Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection

Jiyang Zhang
jiyang.zhang@utexas.edu
University of Texas at Austin
Austin, TX, USA

Yu Liu
yuki.liu@utexas.edu
University of Texas at Austin
Austin, TX, USA

Milos Gligoric
gligoric@utexas.edu
University of Texas at Austin
Austin, TX, USA

Owolabi Legunsen
legunsen@cornell.edu
Cornell University
Ithaca, NY, USA

August Shi
august@utexas.edu
University of Texas at Austin
Austin, TX, USA

ABSTRACT

Regression testing—rerunning tests on each code version to detect newly-broken functionality—is important and widely practiced. But, regression testing is costly due to the large number of tests and the high frequency of code changes. Regression test selection (RTS) optimizes regression testing by only rerunning a subset of tests that can be affected by changes. Researchers showed that RTS based on program analysis can save substantial testing time for (medium-sized) open-source projects. Practitioners also showed that RTS based on machine learning (ML) works well on very large code repositories, e.g., in Facebook's monorepository. We combine analysis-based RTS and ML-based RTS by using the latter to choose a subset of tests selected by the former. We first train several novel ML models to learn the impact of code changes on test outcomes using a training dataset that we obtain via mutation analysis. Then, we evaluate the benefits of combining ML models with analysis-based RTS on 10 projects, compared with using each technique alone. Combining ML-based RTS with two analysis-based RTS techniques–Ekstazi and STARTS–selects 25.34% and 21.44% fewer tests, respectively.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Regression testing, regression test selection, machine learning, program analysis.

ACM Reference Format:
Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524481.3527230

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AST '22, May 17–18, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527230

1 INTRODUCTION

Regression testing [34, 44, 82] is the primary means by which most developers carry out software quality assurance. In regression testing, tests are rerun on each code version to check that the changes did not introduce new faults. The term, *RetestAll*, is often used to describe rerunning all tests in a project after every code change. RetestAll often incurs high costs in terms of the time that developers have to wait for test results and in terms of compute resources needed. These costs have been growing quadratically because of continued rapid growth in the number of tests that developers write and in the rate at which code is changed [29, 83].

To combat the rising costs of regression testing, researchers and practitioners proposed *regression test selection* (RTS) [10, 22, 31, 56, 59, 82]. The goal in RTS is to reduce the cost of RetestAll by only rerunning *affected tests*, i.e., those tests whose pass/fail behavior can be altered by code changes. The quality of an RTS technique is evaluated in terms of its safety, precision, and performance. An RTS technique is *safe* if it selects all affected tests, *precise* if it selects only affected tests, and *performant* if the time to select and rerun affected tests is (on average) less than the time for RetestAll.

Analysis-based RTS techniques use static and/or dynamic program analysis to find code elements (like statements, control-flow edges, methods, or classes) that each test depends on. Then, on a new code version, the techniques select tests that depend on modified elements. The insight is that affected tests must depend on changed code for their pass/fail outcomes to be altered. So, analysis-based RTS techniques compute affected tests as those that depend on changed code elements. Recent work showed analysis-based RTS techniques to be safe (for code changes), imprecise (i.e., they select many unnecessary tests), and performant [25, 42].

Large software companies, like Facebook, Google, and Microsoft, did not yet adopt analysis-based RTS due to analysis costs and these companies' frequent use of multiple programming languages in a repository [14, 50]. Facebook proposed and evaluated an RTS technique based on machine learning (ML) [50]; it learns from features like historical test failure rates, distance between tests and changed files, and files' change history to predict tests that are likely to fail after new code changes. *ML-based RTS* techniques may not select all affected tests, but they can substantially reduce testing costs [50]. Unfortunately, Facebook's ML-based RTS does not directly apply to (medium-sized) open-source projects because the approach is specific to the structure of the Facebook monorepository.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

Recent work evaluated an ML-based RTS technique, inspired by Facebook's ML-based RTS, on one open-source project [48]. But that ML-based RTS technique relies on features of a dependency graph that are only available after leveraging an analysis-based RTS technique. So, the quality of that ML-based RTS technique depends on internals of the analysis-based RTS technique that it leverages. *Our goals are to train ML-based RTS models independently of analysis-based RTS internals, and to study the viability of using such models* together *with analysis-based RTS to select fewer tests.*

In this paper, we present novel design and implementation of several lightweight ML-based RTS techniques that can be combined with analysis-based RTS and used to improve the precision of the latter on open-source projects. Specifically, *we apply ML-based RTS to further reduce the number of non-failing tests that are selected by analysis-based RTS.* Our envisioned usage scenario is that a developer adopts analysis-based RTS but wishes to further improve it (because the analysis-based RTS technique is imprecise) by only rerunning the subset of selected tests that are more likely to fail. In other words, our goal is to preserve safety and improve precision.

We obtain a training dataset for our ML-based RTS models via mutation analysis. We use mutation analysis [3, 36] to seed mutants that represent artificial faults into a program. We then train two types of ML models: one that learns based on tests that fail when run on mutants, and another that learns based on tests that get selected by an analysis-based RTS technique after analyzing the change between the mutant and the original code, thereby mimicking the selection result of the analysis-based RTS technique. We use 10 projects from prior regression testing research in our evaluation.

We make two main findings from our experiments. First, when combining ML-based RTS models with analysis-based RTS techniques, our best ML-based RTS model reduces the average selection rate of two analysis-based RTS techniques, Ekstazi [25] and STARTS [42], by 25.34% and 21.44%, respectively, while still selecting all failing tests. Second, combining ML-based RTS models with an analysis-based RTS technique results in reduced end-to-end testing time, suggesting that (1) the overhead of the ML-based RTS models are small compared with the analysis-based RTS techniques, and (2) substantial time savings result from running fewer tests than what the analysis-based RTS technique selects.

The main contributions of this paper include:

★ **Design and implementation.** We design and implement novel ML-based RTS models, based on mutation analysis and analysis-based RTS, that can work on medium-sized projects.

★ **Synergy.** We combine ML-based RTS with analysis-based RTS (Ekstazi and STARTS) to improve the precision of the latter.

★ **Comparison.** We perform an empirical evaluation using 10 open-source projects to compare our approaches with prior work on analysis-based RTS.

★ **Data.** Our evaluation dataset is publicly available.[1]

2 BACKGROUND

In this section, we briefly describe several aspects of analysis-based RTS and ML-based RTS that are used in later sections.

[1]https://github.com/EngineeringSoftware/predictiverts

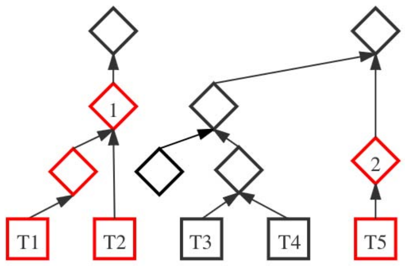

Figure 1: An example dependency graph: diamonds are code entities (e.g., modules/classes/methods) and squares are tests. An edge A → B means that A directly depends on B. In this figure, entities 1 and 2 changed (colored red). Tests T1, T2, and T5 transitively depend on entities 1 and 2 (also colored red), so they are selected.

2.1 Analysis-based RTS

An analysis-based RTS technique takes as input the old and new program versions along with the tests in those versions. The outputs are affected tests—the subset of tests that should be run on the new version. *Affected tests* are those whose pass/fail behavior can differ between running on the old and new program versions. Affected tests must therefore depend on changed code for their behavior to differ. An analysis-based RTS technique collects dependencies (parts of the program that each test depends on) and selects, as affected tests, those whose dependencies changed. Figure 1 illustrates a dependency graph of entities that tests depend on, where tests that transitively depend on changed entities are selected. An analysis-based RTS technique may collect dependencies at different *granularity*, such as method-level dependencies [55] or class-level dependencies [24, 43, 84].

As representative analysis-based RTS techniques, we use Ekstazi [25] and STARTS [42] that support Java projects. These two techniques share similarities in the granularity-level at which they track dependencies and in how they track changes. Both techniques track class-level dependencies, namely how test classes (and therefore all test methods in a test class) may depend on other classes in the program. In Java, each class is compiled into its own separate class file on disk, and both Ekstazi and STARTS check which classes change between versions by relying on the same algorithm for comparing checksums of these class files. A checksum represents the contents of each compiled class file, so a class file with a different checksum between versions means that the class changed between those versions. Ekstazi and STARTS differ in how they collect dependencies and how they compute affected tests[2].

2.1.1 *Computing affected tests in Ekstazi.*

When running tests in the old program version, Ekstazi dynamically (i.e., at runtime) collects, as dependencies, all other classes that each test uses. This mapping between tests and other classes is persisted to disk. On the new program version, Ekstazi computes affected tests as those

[2]In this paper, we also track tests at the granularity of test class. So, unless otherwise stated, when we say "test" we are referring to a test class.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

for which at least one dependency changed, plus any newly-added tests; Ekstazi then reruns these affected tests (test dependencies are collected again during this run).

2.1.2 Computing affected tests in STARTS. In the old program version, STARTS *statically* builds a class dependency graph of all classes in the program and persists to disk a file that maps each class to the set of all tests that transitively and reflexively reach that class in the dependency graph. Then, in the new program version, STARTS computes affected tests as all tests that are mapped to a class that changed along with any newly-added tests. Note that STARTS computes affected tests without running any tests.

2.2 ML-based RTS

An ML-based RTS technique involves training a model that learns from (past) data on what tests that were selected or had a different test outcome due to some change, and it aims to predict which tests to select after future changes. Note that an ML-based RTS model does not directly analyze the code changes and their relationship with the tests. Rather, ML-based RTS aims to extract features from the change that the model can use to classify a test as one that should be selected or not. Instead of selecting the tests that *could* be affected by a change, ML-based RTS aims to select only the tests that it predicts can *fail* after the change. As such, ML-based RTS potentially selects fewer tests than an analysis-based RTS technique (i.e., it ignores tests whose outcomes do not change despite depending on changed code). Also, a model that is trained on test outcomes bypasses the need to engineer sophisticated program analyses, which can be difficult to use on large projects and projects that use multiple programming languages [14].

As a representative ML-based RTS technique, consider the predictive RTS approach proposed by Machalica et al. [50]. They trained a model using a large dataset of historical code changes and test outcomes internal to Facebook. The goal of this model is to select only tests that fail when run after the change (but which passed on the old program version). Simply put, the goal of the Machalica et al. approach is to speed up regression testing and to not miss tests that would fail after code changes.

Machalica et al. trained a gradient-boosted decision-tree model with multiple features: code change-dependent and test target-dependent features, as well as cross features between them. Then, by extracting a feature-based abstraction of a new code change, the model estimates the probability that each test will fail when run on that change. Their ML-based RTS technique then selects the top $N\%$ of tests with the highest probability for failure, where the value of N can be adjusted by the user.

3 TECHNIQUE

We combine two analysis-based RTS techniques with ML-based RTS models to select fewer tests. We train two different types of ML-based RTS models: (1) a model that predicts whether a test would fail due to a change, and (2) a model that predicts whether a test could be affected by a change, i.e., the second model aims to learn the behavior of an analysis-based RTS technique. Given a change and a test suite, the ML-based RTS model extracts semantic features from the change and the tests. The model then assigns to each test a score that represents the relevance of that test to the change: the likelihood of the test failing after the change or the likelihood that an analysis-based RTS technique would select the test after that change. Users can tune the threshold score, so that only tests with scores above the threshold are selected.

Section 3.1 describes how we automatically construct the training datasets for the different models, Section 3.2 describes the features that we extract from changes to train the model, Section 3.3 describes how we create the ML-based RTS models using the features and labels from the training dataset, Section 3.4 describes the baseline models we compared to, and Section 3.5 describes how we combine ML-based RTS and analysis-based RTS.

3.1 Training Dataset from Mutants

Machine learning models require training data to learn a classification task, requiring a large amount of training data to achieve better performance. Machalica et al. constructed their training dataset using internal company logs, such as past continuous integration build logs that contain the changes and the test outcomes [50]. Unfortunately, a single open-source project usually does not have enough of such data on changes and test outcomes, even when using continuous integration services such as Travis CI [2, 6]. There generally are not enough build logs representing changes with true (non-flaky [5, 49, 66]) test failures to help with constructing a training dataset. Further, given our goal to compile and run tests in a project when using analysis-based RTS techniques, we cannot use data from very old historical versions that no longer compile.

Inspired by prior work [48], we utilize mutation testing [3, 36, 37] to create the training dataset automatically. Mutation testing leverages a number of mutation operators that describe rules for changing code under test, creating variants of the code called mutants. After creating mutants, tests are run on each mutant, and tests that fail when run on a mutant (but pass on the original code) are considered to have *killed* the mutant.

We construct a large, labeled training dataset by providing positive and negative labels to each mutant-test pair. We label the mutant-test pairs differently based on the type of ML-based RTS model we are training. (A) For the ML-based RTS model that predicts test failure after a change, we assign a positive label to a mutant-test pair if the test kills the mutant; otherwise, we assign a negative label. (B) For the ML-based RTS model that predicts what an analysis-based RTS technique would select after a change, we assign a positive label to a mutant-test pair if the corresponding analysis-based RTS technique would select that test after analyzing the change represented by that mutant; otherwise, we assign a negative label. The process is illustrated in the Figure 2 (left side), where we label based on whether the test fails on the mutant and whether Ekstazi selects the test after the program mutation. Plus (+) means that the test fails or is selected by Ekstazi while minus (-) has the opposite meaning.

Unfortunately, our initial training dataset is very imbalanced, because only a small percentage of tests would have a positive label (either fail or be selected by an analysis-based RTS technique). Following prior work on code search [28, 76], we address this issue by building a training dataset where the instances are triples of the form $\langle c, t^+, t^- \rangle$: for each code diff c (representing a change between original code and a mutant), there are positive-labeled tests t^+ and

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

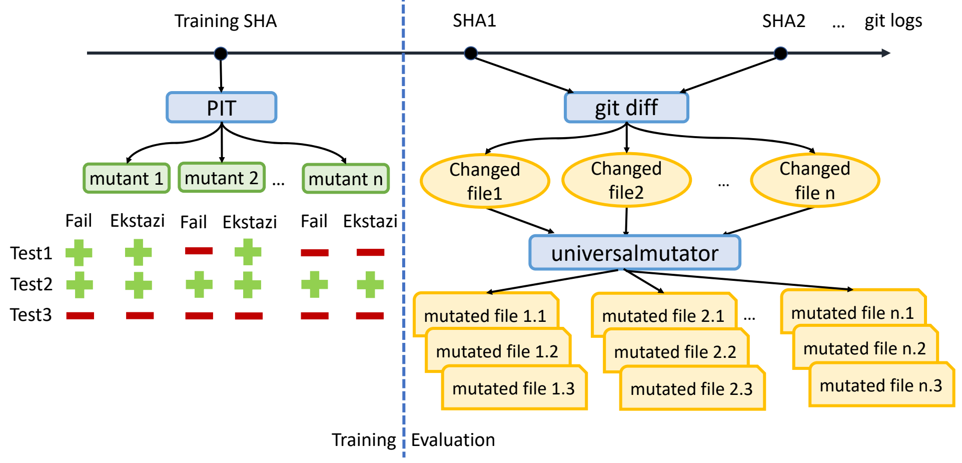

Figure 2: Overview of labeling the dataset.

negative-labeled tests t^-. For every positively labeled t^+ per code diff c, we randomly select 20 negatively labeled tests t^- for that c from the test suite. We then use this dataset of triples to train the models, thereby minimizing ranking loss (Section 3.3).

3.2 Pre-processing and Feature Extraction

We take two steps to pre-process every instance in our training dataset before using it to train a model. First, for all test classes (e.g., `StrTokenizerTest`), we split the test class name into tokens via *camelCase* and *snake_case* and convert to lower case (e.g., `str tokenizer test`). The resulting sequence of tokens represents a test class to be used as a feature. Next, for a code diff c, we pre-process and extract features from the change. We use three approaches for feature extraction: *Basic*, *Code*, and *ABS* (which stands for abstract). We explain these approaches next.

3.2.1 Feature Extraction Approach 1: Basic. We use the class name (i.e., file name) of the changed class file as a feature. Our intuition is that class names often give some indication of the class functionality and the relationship between tests and code under test. We tokenize changed class names in the same way as the test class name (e.g., `StrTokenizer` would be tokenized into `str tokenizer`).

3.2.2 Feature Extraction Approach 2: Code. We consider a file-level code diff that consists of added and modified lines in the commit (we do not consider deleted lines, because they are not in the next version). We assume that the size and contents of the code diff will help the model to select positively-labeled tests. Specifically, for each changed file, we extract the sequence of tokens on added and modified lines. We tokenize lines by space and punctuation, and then we tokenize further in the same manner as the test class names (e.g., `final StrTokenizer tok = getCSVClone();` would be tokenized into `final str tokenizer tok = get csv clone ( ) ;`). We concatenate the tokenized version of the changed class name with the tokens obtained from the added and modified lines.

3.2.3 Feature Extraction Approach 3: ABS. The actual code diff and changed lines may not generalize to unseen (or future) code changes, which can conceptually represent the same kind of change. To mitigate this issue, we define rules that map parts of the changed lines of code to general operators, namely the mutation operators that we used to create the training dataset. Table 1 shows the rules we use. Specifically, given a code diff, we go through each line and check if it matches the defined patterns. For example, if `return true;` is among the added lines for `StrTokenizer.java`, then we will use BooleanReturnValsMutator as the feature for the code diff. After matching, we abstract the code diff as a sequence of mutation operators. We concatenate the tokenized version of the changed class name with the sequence of mapped operators (e.g., `str tokenizer BooleanReturnValsMutator`).

Table 1: Rules to Abstract the Code Diff.

Mutation operator	Matching pattern
BooleanReturnValsMutator	“return true" “return false"
NullReturnValsMutator	“return”
ConditionalBoundaryMutator	“<” “<=” “>” “>=”
NegateConditionalMutator	“==” “! =” “!”
MathMutator	“+” “-” “*” “/” “%” “&” “|” “^” “<<” “>>”
IncrementsMutator	“++” “+=” “−” “-=”
EmptyObjectReturnValsMutator	“/∗” or natural language comments
VoidMethodCallMutator	otherwise

3.3 Models

We train an ML-based RTS model that, given extracted features from a code change, gives a score to each test based on the likelihood of that test being selected. More specifically, given a pair $\langle c, t \rangle$ consisting of code diff c and test t, the ML-based RTS model gives a score between 0 and 1 that measures the probability that test t should be selected based on c. Tests with scores that are greater than a user-defined threshold N are selected.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

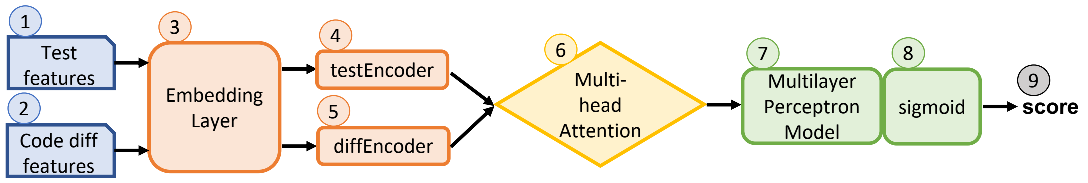

Figure 3: Architecture overview of our ML-based RTS models.

Figure 3 shows an architectural overview of our ML-based RTS modeling. We first embed the extracted features (①②) described in Section 3.2 into sequences of vectors through an embedding layer (③) and then use two machine learning based encoders to encode the embedded code diff features (⑤) and test features (④) separately. The code diff and test encodings are then combined using multi-head attention (⑥) to learn a deep representation that captures the relation between them. Finally, we apply a vanilla multilayer perceptron (MLP) model [35] (⑦) with *sigmoid* activation function (⑧) [46, 47] to predict the likelihood of the test being positively related to the diff (⑨). We next describe each part of the model in more detail.

Encoders. We use two encoders. The first encoder, `diffEncoder`, encodes features from the code diff. The other encoder, `testEncoder`, encodes features from the tests. Both encoders are instances of a bidirectional Gated Recurrent Unit (GRU) network [18], a variant of a recurrent neural network [7]. Specifically, considering a sequence (of length k) of vectors representing the embedded code diff features, $c = \{c_1, c_2, ..., c_k\}$, the `diffEncoder` encodes that sequence and outputs the sequence of encoded vector representations, h^c:

$$h^c = \text{GRU}(c)$$
$$h^c = \{h^c_1, h^c_2, ..., h^c_k\}$$

Similarly, the `testEncoder` outputs the sequence of encoded vector representations, h^t, for test features (of length q):

$$h^t = \text{GRU}(t)$$
$$h^t = \{h^t_1, h^t_2, ..., h^t_q\}$$

Multi-head attention mechanism. To capture the relation between code diff and a test, we apply an attention mechanism [75]. An attention is a function to map a query (Q) and key-value (K-V) pairs to an output, which is computed as a weighted sum of the values (V), where the weight is calculated by applying some compatible function like *softmax* [12] on key (K) and query (Q). In our ML-based RTS model, the query and value are the projected code diff representation vector and the key is the projected test representation vector:

$$\begin{aligned}\text{head} &= \text{Attention}(h^c, h^t, h^c)\\ &= \text{softmax}(\frac{(h^c W_i^{h^c})(h^t W_i^{h^t})}{\sqrt{d_t}})h^c W_i^{h^c}\end{aligned}$$

where, $W_i^{h^c}, W_i^{h^t}$ are trainable parameters that project code diff vectors (h^c) and test vectors (h^t), and d_t is the scaling factor that is equal to the dimension of vector $h^t W_i^{h^t}$. Instead of single attention head, we use multi-head attention [75]. The output representations from different attention heads are concatenated together, and we do another projection to get the final vector that represents the code diff, h^{att}:

$$\begin{aligned}h^{att} &= \text{MultiHead}(h^c, h^t, h^c)\\ &= \text{concat}(\text{head}_1, ..., \text{head}_{|\text{H}|})W^O\end{aligned}$$

where W^O represents trainable parameters for projections.

Predicting likelihood of positive relation. Finally, we feed the learned representation h^{att} into an MLP network, which is a neural network with two linear layers and a ReLU activation function [52], with sigmoid function to output a score between 0 and 1:

$$s = \text{sigmoid}(\text{MLP}(h^{att}))$$

For a code change, if there are multiple changed files, the ML-based RTS model computes a separate score for the changes in each of the changed files, taking a maximum score across all changed files as the final score for the overall code change. We take the maximum to reduce the effect of potential very low scores due to minor changes in many files.

Recall that the training instances from our dataset are triples of $\langle c, t^+, t^- \rangle$. We train the ML-based RTS model to predict the scores s^+ and s^- for $\langle c, t^+ \rangle$ and $\langle c, t^- \rangle$ pairs, respectively, and then we minimize the ranking loss [65, 78]:

$$Loss = \sum_{\langle c, t^+, t^- \rangle \in D} max(0, m - (s^+ - s^-))$$

where D is the training dataset, and m is the user-defined margin. Intuitively, the ranking loss encourages the score between code changes and positively labeled test to go up and the score between code change and negatively labeled test to go down.

ML-based RTS models. From a combination of what the ML-based RTS model is trained to learn and the type of features that are extracted from code changes, we train a total of six models, shown in Table 2. We use Ekstazi as the analysis-based RTS technique for labeling tests selected after a change during training, because, in some cases, STARTS selects all tests and thus we are unable to have negative examples (i.e., the trained model would select all tests).

Table 2: Names of our ML-based RTS Models.

	Basic	Code	ABS
Failing Tests	Fail-Basic	Fail-Code	Fail-ABS
Ekstazi-Selected Tests	Ekstazi-Basic	Ekstazi-Code	Ekstazi-ABS

3.4 Other ML-based RTS Models

3.4.1 IR-based RTS. Information retrieval (IR) can be used to rank documents based on their relevance to a given query. Recent work

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

in test-case prioritization proposed using IR to rank tests based on their relevance to code changes by treating each test file as a document and the code changes as a query [58, 64]. The ranking is based on assigned scores to each test, which is similar to our ML-based RTS models. We build IR-based RTS based on BM25, which Peng et al. found to work best for test-case prioritization [58][3]. We set a threshold to allow for selecting the top, most related tests (the exact ordering of tests is irrelevant for our purpose).

3.4.2 EALRTS. Lundsten et al. presented an ML-based RTS tool called EALRTS [48], which was trained using Random Forest or XGBoost to select tests, using features from a dependency graph extracted by STARTS and the project's Git commit history. Further, they used the mutation testing tool, PIT [1], to create mutants for use as their training dataset. By inserting 1-15 randomly selected mutants into project for multiple iterations, they simulated multiple times code change. Since EALRTS is not designed to be applied to commits that were not used for training, we adapt some of their features. Specifically, the *failure rate* feature that they used for each test is calculated by running PIT and computing killed mutants per test, but running PIT on each commit is too costly. Instead, we use the failure rate computed through running PIT on the training commit in our evaluation. We re-implement EALRTS using the suggested Random Forest algorithm and its features.

3.5 Combining Analysis-based RTS and ML-based RTS

Finally, we combine analysis-based RTS and ML-based RTS by using our proposed ML-based RTS models on top of an existing analysis-based RTS technique. After a developer makes a change, an analysis-based RTS technique can first analyze those changes to do an initial selection of tests. Then, ML-based RTS can use a previously trained model to rank the tests selected by analysis-based RTS by assigning a score to each test based on its relevance to the change. Note that ML-based RTS would only rank tests that are selected by analysis-based RTS and would not rank unselected tests. That is, ML-based RTS selects a subset of tests that analysis-based RTS selects based on the likelihood of failure after a code change. Finally, only tests with scores that are above a user-defined threshold N would be rerun (Section 3.3). As such, this combination of analysis-based RTS and ML-based RTS always selects at most the number of tests selected by the analysis-based RTS technique.

The combination with ML-based RTS only filters tests that analysis-based RTS selects, meaning that it incurs the same cost as analysis-based RTS analysis for finding tests that are affected by the change. It also incurs an additional cost of running the ML-based RTS model to rank the affected tests and to select only the most relevant tests. The cost of using an ML-based RTS model is usually quite small, expected to be less than the cost of analysis-based RTS analysis [48]. We expect that the combination of analysis-based RTS with ML-based RTS would rerun fewer tests than analysis-based RTS at a similar overall cost, leading to lower end-to-end testing time than analysis-based RTS alone.

Note that combining analysis-based RTS and ML-based RTS may not rerun some tests that analysis-based RTS finds to be affected by the changes if the ML-based RTS model predicts that the final pass/fail outcome of such tests will remain the same. Recall that analysis-based RTS techniques rely on updating the proper test dependencies after a change to ensure the correct selection across multiple changes. While a static analysis-based RTS technique like STARTS can statically re-analyze the code after a change to update those dependencies, a dynamic analysis-based RTS technique like Ekstazi needs to rerun the test and collect coverage to update test dependencies. So, when combining with a dynamic technique like Ekstazi, all affected tests found by the analysis-based RTS technique should be run in a separate offline phase [25] after developers have seen the results from running the tests that ML-based RTS selects.

[3]Following Peng et al.'s work [58], we specifically configure to use Low_{token} preprocessing rule and whole-file context.

Table 3: Projects and statistics of dataset.

Projects	Train SHA	Train # Mutants	Evaluation # SHAs	Evaluation # Mutants
Asterisk	a630a125	7,801	4	21
Bukkit	8a1dbc38	7,263	5	12
Configuration	801f4f4b	6,117	8	48
Csv	2210c0b0	536	18	90
Lang	ba8c6f6d	10,937	36	218
Net	dfd5f19d	5,783	17	125
Validator	97bb5737	2,008	15	78
Gedcom4j	fcf39a01	8,618	21	156
Vectorz	1e6769ef	26,025	20	165
ZtExec	acfe9d41	246	14	34

4 DATASET

We perform our evaluation on 10 open-source Java projects that were previously used in prior work on regression testing [42, 53]. These projects use Maven and JUnit 4, as required by Ekstazi [25] and STARTS [42]. Table 3 shows the list of projects (the first column). For each project, we select an early commit where we could build the project with no test failures and run analysis-based RTS tools while still having sufficient commits for use in our evaluation. We then use this commit, shown in Table 3, for training.

We collect commits for evaluation going forward from this initial training commit. Commits that we collect satisfy three requirements: (1) the project must compile successfully on the commit, (2) there must be a bytecode change to a compiled Java class file between the commit and the previous collected commit, (3) there must be some added or modified (not only deleted) Java source code. Requirement (2) ensures there is a change to Java code for Ekstazi or STARTS to analyze, and requirement (3) ensures that commits contain changes that are feasible for the ML-based RTS models. We skip commits that do not satisfy these requirements. We aim to collect 50 commits per project for our evaluation. In total, we filter out 15 commits with compilation failures, 836 commits with no bytecode change, and 16 commits that only delete lines.

On the initial training commit, we construct the training dataset by running PIT [1], a mutation testing tool for Java. We apply all of PIT's default mutation operators to generate mutants (see Table 1). These mutants constitute the set of code changes that we use for

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

constructing a training dataset (Section 3.1). We show the total number of mutants generated by PIT in Table 3.

Mutation evaluation dataset. Unlike prior work [48], we do not train the models using some mutants and then evaluate on the remaining mutants. Instead, we build an evaluation dataset for each project by leveraging its real code evolution. Figure 2 (right side) illustrates the approach that we use to construct the evaluation dataset. Of the commits we collected that come after the initial commit, we run their tests to see if they all pass on those commits. On each of these commits, we introduce failing tests by mutating the code at that commit. We compute the Java source file lines that differ between the commit and the previous valid commit. We then apply a different mutation testing tool, `universalmutator` [27], to create mutants that modify *only the changed lines.* To avoid the problem that most mutants modify the same file, we limit the number of mutants per file to 3, namely randomly selecting 3 generated mutants per file. The number of evaluation commits per project (shown in Table 3) is fewer than 50, because we exclude commits where no mutant is killed or all mutants introduce infinite loops or compilation errors.

We run all tests on each pair of mutant and commit to collect tests that can fail for each combination of change and mutant. We use the resulting set of code changes and test failures as our evaluation dataset. In other words, we run an ML-based RTS model on a code change (combination of real changes and mutation) to compute the tests that would be selected, comparing against the tests that fail on that pair of mutant and commit. We run Ekstazi and STARTS as our representative analysis-based RTS techniques across all commits.

5 EVALUATION

We answer the following research questions:

RQ1: How much can ML-based RTS improve analysis-based RTS techniques when they are combined?

RQ2: How do ML model configurations impact the effectiveness of the combined techniques?

RQ3: How do the overheads of combining ML-based RTS and analysis-based RTS compare with only using analysis-based RTS techniques?

5.1 Evaluation Setup

We ran all experiments on a GeForce GTX 1080 GPU, 3.20 GHz Intel(R) Core(TM) i7-8700 machine with 64GB of RAM, with Ubuntu Linux 18.04.4 LTS and Oracle Java 64-Bit Server version 1.8.0_241. We used Pytorch 1.7.1 to implement our ML-based RTS models.

We run each analysis-based RTS technique on each commit, analyzing the diff between a commit and the previous commit to obtain the percentage of all tests that is selected. The lower the selection rate, the better.

We run our ML-based RTS models on each pair of mutant and commit, as collected in our evaluation dataset (Section 4). We measure for each pair of mutant and commit the percentage of tests (out of all tests in the test suite) that the model would need to select to run all failing tests, i.e., to achieve 100% recall. Conceptually, the largest such selection rate across all pairs of mutants and commits in our evaluation set represents the selection rate that would ensure the model selects all failing tests across all pairs of mutants and commits. Again, lower selection rates are better.

Finally, we measure the overhead of using each analysis-based RTS technique and combining it with ML-based RTS models. For each commit, we measure the time for each analysis-based RTS technique to select tests and to run those selected tests. For each pair of mutant and commit, we measure the time each ML-based RTS model takes to select a subset of the tests that are initially selected by an analysis-based RTS tests.

5.2 RQ1: Combining analysis-based RTS and ML-based RTS

Tables 4 and 5 show the results of our experiments in which we applied our ML-based RTS models on the affected tests that are selected by Ekstazi and STARTS, respectively. The evaluated models are those presented in Table 2 as well as the BM25-model as another baseline. We report the *best safe selection rate*, which is the largest selection rate needed to select *all failing tests* across all pairs of mutants and commits in each project (ensuring safe selection). Users may configure the optimal threshold scores different per model, so we report the best safe selection rate as a means to compare the different models. The selection rates for Ekstazi or STARTS are also included in the table.

The "# Best" and "# Worst" rows show the number of projects in which each model or technique had the best and worst selection rates, respectively (multiple models may have the best/worst selection rate for the same project). The rates in bold fonts signify the model(s) or technique(s) that perform best across all models for each project. The numbers with gray background signify the model(s) or technique(s) that perform the worst for each project.

In Table 5, we additionally report the results from running EALRTS, because EALRTS also builds upon STARTS. We only report EALRTS results on 9 projects, excluding Csv due to insufficient training data. PIT generates mutants on Csv, but after randomly inserting 1-15 selected mutants for multiple iterations, each iteration results in compilation errors or 0 test selected by STARTS.

We highlight three main findings based on results in Tables 4 and 5:

(1) There is no single ML-based RTS model that consistently outperforms or underperforms the rest. Fail-Basic is the model with the highest "# Best", followed by Fail-ABS. We find that ML-based RTS models trained to learn when a test fails (Fail-*) are uniformly better than ML-based RTS models trained to replace an analysis-based RTS tool. Using only the class name of the changed class file as the code diff feature (Basic) is better than including raw code changes (Code) or their abstraction (ABS). We provide detailed analysis in RQ2.

(2) In most cases, ML-based RTS outperforms the BM25-model baseline and EALRTS. When selecting from Ekstazi subset, our best ML-based RTS Fail-Basic is better than the BM25-model for all the projects. When selecting from STARTS, Fail-Basic is better than BM25-model for 8 projects out of 10 projects, and it is at least as good as EALRTS (which is designed for improving STARTS) on 7 out of 9 projects.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

Table 4: Comparison of best safe selection rate of models that select from subset of Ekstazi.

Projects	Fail+			Ekstazi+			Baseline	Ekstazi
	Basic	Code	ABS	Basic	Code	ABS	BM25	
Asterisk	**0.13**	0.22	0.15	0.15	0.15	0.15	0.15	0.22
Bukkit	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**	0.17
Configuration	**0.29**	0.55	0.30	0.57	0.59	0.56	0.52	0.59
Csv	**0.58**	0.71	0.71	0.71	0.62	0.60	0.73	0.79
Lang	0.12	0.14	**0.11**	0.23	0.19	0.26	0.26	0.28
Net	**0.05**	**0.05**	**0.05**	0.12	0.12	0.12	0.09	0.12
Validator	**0.09**	0.23	0.19	0.24	0.31	0.14	0.23	0.33
Gedcom4j	0.80	0.90	0.81	**0.79**	0.90	0.86	1.00	1.00
Vectorz	0.91	0.99	**0.83**	0.93	0.99	0.93	0.99	0.99
ZtExec	**0.90**	**0.90**	**0.90**	0.92	**0.90**	**0.90**	0.92	0.93
# Best	7	3	5	2	2	2	1	N/A
# Worst	1	3	1	3	5	3	6	N/A

Table 5: Comparison of best safe selection rate of models that select from subset of STARTS.

Projects	Fail+			Ekstazi+			Baseline	EALRTS	STARTS
	Basic	Code	ABS	Basic	Code	ABS	BM25		
Asterisk	0.33	0.43	0.30	0.24	0.39	0.46	**0.20**	0.35	0.54
Bukkit	0.56	0.64	0.47	0.58	0.64	0.63	0.61	**0.45**	0.64
Configuration	**0.35**	0.55	0.36	0.57	0.69	0.56	0.62	0.40	0.69
Csv	**0.60**	0.73	0.73	0.73	0.67	0.67	0.80	N/A	0.80
Lang	0.82	0.83	0.83	**0.73**	0.83	0.83	0.79	0.80	0.83
Net	**0.05**	**0.05**	**0.05**	0.12	0.12	0.12	0.09	**0.05**	0.12
Validator	**0.09**	0.23	0.19	0.24	0.31	0.14	0.24	0.27	0.34
Gedcom4j	0.80	0.90	0.81	**0.79**	0.90	0.86	1.00	1.00	1.00
Vectorz	0.91	0.99	**0.83**	0.93	0.99	0.94	0.99	0.99	0.99
ZtExec	**0.90**	**0.90**	**0.90**	0.92	**0.90**	**0.90**	0.92	0.92	0.93
# Best	5	2	3	2	1	1	1	2	N/A
# Worst	0	3	1	2	6	3	4	3	N/A

(3) Combining ML-based RTS with analysis-based RTS improves the precision of Ekstazi and STARTS. In 47 of the 60 project/ML-based RTS model combinations (Fail-* and Ekstazi-*), the models selected fewer tests than Ekstazi. For STARTS, that ratio is 48 out of 60. While selecting from Ekstazi, Fail-Basic on average has a selection rate of 40.39% and reduces Ekstazi's average selection rate by 25.34%. For STARTS, Fail-Basic on average has a selection rate of 54.03% and reduces STARTS's average selection rate by 21.44%. This finding lends credence to Lundsten's findings about the high potential of ML-based RTS to help improve the precision of imprecise RTS techniques like STARTS [48].

5.3 RQ2: Impact of Model Configurations

We design two ML-based RTS models; one is trained to select failing tests (Fail-*), and the other one is trained to select affected tests (Ekstazi-*). Both models have three variants using different features: Basic, Code and ABS described in 3.2

Fail-* ML-based RTS are uniformly better than Ekstazi-* ML-based RTS when combined with analysis-based RTS. For example, the average best safe selection rate for Fail-Basic across 10 projects is 40.39% compared to Ekstazi-Basic's 48.31% when combined with Ekstazi (Table 4). As such, models that mimic analysis-based RTS seems to weaken the overall effectiveness when combined with analysis-based RTS.

Using the changed file's class name as the feature (Basic) performs the best followed by using both class name and the abstract code changes (ABS). Using test class name together with raw code changes (Code) turns out to be the worst. The reason is likely because the actual code diff and changed lines cannot generalize well to unseen code changes, introducing extra noise into the models' classification. This problem can be mitigated by using abstract code changes as shown by comparing *-ABS and *-Code models. However, it is impossible to precisely categorize all the code changes into

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

the mutation operators in Table 1, suggesting that better abstraction rules may be necessary.

5.4 RQ3: ML-based RTS vs. Analysis-based RTS Overheads

Table 6 and Table 7 show the end-to-end testing time for the combination of ML-based RTS models with Ekstazi and STARTS compared against that time for Ekstazi and STARTS alone. End-to-end testing time is defined as the summation of the time to select tests and the time for running the selected tests such that all the failing tests are included in the selected test set. Note that the selection time for models includes the time to process the files, extract features and give scores to the tests. For ML-based RTS and BM25-model, we use the selection rate reported in Table 4 and Table 5 for each project on all pairs of mutants and commits. The times per project are averaged across all pairs of mutants and commits.

Tables 6 and 7 show that the overall end-to-end testing time for our ML-based RTS models combined with an analysis-based RTS technique for the most part outperform the corresponding analysis-based RTS technique. Among the 60 model-project combinations (Fail-* and Ekstazi-*), 46 of them took less time when combined with Ekstazi than running just Ekstazi. When combined with STARTS, 47 of them took less time than running just STARTS. BM25-model usually takes more time because computing the BM25 scores requires parsing the changed source files. EALRTS takes more time because of parsing the dependency graph and collecting historic data as features from git histories. Overall, our results highlight how ML-based RTS models can help improve testing time by selecting fewer tests without adding too much overhead.

6 THREATS TO VALIDITY

The projects that we used to evaluate RTS may not be representative of all projects. We used projects from prior work on regression testing, which represented a variety of applications and domains. Our project selection was also constrained by the analysis-based RTS tools that we used: Ekstazi and STARTS.

We relied on mutation testing to obtain failing tests for both training and evaluation. But, we note that we used different mutation testing approaches in these two phases. While mutants are not real faults, prior work found them to be representative [3, 37].

We studied analysis-based RTS techniques that use coarse-grained dependencies, i.e., classes. Using analysis-based RTS with method-level dependencies could lead to very different conclusions. Prior work showed that Ekstazi and STARTS can substantially reduce regression testing cost, and they can outperform fine-grained RTS techniques in terms of end-to-end testing time due to their lightweight analysis. Future work should combine finer-grained analysis-based RTS (e.g., method or statement level) with ML-based RTS.

7 RELATED WORK

ML in RTS. Our work is similar to Lundsten's EALRTS [48], a ML-based RTS technique that selects tests that are likely to fail and combines ML-based RTS with STARTS. EALRTS also trained an ensemble model (Random Forest) based on mutation analysis, and it uses some of the same features as Machalica et al.'s work [50], plus additional features based on STARTS's dependency graph. On a single medium-sized open-source project, EALRTS selected $2x$ fewer tests than STARTS with a recall rate of 95%. However, EALRTS requires running STARTS to collect necessary features, whereas our models are more lightweight by relying on just source code changes. We compare and combine analysis-based RTS and ML-based RTS on a larger set of models and open-source software, using both Ekstazi and STARTS.

Chen et al. [17] used a semi-supervised K-means algorithm to improve cluster-based RTS. Cluster-based RTS [20, 81] clusters tests, e.g., based on behavioral profiles, and uses sampling techniques to select representative tests from each cluster. We do not perform clustering, and our models are trained on static information about tests and code. We label tests based on their pass/fail outcomes.

Mayo and Spacey [51] trained ML classifiers on partially executed regression tests for predicting whether the remaining regression test will fail. Their features are dynamically-collected performance metrics, and they use support vector machines, Random Forest, and Naive Bayes as classifiers. In contrast, we do not use any dynamic features and aim to select all failing tests that should be rerun after a code change.

ML in other regression testing techniques. Prior work proposed utilizing ML, Natural Language Processing, and Information Retrieval for test-case prioritization (TCP) [8, 13, 15, 16, 19, 39, 41, 71, 73]. TCP ranks tests, ordering first the "better" tests in terms of specified testing goals, e.g., code coverage. We also rank tests, but we select a percentage of the "best" tests, without regard to order.

Elsner et al. [21] empirically evaluated the use of only metadata available from the version control system and continuous integration system to perform TCP and RTS by leveraging ML models. Their evaluation found that relatively simple models are seemingly more effective than the more complicated ones proposed in prior work. In contrast, we aim to improve analysis-based RTS by combining with ML-based RTS, which requires compiling and running tests on past commits along with running the analysis-based RTS tools, extra information that is unavailable from just the version control system and continuous integration system metadata.

Kim et al. [38] proposed a predictive mutation analysis technique, Seshat, to learn the relationship between a mutant and a test case. Compared with generating mutants and executing tests on them, Seshat makes use of Natural Language Processing to learn the relationship between the syntactic and semantic concepts of each test and the mutant it can kill. Similarly, instead of using just analysis-based RTS, which relies on test dependencies, we propose enhancing it with a predictive test selection technique that learns the relationship between the syntactic and semantic concepts of code change and failed tests.

Analysis-based RTS. Analysis-based RTS has been studied for over three decades [4, 9–11, 22, 23, 26, 31–33, 40, 44, 45, 54, 56, 57, 59–63, 69, 70, 72, 79, 80, 85]; Yoo and Harman presented a survey [82]. Recent results on RTS include those that

(1) evaluate and compare RTS at different levels of program granularity, such as file-level plus method-level RTS, file-level vs. module-level RTS, or file-level RTS at ultra-large software ecosystem scale [30, 68, 74, 84];
(2) handle specific programming language constructs, such as how to deal with reflection when performing static file-level

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

Table 6: Average end-to-end testing time (seconds) combining models with Ekstazi.

Projects	Fail+			Ekstazi+			Baseline	Ekstazi
	Basic	Code	ABS	Basic	Code	ABS	BM25	
Asterisk	**15.75**	20.85	17.74	17.14	17.74	17.74	17.35	20.22
Bukkit	6.71	6.93	6.93	6.71	6.93	6.93	6.93	**6.69**
Configuration	**110.90**	153.77	115.03	153.63	159.38	155.70	149.12	156.55
Csv	**23.17**	27.37	27.38	27.05	24.49	23.68	28.75	29.70
Lang	**29.96**	33.06	30.19	41.04	37.84	45.35	45.22	47.13
Net	**9.62**	10.03	10.24	12.05	12.46	12.46	11.88	12.03
Validator	**9.30**	10.33	10.09	10.16	10.78	9.85	10.26	10.59
Gedcom4j	64.06	69.57	66.09	**63.47**	69.54	67.98	70.89	69.80
Vectorz	51.28	52.92	**48.58**	51.48	52.92	51.79	53.94	52.59
ZtExec	**26.44**	26.77	26.77	26.78	26.77	26.77	27.41	27.05

Table 7: Average end-to-end testing time (seconds) combining models with STARTS.

Projects	Fail+			Ekstazi+			Baseline	EALRTS	STARTS
	Basic	Code	ABS	Basic	Code	ABS	BM25		
Asterisk	35.33	42.70	34.42	28.69	39.97	44.12	**26.49**	49.67	47.93
Bukkit	27.71	31.24	**24.28**	28.97	31.24	30.71	31.19	36.02	31.00
Configuration	**169.18**	212.78	173.86	213.20	234.61	215.09	223.80	193.64	232.74
Csv	**26.31**	31.58	31.58	31.26	29.74	29.70	35.02	N/A	34.43
Lang	185.97	186.44	186.44	**169.11**	186.44	186.45	184.32	193.58	186.04
Net	**10.86**	11.28	11.50	14.33	14.73	14.74	13.68	16.64	14.31
Validator	**11.58**	13.28	13.05	13.11	13.73	12.80	13.32	14.41	13.61
Gedcom4j	72.77	78.49	74.91	**72.17**	78.46	76.91	79.88	81.90	78.62
Vectorz	121.14	131.25	**109.21**	121.74	131.24	125.73	133.67	145.25	130.89
ZtExec	**26.78**	27.10	27.10	27.56	27.10	27.11	28.20	28.14	27.83

analysis-based RTS [67] or how to leverage the semantics-preserving nature of refactoring to improve the precision of dynamic RTS [77];

(3) apply RTS to other programming languages apart from Java such as .NET [74] or even applications that use multiple programming languages [14];

(4) perform static analysis-based RTS as a Java compiler pass [55];n

(5) find safety, precision, and performance bugs in RTS tools [86].

Our goal is to evaluate the use of ML for improving the precision of static and dynamic analysis-based RTS. It would be interesting future work to see whether ML can also be used to improve some of the aforementioned recent work on analysis-based RTS.

8 CONCLUSION

We evaluate the use of ML-based RTS for improving analysis-based RTS on medium-sized open-source software. Prior work evaluated ML-based RTS on large-scale software (e.g., the monorepository at Facebook), or performed preliminary evaluation of ML-based RTS on one medium-sized project. We use mutation analysis to obtain many failing tests for creating our training dataset. We also compare the effectiveness of ML-based RTS and analysis-based RTS for selecting failing tests after real code changes. ML-based RTS and analysis-based RTS are often complementary, and we evaluate the combination of both. Results show that combining the approaches improves the precision of analysis-based RTS. The synergy between analysis-based RTS and ML-based RTS is a step forward in reducing regression testing cost. Future work should study further combination of ML and finer-grained analyses techniques.

ACKNOWLEDGMENT

We thank Nader Al Awar, Pengyu Nie, Zhiqiang Zang, and the anonymous reviewers for their comments and feedback. This work is partially supported by a Google Faculty Research Award, and these US National Science Foundation grants: CCF-1652517, CCF-2019277, CCF-2045596 and CCF-2107291.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] 2018. PIT Mutation Testing. http://pitest.org.
[2] 2018. Travis-CI. https://travis-ci.org.
[3] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. 32, 8 (2006), 608–624.
[4] Thomas Ball. 1998. On the Limit of Control Flow Analysis for Regression Test Selection. 134–142.
[5] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. 433–444.
[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. 447–450.
[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. 5, 2 (1994), 157–166.
[8] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. 1–12.
[9] John Bible, Gregg Rothermel, and David S. Rosenblum. 2001. A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques. *ACM Transactions on Software Engineering and Methodology* 10, 2 (2001), 149–183.
[10] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica* 35, 3 (2011), 289–321.
[11] Lionel Briand, Yvan Labiche, and Siyuan He. 2009. Automating Regression Test Selection Based on UML Designs. 51, 1 (2009), 16–30.
[12] John S Bridle. 1990. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Advances in neural information processing systems*. 211–217.
[13] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. 975–980.
[14] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *Symposium on the Foundations of Software Engineering*. 809–820.
[15] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. 700–711.
[16] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. 656–667.
[17] Songyu Chen, Zhenyu Chen, Zhihong Zhao, Baowen Xu, and Yang Feng. 2011. Using semi-supervised clustering to improve regression test selection techniques. 1–10.
[18] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734.
[19] Hagai Cibulski and Amiram Yehudai. 2011. Regression test selection techniques for test-driven development. 115–124.
[20] William Dickinson, David Leon, and A Podgurski. 2001. Finding failures by cluster analysis of execution profiles. 339–348.
[21] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. 491–504.
[22] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. 52, 1 (2010), 14–30.
[23] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. 22–31.
[24] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection, Vol. 2. IEEE, 713–716.
[25] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. 211–222.
[26] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 1998. An Empirical Study of Regression Test Selection Techniques. 188–197.
[27] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation. 25–28.
[28] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. 933–944.
[29] Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[30] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. 112–122.
[31] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. 312–326.
[32] Mary Jean Harrold and Mary Lou Soffa. 1988. An incremental approach to unit testing during maintenance. 362–367.
[33] Jean Hartmann. 2007. Applying Selective Revalidation Techniques at Microsoft. 255–265.
[34] Jean Hartmann. 2012. 30 Years of Regression Testing: Past, Present and Future. 119–126.
[35] Simon Haykin. 1994. *Neural networks: a comprehensive foundation.* Prentice Hall PTR.
[36] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. 37, 5 (2011), 649–678.
[37] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing? 654–665.
[38] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. 2021. Predictive Mutation Analysis via Natural Language Channel in Source Code. *arXiv preprint arXiv:2104.10865* (2021).
[39] Patipat Konsaard and Lachana Ramingwong. 2015. Total coverage based regression test case prioritization using genetic algorithm. In *International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. 1–6.
[40] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. 1994. Change Impact Identification in Object Oriented Software Maintenance. 202–211.
[41] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-level test case prioritization using machine learning. In *International Conference on Machine Learning and Applications*. 361–368.
[42] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. 583–594.
[43] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. 949–954.
[44] H. K. N. Leung and L. White. 1989. Insights into Regression Testing. 60–69.
[45] H. K. N. Leung and L. White. 1991. A Cost Model to Compare Regression Test Strategies. 201–208.
[46] William A. Little. 1974. The existence of persistent states in the brain. *Mathematical biosciences* 19, 1-2 (1974), 101–120.
[47] William A. Little and Gordon L. Shaw. 1978. Analytic study of the memory storage capacity of a neural network. *Mathematical biosciences* 39, 3-4 (1978), 281–290.
[48] Erik Lundsten. 2019. *EALRTS: A predictive regression test selection tool.* Master's thesis. KTH Royal Institute of Technology, Sweden.
[49] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. 643–653.
[50] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. 91–100.
[51] Michael Mayo and Simon Spacey. 2013. Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics. In *International Symposium on Search Based Software Engineering*. 158–171.
[52] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. 807–814.
[53] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. 249–259.
[54] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression Testing in an Industrial Environment. *Commun. ACM* 41, 5 (1998), 81–86.
[55] Jesper Öqvist, Görel Hedin, and Boris Magnusson. 2016. Extraction-based regression test selection. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–10.
[56] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. 241–251.
[57] David Parsons, Teo Susnjak, and Manfred Lange. 2013. Influences on regression testing strategies in agile software development environments. (2013), 1–23.
[58] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically revisiting and enhancing IR-based test-case prioritization. 324–336.
[59] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, Ophelia Chesley, and Julian Dolby. 2003. *Chianti: A Prototype Change Impact Analysis Tool for Java.* Technical Report DCS-TR-533. Rutgers University CS Dept.
[60] Gregg Rothermel and Mary Jean Harrold. 1993. A Safe, Efficient Algorithm for Regression Test Selection. 358–367.
[61] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. 22, 8 (1996), 529–551.
[62] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210.
[63] Gregg Rothermel and Mary Jean Harrold. 1998. Empirical Studies of a Safe Regression Test Selection Technique. *ACM Transactions on Software Engineering*

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.

and Methodology 24, 6 (1998), 401–419.
[64] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. 268–279.
[65] Matthew Schultz and Thorsten Joachims. 2004. Learning a distance metric from relative comparisons. *Advances in Neural Information Processing Systems* 16 (2004), 41–48.
[66] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-Deterministic Specifications. 80–90.
[67] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. 187:1–187:29.
[68] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. 228–238.
[69] Mats Skoglund and Per Runeson. 2005. A Case Study of the Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System. 74–83.
[70] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. 17, 3 (2007), 359–378.
[71] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. 12–22.
[72] TestImpactAnalysisPage 2013. Streamline Testing Process with Test Impact Analysis. http://msdn.microsoft.com/en-us/library/ff576128%28v=vs.100%29.aspx.
[73] Paolo Tonella, Paolo Avesani, and Angelo Susi. 2006. Using the case-based ranking methodology for test case prioritization. 123–133.
[74] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for .NET. 848–853.
[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illiã Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[76] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. 13–25.
[77] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. 233–244.
[78] Kilian Q Weinberger and Lawrence K Saul. 2009. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research* 10, 2 (2009).
[79] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database Driven Applications. 421–430.
[80] Guoqing Xu and Atanas Rountev. 2007. Regression Test Selection for AspectJ Software. 65–74.
[81] Shali Yan, Zhenyu Chen, Zhihong Zhao, Chen Zhang, and Yuming Zhou. 2010. A dynamic test cluster sampling strategy by leveraging execution spectra information. 147–154.
[82] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. 22, 2 (2012), 67–120.
[83] Nathan York. 2011. Tools for Continuous Integration at Google Scale. https://www.youtube.com/watch?v=b52aXZ2yi08.
[84] Lingming Zhang. 2018. Hybrid Regression Test Selection. 199–209.
[85] Jianjun Zhao, Tao Xie, and Nan Li. 2006. Towards regression test selection for AspectJ programs. 21–26.
[86] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. 430–441.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO AMAZONAS. Downloaded on November 27,2025 at 21:34:29 UTC from IEEE Xplore. Restrictions apply.