Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns

Khaled Walid Al-Sabbagh[1][0000−0003−2571−5099], Miroslaw Staron[1][0000−0002−9052−0864], Regina Hebig[1][0000−0002−1459−2081], and Wilhelm Meding[2]

[1] Chalmers | University of Gothenburg, Computer Science and Engineering Department, Gothenburg, Sweden
[2] Ericsson AB
{khaled.al-sabbagh, miroslaw.staron, regina.hebig}@gu.se
wilhelm.meding@ericsson.com

Abstract. Background: Continuous Integration (CI) is an agile software development practice that involves producing several clean builds of the software per day. The creation of these builds involve running excessive executions of automated tests, which is hampered by high hardware cost and reduced development velocity. **Goal:** The goal of our research is to develop a method that reduces the number of executed test cases at each CI cycle. **Method:** We adopt a design research approach with an infrastructure provider company to develop a method that exploits Machine Learning (ML) to predict test case verdicts for committed source code. We train five different ML models on two data sets and evaluate their performance using two simple retrieval measures: precision and recall **Results:** While the results from training the ML models on the first data-set of test executions revealed low performance, the curated dataset for training showed an improvement on performance with respect to precision and recall. **Conclusion:** Our results indicate that the method is applicable when training the ML model on churns of small sizes.

Keywords: Machine Learning, Verdicts, Code Churn, Test Case Selection

1 Introduction

CI is a modern software development practice, which is based on frequent integration of codes from developers and teams into a product's main branch [23]. One of the cornerstones of its popularity is the promise of higher quality delivered by frequent testing and the ability to quickly pinpoint the code that does not meet quality requirements. To achieve this, CI systems execute tests as part of the integration [5]. However, excessive execution of automated software tests is penalized with high hardware cost and reduced development velocity that may consequently hinder agility and time to market.

In order to address this challenge, a CI system should be able to pinpoint exactly which test cases should be executed in order to maximize the probability

of finding defects (i.e. to reduce the "empty" test executions). To achieve this, the CI system needs to be able to predict whether a given test case has a chance of finding a defect or not, or at least whether it will fail or pass – predict the verdict of a test case execution.

We set off to address the problem of predicting test case verdicts by training five ML models on a large data set of historical test cases that were executed against changes made to a software developed at company A. The term "code churn" is defined as a measure that quantifies these changes. Throughout the remaining sections of this paper, we use this term to refer to committed code made during different CI cycles.

Our research is inspired from a previous study conducted by Knauss et el. [13], where the authors explored the relationship between historical code churns and test case executions using a statistical model. Their method used precision and recall metrics in predicting an optimal suite of functional regression tests that would trigger failure. In this paper we expand on that approach by going one step further – conducting a textual analysis of what is the code that is actually being integrated. For example, instead of using code location as the parameter, we use such measures as the number of 'if' statements or whether the code contains data definitions. Similarly, our choice of using code churns is inspired from the work of Nagappan and Ball [14], in which the authors presented a technique for early faults prediction using code churn measures. In their publication, the authors identified a positive correlation between the size of code churns and system defects density. Our method builds on this and uses the Bag of Words (BOW) approach to extract features from code churns. This enables the identification of statistical dependency between keywords and test case verdicts. For example, a churn containing a frequent occurrence of keywords like new or delete might trigger specific tests to fail. More precisely, we aim at investigating the following research question:

> *How to reduce the number of executed test cases by selecting the most effective minimal test suite when integrating new code churns into the product's main branch?*

Our study was conducted in collaboration with a large Swedish-based infrastructure providing company. We study a software product that has evolved over a span of a decade by different cross functional teams. As a result of our study, we present a method that uses ML to predict test case verdicts (MeBoTS).

To address this research question, we conduct a design research study, where we develop a new method and evaluate it on the company's data set. Our method is based on the research by Ochodek et al. [17], which uses textual analyses to characterize source code. Our MeBoTS method builds on that by using historical test verdicts as predicted variables and uses Random Forest algorithms to make the predictions.

The remainder of this paper is organized as follows: Section II provides background information about two categories of ML. The sections that follow provide: an overview of the most related studies in this area, a description of the

method that we developed in our study as well as the results, validity analysis, recommendations, and finally, conclusion.

2 Background

2.1 Categories of Machine Learning

Machine learning is a class of Artificial Intelligence that provides systems the ability to automatically make inferences, given examples relevant to a task [8]. The main advantage of using Machine learning over classical statistical analysis, is its ability to deal with large and complex data-sets [12]. These systems can be classified into four categories depending on the type of supervision involved in training: a) Supervised, b) Unsupervised, c) Semi-supervised, d) Reinforcement Learning [8]. Since we view the problem of predicting test case verdicts as a classification problem, we briefly mention the supervised learning category.

In supervised learning the training data-set fed into the ML model contains the desired solution, called labels. The model tries to find a statistical structure between these examples and their desired solutions [12]. A typical task for this kind of learning is classification.

2.2 Tree-based and Deep Learning Models

In Machine learning, a decision tree is an algorithm that belongs to the family of supervised learning algorithms. The algorithm has an inherent tree-like structure and is commonly used for solving classification and regression problems [20]. Starting from the root node, the algorithm uses a binary recursive scheme to repeatedly split each node into two child nodes, where the root node has the complete training sample [1]. The resulting child nodes correspond to features in the training data, whereas the leaf nodes correspond to class labels (binary or multivariate). Other algorithms, such as Random Forest and Adaptive Boosting, use Decision trees as a primary component in their structure. These algorithms build a collection of decision trees, called an ensemble, to increase the overall learning of the classification or regression task at hand [12].
Deep Learning is a branch of ML that was founded on the premise of using successive learning "layers" to achieve more useful representation of the data [8]. The learning of these successive layers are achieved via models called Neural Networks (NN) [8]. A multilayer NN is one that consists of at least three layers: 1) one input layer, 2) at least one hidden layer, 3) and an output layer of artificial neurons [10]. Similarly, a Convolutional network (CNN) consists of a set of learning layers [8]. The main difference between the two networks is in the way they search for patterns in the input space [12]. More precisely, a CNN works by sweeping a matrix-like window, called filter, over every location in every patch to extract patterns from the input data [12]. As opposed to Decision Trees, ANN have a black box nature, which means that no insight about how their predictions were made can be easily accessible [12]. Nevertheless, the main advantage of using deep learning comes from their ability to handle large and complex data-sets of features.

2.3 Code Churns

The amount of changes made to software over time is referred to as code churn [14]. As new churns are added, new risks of introducing defects into the system emerge [21]. According to Y. Shin et al. [21], each check-in made into a version control system includes newly added or deleted code that increase the chances of triggering failures. At some point in time, an evolving system may be vulnerable, on average, to one extra fault for every new additional change [11]. For example, in C programming, the declaration of 'static' local and global variables are among the most confused keywords by developers, as each static local and global declaration has a different effect on how the data will be retained in the program's memory [3].

3 Related Work

In the following we discuss related work on the specific use of machine learning for test case selection or prioritization.

3.1 ML-based Test-Case Selection

Around 2015/16, we find the first machine learning based approaches for test-case selection. With only 4 studies included in the systematic mapping study by Durelli et al. [9], the use of machine learning for test case prioritization seems to be new.

Busjaeger and Xie [4] present an industrial case study in which a linear model is trained with the SVMmap algorithm using the features Java code coverage, text path similarity, text content similarity, failure history, and test age. The evaluation on the industrial case study, considering 2000 changes and over 45 000 test executions shows an Average Percentage of Faults Detected (APFD) of around 85%.

Chen et al. [6] prioritize test programs for compilers "identifies a set of features of test programs, trains a capability model to predict the probability of a new test program for triggering compiler bugs and a time model to predict the execution time of a test program."

Spieker et al. [22] introduced Retecs, a reinforcement learning-based approach to test case selection and prioritization. Retecs considers duration of a test case's execution, previous last execution and failure history. Online learning is used to improve test case selection between continuous integration cycles. The approach was evaluated on 3 industrial data sets, including together more than 1.2 million verdicts, and achieved a normalized Average Percentage of Faults Detected (APFD) of around 0.4 to 0.8 depending on the data set.

Most recently, Azizi and Do [2] perform test case prioritization by calculating a ranked list of components considering the access frequency of a component as well as a fault risk. The fault risk for each component is thereby predicted using a linear model of change and bug histories. Test cases associated with highly

ranked components are prioritized. The approach was evaluated on three web-based systems and where it could reduce the number of test cases by 20% while still finding over 80% of the errors.

Palma et al. [18] replicate and extend a work of Noor and Hemmati [16] and [15], to predict test case failure based on a machine learned model basing on test quality metrics as well as similarity-based metrics.

However, to the best of our knowledge, no other learning-based method works for code-churns. The only exception is one of our previous collaboration with Knauss et al. [13]. The introduced code-churn based test selection method (CCTS) analyzes correlations between test-case failure and source code change. The approach was evaluated in several configurations, leading to results ranging from 26% precision up to a 54% with a 97% recall. We deem these results promising and one of the main motivations for this study.

4 Method using Bag of Words for Test Selection (MeBoTS)

The following section is a description of the MeBoTS method used in this research, which comprises of three sequential steps, as shown in Figure 1. The method utilizes two Python programs and an open source textual analyzer program, called CCFLEX [17].

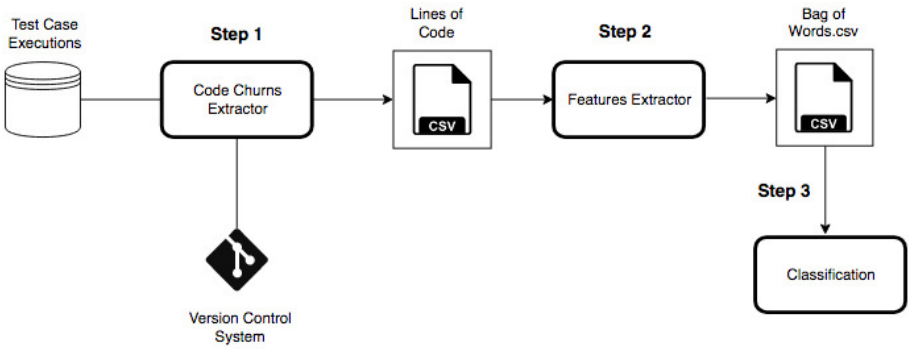


Fig. 1. The MeBoTS method.

4.1 Code Churns Extraction (Step 1)

A Python-based code churn extraction program was created to collect and compile code churns committed in the source code repository. The program takes one input parameter: a time ordered list of historical test case execution results extracted from a database, where each element in the list represents an instance

of a previously run test case and holds information about: the name of the executed test case, the baseline code in Git against which the test case was executed, and the verdict value - as shown in Table 1.

Table 1. An Excerpt of the Historical Test Case Executions List

Baseline	Test Case Name	Verdict
ca82a6dff817ec66f	ST-case 22	FAILED
ca82a6dff817ec66f	FT-case 42	PASSED
34bb5e22134200896	FT-case-333	FAILED
34bb5e22134200333	FT-case-3	PASSED

The program first loops through the extracted list of tests and looks at the change history log maintained by Git and performs a file comparison utility (diff) on a pair of consecutive baselines in the tests list. Note that each baseline value is a hash representation of a revision (build), pointing to a specific location in Git's history log. The result is a fine-grained string that comprises the committed code churns, where each LOC in the churn is compiled with its: 1) filename, 2) physical file path, 3) test case verdict, 4) baseline hash code.

The resulting string is then arranged in a table-like format and written in a csv file, named as 'Lines of Code' in Figure 1.

4.2 Textual Analysis and Features Extraction (Step 2)

The result of the extract from Git is saved as an array (code churn, filename, physical file path, test case verdict and baseline). This file is the input to our textual analysis and feature extraction. The textual analysis and feature extraction use each line from the code churn and:

- creates a vocabulary for all lines (using the bag of words technique, with a specific cut-off parameter),
- for the words that are used seldom (i.e. fall outside of the frequency defined by the cut-off parameter of the bag of words), a token is created,
- finds a set of predefined keywords in each line,
- check each word in the line whether it is part of the vocabulary, it should be tokenized or if it is a predefined feature.

An example input is presented in Table 2. The input contains an example code in C.

For the textual analyses, we can pre-define (arbitrarily for this example) two features: "`if`" and "`int`". The bag of words analysis also found the word "`printf`" as frequent. It has also defined the following tokens:

- "`a`" – to denote the words (of any length) that contain only lowercase letters (e.g. "`condition`"),

Table 2. Input to the textual analysis and feature extraction

Filename	Path	Content	Hash
firstFile.c	c:/folder	`if (condition == true) printf('Hello World');`	aa00111
firstFile.c	c:/folder	`printf('\n');`	aa00111
secondFile.c	c:/folder	`int i = 10;`	aa00111

- "`Aa`" – to denote the words that start with capital letters and continue with lowercase letters (e.g. "`Hello`"),
- "`0`" – to denote the numbers, i.e. sequence of numbers of any length (e.g. "`10`")

The manual features and the bag of words results are then used as features in the feature extraction. Table 3, which corresponds to the input from Table 2.

Table 3. Output from the feature extraction algorithm

Filename	Path	`if`	`int`	`a`	`Aa`	Content
firstFile.c	c:/folder	1	0	3	2	`if(condition == true) printf("Hello World");`
firstFile.c	c:/folder	0	0	2	0	`printf("\n");`
secondFile.c	c:/folder	0	1	1	0	`int i = 10;`

Table 3 is a large array with the numbers, each representing the number of times a specific feature presents in the line. This way of extracting information about the source code is new in our approach, compared to the most common approaches of analyzing code churns. Compared to the other approaches, MeBoTS recognizes what is written in the code, without understanding of the syntax or semantics of the code. This means that we can analyze each line of code separately, without the need to compile the code or without the need to parse it. This means, that we can take code churns from different files in the same baseline and analyze them together. MeBoTS also goes beyond such approaches like Nagappan et al. [14], which characterizes churns in terms of metrics like number of churned lines or churn size.

4.3 Training and Applying the Classifier Algorithm (Step 3)

We exploit the set of extracted features provided by the textual analyzer in step 2 as the independent variables and the verdict of the executed test cases as the dependant variable, which is a binary representation of the execution result (passed or failed). The MeBoTS method uses a second Python program that utilizes and trains an ML model to classify test case verdicts. The program reads the BOW vector space file in a sequence of chunks, merging the extracted feature vectors and the verdicts vector into a single data frame that gets split

into a training and testing set before it is fed into the models for training. Table 4 shows an excerpt of the generated data frame.

Table 4. Input to the Classifier Model

File Name	Line Number	F1	F2	F3	F4	F5	..	F500	Verdict
firstFile	1	0	0	6	1	0	..	0	PASSED
firstFile	2	0	0	5	3	2	..	0	PASSED
secondFile	1	0	0	6	1	0	..	0	FAILED

5 Research Design

5.1 Collaborating Company

The study has been conducted at an organization, belonging to a large infrastructure provider company. The organization develops a mature software-intensive telecommunication network product. The organization consists of several hundred software developers, organized in several empowered agile teams, spread over a number of continents. Given that they have been early adopters of lean and agile software development methodologies, they have become mature in these areas of work. They have also implemented CI and continuous deliveries.

The organization is also mature with regard to measuring. For instance every agile team, as well as leading functions/roles, uses one or more monitors to display status and progress in various development and devops areas. A well-established and efficient measurement infrastructure, automatically collects and processes data, and then distributes the information needed by the organization.

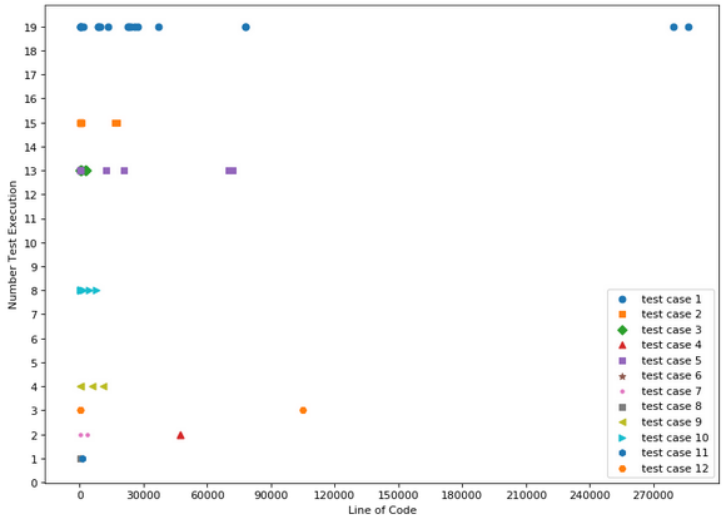
5.2 Dataset

The data-set provided by company A contained historical test case execution results for a mature software product that has evolved for almost a decade. The analyzed product consisted of over 10k test cases and several million lines of code written in the C language. We decided to test the MeBoTS on a set of randomly selected tests that, presumably, reacted to changes in the source code during different CI cycles. Our selection of test cases was based on the granularity of test executions whose verdicts changed from one state to another(see Table 1).

The extracted data-set belonged to twelve test cases that were executed 82 times during different CI cycles. The size of the extracted churns was 1.4 million lines of code, among which 618k lines were labeled as passed and 776k as failed. To better understand the shape of the data-set, we visually inspected the size of code churns covered by each test execution. The scatter plot in Figure 2 shows the distribution of the 82 extracted test executions, belonging to the twelve test cases. Each mark on the plot represents one test case execution. The x-axis represents the number of lines in the code churn the test case was executed on.

The y-axis represents the overall number of test executions for the executed test case. Test executions of the same test case are marked with the same symbol. The visual inspection of the scatter plot suggests that our data-set comprised of churns of varying sizes (large and small). We interpreted this distribution

Fig. 2. Churn Size per Test Execution Plot.



with uncertainty of whether large churns contain additional noise that would adversely affect the training of the ML models. As a result, we decided to curate the original data-set by filtering out tests that were executed on churns whose total size exceeded 110k lines. The visual inspection of the curated data-set is represented in Figure 3, comprised of 290k lines of code, containing a fairly balanced representation of the binary classes (passed and failed), with 110k lines belonging to the passed class and 180k lines belonging to the failed class.

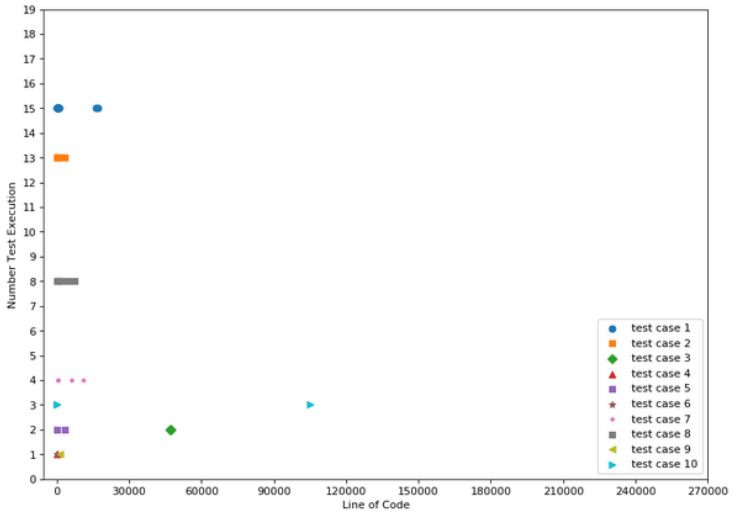
The two data-sets described above were used for training the ML models. The first data-set was used in the first phase, whereas the second curated data-set was used in the second phase of ML training, and ultimately became our focus due to data size homogeneity.

5.3 Evaluating and Selecting a Classification Model

To select the most suitable model for classifying test verdicts, we selected five different ML models and trained them sequentially. The five models are: 1) Decision Tree, 2) Random Forest, 3) AdaBoost, 4) Multilayer NN, 5) CNN

The choice of selecting the three tree-based models was due to their low computational cost and white box nature, whereas the selection of ANN models were based on their ability to abstract large and complex number of features.

Fig. 3. Churn Size per Test Execution After Curation Plot.



Each of the five classification models for test verdicts uses i) the historical test case verdicts ii) the feature vectors in the bag of words table as the baseline of prediction. The evaluation was done in two iterations. In the first iteration, the five models were trained on the original data-set, which contained a mix of large and small churns, comprising a total of 1.4m lines of code for 500 feature vectors. The second iteration involved training the models on the curated data-set, which almost contained 290k lines for the same 500 features, as in the first iteration. Both data sets, curated and original, were split as follow: 70% for the training set and 30% for the test set.

To save the long run time required by automated hyper-parameter tuning tools such as grid and random search, the configuration of the models was done manually. Table 5 summarizes the hyper-parameters used for training the models. We used the implementation of Decision Tree, Random Forest, and AdaBoost algorithms available in the Python scikit-learn library [19] and then used the Keras library [7] for the implementation of Multilayer NN and CNN.

The hyper-parameters of the three tree-based models were kept in their default state as found in the scikit-learn library. The only alterations made were in the 'random state' value in Decision Trees and the n_estimator (number of trees) in both Adaboost and Random Forest. With respect to the ANN models, the architecture of the multilayer ANN was represented with three sequential dense layers that consisted of: one input layer, one hidden layer, and one output layer. For the CNN, the stack of layers comprised of: a Reshape layer, a Convolution layer, a Maxpooling layer, and four Dense layers. The learning in both models was induced over 100 epochs (iterations), as can be seen in table 5

Table 5. The Evaluated Models and Their Hyper-Parameters

Classifier	Random State	Number of Trees	Number of Layers	Epochs
Decision Tree	123	-	-	-
Random Forest	-	50	-	-
AdaBoost	-	100	-	-
Multilayer NN	-	-	3	100
CNN	-	-	8	100

The performance of the classifiers were evaluated using simple retrieval measure: recall and precision.

These measures are based on the following four categories of errors:

- True positives: correct prediction of test executions that pass
- True negatives: correct prediction of test executions that fail
- False positives: incorrect prediction of test executions that pass
- False negatives: incorrect prediction of test executions that fail

Precision is the number of correctly predicted tests divided by the total number of predicted tests, calculated as follows:

$$precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

Recall is the number of correctly predicted tests divided by the total number of tests that should have been positive.

$$recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

While recall and precision measures relate to one another, precision is a measure of exactness, whereas recall is a measure of completeness indicating the percentage of all predicted failed tests in our case. Since the goal of this study is to reduce the amount of test executions without altering the effectiveness of testing, we needed to minimize the risk of missing tests that will actually fail and, therefore, accept some false alarms in the prediction of failed tests. Thus, we believe that the measure of the mode's precision is more important than its recall.

6 Results

To answer the research question, we present the results of using MeBoTS for predicting test case verdicts. We interpret the results of the analysis in light of the reported rates of precision and recall and the values of the four categories of errors: true negatives, false positives, false negatives, and true negatives, as shown in Tables 6 and 7 list the results of training the five models using the original and curated data sets.

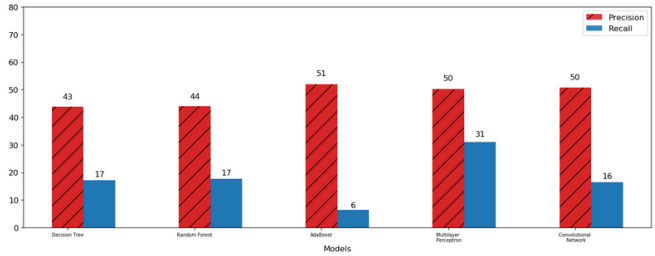
6.1 Training the Models on Churns of Varying Sizes

The evaluation of the five models in the first iteration reports a mean precision of 47% and a mean recall of 17%, suggesting that all models achieved low performance. The best result was obtained by the Multilayer NN model with a precision rate of 50.3% and a recall rate of 31%. The precision and recall rates for the five models can be seen in Figure 4. The interpretation of these values suggest that out of the 406,948 lines of code that actually triggered a test case failure, the model correctly predicted test failure for 226,994 lines, whereas it could correctly predict a passing test verdict for 5,758 out of 11,428 lines. Similarly, the results of the four categories of errors for the other models can be seen in Table 6 and interpreted in the same fashion.

Table 6. Models Evaluation before Data Curation

Model	Precision	Recall	True Neg	False Pos	False Neg	True Pos
Decision Trees	43.9%	17.2%	191,883	40,781	153,709	32,003
Random Forest	44%	17.7%	190,864	41,800	152,794	32,918
AdaBoost	51.9%	6.5%	221,472	11,192	173,590	12,122
Multilayer NN	50.3%	31%	226,994	5,670	179,954	5 758
CNN	50.7	16.5	202,745	29,919	154,890	30,822

Fig. 4. Precision and Recall of The Models Before Data Curation



6.2 Training the Models on Churns of Small Sizes

The second iteration of analysis involved training the same set of models on the curated data-set, which excluded tests covering churns with quantities above 110k lines of code. The results, shown in Figure 5, indicate an improvement in precision and recall when compared to the results in the first iteration for the same types of models. Table 7 reports the results from the second round of training, showing a mean precision of 70% and a mean recall of 44.5%. The Multilayer NN model performed best in prediction, such that, it correctly predicted 48,755

lines that actually triggered a test case failure, out of 67,363 lines in the test set.

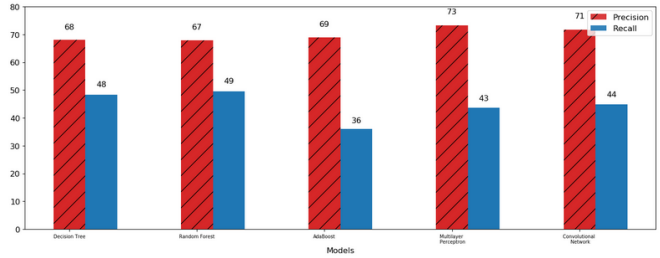
Table 7. Models Evaluation after Data Curation

Model	Precision	Recall	True Neg	False Pos	False Neg	True Pos
Decision Trees	68%	48.4%	46,445	7,548	17,011	15,981
Random Forest	67.9%	49.5%	46,252	7,741	16,637	16,355
AdaBoost	69%	36%	48,656	5,337	21,075	11,917
Multilayer NN	73.3%	43.6%	48,755	5,238	18,608	14 384
CNN	71.75%	44.9%	48,162	5,831	18,179	14,813

6.3 Implication

The results show that we can predict the verdict of a test case with a precision of 73.3%. This means that we can use the results to reduce the test suite by excluding tests that are predicted to pass, but we need to know that there is 26.7% probability that we miss a test case failure. This means that the reduction of the test suite comes with a cost. This cost can be reduced, for example, if we collect the test cases which were not executed and execute them with lower frequency (e.g. during a nightly test suite instead of hourly builds).

Fig. 5. Precision and Recall of The Models After Data Curation



7 Validity analysis

When analyzing the validity of our study, we used the framework recommended by Wohlin et al. [24]. We discuss the threats to validity in two categories: internal and external. Typically, a number of internal threats to validity emerge in studies that involve designing an ANN architecture, namely that the number of hyper-parameters to tune is large that we cannot cover all combinations to decide on the best configuration for the network. To minimize this threat, we used two different multilayer neural networks and trained them during the

two iterations of analysis. This provided us with a sanity check on whether the networks produce similar results. Another threat to internal validity is in the random selection of test cases. There is a chance that the extracted test executions contain one or more test that failed due to factors that do not pertain to functional deficiencies, but due to, for instance, an environment upgrade or machinery failure at execution time. Similarly, there is a chance that the extracted test executions may have failed as a result of defects in the test script code and not the base code. In order to minimize this threat, we collected data for multiple test cases, thus minimizing the probability of identifying test cases which are not representative.

The major threat to external validity for this study comes from the number of extracted tests that were used for training the classifiers. We only studied one company and one product and a limited number of test cases. This was a design choice as we wanted to understand the dynamics of test execution and be able to use statistical methods alongside the machine learning algorithms. However, we are aware that the generalization of the results for different types of systems require further investigations using tests and churns from different systems.

8 Recommendations

In this section, we provide our recommendations for practitioners who would like to utilize MeBoTS for early prediction of test case verdicts.

- The choice of using deep learning or tree-based models for solving this supervised ML problem does not lead to better prediction performance. For this reason, we suggest the use of Decision Trees, since they require less computational time and provide knowledge as to how the results of classification were derived.
- We suggest to only utilize code churns that are homogeneous and small in size prior to applying features extraction with BOW. Small code churns introduce less noise and therefore the quality of the predictions is higher. This can also save practitioners ample time for data curation.
- We recommend that practitioners only extract historical test executions that have failed due to reasons related to functional defects for training the ML model. This knowledge can be obtained from testers/developers who are familiar with the recurrent issues in the source code.

9 Conclusion and Future Work

This paper has presented a method (MeBoTS) for achieving early prediction of test case verdicts by training a machine learning model on historical test executions and code churns. We have evaluated the method using two data sets, one containing a variation of large and small churns, and a second containing only small churns. The results from training the models on small churns revealed a precision rate of 73% and a recall of 43.6%, suggesting that the application of the

method is promising, yet more investigation is required to validate the findings. Moreover, contrary to other existing methods that use statistical correlations for predicting test verdicts, the main advantage of MeBoTS is the ability to predict verdicts of new code changes as they emerge during development and before they get integrated into the main branch.

We believe that the results of this study open new directions for studies to investigate the effectiveness of MeBoTS on different types of systems using larger set of small churns with more test case executions. Finally, studies that investigate the impact of using different feature extraction techniques, such as word embedding are encouraged to identify any changes in the overall performance of MeBoTS.

Acknowledgment

This research has been partially carried out in the Software Centre, University of Gothenburg, and Ericsson AB.

References

1. Awad, M., Khanna, R.: Efficient learning machines: theories, concepts, and applications for engineers and system designers. Apress (2017)
2. Azizi, M., Do, H.: A collaborative filtering recommender system for test case prioritization in web applications. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1560–1567. SAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167132.3167299, http://doi.acm.org/10.1145/3167132.3167299
3. Beningo, J.: Using the static keyword in c. https://community.arm.com/developer/ip-products/system/b/embedded-blog/posts/using-the-static-keyword-in-c (2014)
4. Busjaeger, B., Xie, T.: Learning for test prioritization: an industrial case study. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 975–980. ACM (2016)
5. Çalikli, G., Staron, M., Meding, W.: Measure early and decide fast: transforming quality management and measurement to continuous deployment. In: Proceedings of the 2018 International Conference on Software and System Process. pp. 51–60. ACM (2018)
6. Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., Xie, B.: Learning to prioritize test programs for compiler testing. In: Proceedings of the 39th International Conference on Software Engineering. pp. 700–711. IEEE Press (2017)
7. Chollet, F., et al.: Keras. https://keras.io (2015)
8. Chollet, F.: Deep Learning with Python. Manning (2017)
9. Durelli, V.H., Durelli, R.S., Borges, S.S., Endo, A.T., Eler, M.M., Dias, D.R., Guimarães, M.P.: Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability (2019)
10. Gondra, I.: Applying machine learning to software fault-proneness prediction. Journal of Systems and Software **81**(2), 186–195 (2008)

11. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Transactions on Software Engineering **26**(7), 653–661 (July 2000). https://doi.org/10.1109/32.859533
12. Gron, A.: Hands-On Machine Learning with Scikit-Learn and TensorFlow. Oreilly (2015)
13. Knauss, E., Staron, M., Meding, W., Söder, O., Nilsson, A., Castell, M.: Supporting continuous integration by code-churn based test selection. In: Proceedings of the Second International Workshop on Rapid Continuous Software Engineering. pp. 19–25. IEEE Press (2015)
14. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering. pp. 284–292. ACM (2005)
15. Noor, T.B., Hemmati, H.: A similarity-based approach for test case prioritization using historical failure data. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). pp. 58–68. IEEE (2015)
16. Noor, T.B., Hemmati, H.: Studying test case failure prediction for test case prioritization. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 2–11. ACM (2017)
17. Ochodek, M., Staron, M., Bargowski, D., Meding, W., Hebig, R.: Using machine learning to design a flexible loc counter. In: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on. pp. 14–20. IEEE (2017)
18. Palma, F., Abdou, T., Bener, A., Maidens, J., Liu, S.: An improvement to test case failure prediction in the context of test case prioritization. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 80–89. PROMISE'18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3273934.3273944, http://doi.acm.org/10.1145/3273934.3273944
19. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
20. Saxena, R.: Introduction to decision tree algorithm (2017), https://dataaspirant.com/2017/01/30/how-decision-tree-algorithm-works/
21. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering **37**(6), 772–787 (Nov 2011). https://doi.org/10.1109/TSE.2010.81
22. Spieker, H., Gotlieb, A., Marijan, D., Mossige, M.: Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 12–22. ACM (2017)
23. Ståhl, D., Bosch, J.: Experienced benefits of continuous integration in industry software product development: A case study. In: The 12th IASTED International Conference on Software Engineering,(Innsbruck, Austria, 2013). pp. 736–743 (2013)
24. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)