross-validation on each of the datasets, the datasets are split into 80% training data and 20% testing data. In each of the folds, the neural net is trained and tested again. For projects with a dominant number of either buggy or clean examples, we implement oversampling by duplicating random examples of the smaller proportion. This greatly improves recall, while lowering precision within acceptable limits.

2.2.4 Results

The performance of the standard regression and the RbC model is measured by calculating the RMSE (root-mean-square error). The RMSE is a frequently used measure to describe the deviation of predicted values, compared to the actual values. To measure the effectiveness of multi-class classification and the CbR model, the area under the receiver operating characteristic curve (ROC AUC) is computed. The AUC serves as a good measure of the efficiency of a classifier, since it represents the performance of a classifier operating on various thresholds rather than a fixed value.

The AUC is calculated by selecting possible numbers of bugs as thresholds and plotting the the true positive rate on the y axis against the false positive rate on the x axis at each of the selected thresholds. The plotted graph, illustrated in Figure 2.3 describes the ROC curve, while the area under the curve is declared as the AUC. The AUC generally represents the performance of a classifier, whereas a value of 1 describes a perfect classifier and a value of 0.5 implies the performance of a random guesser. A value below 0.5 would indicate a negative correlation, so the model would be better fitted to predict the negatives rather than positives.

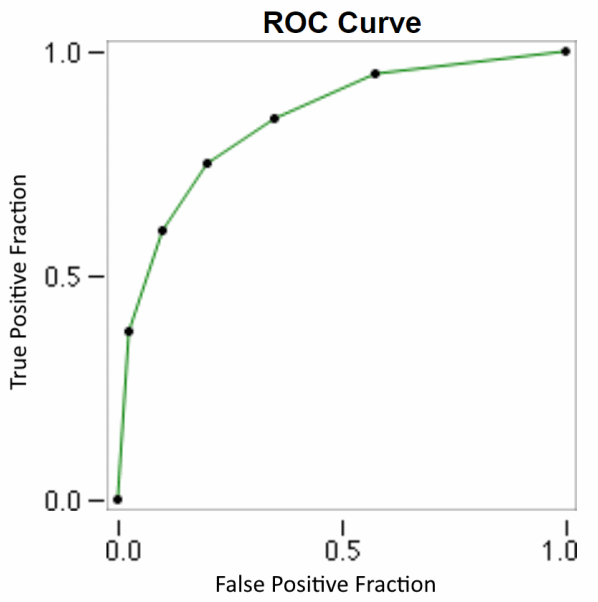


Figure 2.3: Example of a receiver operating characteristic (ROC) curve. The area under the curve describes the AUC.

As the standard classification model returns integer class labels describing the estimated number of bugs, we use all possible numbers of bugs (1,2,3...) as thresholds in the AUC calculation for standard classification. The CbR model uses the continuous value output from regression for the classification step. This allows us to select much smaller steps for the thresholds in AUC calculation. In our experiments, the thresholds to plot the ROC curve of CbR were selected in steps of 0.1*RMSE of regression, ranging from 0 to 2*RMSE.

As can be seen in Figure 2.4, CbR seems to outperform standard classification (Cla) in more than a half of all cases. We can also see that in some projects, where classification performs at the level of a random classifier (with an AUC of 0.5), CbR seemingly manages to increase the AUC up to a certain degree and gains the ability to actually classify correctly in some cases.

To validate these results, we want to verify the hypothesis that the performance of the models is different in a statistically significant way. To verify this hypothesis, we calculate the p-value, as well as effect size (chi-squared) [4] using a Kruskal-Wallis test. If the p-value is lower than 0.05, we have a 95% confidence interval that the results are different. Using the effect size, we can determine whether the difference is actually statistically significant and how big the effect is. We are using the following scale as proposed by Cohen [4] for a chi-squared effect size.

<0.1 = trivial effect
0.1 - 0.3 = small effect
0.3 - 0.5 = moderate effect
>0.5 = large effect

Table 2.6 shows the calculated p-values and effect sizes of the results. Most of the projects actually show a p-value lower than 0.05, but considering the effect size, the difference achieved in CbR compared to classification is often small to trivial. Still it manages to deliver better and statistically significant results in 7 of the 14 projects.

Project	P-Value	Effect Size
Eclipse JDT Core	9.6 E-159	-0.202
Eclipse PDE UI	6.1 E-164	-0.205
Equinox	1.0 E-145	-0.142
Mylyn	3.1 E-162	-0.147
Lucene	4.7 E-98	+0.130
Xalan 2.5	0.127	-0.003
Xalan 2.6	5.3 E-122	-0.137
Camel-1.2	0.016	-0.004
Prop-1	0.005	+0.006
Prop-2	1.8 E-118	-0.079
Prop-3	0.128	+0.004
Prop-4	2.1 E-96	-0.059
Prop-5	0.101	-0.004
Xerces	3.5 E-151	-0.151

Table 2.6: P-values and effect size of classification compared to CbR for all projects. The tests are carried out with a confidence interval of 95%. Highlighted effect sizes show statistically significant results.

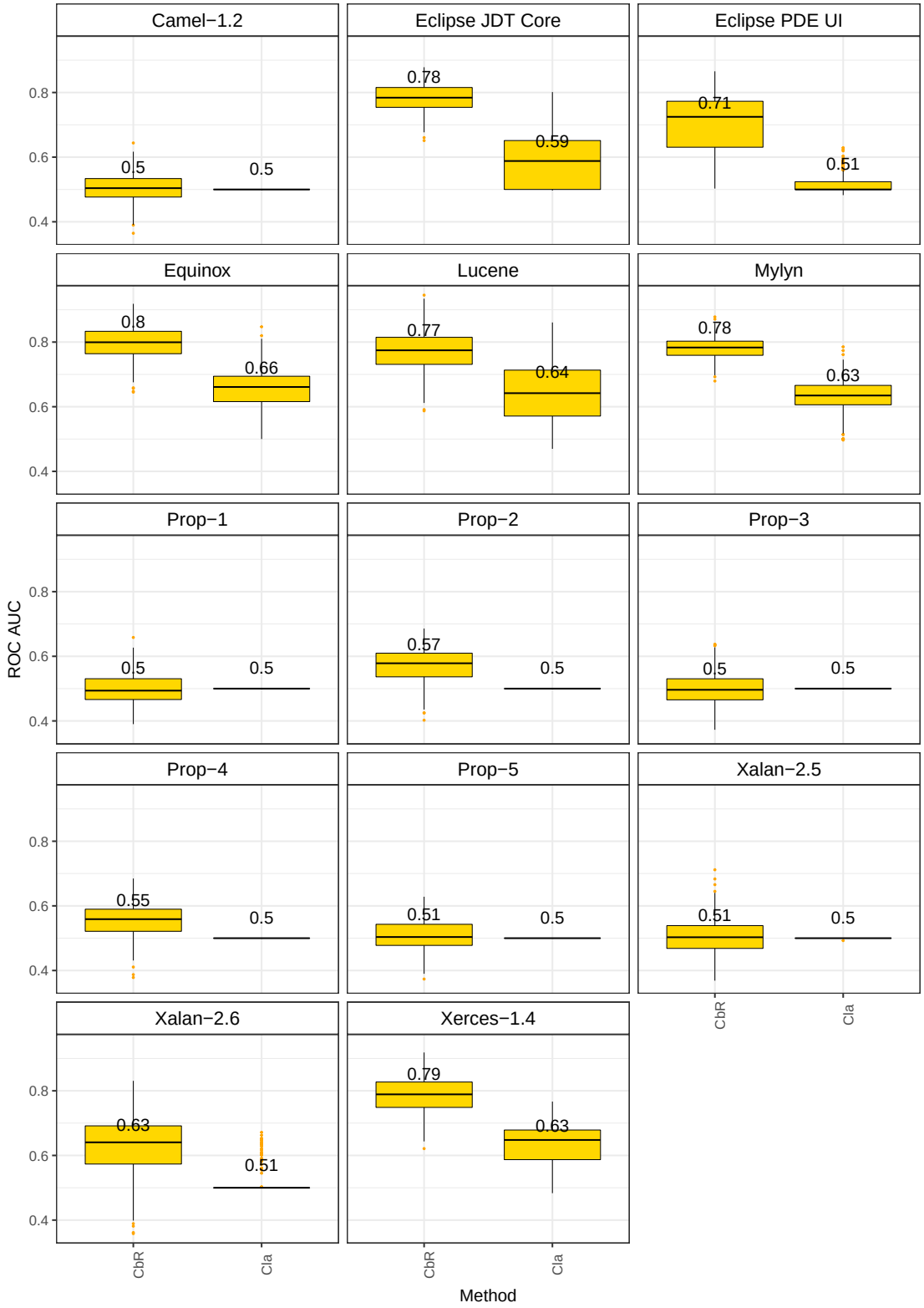


Figure 2.4: Boxplots of the experiments using classification and the CbR model on all projects. The y-axis denotates the ROC AUC. The two different classifiers are listed on the x-axis.

Figure 2.5 shows the RMSE of the regression model and the RbC model at different class caps (2,5,10 and 100). By limiting the number of possible classes, the RMSE obviously gets smaller, since the maximal difference between prediction and actual value is reduced.

As before, we validate our results by calculating the p-value and the effect size. In Table 2.7 we compare the results of standard regression to RbC with a cap of 100 and regression with RbC2 in Table 2.8. Comparing regression to RbC with a lower cap than 100 always decreases the p-value, while increasing the effect size for all projects and therefore all results with RbC2, RbC5 and RbC10 can be considered statistically relevant with moderate to large effect in all cases. As can be seen in the Figure 2.5, the RMSE of regression is already significantly decreased in many cases by using RbC100. With a minimum effect size of 0.318 and p-values smaller than 0.05 in all projects except for Xerxes, we can say that the results are statistically relevant. Using RbC with a lower cap always decreases the RMSE, while also increasing corresponding effect size. Looking at the results, we can safely state that the RbC model is able to improve the RMSE in predictions compared to standard regression.

Project	P-Value	Effect Size
Eclipse JDT Core	9.33E-72	-0.567
Eclipse PDE UI	3.54E-55	-0.495
Equinox	1.41E-26	-0.338
Mylyn	9.44E-98	-0.664
Lucene	4.54E-119	-0.845
Xalan 2.5	1.69E-55	-0.498
Xalan 2.6	7.39E-47	-0.455
Camel 1.2	9.95E-24	-0.318
Prop-1	3.19E-54	-0.491
Prop-2	1.39E-43	-0.439
Prop-3	3.07E-69	-0.557
Prop-4	7.40E-54	-0.489
Prop-5	2.79E-52	-0.482
Xerces	0.0677	+0.058

Table 2.7: P-values and effect size of regression compared to RbC100 for all projects. The tests are carried out with a confidence interval of 95%. Highlighted effect sizes show statistically significant results.

Project	P-Value	Effect Size
Eclipse JDT Core	0	-1.384
Eclipse PDE UI	0	-1.322
Equinox	0	-1.217
Mylyn	0	-1.345
Lucene	2.69E-294	-1.161
Xalan 2.5	0	-1.304
Xalan 2.6	0	-1.286
Camel 1.2	0	-1.276
Prop-1	0	-1.385
Prop-2	0	-1.386
Prop-3	0	-1.387
Prop-4	0	-1.386
Prop-5	0	-1.386
Xerces	5.13E-304	-1.180

Table 2.8: P-values and effect size of regression compared to RbC2 for all projects. The tests are carried out with a confidence interval of 95%. Highlighted effect sizes show statistically significant results.

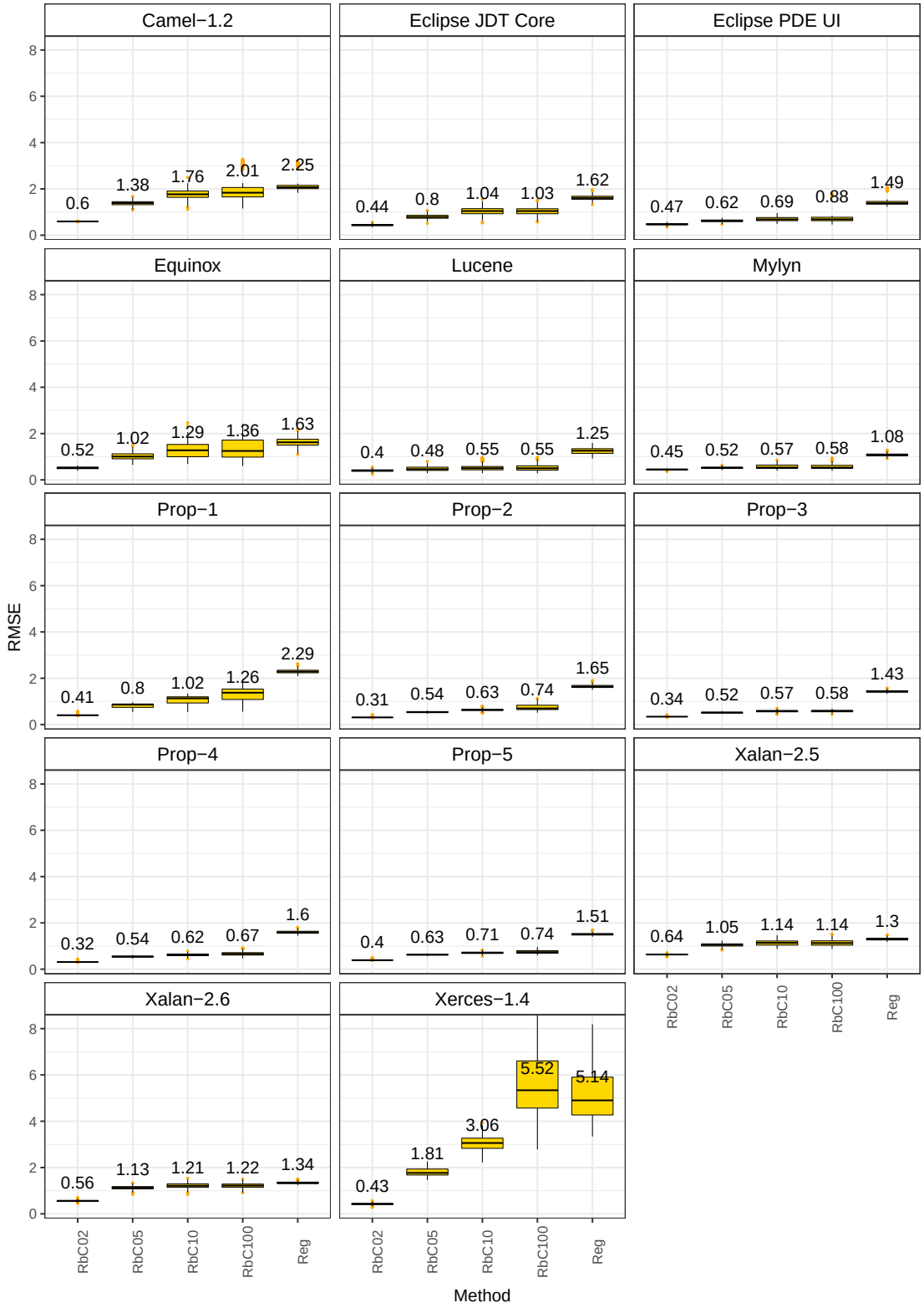


Figure 2.5: Boxplots of the experiments using regression and the RbC model on all projects. The y-axis denotates the root mean squared error (RMSE). The 5 different methods are listed on the x-axis.

2.2.5 Conclusions

The results of our experiments using a neural network on code metrics show that a neural net is able to correctly identify bugs in code and that its performance can be significantly increased by using either a regression-based classifier or a classification-based regressor. Another positive fact about the CbR and RbC models is that they manage to deliver correct predictions in some cases, where basic classification or regression fails to predict any bugs. With peak performances of an AUC up to 0.8 and a minimum RMSE down to 0.31, the neural net shows the ability to classify bugs correctly in many cases, heavily depending on the chosen model, dataset and parameters, which all still could be improved in various ways [12].

RbC outperforming regression can be explained by classification delivering a 0 error value when it correctly predicts the actual defect count. If the examples are classified correctly often enough, the error will most probably be smaller than the average error with regression, so RbC delivers a lower RMSE.

The performance increase with CbR can be explained by the granularity of data that is used. CbR performs binary classification based on all predicted probabilities and responds to small differences in a prediction. Classification on the other hand is only able to classify in numeric values, only selecting the most probable output and discarding any additional information. This leads to bigger differences of correctly and falsely classified examples.

While these results may not be true for all datasets and models, we generally can recommend the use of CbR or RbC for datasets with discrete response variables to improve the AUC or RMSE respectively.

3 Related Work

3.1 Text-Based Bug Prediction

Using vectorization of text to perform classification and sentiment analysis is a topic already researched by multiple studies in the past [8] [7] [2] and has delivered good results in the experiments. By analyzing the used words and vectorizing paragraphs of text, the model is able to identify topics and perform sentiment analysis correctly in many cases. While there exist different experiments using Doc2Vec for vectorization and analysis of texts prior to our study, there are no known attempts to use this approach for bug prediction. To the best of our knowledge, this is the first study researching the question, whether vectorization with Doc2Vec is able to perform successful bug prediction.

Many previous studies using a neural neural net for bug prediction are using some form of code features as input [20] [1] [15] [16] [18] [6] to perform classification or regression. As far as our research shows, this thesis is also the first study to attempt bug prediction using plain text bug fix commits in the experiments.

This thesis therefore contributes to understanding the complexity of bug prediction, as we are able to show that source code cannot be treated as simple text, but should rather contain additional contextual information like code metrics do.

3.2 Metrics-Based Bug Prediction

Many previous studies use code features as input for machine learning algorithms to perform bug prediction [20] [1] [15] [16] [18] [6]. While most of the experiments strive to achieve binary classification or discretized defect counts with according datasets, none of them use the full potential information contained in datasets with discrete defect counts to perform binary classification. Also, no previous experiments have been found using a multi-class classifier as a regressor in an attempt to reduce the measured RMSE.

Shortly after the beginning of our experiments though, studies using a similar approach to ours has been published in the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. The studies are documented in the paper "The Impact of Using Regression Models to Build Defect Classifiers" [17]. In the experiments, the authors apply different machine learning algorithms to perform classification as well as regression-based classification and show an improvement using the latter rather than a discretized defect classifier, while also suggesting to use a random forest classifier for best performance. The paper mainly focuses on comparing the performance of different machine learning models with and without applying regression-based classification.

Our studies are different from the ones mentioned above, as we set our focus on investigating the improvement achieved by a neural net in bug prediction when using a regression-based classifier or classification-based regressor. Instead of comparing the performance of different machine learning algorithms, we research the results of a single neural network layout performing different approaches. Additionally, no previous experiments are to be found using a similar approach to the classification-based regressor proposed in our experiments.

This thesis is therefore the first to specifically research the improvement achieved in bug prediction when using a regression-based classifier or classification-based regressor, compared to using classification or regression with a neural net.

4
Conclusions and Future Work

4.1 Doc2Vec

The experiments with the standard Doc2Vec model do not achieve any of the effects hoped for. While expecting to achieve some contextual knowledge and predict some cases correctly, the classifier based on Doc2Vec does not perform better than any random guesser. Using either of the two standard implementations, a vectorized form of source code does not seem to contain enough information to perform bug prediction on plain text. The experiments demonstrate that code should not be treated as simple text in paragraphs and needs a more complex models to achieve the expected results.

4.2 Feedforward neural net

As the experiments show, a feedforward neural net performing different regression- and classification-based approaches on code metrics is clearly able to improve results in bug prediction. While standard regression and classification already are able to achieve useable results in some cases, the performance can potentially be increased by using a regression-based classifier or a classification-based regressor, depending on the used model, dataset and parameters. The experiments demonstrate that to improve results of a neural network, models like RbC and CbR can provide a good solution.

4.3 Future Work

Instead of using Doc2Vec to turn whole paragraphs of plain text into vectors, the model should be able to gain some more local contextual knowledge of the source code to correctly classify bugs.

This could maybe be achieved by using a Recurrent Neural Network (RNN) in form of a sequence to sequence model like a Long Short Term Memory (LSTM) network. Using word vectors as input, the model could be trained to predict the next words for either a buggy or fixed context and then be used to classify based on the probability of some words occurring.

As an alternative, instead of just using simple plain text in the Doc2Vec model, the input could be extended with additional inputs, like the type of a word as parsed by an AST. With additional inputs, the model has more contextual information available and would potentially be able to classify bug patterns correctly. An example of such a model is illustrated in Figure 4.1.

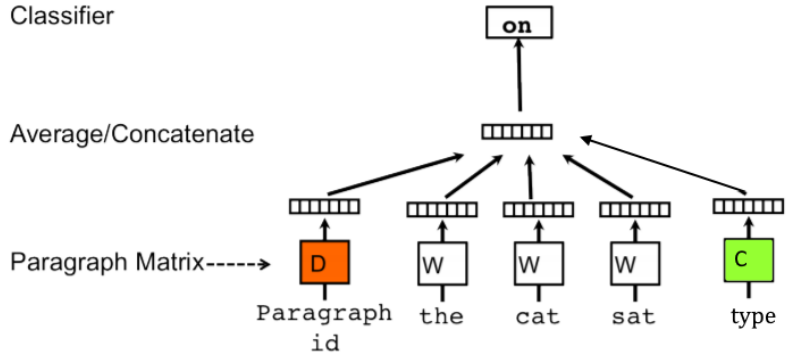


Figure 4.1: A suggestion for an extended PV-DM model, using word type as additional input

Concerning the experiments with a feed forward neural network, the performance could certainly further be improved by optimizing hyperparameters [13] for each project individually and by better feature selection and different datasets [12]. A deeper network might also be able to improve performance if configured correctly.

Bibliography

[1] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. Benchmarking machine learning techniques for software defect detection. *International Journal of Software Engineering and Application Vol.6 No.3*, pages 1–13, 2015.

[2] Amira Barhoumi, Yannick Estve, Chafik Aloulou, and Lamia Hadrich Belguith. Document embeddings for arabic sentiment analysis. *Language Processing and Knowledge Management*, 2017.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering vol 20. No 6.*, pages 1–16, 1994.

[4] Jacob Cohen. A power primer. *Psychological Bulletin Vol. 112, Issue 1*, 1992.

[5] JavaParser. Java 9 parser and abstract syntax tree for Java. Accessed: 2017-10-08.

[6] O. Kutlubay, Dou Gl Mehmet Balman, and Aye B. Bener. A machine learning based model for software defect prediction. *IEEE Transactions on Software Engineering Vol.39 No.4*, pages 1–17, 2005.

[7] Jey Han Lau and Timothy Baldwin. An empirical evaluation of Doc2Vec with practical insights into document embedding generation. *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 78–86, 2016.

[8] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. *ICML'14 Proceedings of the 31st International Conference on International Conference on Machine Learning, Vol. 32*, pages 1188–1196, 2014.

[9] DAmbros M, Lanza M, and R. Robbes. An extensive comparison of bug prediction approaches. *IEEE CS Press*, pages 31–40, 2010.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems, Vol. 2*, pages 2–5, 2013.

[11] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *ICSE '08 Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[12] Haidar Osman. *Empirically-Grounded Construction of Bug Prediction and Detection Tools*. PhD thesis, University of Bern, December 2017.

[13] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. Hyperparameter optimization to improve bug prediction accuracy. *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*, pages 1–3, 2017.

[14] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. Mining frequent bug-fix code changes. *2014 Software Evolution Week - IEEE Conference on Software Maintenance*, pages 1–3, 2014.

[15] Shruthi Puranik, Pranav Deshpande, and K.Chandrasekaran. A novel machine learning approach for bug prediction. *Elsevier*, pages 924–930, 2016.

[16] Shruthi Puranik, Pranav Deshpande, and K.Chandrasekaran. A novel machine learning approach for bug prediction. *Elsevier*, pages 924–930, 2016.

[17] G. K. Rajbahadu, S. Wang, Y. Kamei, and A. E. Hassan. The impact of using regression models to build defect classifiers. *MSR '17 Proceedings of the 14th International Conference on Mining Software Repositories*, pages 135–145, 2017.

[18] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing features to improve bug prediction. *IEEE Transactions on Software Engineering Vol.39 No.4*, pages 552 – 569, 2013.

[19] Tera-PROMISE. Tera-promise repository. Accessed: 2017-05-10.

[20] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Elsevier*, pages 1–7, 2009.

5 Anleitung zu wissenschaftlichen Arbeiten

This part contains a tutorial on how to set up, train and evaluate a Doc2Vec model for plain text classification and a feedforward neural network to perform bug prediction on code metrics using the DL4J library.

5.1 Prerequisites

To set up the experiments, we first need to choose and install a preferred Java IDE with Maven support. In our experiments we use the IntelliJ IDEA (community edition 2016). To implement and run the experiments, the Java Development Kit (JDK) and the Java Runtime Enviroment (JRE) is required, offering compatibility for a broad range of systems. To access the plain text data used in the experiments with Doc2Vec, MySQL also has to be installed on the system.

All models used in the thesis are built and run using DL4J, a deep learning library for Java. To include DL4J in our project, we start by creating a new Maven project and generate the pom.xml file. In the pom.xml file the following dependencies are to be included:

org.deeplearning4j with artifacts (Contains the core functionality of DL4J): deeplearning4j-core, deeplearning4j-nlp, deeplearning4j-ui

org.nd4j with artifacts (A scientific library for handling N-Dimensional arrays): nd4j-native-platform, nd4j-native

5.2 Doc2Vec

5.2.1 Setup

After setting up the project and including all the required dependencies, we can now begin with the implementation of Doc2Vec. To access the data, we use a simple SQL connector and access the database to get code chunks as strings. We use 80% of all entries as training set and the remaining 20% as testing set. To train our model, we need to create an iterator that implements the "LabelAwareIterator" interface and iterates through labeled paragraphs of text. In our example, we create a "DocIterator" class that iterates through source code examples, either labeled "bug" or as "fix". To achieve this we pass it a list of all examples and a list with corresponding labels. For our model, we also need to declare a tokenizer factory to separate words in the used paragraphs. We use a default tokenizer factory for this.

```
LabelAwareIterator iterator =
      new DocIterator(exampleList, labelList);

TokenizerFactory tokenizerFactory =
      new DefaultTokenizerFactory();
```

Using the PraragraphVectors implementation of DL4J, the model is easily defined and initialized. We start by defining a new ParagraphVectors object and call a chain of methods on the builder to configure the model. We define a learning rate, batch size, number of epochs and whether we want to train word vectors. Furthermore we pass it our previously created iterator and a default tokenizer factory, used to tokenize the texts. Lastly, we choose the sequence learning algorithm. Here we can choose between the two implementations of paragraph vectors: DBOW or DM. After setting all parameters, we call the build method to complete the configuration.

```
ParagraphVectors paragraphVectors =
      new ParagraphVectors.Builder()
                .learningRate(learningRate)
                .batchSize(batchSize)
                .epochs(epochs)
                .iterate(iterator)
                .trainWordVectors(true)
                .tokenizerFactory(tokenizerFactory)
                .sequenceLearningAlgorithm(new DM())
                .build();
```

After defining and building the complete configuration, we can train the model by simply calling the .fit() method on the ParagraphVectors object. This step will start the learning process and generate the paragraph vectors.

```
paragraphVectors.fit();
```

5.2.2 Training and Inference

To use the trained paragraph vector model for classification, we can compare the vectorized form of a previously unseen paragraph to already learned paragraph vectors. To accomplish this, a "MeansBuilder" object builds a vector for the selected text (document), based on the trained paragraph vector model. A "LabelSeeker" object then searches the nearest labels of the vectorized paragraph and returns a list of labels with a similarity indicator between 0 and 1. If the example is more similar to a "bug" labeled paragraph, we classify it as a "bug", otherwise as a "fix".

```
LabelledDocument sourceCode =
            testSetIterator.nextDocument();
INDArray codeAsVector =
            meansBuilder.documentAsVector(sourceCode);
List<Pair<String, Double>> scores =
            seeker.getScores(codeAsVector);

Double highest = 0.0;
String label = "";
for (Pair<String, Double> score : scores) {
   if (score.getSecond()>highest){
         highest = score.getSecond();
         label = score.getFirst();
     }
}
```

5.3 Feedforward Neural Network

5.3.1 Setup

A neural feedforward network in DL4J is created by defining and building a multi-layer configuration. The different layers and parameters of the network all have to be stored and built in this configuration, before we initialize and train the net.

The first layer is an input layer of type "DenseLayer", which takes the features as input to the network and forwards it to the next layer. The input size of the layer consists of the number of features and it returns one output for each input node in the next layer.

```
DenseLayer layer0 = new DenseLayer.Builder()
                        .nIn(numFeatures)
                        .nOut(numFeatures)
                        .build())
```

The second (the 'hidden') layer is also of type "DenseLayer". It uses the number of features as input from the first layer and returns a specified number of outputs. In our experiments, we define the number of outputs as the average between number of features and number of classes.

```
numHiddenLayer = (numFeatures + numClasses)/2;

DenseLayer layer1 = new DenseLayer.Builder()
                        .nIn(numFeatures)
                        .nOut(numHiddenLayer)
                        .build())
```

The third and final layer is defined as an "OutputLayer". It uses the output from the second layer as its input and returns an output in the size of the of all possible classes. Each output node of this layer corresponds to a specific class, in our case the number of bugs. Each of the output values describes the probability of the current example being part of this class. In this layer, we also specify the loss and activation function used in the network. To achieve a probability distribution over all classes, the output layer uses a "negative log likelihood" loss function and a "softmax" activation function.

```
OutputLayer layer2 = new OutputLayer
                .Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                .activation("softmax")
                .nIn(numHiddenLayer)
                .nOut(numClasses)
                .build()
```

After specifying all layers of our neural network, we add them to our configuration and define the general parameters, before building the configuration. The parameters contain the number of iterations, a seed for random number generation, the initial weights, a learning rate, whether we want backpropagation and if we want the model to use pretrained weights. The ".list()" method creates a ListBuilder object to hold multiple layers and must be run before specifying the layers. The parameters can be optimized for each project independently and changing them will influence the results.

The complete configuration of the network looks as follows and has to be built with the indicated chained methods.

```
MultiLayerConfiguration config = new
      NeuralNetConfiguration.Builder()
                        .seed(seed)
                        .iterations(iterations)
                        .weightInit(WeightInit.XAVIER)
                        .learningRate(learningRate)
                        .list()
                        .layer(0, layer0)
                        .layer(1, layer1)
                        .layer(2, layer2)
                        .backprop(true)
                        .pretrain(false)
                        .build();
```

With our built configuration, we can now initialize the model at any time by defining and initializing a MultilayerNetwork.

```
MultiLayerNetwork model = new MultiLayerNetwork(config);
model.init();
```

Using the initialized model, we can now train it and test the performance of our implementation.

5.3.2 Training and Inference

To train our model, we need a ND4J DataSet object that can be passed to our network. With a simple CSVReader, we read our dataset and iterate through all available source code metrics of different projects. Using a KFoldIterator (with included stratification), we then create our separate training and testing set for each project. For every single run of the experiments, a new training and testing set is created.

With a complete dataset of features and labels, we can easily train the network by calling the model.fit() method and passing the dataset as parameter.

```
model.fit(trainingData);
```

The training step uses up most of the time when working with neural nets. As soon this step is finished, we can quickly evaluate predictions on our previously unseen testing data, only passing the features of the testing set to the model:

```
INDArray output = model.output(testData.getFeatureMatrix());
```

This method call results in a matrix of type INDArray, containing as many rows as there are examples in the testing set and a column for every possible class. For the regression, classification and CbR models, the number of classes is defined by the maximum defect count contained in the metrics set of a project. In RbC, we manually set the maximum number of output classes to either 2,5,10 or 100.

For regression we then calculate the predicted value by adding up the number of bugs multiplied by the predicted probability of containing this many bugs for every row of the output.

```
for (int i = 0; i < output.rows(); i++) {
   for (int j = 0; j < output.columns(); j++) {
      guessValue = output.getFloat(i, j);
      bugsRegression[i] += guessedValue * j;
   }
}
```

For classification we simply select the index of the highest value output in each column, describing the most probable output and therefore the predicted number of bugs.

```
for (int i = 0; i < output.rows(); i++) {
   for (int j = 0; j < output.columns(); j++) {
      guessValue = output.getFloat(i, j);
      if (guessValue > highestGuess) {
         highestGuess = guessValue;
         bugs[i] = j;
      }
   }
}
```

To evaluate the predictions of our model, we can eventually compare the actual labels of the testing data, with the predicted output by the model. In regression we calculate the RMSE to measure the performance. in classification, we determine the AUC as explained in previous chapters.