Test Case Selection with Incremental ML

Markus Mulkahainen,
Kari Systä[0000−0001−7371−0773], and Hannu-Matti Järvinen[0000−0003−0047−2051]

Tampere University
Unit of Computing Sciences
markus.mulkahainen@hotmail.com, kari.syst@tuni.fi,
hannu-matti.jarvinen@tuni.fi

Abstract. *Context:* Software projects applying continuous integration should run the tests very frequently, but often the number of test is huge and their execution takes a long time. This delays the feedback to the developer. *Objective:* Study if heuristic and especially incremental machine learning can help in finding an optimal test set that still finds the errors. *Method:* Several methods for reducing the tests were tested. Each method was applied to the example software its commit history, and the performance of the methods were compared. *Results:* The test set size can be radically reduced with automatic approaches. Furthermore, it was found that the incremental machine learning based test selection techniques eventually perform equally well or better than the best heuristic.

Keywords: test case selection · continuous integration

1 Introduction

This paper documents research that has been inspired by a practical need in a company. The research was originally conducted for a master thesis [13], and this paper summarizes the research for an international audience.

Continuous Integration (CI) is a software engineering practice that automates software integration and encourages developers to commit more often. Testing in CI should be automatic [18] and extensive. Every change in the software should be validated in the context of the whole software to maintain quality.

Test suites tend to grow large during the development and lead to long-lasting test suites. In the context of the case company, Space Systems Finland[1], the problem culminates in validation tests, which exercise multiple end-to-end tests, and even the execution of a single test can take a long time. Test case selection and prioritization can be used in such situations to reduce the time for developer feedback.

The case context was satellite instrument control software for the Meteosat Third Generation Sounder (MTG-S) satellites. The instrument, namely Sentinel-4/UVN, is a high-resolution spectrometer and will be used to monitor air quality parameters over Europe, and the case of this paper was a command and control

[1] http://www.ssf.fi/

software for the Sentinel-4/UVN instrument. The software and its validation tests are programmed in Ada.

Test case selection (TCS) selects a subset of the test suite for repeated testing. Ideally, the selected subset of the tests finds the same number of faults with lesser effort compared with the original test suite. However, the reduced test set may not include all fault-revealing test cases, but TCS discards some of them. Finding the optimal subset in TCS is a non-obvious task, especially in large systems with a large number of test cases. Machine learning (ML) is an interesting approach for this. Because each project is new and different from the others, incremental ML may be more suitable than approaches with a separate learning phase.

The goal of this research was to evaluate different TCS techniques in order to speed up the testing and facilitate continuous integration. Specifically, the research question was *How effective is incremental machine learning for test case selection and how does it compare to heuristics-based methods?*

Machine learning has become more popular in the domains of TCS [2,8,9,17]. These studies have shown promising results in using ML for TCS.

The paper is organized as follows. Section 2 introduces the background and Section 3 summarizes the related research. The experiment is described in Section 4 and the results are presented in Section 5. The results are discussed in Section 6. Finally, Section 7 gives the conclusions.

2 Background

2.1 Dependency coverage

All heuristic methods used in this study rely on dependency coverage. A modification can break functionality in the modified module, but also in the modules that depend on the modified module. This can be troublesome as faults can show up in surprising components or sub-systems [19].

A rather safe way to reduce the number of tests, but to still reveal faults in the dependent modules, is to recognize tests that target the modified modules and their dependents and execute them. The aim is to recognize every test case that could be transitively affected by a change. A test case is affected if the test or any of its dependencies is modified [19].

2.2 Machine Learning

The gist of ML is a piece of software, that is capable of improving its performance in a set of tasks, based on experience [12]. In this paper, incremental machine learning means the application of machine learning algorithms so that the model continuously learns from the new input while being used.

Before machine learning techniques can be applied to test cases, the problems need to be coded to the ML algorithm, i.e., they have to be represented as *feature vectors*. The values used in feature vectors are described below.

Seven features are used to represent test cases as feature vectors, namely *statement coverage*, *modification coverage*, *similarity score*, *duration*, *failure rate*,

latest pass and *history length*. The features are somewhat similar to features used in a related study [2]. The following describes how these features are attained.

Statement coverage is a floating-point number with a closed interval between 0 and 1. It is the total percentage of statements (or lines) covered by a test case. The statement coverage is updated every time the test case is executed and remains unchanged until the test case is re-executed. Line coverage is used instead of statement coverage, because that information was available.

Modification coverage is similar to statement coverage, but the coverage is calculated over the modified lines of a commit instead of all software code lines. More closely, the modification coverage is calculated as $\frac{|C_t \cap M|}{|M|}$ where C_t is the coverage of the test case and M is the set of modified lines. The test coverage, C_t, is produced by *gcov*-tool. *Git diff* command was used to find M. The full coverage of previous test executions is needed to deduce coverage over the modified lines, e.g. $|C_t \cap M|$. Therefore, the full coverage produced by *gcov*-tool needs to be persisted and transferred between any two commits.

Similarity score is a similarity measure between a code change (git commit) and a test case, where a higher value means that certain keywords occur more often in both texts suggesting a higher similarity. The similarity score is calculated with TF-IDF transformation and cosine similarity [10]. A similarity score is a floating-point number with a closed interval between -1 and 1.

Duration is the test execution time of the test case in seconds. Duration is updated every time the test case is executed.

Failure rate is a floating-point number in a closed range between 0 and 1, and it is calculated by $\frac{T_f}{T_p+T_f}$ where T_f is the total number of failures and T_p is the total number of passes of a single test case. Every test execution updates this value because either T_f or T_p is incremented.

Latest pass is a left-closed and right-unbounded discrete value from 1 to infinity. It is the number of failing test executions that precedes a passing test execution. Every failing test execution increments this value by one and passing execution resets the value back to 1. The initial value 1 is set because of the assumption that initially, every test case is passing.

History length is a left-closed and right-unbounded discrete value from 1 to infinity which denotes the number of executions for the test case. The initial value is 1 and each test execution increments the value by one.

2.3 Test case selection

Test case selection techniques are a group of regression testing techniques where a subset of the test suite is selected for execution. It reduces the execution time, but at the same time risks neglecting *fault-revealing* test cases.

Test case selection techniques have been evaluated in the literature using metrics such as test suite reduction (TSR) and reduction in fault detection effectiveness [4] [16] [3]. Test suite reduction is expressed as [16]

$$\mathrm{TSR} = 1 - \frac{|T'|}{|T|} \tag{1}$$

where T is the original test suite and T' is the reduced test suite. Reduction in fault detection effectiveness (RFDE) is given as [16]

$$\text{RFDE} = 1 - \frac{|F_{T'}|}{|F_T|} \tag{2}$$

where F_T is the set of faults found by the original test suite T and $F_{T'}$ is the set of faults found by the reduced test suite T'. The test case selection techniques should maximize the test suite reduction and minimize RFDE.

In this study, F_T is unknown, e.g. the number of actual faults in the system is not known. The failing test cases are known, but a failing test does not always reveal one unique fault. One failing test can reveal any number of actual faults. Because F_T is not known, RFDE cannot be used to measure the performance of TCS techniques. Instead, it is assumed that finding the failing test cases helps to find the actual faults in the system. Therefore, TCS techniques are used to find the failing test cases, f_T, from T. The reduced test suite is not wanted to contain anything else but the failing test cases. In addition to TSR, the objective becomes to maximise the proportion of test failures in the reduced test suite T':

$$\frac{|f_{T'}|}{|f_T|} \tag{3}$$

where $f_{T'}$ is the set of failing tests in T', and f_T is the set of failing tests in the original test suite T. The expression 3 can be rewritten with *true positives* (T_p), *false positives* (F_p), *true negatives* (T_n) and *false negatives* (F_n) by using Knauss et al. [7] descriptions:

- T_p: Test cases that were correctly selected to the reduced test suite (predicted to fail and failed).
- F_p: Test cases that were incorrectly selected to the reduced test suite (predicted to fail but passed).
- T_n: Test cases that were correctly omitted from the reduced test suite (predicted to pass and passed).
- F_n: Test cases that were incorrectly omitted from the reduced test suite (predicted to pass but failed).

The expression 3 can be rewritten with T_p, F_p, T_n and F_n:

$$\frac{|f_{T'}|}{|f_T|} = \frac{|\text{T}_\text{p}|}{|\text{T}_\text{p}| + |\text{F}_\text{n}|} \tag{4}$$

which is the same as *recall* in information retrieval theory [5] [15]. The same can be done for test suite reduction, and rewrite it with T_p, T_n, F_n and F_p:

$$\begin{aligned} \text{TSR} &= 1 - \frac{|T'|}{|T|} = \frac{|T| - |T'|}{|T|} \\ &= \frac{(|\text{T}_\text{n}| + |\text{F}_\text{n}| + |\text{F}_\text{p}| + |\text{T}_\text{p}|) - (|\text{T}_\text{p}| + |\text{F}_\text{p}|)}{|\text{T}_\text{n}| + \text{F}_\text{n} + |\text{F}_\text{p}| + |\text{T}_\text{p}|} \\ &= \frac{|\text{T}_\text{n}| + |\text{F}_\text{n}|}{|\text{T}_\text{n}| + |\text{F}_\text{n}| + |\text{F}_\text{p}| + |\text{T}_\text{p}|} \end{aligned} \tag{5}$$

There are now two conflicting performance scores for TCS techniques: test suite reduction and recall. The goal is to find a TCS technique that maximises both of these scores. It is not trivial, because increasing one potentially decreases the other and vice versa.

Thus, the Matthews correlation coefficient (MCC) is introduced. MCC was found to be a good surrogate for a combination of test suite reduction and recall. The MCC score is a single value and gives us a more robust way to compare the performances of TCS techniques. B.W. Matthews introduced the MCC-score in 1975 [11] and defined it as:

$$MCC = \frac{|\mathrm{T_p}| \times |\mathrm{T_n}| - |\mathrm{F_p}| \times |\mathrm{F_n}|}{\sqrt{(|\mathrm{T_p}| + |\mathrm{F_p}|)(|\mathrm{T_p}| + |\mathrm{F_n}|)(|\mathrm{T_n}| + |\mathrm{F_p}|)(|\mathrm{T_n}| + |\mathrm{F_n}|)}} \tag{6}$$

The highest possible MCC score is 1. It is achieved when $\mathrm{F_p} = 0$, $\mathrm{F_n} = 0$, $\mathrm{T_p} \neq 0$ and $\mathrm{T_n} \neq 0$. In such a case, there are no incorrect predictions. Selecting only the failing predictions, the "perfect selection" is got. The perfect selection never fully satisfies test suite reduction, e.g. TSR $\neq 1$, but always results in the maximum recall value of 1. In other words, MCC = 1 evaluates to the highest possible test suite reduction for a recall of 1.

2.4 Evaluated algorithms for TCS

In the case study, there were three heuristic methods based on data coverage; the fourth one is a random method.

Random. In the random technique, both the number of tests and the tests themselves are selected randomly. This means that from $|T|$ test cases n random tests are selected. Hence, n (the size of the selected test suite) can have any value between 0 and $|T|$.

Coverage. The coverage technique selects every test case that covers a modified statement as described in Section 2.1. The size of the selected test suite varies between 0 to $|T|$ as in the case of the Random technique.

Coverage(H). This technique includes all the cases of the Coverage technique and tests that have failed in the previous iteration. Naturally, the size of the test suite is between 0 to $|T|$.

Coverage(PH). In this technique, the first step is to select the tests in similarly to Coverage(H). Furthermore, if the selection size is greater than 2% of $|T|$, the selected tests are prioritized and n top test cases are selected until the 2% limit is reached. The prioritization step calculates the average of test history and coverage, sorts the test cases descending and selects n top test cases from the sorted list. In this technique, the test suite size falls between 0 and $0.02 \times |T|$.

The machine learning techniques apply binary classification over the test case samples and categorize the samples into bins of passing and failing. The selected techniques, except the unlimited version of *RandomForest(U)*, guarantee 98% test suite reduction.

RandomForest Select every test case that is predicted failing using random forest classifier from *scikit-learn* toolkit [14]. The random forest implementation

follows Breiman's implementation [1]. Furthermore, if selection size is greater than 2% or less than 2, prioritize the test suite T using class probabilities and select 2% of the most promising tests. This means that the size of the test suite is between 2 and $0.02 \times |T|$.

RandomForest(U) As above, but the test suite size is not limited. Hence, prioritisation is not needed and the size of the test suite is between 2 and $|T|$.

LogReg Select every test case that is predicted failing using logistic regression classifier from *scikit-learn*. If the selection size is greater than 2% or less than 2, use the same prioritisation method as in *RandomForest*. This leads to the test suite size of 2 to $0.02 \times |T|$.

XGBoost Select every test case that is predicted failing using gradient boosting technique (XGBClassifier) from *xgboost*-library. Also in this case, if the selection size is greater than 2% or less than 2, prioritisation is done as in the case of *RandomForest* resulting in the test suite size between 2 and $0.02 \times |T|$.

3 Related work

Spieker et al. [17] used reinforced learning and multi-layer perceptron to predict failing test cases based on test history. They actualized both test case selection and prioritization in test suites. The idea was to 1) prioritize the test suite T, and 2) repeat selecting the topmost test from T as long as the summed duration of the selected tests goes under a time threshold M. They used the normalized average percentage of faults detected (NAPFD) to measure the performance of their technique and concluded that approximately 60 CI cycles are needed to perform equally or better than the reference techniques. The reference techniques were a random technique, which ordered test cases randomly, a sorting technique, which ordered recently failed test cases with higher priority, and a weighing technique, which ordered test cases by a weighted sum of the test features. Spieker et al. were the first to apply reinforcement learning in TCS.

Di Nardo et al. [11] applied TCS in an industrial system with real regression faults. They measured reductions in test suite sizes and fault detection effectiveness with their coverage-based TCS techniques. They were barely able to reduce test suite sizes at all. The maximum reduction was 2%. Because of the small reductions, fault detection was not compromised.Di Nardo et al. discussed, that the small reductions in test suite sizes were likely due to modifications to the core components of the software. Such parts are covered by a multitude of test cases. Additionally, Di Nardo et al. examined only four different software versions, and the modifications between versions were arguably large.

Beszédes et al. [6] used priority-based TCS to reduce test suite size in the WebKit web browser engine. In their initial experiments, they selected every test case that covered the modified procedures in the software, or that had failed previously. Using this initial selection, they witnessed a test suite reduction of 79.43% with 95.08% recall on average. In their study, Beszédes et al. used the term "inclusiveness" instead of a recall, but both measures are the same.

When Beszédes et al. applied their selection technique in an actual live system they witnessed a test suite reduction of 51% with 75.38% recall on average. Beszédes et al. extended their selection technique with an extra prioritization step. The prioritization was based on coverage information. With this extra step, Beszédes et al. were able to further reduce the selection size. With this technique, they showed a test suite reduction of over 90% with half the recall compared with the non-prioritized test suite. Thus, the recall was interpreted to be approximately 38%. Comparing this result with the result by Busjaeger and Xie [10], the ML-based TCS technique seems to have superior performance.

Harrold et al. [19] experienced fluctuating test suite reductions with their code-based regression-test-selection technique. Their TCS technique relied on code coverage information. Harrold et al. recorded test suite reductions from 0% to almost 100%. They discussed, that the large reductions were due to small modifications in the software, where only a few methods covered by a few tests were changed. Harrold et al. did not analyze thoroughly the reasons behind the small reductions but mentioned that the location of a change can affect test suite reduction. As Harrold et al. applied their technique over four different software with less than eleven software versions, it is possible that the modifications between two consecutive versions were still quite large. Applying TCS in such versions can bring no reduction in test suite size.

Gligoric et al. [17] used dynamic dependency tracking from tests to files to reduce the number of tests. Their tool, "Ekstazi", can track any changes in files that are dependent on the tests, and execute only part of the test suite that is relevant for a set of file changes. The tool is capable of tracking source code files, but also configuration files. The tool monitors the execution of tests running on JVM and collects the accessed files using bytecode instrumentation and listening to all standard Java library methods that might open a file. After the collection of the dependent files is done, the tool can select a subset of tests to be executed for any change made in the dependent files. Gligoric et al. report, that their tool is capable to reduce end-to-end testing time by 32%.

Yoo et al. [45] used dependency coverage among other features to select and prioritize tests. The optimization technique by Yoo et al. balanced three competing objectives: dependency coverage maximization, historical fault detection maximization and execution time minimization. Yoo et al. reported an average test suite reduction of 68% with their technique.

4 The Experiment

4.1 Data Collection

The research is based on the version control history of an existing project (528 tests and 87 commits – for further details see [13]). The first phase collected data about tests in each commit in the version control history. An essential part of this data collection was re-executing tests for each version of the software. Then, different test case selection algorithms were applied.

Because data was collected by executing the tests, it took a long time. To reduce the time, handling of source modifying commits and test modifying commits were separated. When a commit modifies only *test/* directory and not *src/* directory at all, a transitive dependency selection includes only test cases that are transitively affected by the modification. This reduced the number of tests to execute. To further optimize the data collection, consecutive instances of test/* modifying commits were merged. This was done as a preprocessing step before running the data collection algorithm. Every commit that had no source or test modifications was also removed since they had no effect on the functionality of the software. The data collection algorithm used is the following:

1. Checkout newest commit
2. Repeat:
   (a) If the current commit has src/* modifications:
      i. Execute test suite
   (b) Else if the current commit has test/* modifications:
      i. Find modified tests through transitive dependency selection (see 2.1)
      ii. Execute modified tests
   (c) Save executed test verdicts, coverage and durations
   (d) Checkout previous commit

The output of this algorithm is an ordered set of tuples $D = \{commit,\ tests\}$, where *commit* is a commit's checksum (identification in Git) and *tests* is a set of tuples $\{verdict,\ coverage,\ duration\}$. *Verdict* is the output: pass or fail, *coverage* is the full gcov-coverage for the test, and *duration* is the length in seconds.

Step 2ai, test suite execution, lasted about 17 hours. The algorithm was continuously being executed for approximately two months for the preprocessed version control history. 87 commits ended up in the dataset, where 45 commits had only source code modifications, 17 commits had both test and source code modifications and 25 commits had only test modifications. Unfortunately, the test suite contained many non-deterministic test cases due to differences in the test environments. Those tests were removed from the dataset As a result, a portion of the commits ended up having no faults. These commits were not removed from the dataset.

The characteristics of the collected dataset are shown in Table 1. Note, that the build or execution failures are test cases that had passed at least once before. Every test case that was recently added and had build or execution failures were removed because it was impossible to gather coverage information for them. As soon as the removed tests passed again in the following commits, they were added back to the test suite. The oldest commit in the dataset did not luckily contain any failing tests after the non-deterministic tests were removed.

Surprisingly, many of the test cases failed because of build or execution errors. The reason behind this was not thoroughly studied, but it could possibly relate to differences between the test environments used in this research and the real one. It is also possible that the developers were aware of these build failures all along, and they had no intention to fix them.

More details about the data and its collection can be found in the original thesis [13].

Table 1. Charasteristics of the data used in this research.

Modifications	**Source code**	**Source and test code**	**Test code**
Commits	45	17	25
Commits with at least one failing test case	36	14	11
Failing tests	142	124	119
Normal failures	32	66	39
Build or execution failures	110	58	80
Passing tests	22590	8513	7109

4.2 Test case selection

All test case selection algorithms were applied by iterating through the collected data. The algorithm below presents the procedure. The algorithm was run for every test case selection technique t and for every tuple $d \in D$:

1. If *d.commit* has test/* modifications:
   (a) Let T_{tmod} be the tests selected with transitive dependency selection
2. If *d.commit* has src/* modifications:
   (a) Let T_{smod} be the tests selected with t according to current knowledge C
3. Let $T' = T_{tmod} \cup T_{smod}$
4. Simulate the execution of T'
5. Update current knowledge C

C represents the current knowledge about the test cases. This includes the coverage, duration, and test verdict histories (history of passes and fails) for every test case. In the first commit, this information is not available, and therefore one commit is needed to initialize the test case selection techniques. During the first commit, the initial coverage, duration, and test verdicts were collected.

In the first step, a transitive dependency selection to the *d.tests* is applied, if *d.commit* type is "test" or "source&test". In the second step, the TCS technique t to *d.tests* using the current knowledge C is applied. The selected tests were saved in T_{smod}. In the third step, the transitively affected tests T_{tmod} and the selected tests T_{smod} are combined. T_{tmod} is empty, if *d.commit* type is "source". T_{smod} is empty, if *d.commit* type is "test", respectively. If *d.commit* type is "source&test", both T_{tmod} and T_{smod} can contain test cases, but not the same test cases. In the fourth step, it is not necessary to execute the reduced test suite T', because it was already done during the data collection phase. Instead, the existing information of T' was used, and current knowledge was updated about test histories.

The last step 5 is rather complex. The selection T_{smod} is turned into feature vectors, but only if *d.commit* type is "source". Using the coverage, duration and verdict the feature vector {*statement coverage, modification coverage, similarity score, duration, failure rate, latest pass*} is created for every test case. This is done for all tests in T_{smod}, and they are saved for the next iteration d_{i+1}. This idea is

applied for every source commit, and eventually, the training data accumulates and grows larger. The training dataset is a set of $\{T_{smod_1}, ..., T_{smod_{n-1}}, T_{smod_n}\}$, where n is an index of a source commit. During every iteration, the machine learning model is re-trained with this training dataset.

It is also necessary to calculate the MCC metric, recall and test suite reduction between the steps 4 and 5 if the *d.commit* is a source commit and the commit has at least one failing test case. If there are no failing tests, the output of MCC is undefined, recall is zero, and test suite reduction would be the only indicator worth measuring. Therefore, measuring performances is skipped when the commit has no failing tests. In addition to non-faulty commits, performances in "test&source" commits or "test" commits were not measured either.

5 Results

Test case selection techniques were compared using Matthews correlation coefficient (MCC) values. The boxplot in Figure 1 shows MCC-scores for each technique over 35 commits. The green triangle is the mean and the orange line is the median. The box presents values from lower to upper quartile. The whiskers display the range of the data, and the dots are outliers. *Coverage(PH)* technique has the highest median and mean MCC-score, and *Random* technique the lowest.

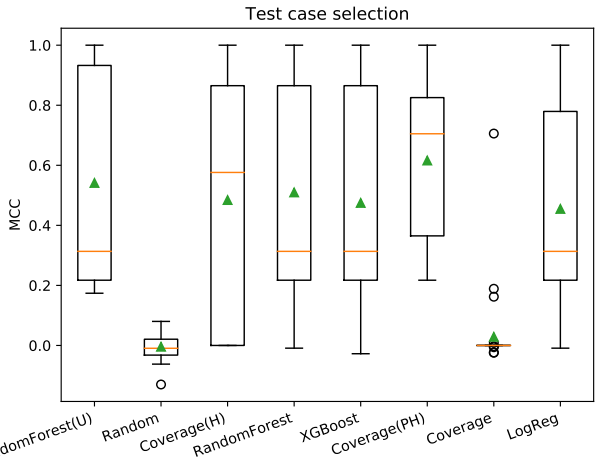

Fig. 1. MCC of each test case selection method over 35 commits.

To examine the significance of the techniques, a Kruskal-Wallis test was done for the MCC scores across 35 source-modifying commits with a failing test. The result showed an H-statistic of 117.9 and the p-value of $2.07 \cdot 10^{-22}$ allowing the rejection of the null hypothesis (medians of the groups are equal). To find which of the groups were different, a pairwise posthoc test was done using Dunn's test with Bonferroni adjustment. The pairwise comparison is shown in Figure 2.

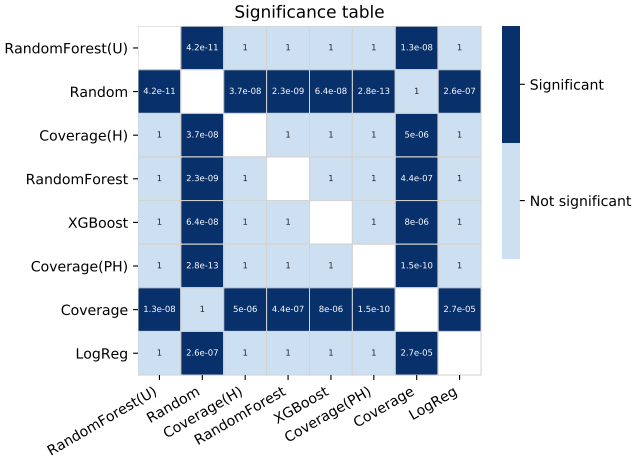

Fig. 2. Pairwise significance analysis using Dunn's test with Bonferroni adjustment. Any value below 0.05 indicates a significant difference in MCC.

Figure 3 shows MCC-trend for each heuristic (top) and each machine learning technique (bottom) across 35 source modifying commits. All machine learning techniques have fairly low MCC values during the first 19 commits. Towards the end, the machine learning techniques improve.

6 Discussion

6.1 Test case selection

This test case selection case study, compared the performance of eight test case selection techniques. Four of the techniques were based on heuristics, and the rest four were based on machine learning. For each technique three different performance indicators were measured, namely test suite reduction, recall, and Matthews correlation coefficient. The MCC-score was used to differentiate the well and poorly performing techniques in a form of significance analysis using Dunn's test with Bonferroni adjustment (Figure 2).

Heuristics The significance analysis revealed, that *Coverage* and *Random* techniques were outweighed by other techniques. Interestingly, *Coverage* and *Random* techniques did not have a statistical difference in their performances.

The coverage based test case selection (*Coverage*) achieved test suite reduction of 64.7% while having a recall of 39.5% on average. The MCC scores had no significant differences from *Random* technique.

The *Coverage(H)* technique was able to resolve part of the issues of *Coverage*, providing significantly better results. It had an additional way to predict a test failure, namely the *latest pass*. It selected every test case that either covered a modification or failed in the previous commit.

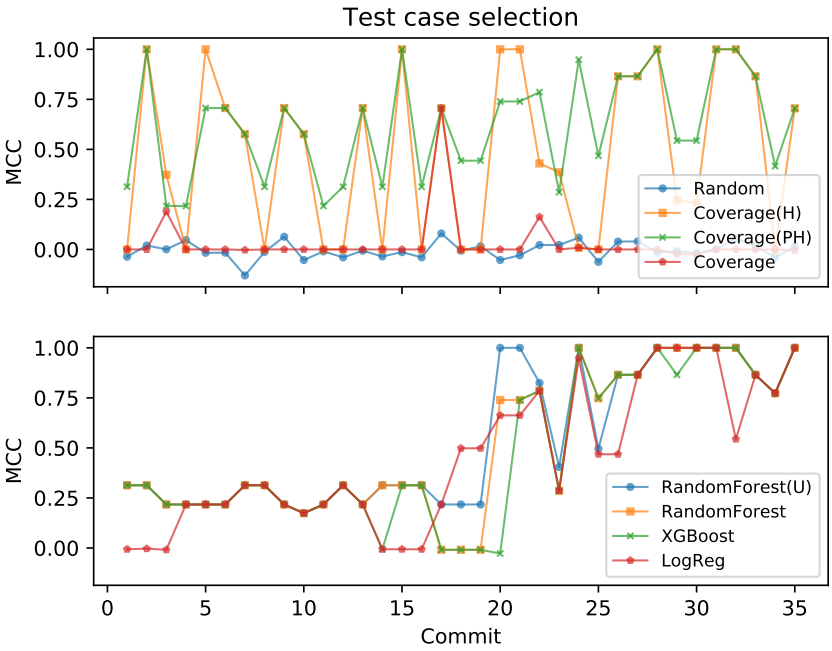

Fig. 3. MCC per method and commit. Trends for heuristics are shown in the top plot, and for machine learning techniques in the bottom plot.

The *Coverage(PH)* technique was the most promising technique among the heuristics of this paper. It used priority-based test case selection over modification coverage and test history. It selected every test case that covered a change or failed in the previous commit. If the selection size was still too large, it reduced the selection by prioritizing the selected tests using failure rate, latest pass, and modification coverage.

Machine learning The assumption was that the performance of the machine learning models gradually increases as tests are being executed and new labeled data samples are accumulated in the training dataset. It was interesting to know, whether the machine learning techniques eventually reach the same performance as the heuristics, and if so, then how long time does it take to reach a similar performance? To investigate this, Figure 3 shows the performance of each technique over time. Indeed, every machine learning technique shows a positive trend for the MCC scores, where the techniques performed better in the end than in the beginning and towards the end they performed equally or better than the heuristics. It looks like that at commit number 20 all techniques gained a positive boost, and they perform better than in commit 19.

During the first 19 commits, the machine learning techniques had fairly low MCC scores possibly due to the low amount of negative samples in the training

data. Between commits 20 and 35 however, the machine learning techniques seem to perform better. The Table 2 collects the recall and test suite reduction values of each technique between the commits 20 and 35.

Table 2. Average recall, test suite reduction and Matthews correlation coefficient for each test case selection technique.

	Commits 1-19			Commits 20-35		
Technique	**Recall**	**TSR**	**MCC**	**Recall**	**TSR**	**MCC**
Random	0.316	0.588	-0.010	0.489	0.528	0.003
Coverage	0.553	0.524	0.047	0.215	0.793	0.007
Coverage(H)	0.886	0.523	0.387	0.947	0.778	0.600
Coverage(PH)	0.781	0.988	0.515	0.790	0.986	0.736
LogReg	0.412	0.982	0.189	0.783	0.987	0.771
RandomForest	0.623	0.980	0.220	0.790	0.990	0.854
RandomForest(U)	0.702	0.980	0.255	0.871	0.983	0.881
XGBoost	0.570	0.980	0.203	0.755	0.990	0.798

Comparing the columns of commits 1-19 with columns of commits 20-35, the machine learning techniques had increased their recall but also improved test suite reduction a bit. *RandomForest(U)* technique outperformed *Coverage(PH)* in recall with a slightly lesser test suite reduction. The rest of the techniques also provided competitive results to *Coverage(PH). Coverage(H)* still remained the technique with the highest recall.

The best performing machine learning model was an unlimited random forest (*RandomForest(U)*), which achieved a test suite reduction of 98.2% and recall of 73.1% on average. Towards the end the recall was notably higher, rendering MCC score also higher. This is a promising result for incremental learning-based test case selection and shows that machine learning techniques have the capability to outperform heuristics in a relatively small number of commits.

Spieker et al. [17] used reinforcement learning to select and prioritize test cases, and their technique required 60 consecutive commits to perform equally or better than comparison techniques. The test case selection results in this study suggest, that approximately 20 source code modifying commits to provide similar results with the comparison techniques. The machine learning techniques provided similar or better MCC scores compared to the *Coverage(PH)* technique after 20 commits. This could indicate, that using a different model (e.g. random forest classifier instead of multilayer perception), accumulating training data and re-training the machine learning model in every iteration, and using more features in addition to test histories, such as coverage information and text similarity scores, can help to reach the saturation point faster. The results achieved in this study, are not outright comparable to the results of Spieker et al., because experimentation setups were different, the comparison methods were different and the used measures were different, namely NAPFD and MCC. Also, the results were not validated with other projects but the techniques were ap-

plied to a single software project only. Therefore, more investigation is required to compare results more reliably with Spieker et al. and this research.

Busjaeger and Xie [2] used supervised learning and pointwise ranking to prioritize test cases. They were able to select 3% of the topmost test cases and provide 75% recall. Such selection equals to 97% test suite reduction. The results of this study are approximately similar, but the results were achieved with less training data. The results in this study suggest, that even if initial training data does not exist, incremental learning can eventually achieve similar performance to supervised batch learning. The saturation point was at about 20^{th} commit, and then the performance was similar to [2].

In many cases, false positives and false negatives have different impacts. One disadvantage of the MCC score is that it values false positives and false negatives similarly, i.e. it is invariant to the changes in false positives and false negatives when their sum is constant. Small amount of false negatives and a greater amount of false positives is more beneficial than the contrary in test case selection. The MCC score could be biased to penalize false negatives more than false positives, but this is left to future research.

6.2 Threats to validity

There are many threats to validity. Firstly, all non-deterministic tests were deleted from the test suite before the experiments. This arguably distorts the results. However, the test history features, such as the latest pass and failure rate described in section 2.2 are the key features to explain even the non-deterministic test case failures.

A different test environment was used in the data collection (section 4.1) than in the actual project. These two test environments are similar, but they use a different amount of hardware simulation. This could have brought excessive discrepancies in test verdicts between the test environments.

Because of separating how code (src/*) and test (test/*) commits are handled, the experiment setup became complex. MCC, recall and test suite reduction were calculated for source modifying commits only and ignored the values for the test commit types. Using dependency coverage as a new machine learning feature could have fixed this issue.

Code-coverage-based test case selection is not able to trace every kind of change in the codebase. These are generally the non-instrumental parts of the code repository, such as meta- or configuration files, but also source code. For example, a global variable value change cannot be traced.

The coverage information produced by *gcov*-tool was not accurate when a statement contains line breaks. In such situations, the first line is only detected by gcov, and the rest of the lines are ignored. The software code contained statements that split into multiple lines. Therefore, the coverage-based selection techniques could have been affected.

Finally, the test case selection was applied to one software project only. This suggests that external validity can be affected. The plan included another project but there was not enough time.

7 Conclusions

Because CI aims to provide rapid feedback for the developers, slow testing can be harmful [6]. As software evolves, the test suites become large and at some point, they can no longer be executed in a short time. The aim of this research was to find ways to enhance or speed up testing in order to facilitate CI, and that test case selection techniques can be used to reduce the time required for testing. The incremental machine learning was found especially interesting for its capability to eventually outperform comparison heuristics.

Incremental machine learning was used to predict failing tests out of the test suite using information such as test history, code coverage, and modifications introduced in a commit. With these predictions, the system effectively selected a small number of test cases for execution when a new commit was made to the software repository. The incremental machine learning-based test case selection techniques eventually performed equally well or better than the best heuristic. Similar results have already been suggested by Spieker et al. [17], who used reinforcement learning and neural networks to select a subset of tests based on test history. Their technique required 60 consecutive CI cycles to perform equally well or better than the comparison techniques in NAPFD values. The research reported in this pare was based on the MCC score, and the ML techniques produced equal or better MCC scores than the best heuristic after 20 source code modifying commits. The research supports the results of Spieker et al. and brings in more evidence that when initial training data does not exist, machine learning can be applied incrementally to eventually produce as good or better results as comparison techniques. In addition to that, the results give a cautious hint that accumulating training data and re-training the models in every iteration, using more features such as code coverage and similarity score and using a different classifier, e.g. random forest, can make the models learn faster and predict failing tests correctly earlier.

Despite the positive results in favour of using machine learning in test case selection, the results need further verification. There was a single software project in the case study, and therefore external validity is risked. Secondly, the machine learning models were fully trained in every commit, which can become infeasible when the training data increases.

References

1. Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.
2. Benjamin Busjaeger and Tao Xie. Learning for test prioritization: An industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 975–980, New York, NY, USA, 2016. ACM.
3. Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015.

4. Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14 – 30, 2010.
5. Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861 – 874, 2006. ROC Analysis in Pattern Recognition.
6. Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.
7. E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell. Supporting continuous integration by code-churn based test selection. In *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, pages 19–25, May 2015.
8. R. Lachmann. Machine learning-driven test case prioritization approaches for black-box software testing. In *European Test and Telemetry Conference ettc2018*, pages 300–309, Jun 2018.
9. R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 361–368, Dec 2016.
10. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
11. B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975.
12. Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
13. Markus Mulkahainen. *Test case selection and prioritization in continuous integration environment*. Master Thesis, Tampere University, Faculty of Information Technology and Communication, Tampere, 2019.
14. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
15. David Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation. *Mach. Learn. Technol.*, 2, 01 2008.
16. G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43, Nov 1998.
17. Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 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*, ISSTA 2017, pages 12–22, New York, NY, USA, 2017. ACM.
18. M. Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82, May 2015.
19. Shin Yoo, Robert Nilsson, and Mark Harman. Faster fault finding at google using multi objective regression test optimisation. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, 2011.