TU WIEN Informatics

Predictive Test Selection

A Replication Study

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Stefan Aichmann, BSc
Matrikelnummer 01125470

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 1. Jänner 2025

Stefan Aichmann	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Predictive Test Selection

A Replication Study

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Stefan Aichmann, BSc
Registration Number 01125470

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, 1st January, 2025

Stefan Aichmann	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefan Aichmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2025

Stefan Aichmann

Kurzfassung

Die Auswahl von zu testenden Softwaretests ist von entscheidender Bedeutung für die Verbesserung der Effizienz durch die Verringerung der mit dem Testen verbundenen Zeit, Rechenressourcen und Kosten. Indem wir auf der Grundlage von Code-Änderungen oder anderen Kriterien selektiv nur die relevantesten Tests durchführen, können wir die Testzeit erheblich verkürzen und unnötige Berechnungen vermeiden. Dies beschleunigt nicht nur den Lebenszyklus der Softwareentwicklung, sondern führt auch zu Kosteneinsparungen, da weniger Ressourcen für die Ausführung von Tests benötigt werden. Eine effiziente Testauswahl stellt sicher, dass das Testen sowohl effektiv als auch ressourcenschonend ist, was es zu einer wesentlichen Praxis für die Optimierung von Softwarequalität und -leistung macht.

In dieser Arbeit versuchen wir, die „Predictive Test Selection"-Studie von Facebook in einem Open-Source-Kontext zu replizieren. Um dies zu erreichen, sammeln wir Daten aus mehreren Open-Source-Projekten und trainieren Modelle für maschinelles Lernen, um relevante Tests auf der Grundlage von Codeänderungen vorherzusagen. Unser Ziel ist es, herauszufinden, ob die Ergebnisse von Facebook auch außerhalb ihrer proprietären Umgebung zutreffen, und die Wirksamkeit der prädiktiven Testauswahl in einem breiteren, öffentlich zugänglichen Software-Ökosystem zu bewerten.

Die größte Herausforderung in dieser Arbeit war das Sammeln der notwendigen Daten und das Training der Modelle, da die ursprüngliche Arbeit nur eine grobe Beschreibung der Merkmale und der Trainingspipeline enthielt. Daher mussten wir bei der Implementierung mehrere Annahmen und Designentscheidungen treffen. Trotz dieser Hindernisse gelang es uns, den Ansatz zu rekonstruieren und die wichtigsten Ergebnisse zu wiederholen, wodurch die Ergebnisse der ursprünglichen Studie in einem Open-Source-Umfeld validiert wurden.

Abstract

Software test selection is crucial for improving efficiency by reducing the time, computational resources, and costs associated with testing. By selectively running only the most relevant tests, based on code changes or other criteria, we can significantly cut down on testing time and avoid unnecessary computation. This not only speeds up the software development lifecycle but also leads to cost savings, as fewer resources are required for executing tests. Efficient test selection ensures that testing is both effective and resource-conscious, making it an essential practice for optimizing software quality and performance.

In this thesis, we aim to replicate the "Predictive Test Selection" study by Facebook within an open-source context. To achieve this, we collect data from multiple open-source projects and train machine learning models to predict relevant tests based on code changes. Our goal is to evaluate whether Facebook's findings hold true outside of their proprietary environment and to assess the effectiveness of predictive test selection in a broader, publicly available software ecosystem.

The key challenge in this thesis was collecting the necessary data and training the models, as the original paper provided only a high-level description of the features and training pipeline. This required us to make several assumptions and design decisions during implementation. Despite these obstacles, we successfully reconstructed the approach and were ultimately able to replicate the key findings, thereby validating the results of the original study in an open-source setting.

Contents

CHAPTER 1

Introduction

No matter how big or small, how well conceived and tested. A large software project will eventually have to be modified, to fix bugs or respond to changes. After such changes to the software usually Regression Testing is conducted to confirm that the software is not affected adversely. Old tests need to pass, and new tests need to be written, while old tests are rarely discarded. Hence as a program evolves and grows the regression test set grows larger [WHLA97, RH97, GHK+01, RH96].

Although regression testing is the primary means by which most developers carry out software quality assurance, repeating all test cases of the test set after every software revision is often impossible due to time and/or budget constraints [WHLA97, ZLG+22]. A more feasible approach is employing a selective regression testing technique, which chooses a subset of a test suite and then uses this subset to test the modified software. The cost of regression testing is reduced, if the cost of running the selection algorithm combined with the cost of running the subset of the test suite is less than running the whole test suite [HRRW01].

To put the cost into numbers, regression testing can take up to 80% of the testing budget and up to 50% of the software maintenance cost, testing cost is challenging to keep up with even for a company with an abundance of computing resources [GEM15b, MGN+17]. Hence regression test selection is an extensively researched field. In the literature, there are two main approaches to be found. One focuses on the information collected from program specifications, and the other focuses on information about the code, the program, and the modified Version [RH96, GEM15b]. In newer research, the latter is enhanced by using machine learning models to further minimize the subset of tests to run [MSPC19, CH20, PBGB22]. Some of these techniques are already used in industrial applications[1] [MSPC19].

[1] 2023: https://hacks.mozilla.org/2020/07/testing-firefox-more-efficiently-with-machine-learning/

Although the results are intriguing, there are inherent problems in concluding single studies. Every stage of an experiment from literature research to result interpretation is prone to errors. Conclusions may not be justified by the analysis or may simply be incorrect. Because of that, there is a need for experimental results to be externally reproducible [GK18, BE12, BI15, BDM+96]. A survey points out that out of 400 machine learning papers only 6% share their algorithms and only a third share their data, half of them share pseudo code [GK18, Hut18].

Machine Learning models are not only sensitive to the exact code used but also to the training data and even the used hyperparameters[2]. In some cases, even the original researchers cannot reproduce their results [GK18, BE12, BI15, Hut18]. The lack of details in machine learning papers does not only affect the reproducibility of studies, in some cases, researchers find it hard to use acknowledged benchmarks for their models because the results are published in not enough detail. This problem is getting to a point where IBM proposed a machine learning model that is designed to recreate other neural networks from papers [Hut18].

The overall aim of this work, however, is to recreate Facebooks [MSPC19] test selection approach as closely as possible on various open-source projects. Since they are not sharing their data collection algorithm, data, or model code. Data will be collected from open source projects, to train one or more models that fit the described model in their paper. If the recreation succeeds, this model can be used to save considerable amounts of time by only running tests that have a high probability of failing. The central research question of this thesis is:

> *To what extent can Facebook's findings be replicated in the open source context, i.e. will a model, as described in the paper, trained on a dataset created from the features described in the paper, still detect tests that will likely fail when inducing a code change?*

To strengthen our results we want to perform additional training on different models and answer the following questions:

> *Are there models, other than the ones described in Facebook's paper, which can replicate its findings on the dataset?*

Answering these questions can strengthen and extend the findings to the open-source context. This is interesting since the open source context greatly differs from a monolithic repository used in an industrial setting, where project structure, git usage, contributors, etc. do not differ as much from project to project. If therefore the precision of Facebook's

[2]Hyperparameters are parameters not learned by the model but set before learning, one can see them as a model's "configuration". E.g. batch size, learning rate, activation functions, etc. While not learned they are often subject to tuning, to find a configuration in which the model performs best.

models cannot be reached, at least this thesis establishes a baseline for predictive test selection on open-source projects. Additionally creating a dataset containing test meta data of the test suites executions. The success of machine learning algorithms is greatly dependent on the dataset quality, hence creating and curating a high-quality dataset is another important aim of this thesis. Furthermore, it aids research by publishing datasets, tools, and models which, in this form, do not exist at the time this thesis is written.

This thesis is structured as follows: Chapter 2 explains the background knowledge essential for this thesis. We give a brief overview of state-of-the-art test case selection techniques in Chapter 3. In Chapter 5 we describe our experiment setup. When doing machine learning research data quality is essential, Chapter 4 focuses on how our data is generated. Lastly, we report our findings in Chapter 6 and answer the above questions in Chapter 7.

CHAPTER 2

Background

2.1 Regression Testing

Regression testing refers to the process of retesting a modified or updated software system to ensure that the changes made to it have not introduced a regression. When talking about regression, we generally refer to a situation where a previously working piece of software or a feature of a bigger software stack stops functioning correctly after introducing a change. Regression testing involves running a set of pre-existing tests covering most, ideally all, software functionality. This practice aims to ensure the stability and reliability of software, thus providing confidence in its overall quality [GHK+01].

2.2 Regression Test Selection

Regression test case selection is the process of determining a subset of test cases from a test suite that should be executed when a code base is modified [RH96]. Defined formally:

> Let P be an application program and P' be a modified version of P. Let T be the test suite developed initially for testing P. A regression test selection technique aims to select a subset of test cases $T' \subseteq T$ to be executed on P', such that every error detected when P' is executed with T is also detected when P' is executed with T' [RH94].

The main goals of regression test case selection algorithms generally are to be safe, precise and cost-effective. A *safe* regression test selection technique selects every test case that can expose faults in the modified program [HJL+01]. In this thesis, the focus lies on test case selection based on code changes. Such a selection technique will further reduce a safe subset of test cases by only selecting test cases affected by a code change.

A *precise* regression test selection technique only selects affected tests. To be *cost-effective*, the cost of running a regression test case selection algorithm, combined with the cost of running the selected subset of tests, must be less than the cost of running the whole test set[1] [ZLG+22]. To scale to big code bases approaches may trade precision and safety for cost-effectiveness [ERS10, YH12, Wah99, BMSS11, MSPC19, MGN+17].

2.3 Test Flakiness

Flaky tests are tests with a non-deterministic outcome for the same code version. Test flakiness can be caused by various factors [LHEM14]:

1. **Async Wait:** Making an asynchronous call and not waiting properly for the result of the call before it becomes available.
2. **Concurrency:** When different threads interact in a non-deterministic manner; e.g., data races, atomicity violations, or deadlocks.
3. **Test Order Dependency:** The outcome of tests depends on the order in which they are executed
4. **Other Root Causes:** Can be resource leaks, network issues, timing issues, I/O issues, using randomness in tests, floating point operations, unordered collections, etc.

When a flaky test fails the developer is not aware at first that the test is flaky, a developer will typically start debugging their changes within their and the previous revision. Test flakiness is one of the biggest problems in software development in many organizations. Because of its non-deterministic nature, test flakiness is hard to detect automatically [LWW+20].

2.4 Machine Learning for Test Selection

In a supervised machine learning setting, to be precise when given a binary classification task, we try to learn from a labeled dataset, consisting of a feature matrix $X = [x_1, ..., x_n]^T \in \mathbb{R}^{n \times d}$ and a target vector $y \in 0, 1^n$ [BN06], to predict a label for a future unseen feature vector. Our dataset is an observation or sub-set of the universe, where the universe refers to the set of all possible input instances. In our case $y_i = 1$ refers to a failed test and $y_i = 0$ refers to a successful test. Where the test $t \in DependendTests(d)$ of a change d [MSPC19]. A vector $\in \mathbb{R}^d$ of our feature matrix, i.e., a row in our dataset, consisting out of test metadata.

[1] In literature running the whole test set is usually used as an additional baseline for selection techniques, called *RetestAll*

The goal of machine learning, when used for regression testing, is to create a prediction function $y(d, t) \in \{0, 1\}$. That can determine whether a test target denoted as t and a change denoted as d, will detect a regression, after a code change. A perfect model would predict all and only tests impacted by a particular code change introducing a regression. Such a model is not feasible since it would require access to information unavailable at test selection time. While we cannot exactly compute the set of impacted tests, machine learning is used to approximate such a set by learning to identify which tests would have reported a failure based on historical data. The main use of such a model is to filter tests, that have a low probability to fail, leaving only tests with a high probability of detecting a regression for the CI Pipeline to run. Optimizing developer experience at the cost of machine resources and developer time. Hence the model doesn't need to find all such tests since the whole test suite can be run at a later time, e.g., before a feature is merged into the main branch [MSPC19, CH20].

2.5 Mentioned Machine Learning Models

The underlying study [MSPC19] only explains features of its dataset on a high level, also there is no further information given about how the variables. However, by using a gradient-boosted decision tree [CG16], as a classifier, they mention the need to deal with unbalanced labels. Hence in addition to XGBoost, we will also focus on models that can handle such label distribution. Furthermore, a short explanation of other models mentioned in this work is given.

2.5.1 Decision Tree

Decision trees are a way of combining models. Instead of averaging the prediction of a set of models, a model is selected for making predictions in different regions of input space. The model selection progress is best described as the sequential traversing of a binary tree, where a decision is made in every node. A decision does split the feature space depending on the feature represented by the node. Decision trees can also be used as tree-based models, representing a prediction function. The input space is still partitioned as described above. Instead of representing a model the leaves represent the output of such a tree-based model. When classifying data the leaves of the tree represent a label, in regression a leaf represents a constant [BN06].

2.5.2 Random Forest

A random forest is an ensemble of decision trees, such that the final prediction is an aggregation of each tree's independent prediction. Each tree is trained on an independently sampled random subset of the training set with the same distribution for all trees. This process, called bagging, tries to mitigate the high variance common for decision trees. Furthermore bagging can be used to give an ongoing estimate of the generalization error of the random forest [Bre01].

2.5.3 XGBoost

Similar to random forests, the XGBoost model is built upon decision trees. However, it distinguishes itself as a gradient tree-boosting model. Unlike random forests, where decision trees are generated independently, XGBoost sequentially incorporates models to rectify errors made by previous models using gradient descent techniques. This iterative process allows for continuous improvement and refinement, enhancing the overall predictive power of the model [CG16].

2.5.4 Logistic Regression

Logistic regression is a classifier used for binary classification, based on a dataset of *independent* variables. It assumes a linear relationship between features. Hence it may not perform well when faced with highly non-linear relationships. The prediction function of a logistic regression classifier outputs a label's probability bounded between $\{0, ..., 1\}$, based on the feature vector used as input [Bre01].

2.5.5 Multi-Layer Perceptron

A multi-layer perceptron is a neural network with at least three layers, an input layer, one or more hidden layers, and an output layer. Each layer operates on the outputs of the preceding layer. When computing an output, a feature vector is fed through the network, for each neuron[2] the input is computed as a weighted sum of all inputs plus a bias. The result is then used as input for the activation function[3] of the neuron. The output of the activation function, in combination with the output of all other neurons of the same layer is then fed into the next layer. During training, the network predicts its output on input data, compares this prediction with the desired output, and computes the error between its output and the desired output. This error is then propagated back through the network while backpropagating the weights and biases in the network are [Bre01].

2.5.6 Semi-supervised Clustering

Semi-supervised clustering combines semi-supervised learning with cluster analysis. Clustering is an efficient technique to mine useful information by organizing datasets into clusters with similar features. Conventional cluster analysis uses unlabeled data, hence without prior knowledge of the domain, clustering methods often encounter inconsistencies between clustering results and the actual partition of the data. Semi-supervised clustering uses a limited number of labeled data to predicate the cluster tag of unlabeled data. Apart from that, in comparison to traditional cluster analysis, a trained semi-supervised clustering model can predict test data [CHY+23].

[2] A *neuron* is a node in the network, usually connected to all previous nodes in the network

[3] An *activation function* is usually a relatively simple function that depends on how we want a network layer to behave. In more complex neurons

2.5.7 Large Language Models and Natural Language Processing

Natural language processing is an approach to teaching machines the meaning of text-based content, recent media attention was gained by models such as ChatGPT. In the beginning natural language processing was largely rule-based [CC20]. With machine learning gaining traction, research shifted toward language models. Language models do not necessarily need to be used on spoken languages, they can be used on, for example, programming languages. In general, such a model aims to represent the generative likelihood of word sequences, to predict future or missing tokens[4]. Preceding large language models were statistical language models, neural language models, and, pre-trained language models, all with their limitations. Researchers find that scaling up model or data size often leads to improved model capacity, hence large language models were the logical consequence of pre-trained language models. At their core, large language models are still deep neural networks[5], with billions of trained parameters [ZZL+23].

2.5.8 Support Vector Machine

Similar to the logistic regression model, the support vector machine (SVM) classifier is also linear in its parameters. A SVM aims to find a hyperplane in an N-dimensional space, where N is the number of features, that distinctly classifies the data points. The hyperplane's goal is to separate data points belonging to the same label. When training an SVM we search for a hyperplane that has a maximum margin between data points of the same label. The data points that lie on this margin are referred to as support vectors, as they play a crucial role in defining the decision boundary and influencing the classification outcome [Bre01].

Additionally, the SVM allows us to specify a kernel to be used to identify similar data points, alongside a regularization parameter C. This is the equivalent to transforming to another feature space using the dot product. To make an SVM work in a high dimensional feature space the Nystroem approximation technique [WS00] is used.

2.6 Interpretability

Having a model that can select tests that have a high failure chance when a change is introduced is good. But it is also important to know why the model thinks a particular code change introduces a regression. Although the features of the underlying study [MSPC19] are not particular helpful for a developer to figure out why their edit had a regression, we want to enhance the feature selection that the underlying study is doing with an interpretability section.

[4] A *token* refers to units of text used during the training of a language model. A *token* can be as short as a single character or as long as a word or even as long as a substring

[5] *Deep neural networks* are *multi-layer perceptrons* with additional layers, additional connections, or specialized neurons

Several studies have explored the use of the SHAP (Shapley Additive Explanations) method as a game-theoretic approach to interpreting machine learning model outputs. This technique assigns a Shapley value to each feature of a given data point, quantifying its contribution to the model's prediction relative to the dataset's average prediction [LL17, Mol20].

When summed up Shapley values represent the predicted value for a data point minus the average predicted value. This means the prediction at a given data point is distributed additively amongst all features. To then explain a given models decision, we can plot features alongside their shapley values for all data points.

CHAPTER 3

Related Work

Regression testing is essential for modern software development. During the development and maintenance of software systems, developers periodically test regression, to find errors caused by their modifications. Even with abundant resources regression testing can be a bottleneck [MGN+17, CH20, EFK+16, MSPC19, ASD14]. To address cost, regression test selection techniques are an extensively researched field [ERS10, SKPS20]. Closely related research topics, but not the focus of this thesis, are *Test suite reduction*, and *Test case prioritization*.

Test case prioritization, in contrast to test case selection, is a technique that aims to optimize test execution order of test cases. The main goal is to run most important or critical tests first to identify issues more quickly and efficiently [DS18, YH12]. *Test suite reduction* seeks to eliminate redundant, obsolete or ineffective tests from a test suite, while maintaining the test coverage and preserving fault detection capabilities, with the goal to reduce execution time [YH12],

In the literature mainly two ways of regression test selection are found, selection based on code changes, and selection based on specification changes [ERS10], this thesis focuses on the former. Various surveys [ERS10, YH12, Wah99, BMSS11] reviewing a wide spectrum of regression test selection techniques, can be found. In this section, we will give a brief overview of the evolution of such techniques focusing mainly on techniques that directly relate to our work. Starting with traditional analysis-based approaches, followed by more novel machine learning approaches and also a combination of the two.

3.1 Evolution of regression test selection techniques

Initially, regression test selection techniques were based on control flow graphs[1] [HS89], data flow [LW90, KGH+95], and slicing [GHS92, RH93, RH94]. Early data flow approaches introduced the concept of a *firewall* proposed for integration testing [LW90] as well as regression testing [KGH+95]. In contrast to the two main strategies at the time, retesting everything and retesting only affected modules, a *firewall* is a boundary enclosing all modified modules needed and related modules that need to be re-integrated and regression tested. The construction of a firewall is another graph-based approach to regression testing, by analyzing data flow inside a call graph[2]. Later a data flow-centered approach using slicing[3] was introduced. Which has the advantage of not needing to recompute the data flow completely after a change [GHS92].

In 1997 Pythia was introduced, a tool to analyze software systems, based the first of its kind using *textual difference*, to determine differences between source files of an old and a new version of a program. Using the `diff` command is safe and fast in selecting tests. While being safe, this technique is not semantic aware, meaning it cannot decide whether, or not a change affects a program, a change that only adds a comment, would still lead to test executions [VF97].

In 2012 a change-centric approach was proposed, called ChEOPSJ [SD12]. This tool captures code changes and transforms them into change objects while running in the background of Eclipse. Changes are silently logged, and modeled with the help of the FAMIX[4] model. With the help of this model, dependencies between changes and test cases are found. Although it is one of the first test selection approaches evaluated on large code bases, it is an unsafe approach [SDZ13].

Later the concept of smart checksums[VPMG17][5,6]. These techniques either work on the file level or the method level. Smart checksums are used to detect changes by comparing a file's old version to the new version. If a change is detected only impacted tests are run. These are tests whose transitive dependencies include file (method) affected by a change [Zha18, VPMG17, GEM15b]. This approach yielded two wildly known test selection tools, a static[7] one called STRATS [LSM17], a dynamic[8] one called Ekstazi [GEM15a], which are still considered state-of-the-art by many, including the following paper.

[1] In a *control flow graph* nodes represent a program's basic blocks, and edges represent the flow of the program

[2] A *call graph* visualizes how function within a program relate, or call each other.

[3] A *Program Slice* consists of program statements, that might be affected by the value of a variable (forward slice), or program statements that might affect the value of a variable (backward slice), at a program point.

[4] 2025: `ttps://scg.unibe.ch/archive/famoos/FAMIX/Famix20/famix20.pdf`

[5] *Smart Checksums* are checksums that ignore debug information, (i.e. do not change if e.g. a comment is edited) often computed on bytecode files.

[6] 2024: `https://crc32c.machinezoo.com/` an example implementation of smart checksums used in the cited paper

[7] *Static* meaning that information available at compile time is used

[8] *Dynamic* meaning that information available on runtime is used

In 2018 SPIRITuS was proposed, it selects tests by using method code coverage and a vector space model[9] on a method level. The vector space model is used to, in essence, compute lexical differences between a new and an old version of a file. After that, with the help of the code coverage model, this tool decides what test cases to run [RSAM18].

Also, ReTEST was introduced in 2018. It uses program changes, test suite source files, and fault history information to select test cases. It improves on previous approaches by reducing code coverage overhead, building a graph on test cases, and collecting metadata from them. This graph is then saved in a graph database which is later queried. Queries are built from program changes, after a graph traversal ReTest will output a list of selected test cases [AD18].

HyRTS [Zha18] is another state-of-the-art test selection tool that aims to combine traditional test selection techniques at different granularity. To achieve this a file and method level analysis takes place. Both of these steps aim to filter tests by their respective dependency on other files or methods. It is build on Ekstazi [GEM15a] to analyze file level changed and Faulttracer [ZKK12] to analyze method level changes.

Recently a more novel approach, FineRTS, to analysis-based test selection was published by Yu Lui et. al.. Where tests are selected based on semantics of introduced changes, and implemented as an extension to STRATS [LSM17], and Ekstazi [GEM15a]. By manually inspecting code changes, 29 kinds of changes where identified into 13 findings of which 11 are semantic modifications and do not need a test rerun. They show that their tool outperforms novel research that focuses on enhancing test selection techniques with machine learning concepts [LZN+23]. Newer research additionally focuses on enhancing test case selection with machine learning techniques, which will be described in the next section.

3.2 Regression Test Selection using Machine Learning

Chen et.al. use semi-supervised clustering for regression test selection. A function call profile is represented as a binary vector, each bit representing whether a function represented by the bit is called. This binary vector in combination with a distance measure between two function call vectors is used to cluster tests together by training a semi-supervised k-means clustering model. Each cluster contains tests of similar behavior. A few tests are sampled randomly from each cluster, if a test in a cluster fails all tests in this cluster will be selected [CCZ+11].

Mayo et.al. are using a genetic algorithm for dynamic metric selection in combination with machine learning algorithms to select test cases. In the first step dynamic execution metrics are collected for the correct execution of a program. After that, a genetic algorithm is used to modify the program. The program is run again, and the metrics from the modified program are combined with those from the unmodified program and

[9] A *Vector Space Model* represents a set of documents as vectors in a common vector space [Man09].

used for training various machine learning algorithms, with the random forests classifier being the best-performing one [MS13].

Facebook's approach is based on collecting test meta-data on tests affected by a change. This meta-data is being collected from commits of a repository, by running the test suite on them. This meta-data is labeled by the outcome of a test, and used for training an XGBoost classifier [MSPC19]. While mining historical test data for test selection isn't a new approach [ASD14], using it in combination with machine learning is. EALRTS is an approach based on Facebook's paper, which uses the same meta-data and models in combination with STRATS to collect data. The dataset is mined by injecting mutants into the code base [Lun19]. Since Facebook's paper represents the underlying study of our work, a lot more detail will be given throughout this thesis.

Spieker et.al. uses reinforcement learning for test case selection. Features used for learning are the duration of a test, the previous execution, and failure history. Their features allow them to purely learn from historical test data, meaning no source code or program access is required [SGMM17].

Zhang et.al. are combining traditional regression test selection techniques based on code changes with machine learning models for their test selection approach. Given a change and a test suite, one model assigns a score to each test representing the relevance of each test, by extracting semantic features from the change and the test. The assigned score represents the likelihood that this test needs to be selected. The second model predicts whether a test would fail due to a change. This model reimplements EALRTS [Lun19] by injecting mutants into the database, but using a different approach to feature collection, and training a random forest classifier on the resulting dataset. The models are used on the output of the traditional test selection, which makes sure that at most the output of the traditional selection can be selected [ZLG+22].

Zhang et.al aim to combine traditional analysis-bases regression test selection approaches with learning bases approaches. To achieve this a data set is, again, constructed via mutation analysis on which novel machine learning classifiers are trained. After that STARTS [LSM17], and Ekstazi [GEM15a] is used to filter the tests, on the output of these analysis based approaches the trained models are run [ZLG+22].

An approach by Sutar et.al. leverages natural language processing to select test cases. They exploit text similarity between historical defect data and test cases [SKPS20].

3.3 Replication Studies in Software Engineering Research

The absolute number of replication studies is still small in software engineering, although it has been growing in the last few years [DSSF+14]. While there is growing concern that most published research findings are false, the probability of a research claim being true increases by replications [MKJ07]. Even though growth in replication studies is evident it does not keep up with newly published research [DSSF+14].

The terms replication and reproducibility are frequently used interchangeably, but they have distinct meanings. Replication refers to the process of independent researchers repeating an experiment in a different environment, with modifications to the original study to achieve consistent results. In contrast, reproduction is to recompile the same materials from a study, including data, analysis, and procedures, to validate the findings and obtain the same results [LP15, MK17, BMG20, BID+14].

In the highly related field of machine learning in software defect prediction, it is found that bias introduced by researchers amounts to the most variance in defect prediction model performance. So the identity of the person conducting the work is often more significant than the work itself. Additionally, the concept of a research group consists of a variety of factors, including prior knowledge, statistical and data processing skills, interests, opportunities, and relationships with practitioners, among others [SBH14].

Replicating machine learning studies poses unique challenges. Machine learning models make use of both learned parameters and parameters set manually, so-called hyperparameters. Furthermore, models make use of randomness, changing variables like seed values for random parameters, which can inflate a model's performance up to twofold. Hyperparameters of models also are not standardized between libraries, by using different libraries, vastly different conclusions can be reached [HIB+18, BN06]. When aiming for replicability three main points should be documented, method used, data used, and experiment setup [GK18].

Even though empirical software engineering studies based on data retrieved from development repositories are especially suitable for reproduction. They are based on data that can be easily accessible and shared, and on tools that can be used, shared or described in great detail. However, their reproducibility varies from easy to almost impossible. Hence, reproducibility of experiments not only strengthens the original findings but is one of the basic rules in the scientific method [GBR12, Rob10].

CHAPTER 4

Data Collection and Labeling

In this chapter, we will describe how our data was collected, starting with a Methodology section outlining, on a theoretical level, how a data set replication should be done. Furthermore, points we need to tackle for our specific data set. After that, a practical description of the collection process is given. Then we will describe the features used in the underlying study followed by a short analysis of our data.

4.1 Methodology

4.1.1 Feature Engineering

For replication, the same features must be used. This ensures that the replicated study closely mimics the original. By using the same features results of the replication can directly be compared with the underlying study allowing a comprehensive assessment. By keeping the features constant we can validate the robustness and reliability of the initial findings [BID+14, RGS+18]. Furthermore, when replicating a dataset, besides aiming for feature parity, one should also aim for a matching distribution of variables. When replicating a data set there is a chance to introduce statistical bias. When replicating a data set distribution shifts in parameters that impact task performance must be controlled [EIS+20].

We not only need to aim for feature parity but also for the same distribution of variables. Below we will describe all Features the underlying study is using. Facebook is not providing an analysis of how features in the data set are distributed. However, in some cases, we can make educated guesses. For example, tests tend to succeed more often than they fail, hence a high failure rate for almost all tests in every commit is highly unlikely. To aid reproducibility we will analyze how our test cases are distributed, furthermore, we will provide all data that is used in this study.

4.1.2 Tackling Test Flakiness

Test Flakiness describes a situation where a test produces inconsistent and unpredictable results across multiple test runs, even though the underlying System did not change, see 2.3 for greater details on test flakiness. The underlying study only employs retry bases deflaking, which may not detect all forms of flakiness. However, to ensure reproducibility we are employing the same methodology. Unsuccessful tests are retried a certain amount. If during these retries the test is successful at least once, it gets recorded as such. This to some degree shall help our models to predict only true test failures. Without flakiness detection, we risk injection noise into our dataset and training the model to detect flaky tests, rather than those that failed because of a change. [MSPC19].

4.1.3 Time Series Data

We will collect our data from commits of open-source repositories. Such repositories represent a sequence of data points indexed in time order. Time series data can be seen as a collection of random variables indexed according to the order they are obtained in time [SSS00], we will describe how we will use this data for our experiment setup in chapter 5. Hence, in addition to the features of the underlying study, we will add a timestamp for every data point (commit) to our data set. This will ensure that we can reconstruct the order of commits independent of how we handle the data later.

4.2 Data Collection

Our data collection is split into the following sections, first we will describe how our training data is collected, after that we will describe how we collect additional data to later compare our models result to analysis bases test selection. After describing how we collect our data we will describe all the features that are present in our data set in detail.

4.2.1 Data Collection

The underlying study [MSPC19] collects features from different projects in different programming languages. Since programming a data collection tool that works with different project dependency management tools, programming languages and build automation tools we will only collect data from Java projects that are using maven for their build and dependency management[1].

The data collection will be as close to the underlying study as possible. However, the underlying study describes its data collection process as highly abstract, hence during our reproduction, we needed to make assumptions, based on the described features, and on how they are collected. First, let us depict a simple arbitrary git tree of some project.

[1]Although a rudimentary Gradle implementation is present in our tool, it is untested and may not work for a wide scope of proejcts

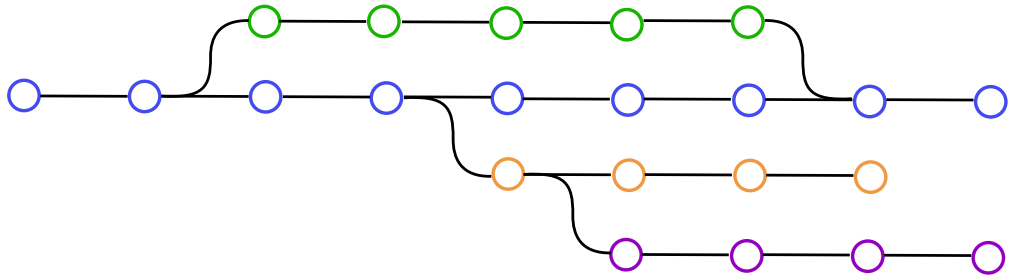

Figure 4.1: A simple arbitrary git graph, with colored nodes, to illustrate an initial situation of the data collection algorithm, commits on the same branch are given the same color.

On a high level, we walk a flattened git graph, sequentially by date, and collect data for each commit yielding a flattened graph like depicted in figure 4.2. In the first step, we fetch all commits and sort them by commit time. Building each commit, and collecting features described in section 4.2.3.

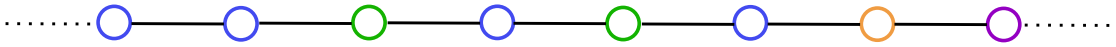

Figure 4.2: Git graph of figure 4.1 flattened and commits sorted by commit time.

To collect all features bookkeeping between commits is necessary, and for each commit, after a successful build, we need to compute a sequence of steps. The Java compiler will yield one or more *.class* files per *.java* file, every such class file represents a node in a dependency graph depicted in 4.3. In the following, we will call *.class* files in the graph class nodes, and *.class* files containing Test annotations test nodes.

For every commit we reset all dependencies between nodes and add new nodes to a registry if we detect new files, then all nodes are connected, forming a dependency graph. A node is connected if they are directly dependent on each other, we interpret a dependency as an import inside a Java file. For each node, we compute which node has changed according to *diff* information for the current commit provided by git. A node is the target of a change if the *.java* file of the corresponding *.class* file is changed.

For each changed node we recursively walk the dependency graph computing the shortest path to each affected test node. After computing the set of affected tests of a commit we run all affected tests. The test result as well as the collected change information is used to compute the various change vectors and features, as described in detail in the next section 4.2.3. Each affected test can have multiple class nodes that depend on it. Hence, a test may yield several data points with different paths and change vectors for each changed file. After every commit all data points are added to the dataset, this allows us to recover from a crash.

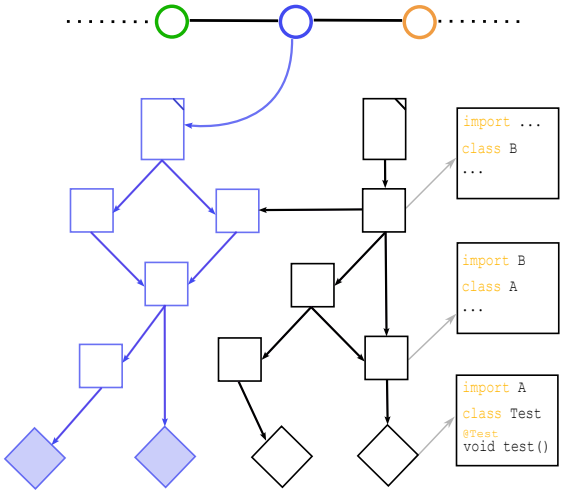

Figure 4.3: A Dependency graph, let square nodes depict *.class* files and diamond nodes *.class* files containing at least one test. An arrow connects two entities if they are directly dependent on each other. We interpret a dependency as an import inside a Java class. Assume the blue commit changes a *.java* file, modifying the blue *.class* nodes of the graph. This impacts all transitively dependent nodes. A trivial test selection strategy would then only run tests (diamond nodes) that we fully colored in the graph, ignoring the uncolored test nodes [MSPC19].

4.2.2 Analysis Based Test Selection

Furthermore, to later compare our trained models we connect which test a traditional test selection algorithm would have selected. For this, we implemented a variation of STARTS [LSM17] selection algorithm into our data collection tool. Similar to the approach discussed until now STARTS works by first building the project and computing a dependency graph of the compiled *.class* files.

When test selecting with STARTS[2] [LSM17] after computing the dependency graph for every *.class* file affected by a change the class files are then hashed using EKTAZIs[3] [GEM15a] hashing algorithm. This algorithm filters out changes that do not impact program behavior[4], by cleaning up the read in byte array of a *.class* file before hashing.

As depicted in figure 4.4 the newly computed hashes are then compared to the last saved hash of the corresponding file. If the newly computed hash is not equal to the last saved hash all dependent test nodes in the dependency graph are selected for test execution. If the hashes are equal the corresponding test nodes are ignored.

[2]2025: STARTS `https://github.com/TestingResearchIllinois/starts`
[3]2025: EKTAZI `https://github.com/gliga/ekstazi`
[4]Mainly whitespace, comments, java doc, method ordering, import statement ordering. This can be found in STARTS *CheckSumUtil* that uses EKTAZIs *Hasher* to compute its hashes.

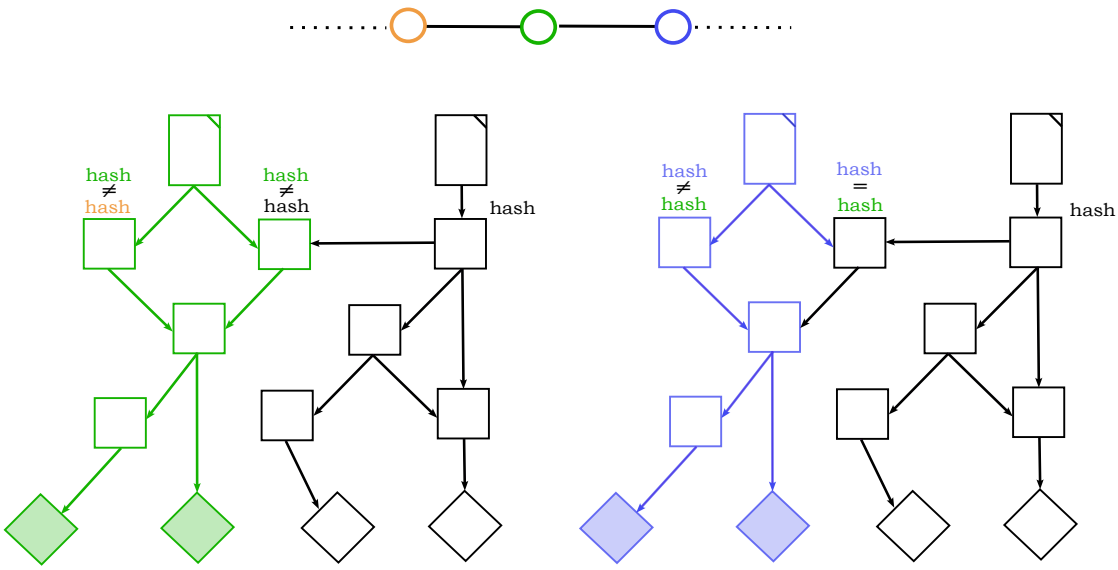

Figure 4.4: The commits are depicted as colored circles, furthermore, two dependency graphs, equal to figure 4.3, are depicted as affected by the blue and green commit. The file changes of a corresponding commit, are depicted as files colored in the color of the change. With hashes that are saved per *.class* file. The hashes on the bottom depict the saved hashes for each file. The hashes on the top depict the newly computed hashes. Each hash is colored in the color of the last change that updated its value.

To speed up execution time STARTS does save a class to test mapping between executions, this allows them to not compute the dependency graph for every execution. STARTS then computes all tests not impacted by a change from these mappings, then computes the impacted tests as the difference between the set of not affected tests and the set of all tests. Our implementation does not do that since, for every execution, we compute a new dependency graph for data collection. For further details on the implementation of STARTS and the resulting differences to our variation view Legusons et. al.'s paper [LSM17].

Since this approach is really similar to our training data collection tool, aside from additional book-keeping we added the described hashing capabilities to our tool. This allows us to additionally collect a list of tests that would be selected by an analysis based test selection approach. We decided against using STARTS directly since it is implemented[5] as Maven plugin. This would restrict us in future extensibility of our tool to other build tools. Furthermore, it is not trivial to inject a Maven plugin into the build of arbitrary projects. In the next section, we will describe the features we are collecting in greater detail.

[5]2025: STARTS `https://github.com/TestingResearchIllinois/starts`

4.2.3 Collected Features

The model inputs, consisting of a change and a test target, offer a natural framework for categorizing features into distinct types: those dependent on the change and target individually, as well as those representing cross-interactions between the two [MSPC19]. The following features are collected by the underlying study:

Change level features:

- *Change History of Files* a vector of the number of changes made to modified files within 3, 14, and 56 days.
- *File Cardinality* Number of files touched by a change. Large changes are harder to review, and we assume they are more error-prone.
- *Test Cardinality* The number of test classes triggered by a change.
- *Extension of Files* We do not collect this feature explicitly, because we are only analyzing Java projects in this thesis. Therefore, this feature is given implicitly.
- *Number of Distinct Authors* collected per class file, might indicate unstable or commonly used code..

Test level features:

- *Historical Failure Rate* of a test class, as baseline probability of failures, as a vector of failure rates in the last 7, 14, 28 and 56 days.
- *Project Name* to identify an area the test class covers and categorize breakage patterns based on a project.
- *Number of Tests* in a test class can be used as a proxy of the code area covered by the test class.

Cross-dependent features:

- *Minimal Distance* between a java class changed by a commit and the test class to predict the outcome of. To approximate the significance of a change's impact on a given test class.
- *Number of Common Tokens Shared* in the path of a test class and a java class as lexical distance, to approximate human perceived relevance[6].

[6]The underlying study does not state how they calculate lexical distance. We calculate the lexical distance using Jaccard similarity [NSNW13].

Facebook's study does not make use of all the features described above, however, we want to aim for feature parity [EIS+20]. We are collecting additional data for each commit we analyze. This data will not be used for model training. But helps us debug our data collection, can be used in further research and will be used to evaluate our trained models against an analysis based test selection approach.

Change level features:

- *Full Target Class Name* will be used to compute *Number of Common Tokens Shared*
- *Starts would select* if the target had been selected by STARTS.
- *Time of Change* the time of the commit.

Test level features:

- *Full Test Class Name* will be used to compute *Number of Common Tokens Shared*
- *Number of All Test Classes* the number of all test classes in a project at the time of the change.
- *Changed Files* a list of all files affected by a change.
- *STARTS selected Test Classes* a list of full class names of all test classes that STARTS would select.
- *STARTS selected Changes* a list of full class names of all *.class* files that STARTS would select after byte code filtering.
- *STARTS path to tTarget* a list of full class names of all *.class* files building the shortest path to the STARTS selected test target.
- *Change impacted Test Classes* a list of full class names of all test classes that are impacted by a change.
- *Selected Changes* a list of full class names of all *.class* files impacted by a change.
- *Path to Target* a list of full class names of all *.class* building the shortest path to the test target.

Cross-dependent features:

- *Build Errors* of commits with their error message and stack trace.

Furthermore, these additional features make sure that we destroy as little information as possible, and ensures computations like lexicographical distance or shortest path can be debugged easily or even changed in the future, since the underlying data still exists in the data-set. In the following section, we will list the differences and limitations of our data collection algorithm, in contrast to the underlying study.

4.3 Postprocessing

In the process of data collection, our tool can exclude test cases that are encountering timeouts or exceptions stemming from attempts to retrieve resources that do not exist anymore. This minimizes the amount of post-processing we need to deploy. We are focusing on the following anomalies:

- Points in time after which all following commits and tests fail.
- Build failures[7].

Run them again and make sure that they fail because of the changes made and not because of our setup[8].

4.4 Label distribution

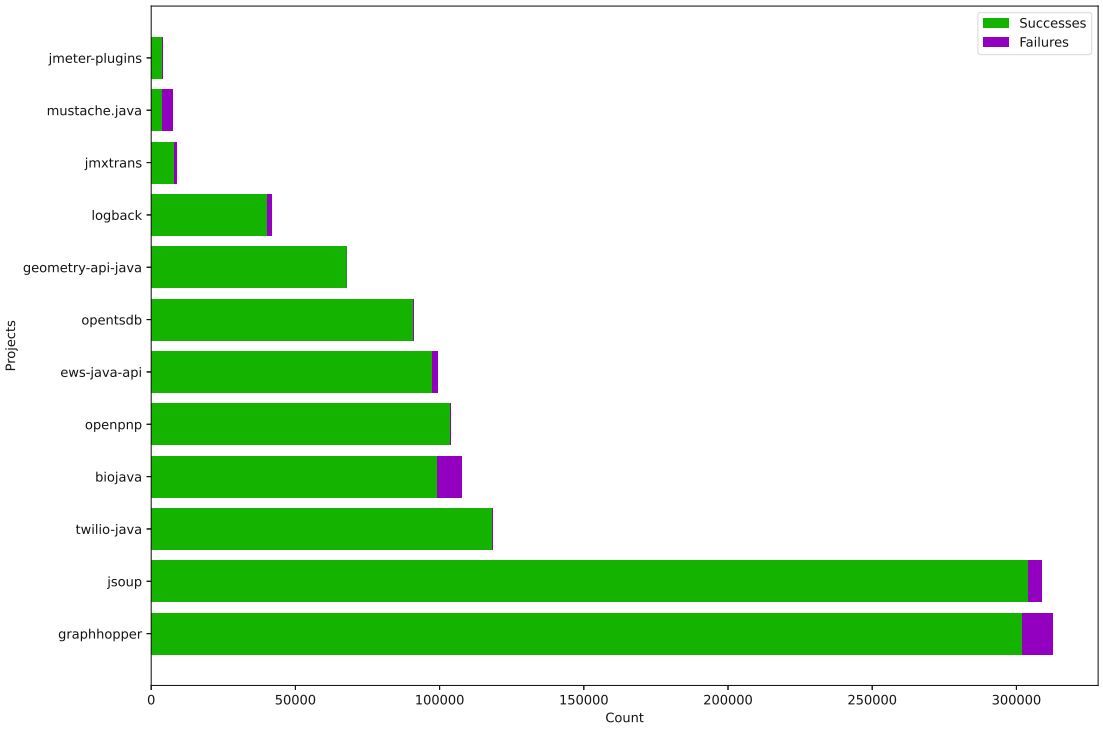

Figure 4.5: Showing the amount of failed and succeeded test cases retrieved from all analyzed projects. With blue being successes, and purple being failed tests.

[7]During our runs we had to introduce build, environment, JVM arguments, to make sure we can at least build (almost) all commits, to skip, for example, maven phases that always fail the builds.

[8]E.g., because of a dependency which is not available in the default maven repository anymore, changed JDK versions, or resources that are not downloadable anymore.

In the figure above we see that the distribution of succeeded and failed tests as well as naming all projects we are gathering data from. Overall we gathered about 1.2 million tests of which approximately 30000 are failing. Although most of the projects have at least some test failures, they are unevenly distributed between them. Furthermore, like discussed before, our classification data is imbalanced.

4.4.1 Limitations of Our Collection Tool

- We do not have the same level of control over the observed Projects, hence, in contrast to the underlying study [MSPC19], we do not run tests that are affected by a dependency change. We also cannot fix builds or react to missing dependencies.
- Commits that cannot be built will be ignored, hence in some cases, the collected change vectors may differ from reality.
- Since git allows rewriting its history there is no best practice how to emulate the real development flow. Like depicted in figure 4.2 we do walk commits like a flattened git graph, since tests are more likely to fail on development branches. Walking the graph like this does not allow us to compute changes out of checksums since commits that follow each other do not necessarily be from the same branch. Therefore, we compute files impacted by a change from gits change information, even before the STARTS hashing takes place, which is not part of STARTS algorithm.
- We encountered several antipatterns[9] in tests, which will make post-processing necessary. We describe it in more detail in section 4.3.
- To avoid possible side effects in tests[10] we run each test separately. This entails spinning up a separate process, JVM and Maven run for each test, which costs a lot of time.
- We do use OpenJDK versions to build the projects. We do not automatically switch the JDK version if it changes in the project during data collection. However, it is possible to recover manually if this is the case. Although we implemented some rudimentary Gradle support, the tool works best on maven based projects.
- We ignore integration tests if they are easily identifiable by pattern matching.

In the next chapter we will discuss the Methodology used to replicate the results of the underlying study [MSPC19].

[9] Tests that will fail if a web resource they depend on is not available anymore, insufficient mocking of dependent resources, unit tests that are integration tests and many more.

[10] We set a generous timeout for each test run, however, we encountered tests that do not clean up properly and trigger these timeouts regularly. Tests that block a resource that is then needed by another test or even tests that crash the JVM.

CHAPTER 5

Methodology

5.1 Data Preparation

To ensure comparability between our work and other papers [Lun19, ZLG+22], we will use the same performance measurement as the underlying study [MSPC19], to compare and evaluate the different machine learning methods. This chapter will first explain how we handle our collected data, then how our models are trained, and finally lists the evaluation metrics used for our final comparison.

5.1.1 Tackling Imbalanced Data

Software engineers rarely introduce code changes that fail tests in a repository, at least the unit tests directly related to a change will be run locally until they pass. This results in a great imbalance in our data set between passed and failed tests. The classifier used in the underlying study, XGBoost, works out of the box on such data without further changes to the data set [MSPC19]. However, since we want to compare other classifiers we will test methods to tackle imbalanced data, also used in defect prediction [TTDM15]. The following methods are tested:

- *Original data* Use the data set as is.
- *Over-sampling* Either randomly duplicate or synthetically create data points from the minority class to match the size of the majority class.
- *Under-sampling* Randomly sample a subset of the majority class to match the size of the minority class.
- *SMOTE* Synthetic over sampling based on k-nearest neighbors [CBHK02].
- *ADASYN* Synthetic over sampling similar to SMOTE but focuses on difficult to learn examples [HBGL08].

5.1.2 Handling Time Series Data

Historically regression test selection performance is measured by letting an algorithm select tests on a code base injected with random mutants and comparing the algorithm selection with the tests that fail due to the injected mutants. As described below, the data we are using is highly time-sensitive and depends on the development of a repository. By collecting a dataset using mutant injection to simulate commits, the data set will lose information that is hidden in an organically evolved repository. In spite of that, this approach is employed by papers that are using machine learning for test case selection [Lun19, ZLG+22].

Although we did an extensive literature review, we did not find related work that dives into the caveats of time-sensitive evaluation for regression test selection by machine learning models. Like in the underlying study [MSPC19], in the highly related field of software defect prediction, it is common to collect data from the commits of a repository. We will use knowledge from this field for our experiment setup.

Because software repositories are evolving over time, shuffling data, like in a standard cross-validation approach[BB12], would induce bias by allowing the model to make predictions on past events with information from the future. As described in chapter4 we are using different vectors representing the history of a file's defects, the model could infer that this file is susceptible to problems even before the regression occurred. Hence, we split our data by date [TTDM15].

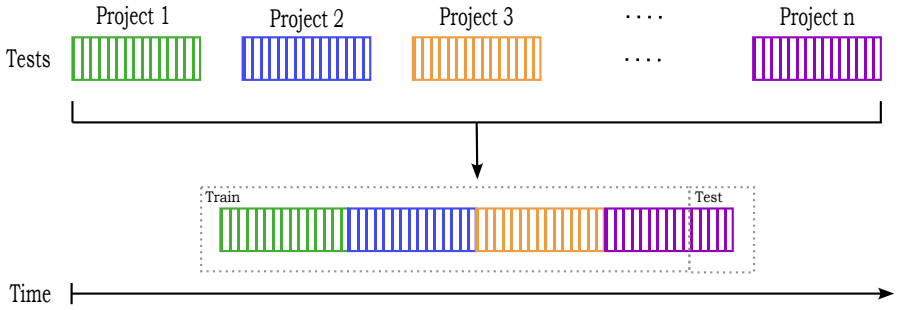

Figure 5.1: Worst case scenario when combining projects test data.

When combining the collected data from different projects, we must keep the different development cycles in mind. In a worst-case scenario, when sorting tests by time each project could have stopped committing one after the other. Resulting in a test timeline where the tests of each project do not intersect with each other, as depicted in figure 5.1.

This would leave the time series intact but lead to at least an overrepresentation of one or more projects in the test set, while other projects may not be present at all. To get around this we will split each project data set into chunks consisting of 5% of its data. The data in these chunks and the chunks overall are ordered by time. After that, we reassemble these chunks as depicted in 5.3.

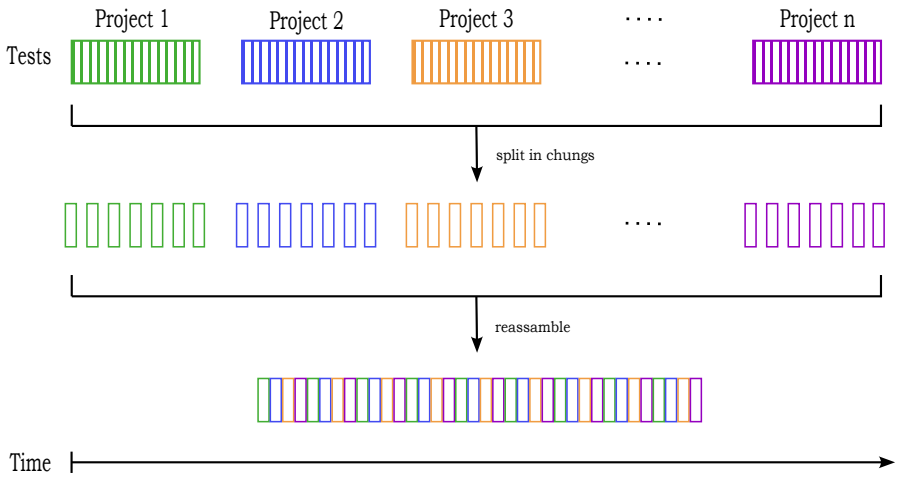

Figure 5.2: Combining project data into a dataset used for training.

This will interrupt the overall time series but leave the time series of each project intact. Hence, the model will still be unable to predict past events with information from the future on a by-project basis. When splitting the data into a training and test set our reassembled data makes sure that we have unseen data from each project in the test set. Furthermore, it ensures that we can extract the same amount of data from each project as from the newly compiled dataset to test the model per project without the risk of using training data for evaluation.

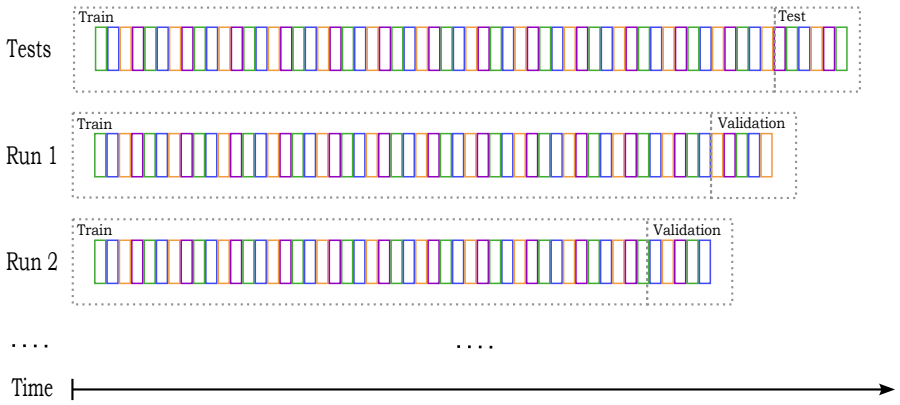

Figure 5.3: Visualization of data split for training and hyperparameter optimization.

After compiling our dataset, we keep the first 90% of the tests for training and use the most recent 10% for testing for every project. Moreover, the training set undergoes multiple divisions into two separate sets, into training and evaluation sets. This setup is often used for time series evaluation [BB12]. We split the data 5 times in total. One set is used for model fitting, while the smaller one serves to test the model's performance on future tests.

This second set is referred to as the validation set and is designed to match the size of the test set. The evaluation runs will be used to identify the optimal hyperparameters for each model. After all evaluation runs, the final evaluation of a model's performance is done on the test set, from which no information was used for training. This approach is similar to cross-validation with the difference that the validation sets are subsets of each other, and the predictions are only made in the future [BB12]. See Figure 5.3 for a visual description of the pipeline.

5.2 Parameter Tuning and Model Selection

5.2.1 Performance Measure

In order to assess and compare the performance of the different machine learning techniques, we utilize five standard classification metrics: precision, recall, F1-score, average precision, and area under the ROC curve [HS15]. These metrics are computed based on the following values:

- *True Positive (TP)* a test that will fail when executed correctly predicted.
- *True Negative (TN)* a test that will be successful when executed correctly predicted.
- *False Positive (FP)* a test that will be successful when executed predicted as failure.
- *False Negative (FN)* a test that will fail when executed predicted as successful.

Precision quantifies the proportion of true positive predictions out of all the instances predicted as positive. This is desirable, predicting fewer false positives, means it is more likely that tests are selected that are failing when executed. Our main goal is to run as few tests as possible while, ideally, predicting all tests that will fail when being executed. Precision is given by:

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

Recall, also known as sensitivity or true positive rate, is a metric used to assess a machine learning model's ability to correctly identify all relevant instances within the positive class. A high Recall indicated that the model correctly identifies most test cases that fail when executed. It is more important to find all failing test cases than to execute a few that may succeed. Recall is given by:

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{P} \tag{5.2}$$

Precision or recall on itself may not provide a complete measure of model performance. Since it is possible to artificially achieve a perfect recall by predicting only the positive

class or perfect precision by predicting the positive class for just a single data point with high certainty. In practice, the goal is to optimize both metrics simultaneously. To evaluate the model's overall predictive ability. To achieve this, the F1-score is commonly used, which balances precision and recall, by using the harmonic mean between them. F1 Score is given by:

$$F1 = \frac{2 \cdot Percision \cdot Recall}{Percision + Recall} \tag{5.3}$$

Many machine learning models allow predictions at varying confidence levels, often referred to as thresholds. For instance, when predictions are made only with high certainty, precision increases, but recall typically decreases as many true positives are missed. Conversely, lowering the confidence threshold leads to higher recall but reduced precision. At an extreme, a perfect recall can be achieved with a confidence threshold of zero. This trade-off between precision and recall affects the F1-score, which varies depending on the chosen threshold. To evaluate model performance and identify the best-performing models, average precision (AP) is commonly used as a comparative measure. Average precision is given by a sum over all thresholds t, summing up precision at a threshold P_t, weighted by the gain at the gain in recall $R_t - R_{t-1}$ at said threshold:

$$Average\ Percision = \sum_{t \in thresholds} (R_t - R_{t-1}) \cdot P_t \tag{5.4}$$

Additionally, we report the Receiver Operating Characteristic (AUC) curve, which is a widely used graphical representation of a classification model's performance across different decision thresholds. It plots the true positive rate (recall) against the false positive rate, which is defined as:

$$ROC\ Curve = \{(FPR_t, TPR_t) : t \in thresholds\} \tag{5.5}$$

Where the true positive rate $TPR_t = TP_t/P$ and false positive rate $FPR_t = FP_t/P$ at a given threshold t, respectively. Where TP_t and FP_t represent the true positives and false positives at threshold t, and P is the total number of actual positive instances. The area under the ROC curve (AUC) is then computed, where an AUC value of 0.5 indicates random guessing, and a value of 1.0 reflects perfect classification performance at every threshold. Since our data is imbalanced[1], a model predicting that all tests fail and therefore need to be run would score a high accuracy $(TP + TN)/(P + N)$, but would be completely useless in practice. Hence, we will not use accuracy scoring in our evaluation.

[1]See sections 5.1.1 4.4 for further details

5.2.2 Feature Selection

We already discussed in section 4.1.1 that our dataset aims for feature parity, by collecting all features of the underlying study [MSPC19]. In a first step we will use all features to train our models, using the best hyperparameters we find according to section 5.2.3. The trained models will be scored according to the performance measures in the previous section. The best model will be used to further analyze performance.

To achieve this we will use SHAP (Shapley Additive Explanations) [LL17] like discussed in section 2.6. This will not only allow us to select the best performing features, but additional do an interpretability analysis of our best performing model. We will then train the models again on the reduced feature space. The best performing model trained on the reduced feature space will then be used for our final performance analysis, and further interpretability discussions. To perform the SHAP analysis the best performing model will be trained with default hyperparameters.

According to the underlying study [MSPC19] the best performing features are *file extensions*, *change history*, *failure rates*, *project name*, *number of tests* and *minimal distance*. Since we have collected real world data like discussed in chapter 4, and did our best to aim for a similar feature distribution, we are expecting to match at least some if not all the best performing features.

5.2.3 Hyper-Parameter Optimization

Hyperparameter optimization can significantly improve model performance [MC24]. We evaluate of different sampling techniques introduced in section 5.1.1. The implementations for logistic regression, support vector machine, random forest, and multi-layer perceptron are sourced from the open-source library scikit-learn [PVG+11]. Additionally, the XGBoost Python package is utilized, as it is compatible with the scikit-learn interface. XGBoost is the classifier we are most interested in since it is the classifier the underlying study is using [MSPC19].

To incorporate the sampling methods into the machine learning pipeline, the scikit-learn extension imbalanced-learn is employed [LNA17]. The interoperability of these libraries with scikit-learn facilitates hyperparameter optimization and the selection of the most suitable sampling method within a unified framework. The different sampling methods that are contained in the search space for every model are listed in the table 5.1 below.

A sequential model-based Bayesian optimization algorithm is utilized from the scikit-learn extension[2]. This approach enables efficient sampling of the parameter search space by leveraging a Bayesian model, reducing the need for exhaustive grid search or less efficient randomized search. The complete configuration of the parameter search space is presented in table 5.1. For all models, 100 samples are drawn from the search space and evaluated. For evaluation average precision scoring is used.

[2] 2025: `https://scikit-optimize.github.io/stable/`

Parameter	Search Space
Sampling	None, Under-Sampling, Over-Sampling, SMOTE, ADASYN
Logistic Regression	
Regularization parameter C	LogUniform([10^{-4},10^{4}])
Penalty	L1, L2
Support Vector Machine	
Regularization parameter C	LogUniform([10^{-4},10^{4}])
Kernel	Linear, Radial Basis Functions, Polynomial
Penalty	3, 5
Random Forest	
Maximum Depth	Unbound, 3, 5, 10 15 20
Minimum Samples to split Leaf	2, 5
Number of Estimators	UniformInteger([5, 150])
Multi Layer Perceptron	
Regularization Parameter α	LogUniform([10^{-4},10^{4}])
Initial Learning Rate	LogUniform([10^{-4},10^{-1}])
Hidden Layer Size	10, 20, 50, 100, 200, 500
Activation Function	ReLU, Sigmoid, tanh
XGBoost	
Maximum Depth	UniformInteger([2, 20])
Minimum Child Weight	1, 2, 5
Maximum Delta Step	0, 1
Number of Estimators	UniformInteger([5, 150])
Regularization Parameter γ	Uniform([0, 1])

Table 5.1: Search spaces for hyperparameter optimization.

5.3 Selection Performance Measures

We use the best hyperparameters determined by our search to train the models. Afterward, we use Average Precision, Receiver Operating Characteristic (AUC) curve[3], and feature selection[4] to select and further optimize the best performing model. Finally, we will use the performance metrics of the underlying study to compare the best performing XGBoost model against our implementation of *STARTS* [LSM17]. *STARTS* is an analysis based state-of-the-art test selection approach we explained in section 4.2.2. For comparison, we will use the metrics provided in the underlying study [MSPC19].

[3]As described in section 5.2.1
[4]As described in section 5.2.2

As described in section 2.4 a trained classifier on our dataset returns a $Score(d,t) \in [0,1]$ for a change d and a test t, which can be interpreted as likelihood of $t \in FailedTests(d)$. Futhermore, we denote $FailedTests(d)$ as the set of all failed test of a change d, and $DependendTests(d)$ as all tests that are dependent on a change d. For our test selection strategy s we introduce a $ScoreCutoff(s) \in [0,1]$ which resembles the threshold we previously explained for our performance measures. Furthermore, we introduce a $CountCutoff(s) \in \mathbb{N}_{\geq 0}$ which resembles the number of top performing targets for a change.

For the following performance metrics we will analyze tests that are $LikelyFailing(s,d)$ which contains all tests $t \in DependendTests(d)$ for which $Score(d,t) \geq ScoreCutoff(s)$. $HighlyRanked(s,d)$ which contains up to $CountCutoff(s)$ of $t \in DependendTests(d)$ with highest $ScoreCutoff(s)$. In the underlying study $SelectedTests(s,d)$ consists out of $LikelyFailing(s,d) \cup HighlyRanked(s,d)$. We will however analyze both sets independently to better compare them to the set of tests that *STARTS* would select. Hence, in our Results $SelectedTests(s,d)$ will not depict the union of both sets but one of each independently. We will use the following metrics for comparison:

$$TestRecall(s,D) = \frac{\sum_{d \in D} |SelectedTests(s,d) \ \cap \ F_d|}{\sum_{d \in D} |F_d|} \tag{5.6}$$

Where d is a change in the set of all changes D and $F_d = FailedTests(d), \exists_{d \in D} F_d \neq \emptyset$, and s is a test selection strategy. Test recall mirrors the recall metric we already explained in section 5.2.1. Test recall can be understood as the empirical probability of a given test selection strategy identifying an individual failure.

$$ChangeRecall(s,D) = \frac{|\{d \in D\} | SelectedTests(s,d) \cap F_d \neq \emptyset|}{|\{d \in D\} | F_d \neq \emptyset|} \tag{5.7}$$

Where d is a change in the set of all changes D and $F_d = FailedTests(d), \exists_{d \in D} F_d \neq \emptyset$, and s is a test selection strategy. Intuitively, change recall represents the empirical probability that a specific test selection strategy will detect at least one failure in a faulty code change. Or in other words the relation of all faulty changes containing at least one true positive predicted test to all faulty changes containing at least one failed test from the test set.

$$SelectionRate(s,D) = \frac{\sum_{d \in D} SelectedTests(s,d)}{\sum_{d \in D} DependendTests(d)} \tag{5.8}$$

Where d is a change in the set of all changes D, s is a test selection strategy. The selection rate reflects the proportion of test targets chosen by a specific strategy in comparison to those selected by the build-dependency-based approach. After computing all the evaluation metrics—such as change recall, test recall, and selection rate—we will compare the results against those obtained using the STARTS approach.

This comparison will help assess the effectiveness and efficiency of our test selection strategy relative to an established baseline.

CHAPTER 6

Results

This chapter presents the results of our replication study, which aims to validate the model selection of the underlying study. We first evaluate the models' classification scores and compare them to other models. To assess feature importance, we use SHAP (SHapley Additive Explanations) values, comparing them with the original study's reported feature rankings. Finally, we will compare our trained model to analysis based model selection, using the performance metrics of the underlying study.

6.1 Model Comparison

To evaluate the predictive performance of the XGBoost model in comparison with alternative classifiers, we report the average precision (AP) and area under the receiver operating characteristic curve (ROC AUC) scores in table 6.1. We report these metrics on the training set and test set, where the models are trained on the training data set containing all features. Additionally, we report the average scores of the 5 validation splits used to select the best hyperparameters for each model. For reference, we report the scores of a dummy classifier which by definition has perfect recall, and a 0.5 AUC Score.

The results indicate that all models perform similarly in terms of both average precision (AP) and ROC AUC scores, with only marginal differences across classifiers. Among them, Random Forest achieves the highest performance, closely followed by XGBoost. The similar performance of Random Forest and XGBoost is expected, as both models are tree-based ensemble methods that effectively capture complex feature interactions and handle non-linear relationships. We can assume that the primary motivation of choosing XGBoost was not performance, but its handling of imbalanced data, and its efficient computation through gradient boosting. Trading a small performance decrease for faster (re)training and inference [CG16].

	train		validation		test	
	AP	AUC	AP	AUC	AP	AUC
dummy classifier	0.025597	0.500000	0.021984	0.500000	0.025168	0.500000
logistic regression	0.861135	0.978375	0.864471	0.976552	0.950223	0.981493
multi layer perceptron	0.879846	0.987935	0.819965	0.975897	0.948529	0.979035
random forest	**0.9392**	**0.9929**	**0.8886**	0.973220	**0.9590**	**0.9875**
support vector machine	0.881557	0.979648	0.852743	0.953754	0.914072	0.984090
XGBoost	0.895560	0.991600	0.880514	**0.9790**	0.947403	0.985290

Table 6.1: Results of fitted models after hyperparameter tuning.

In addition to the reported aggregated metrics we report the ROC-curve and precision-recall curve in figure 6.1, to show the full classification performance at different confidence thresholds. We see that we reach an almost perfect ROC-curve with an almost perfect precision-recall curve for all classifiers, besides the support vector machine.

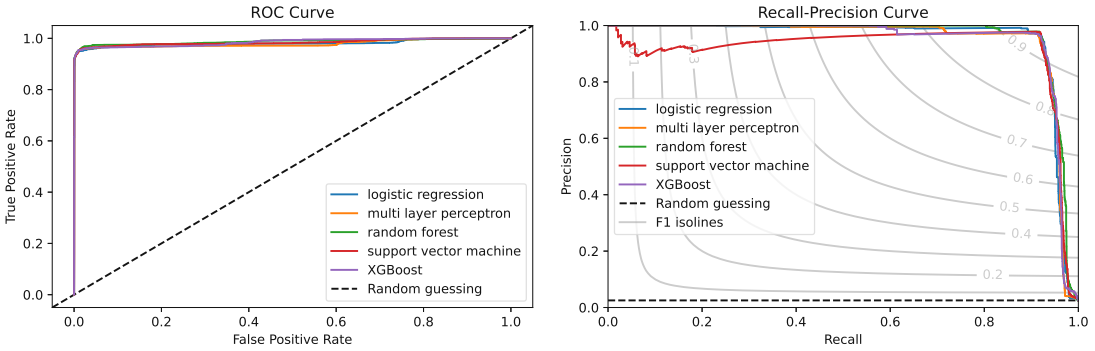

Figure 6.1: ROC and precision-recall curves for all features.

Because the performance metrics obtained appear unexpectedly high, we have concerns about potential overfitting or data limitations. The consistently strong scores on the training, validation and test sets suggest that the model may be overly specialized. This could stem from the relatively small dataset of approximately 1.3 million data points, which may not provide sufficient variability. However, since our dataset is comprised out of several thousand commits and multiple projects we do think that our results are meaningful.

Despite XGBoost not achieving the highest performance in our evaluation, we will continue our analysis using this model. The decision is justified by the overall similarity in performance across all classifiers, with only marginal differences in average precision (AP) and ROC AUC scores. Given that Random Forest slightly outperforms XGBoost, the choice to proceed with XGBoost is primarily motivated by its use in the underlying study [MSPC19], ensuring methodological consistency and facilitating direct comparability of results.

To further investigate overfitting in our XGBoost classifier, we will plot a confusion matrix. This will provide a detailed breakdown of the model's predictions, highlighting instances of misclassification. By examining the confusion matrix, we can assess whether the model is overfitting by identifying patterns such as high accuracy on the training set but poor generalization to the test set.

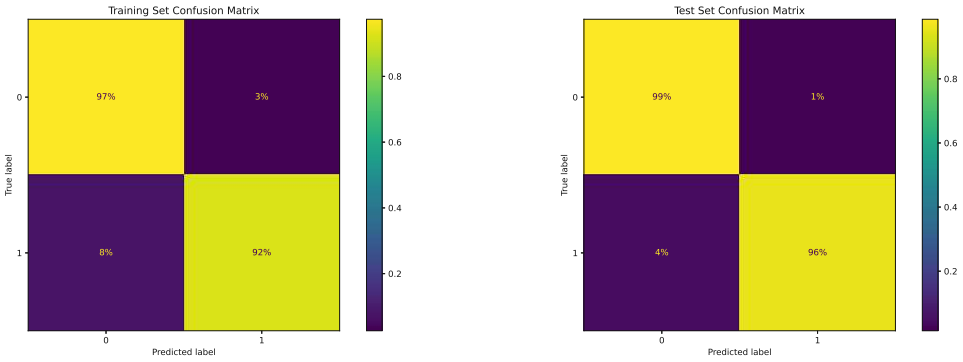

Figure 6.2: Confusion Matrix of the XGBoost Classifier trained on all features. Left predicting on the trainings data set, right predicting on the test data set.

A perfect confusion matrix occurs when a classifier makes no errors, meaning all predictions are correct. In this case, the matrix will have nonzero values only along the diagonal, where each row corresponds to the actual class and each column to the predicted class. Off-diagonal elements, representing misclassifications, will be zero. This indicates 100% accuracy, precision, recall, and F1-score, which is ideal but often unrealistic in real-world scenarios due to noise, class overlap, and data variability [AM22].

In case of overfitting, we would see a near-perfect, or suitable confusion matrix built from the predictions on the training data and a bad or worse confusion matrix on the test data. As depicted in figure 6.2, we can see that we reach almost identical matrices on both sets. With the test set matrix being slightly better. However, we should stay cautious about overfitting, since as described in the next section, our best feature is *project name*. This may hint that we are prone to overfitting since the *project name* is a categorical value[1]. This would become especially apparent after a new project is introduced.

In the next section, we will analyze the importance of each feature and how much it contributes to our XGBoost model prediction. With this information we can compare our most important features with the ones from the underlying study [MSPC19].

[1]A categorical value refers to a type of data that represents distinct groups or categories rather than numerical values. These values can be nominal (e.g., colors) or ordinal (e.g., education level). The models used to measure performance are trained on a dataset where the *project name* is one-hot encoded, while for the SHAP plot, it is label encoded. Tree-based classifiers are especially prone to overfitting on such values. The algorithm needs to determine the optimal branching decision by analyzing the proportion of each categorical attribute's values in relation to the target attribute [KK11].

6.2 Feature Importance and Interpretability

Feature selection is a critical when using machine learning for regression test selection and related tasks, as it directly impacts model performance, computational efficiency, and cost-effectiveness [KAK+25, SWAK12]. By identifying and retaining only the most relevant features, the dimensionality of the input space is reduced, leading to lower computational complexity and faster model training and inference. This, in turn, decreases resource consumption and enhances scalability, this is especially important in regression test selection, since the model is inferred for every test, and the goal of test selection is to reduce the overall runtime and cost.

In this section, we aim to replicate the feature selection process of the underlying study [MSPC19] using SHAP (Shapley Additive Explanations) to identify the most influential features for regression test selection. In addition to replicating the original feature selection, using SHAP provides the added benefit of enabling further interpretability analysis. By examining the distribution and interaction of SHAP values across different test cases, we can gain deeper insights into how individual features contribute to model predictions [LL17].

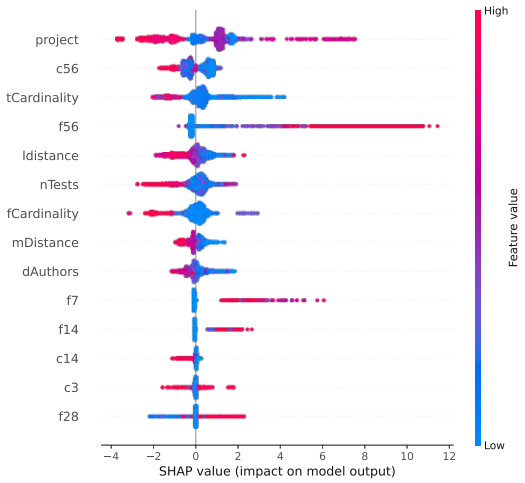

Figure 6.3: Bee swarm plot of features and their impact on model performance of the XGBoost classifier.

Like depicted in figure 6.3 we can see that the best performing features are *project name*, *change history*, *failure rates*, *lexical distance* and *number of tests*. The best-performing features align closely with those identified in the underlying study, with one notable exception. While the original study states *minimal distance* as a key feature, our replication prefers *lexical distance*. Despite this variation the overall consistency of the selected features further strengthens the results of the underlying study. We cannot report how important *file extensions* are, we only analyze *.java* files, hence we do not collect this feature since it is given implicitly.

After dropping *file cardinality*, *number of distinct authors* and *minimal distance*, we did another hyperparameter optimization ensuring that the models remain well calibrated. Following optimization, we retrained the models, using the same pipeline as already described, on the reduced feature set. After training, we collected the same scores as we used in the previous section in table 6.2 and figure 6.4.

	train		validation		test	
	AP	AUC	AP	AUC	AP	AUC
dummy classifier	0.025597	0.500000	0.021984	0.500000	0.025168	0.500000
logistic regression	0.859082	0.976204	0.862128	0.972651	0.954523	0.980096
multi layer perceptron	0.857812	0.982870	0.851078	**0.9821**	0.944660	0.975882
random forest	**0.9338**	**0.9915**	**0.8936**	0.976913	**0.9584**	**0.9844**
support vector machine	0.864453	0.968066	0.866973	0.956475	0.900685	0.983901
XGBoost	0.894446	0.990110	0.879708	0.980222	0.954581	0.982089

Table 6.2: Results of fitted models after hyperparameter tuning

We can again see that all models still perform similar, with XGBoost and random forest still being the best models. We can also report that dropping these features impacts model performance only insignificantly. As depicted in figure 6.4 we still maintain a perfect ROC curve for all classifiers. After training on the reduced feature set, the precision-recall performance slightly declined, especially for lower thresholds, for both the support vector machine and multi layer perceptron model, indicating a minor loss in their ability to correctly identify positive instances.

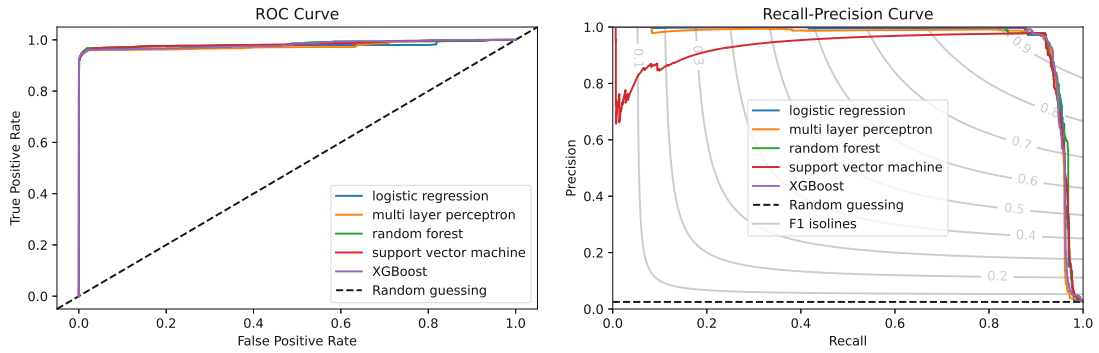

Figure 6.4: ROC and precision-recall curves after feature selection.

In section 8.3 we report additional performance metrics for our models, for before and after our feature selection. In the next section we will report our final findings. We will use the performance metrics of the underlying study [MSPC19] on our XGBoost classifier as well as on our START implementation, this will allow us to compare our trained classifier with an analysis based test selection approach.

6.3 Evaluation

In this evaluation section, we compare the performance metrics of our replication study with those reported in the original research [MSPC19]. The focus of our analysis is on test recall, change recall, and selection rate as described in 5.3. By examining these metrics, we aim to assess how closely our replicated models align with the original results. Since we already showed in previous sections that our performance metrics are nearly perfect, we will focus mainly on the comparison between our trained XGBoost model and *STARTS* [LSM17] an analysis-based test selection approach.

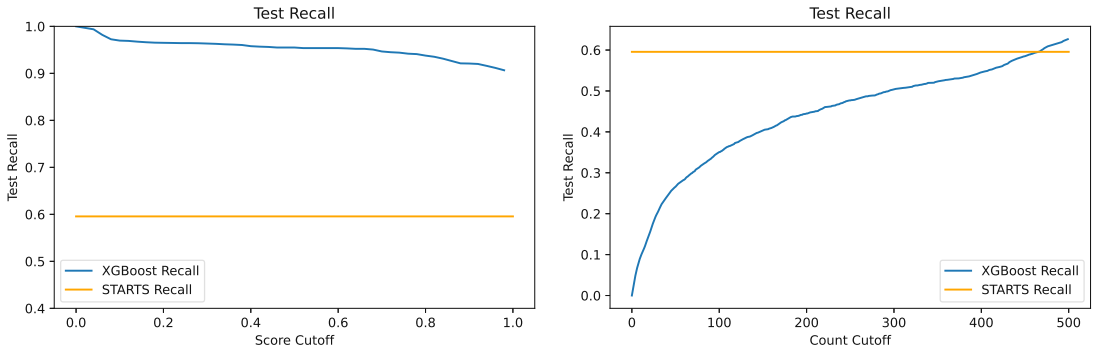

Figure 6.5: Left, $TestRecall(s)$ score as a function of $ScoreCutoff(s)$ of our trained XGBoost model with $CountCutoff(d) = 0$. Right, $TestRecall(s)$ score as a function of $CountCutoff(s)$ of our trained XGBoost model with $ScoreCutoff(d) = 0$. With *STARTS* values in orange.

As shown in Figure 6.5, our approach consistently outperforms *STARTS* in the test $TestRecall$ metric across all values of $ScoreCutoff(s) \in [0, 1]$. Additionally, we observe better performance after applying a $CountCutoff > 500$. Since the final selected test set $SelectedTests(s, d)$ is defined as the union of tests selected by both cutoff strategies, these results indicate that our method outperforms *STARTS* regardless of the chosen cutoff value.

STARTS performs well on $ChangeRecall(s)$ as depicted in figure 6.6. Since we only need to find one true positive change per test to reach a perfect change recall, this is not surprising. It is however counterintuitive that if *STARTS* selects all tests dependent on all changes that do not impact code behavior, that we do not reach perfect $ChangeRecall(s)$. After looking over our data we found an example commit *18f2b6e6625fae157cb427e6038789300138899f9*[2] of biojava, that may explain the slight deprecation in $ChangeRecall(s)$.

[2]2025:https://github.com/biojava/biojava/commit/18f2b6e6625fae157cb427e603878930013899f9

This commit just deletes an empty *else* block, which, after the byte file is cleaned up, will be ignored by *STARTS*. Since the hash of the *.class* file does not change by a change that does not impact code behavior. However, since this change does not impact behavior we still report test failures from previous changes.[3]

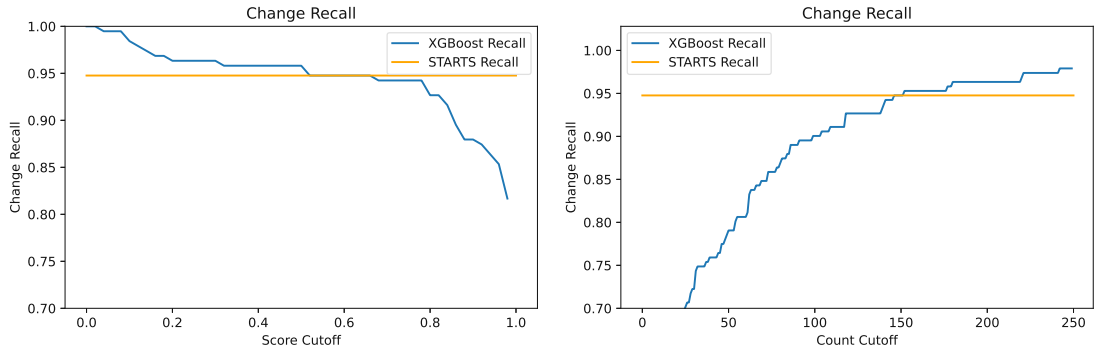

Figure 6.6: Left, $ChangeRecall(s)$ score as a function of $ScoreCutoff(s)$ of our trained XGBoost model with $CountCutoff(d) = 0$. Right, $ChangeRecall(s)$ score as a function of $CountCutoff(s)$ of our trained XGBoost model with $ScoreCutoff(d) = 0$. With *STARTS* values in orange.

Although *STARTS* performs reasonably well in terms of $ChangeRecall(s)$, our approach is still able to outperform it for appropriate cutoff values, as illustrated in figure 6.5. We are performing slightly worse in terms of $ChangeRecall(s)$ as a function of $CountCutoff(s)$ since the underlying study can detect 70% of faulty changes by choosing a cutoff of the two highest-ranking tests. If we relax our definition of $ChangeRecall(s)$ such that we can also use a false positive test to identify a faulty change, as long as the change we select is faulty. This will result not only in a similar curve to the underlying study but also allow us to detect 70% faulty changes by choosing a cutoff of the two highest-ranking tests.

Finally, when comparing $SelectionRate(s)$ of tests selected by our XGBoost classifier we see that we can again choose appropriate cutoff values to outperform *STARTS*, as depicted in figure 6.7. We did not calculate the $SelectionRate(s)$ as a function of $CountCutoff(s)$ for $CountCutoff(s) > 250$ since it is directly proportional to the number of changes times the number of tests in a change. We see that *STARTS* selects approximately half of $DependendTests(d)$.

Although we applied the de-flaking process as part of our data preparation, we did not encounter any flaky tests during data collection. As a result, while the de-flaking step was included for consistency with the original methodology, we could not collect data with flaky tests to analyze or compare in our replication.

[3]Note that this change is taken from the training data set, which is not part of the test dataset. However, the explanation for the test data stays the same.

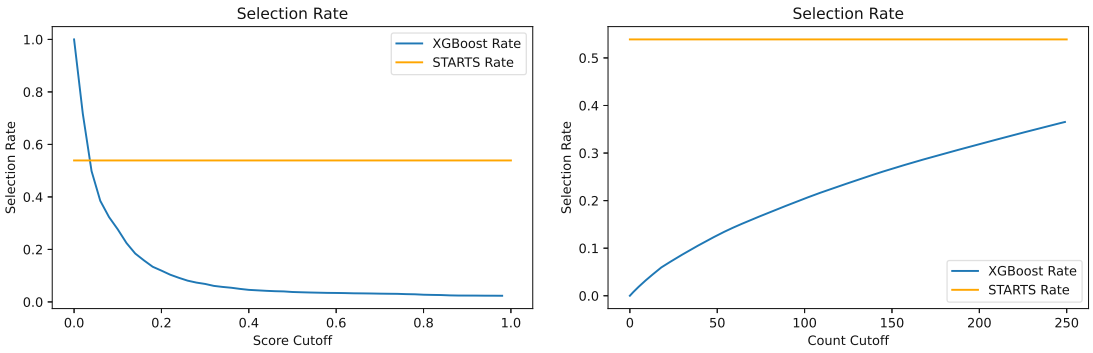

Figure 6.7: Left, $SelectionRate(s)$ score as a function of $ScoreCutoff(s)$ of our trained XGBoost model with $CountCutoff(d) = 0$. Right, $SelectionRate(s)$ score as a function of $CountCutoff(s)$ of our trained XGBoost model with $ScoreCutoff(d) = 0$. With *STARTS* values in orange.

CHAPTER 7

Conclusion

7.1 Summary

We summarize the key findings of this thesis by revisiting the research questions posed in the introduction, see Chapter 1.

> *To what extent can Facebook's findings be replicated in the open source context, i.e. will a model, as described in the paper, trained on a dataset created from the features described in the paper, still detect tests that will likely fail when inducing a code change?*

Facebook's findings show strong potential for replication in the open source context. By constructing a dataset using the features described in the paper several machine learning models, including Multilayer Perceptron, Support Vector Machine, Logistic Regression, and Random Forest, were evaluated. These models demonstrated the ability to predict test failures with reasonable accuracy. Among them, XGBoost, as also used in the original study, was among the best models evaluated. These results support the generalizability and effectiveness of Facebook's approach beyond its proprietary environment.

> *Are there models, other than the ones described in Facebook's paper, which can replicate its findings on the dataset?*

As already described we trained Multilayer Perceptron, Support Vector Machine, Logistic Regression, and Random Forest as additional models which all perform similar to the used XGBoost model.

7.2 Contributions

The main contribution of this thesis to the research of software test selection are as follows[1]:

- An open source implementation of a java project that can create a dataset out of arbitrary java projects using maven as build management tool.
- An open source implementation of the complete machine learning pipeline used in this thesis.
- A high quality dataset containing more than 1.2 million test cases, with many additional features besides the ones described in the underlying study. Constructed from real world data. This sets our thesis apart from many theses in the field which are using mutant injection to create synthetic datasets.
- An in-depth validation of different machine learning models trained on the constructed dataset.
- We show that the findings of the underlying study can be validated within the open source context on real world data.

7.3 Limitations and Future Work

One limitation of our replication study is that it focuses exclusively on Java projects, whereas the original Facebook study applies predictive test selection across multiple programming languages. This constraint may affect the generalizability of our findings, as language-specific characteristics—such as testing frameworks, code structure, and development practices—can influence model performance. As a result, while our results are promising within the Java ecosystem, further evaluation across diverse language environments is necessary to fully validate the cross-language applicability of the approach.

Another limitation of our replication study is the absence of flaky tests in our dataset. Flaky tests—those that exhibit non-deterministic outcomes—are a significant concern in large-scale software development and were addressed in the original study. Consequently, our replication does not validate this aspect of the original findings, and further work with datasets that include flaky tests is needed to assess model robustness in such scenarios.

Finally, due to the high-level description provided in the original study regarding its machine learning pipeline and feature engineering, we cannot be certain that our assumptions about data collection and model training fully align with those of the original work. The lack of detailed implementation specifics may introduce subtle differences in how features are extracted or models are tuned, which could impact the comparability of results. This limitation highlights the importance of transparency and reproducibility in empirical machine learning research.

[1] Code, pipelines, and dataset can be downloaded at: `https://zenodo.org/records/15088426`

7.4 Threats to Validity

External Validity is the extent to which our findings can be generalized. In this thesis we only consider java based project, that are well established in the open source context. However, the underlying study also analyzes different programming language and build tools. We already contributed a great step towards external validity by validating the underlying studies results within the open source context instead of Facebook's proprietary monorepo [MSPC19].

Internal Validity is the extent to which the findings of this work are indeed explained by the presented data and not by other factors. The biggest uncertainty is if the features where collected in the same way as the underlying study. Since only a high level explanation is given by it. While we took particular care to not include future information about the development lifecycle in the data, we cannot guarantee that this is the case. Since the data was collected from previous commits that may or may not been altered by some git commands. We would have needed more control over the underlying projects to inject our data collection into, for example, a CI pipeline to only capture current commits and their changes.

CHAPTER 8

Appendix

Here we will provide a breakdown of all projects used for this thesis. Futhermore, it will bundle the best hyperparameters used for training, and additional performance metrics.

8.1 Projects

Following are all chosen projects for dataset generation. Note that although we will provide arguments for all problems that will allow a successful build. Because of memory and time constraints of, only the projects market with (x) in table 8.2 are used in the thesis.

Project	Runtime	Source
biojava	45h	https://github.com/biojava/biojava
ews java api	4h	https://github.com/OfficeDev/ews-java-api
geometry api	5h	https://github.com/Esri/geometry-api-java
graphhopper	262h	https://github.com/graphhopper/graphhopper
jackrabbit-oak	706h	https://github.com/apache/jackrabbit-oak
jmeter-plugins	9h	https://github.com/undera/jmeter-plugins
jmxtrans	8h	https://github.com/jmxtrans/jmxtrans
jsoup	24h	https://github.com/jhy/jsoup
logback	25h	https://github.com/qos-ch/logback
mustache.java	14h	https://github.com/spullara/mustache.java
openpnp	19h	https://github.com/openpnp/openpnp
opentsdb	38h	https://github.com/OpenTSDB/opentsdb
PDF Box	76h	https://github.com/apache/pdfbox
Twilio	68h	https://github.com/twilio/twilio-java
XChange	115h	https://github.com/knowm/XChange

Table 8.1: Table of analyzed projects and their source origin.

Project	Run Parameters
(x) biojava	−−proj=biojava −−commits=900 −−ignoreDirs=biojava-integrationtest −−javaHome="<PATH TO>jdk-11.0.26+4" −−mvnArgs=-Dmaven.jxr.skip,-Dcheckstyle.skip,-Dfindbugs.skip,-Dmpir.skip
(x) ews java api	−−proj="ews-java-api" −−commits=600 −−javaHome="<PATH TO>jdk8u402-b06"
(x) geometry api	−−proj="geometry-api-java" −−commits=300 −−javaHome="<PATH TO>jdk8u402-b06"
(x) graphhopper	−−proj=graphhopper −−commits=1000 −−javaHome="<PATH TO>jdk-21.0.2" −−mvnArgs=-Dskip.installnodenpm,-Dskip.npm,-Dmaven.antrun.skip
jackrabbit-oak	−−proj="jackrabbit-oak" −−commits=1000 −−javaHome="<PATH TO>jdk-11.0.22+7" −−mvnArgs=-Ddependency-check.skip,-Drat.skip,-Dmaven.jsdoc.skip,-Dcheckstyle.skip,-Dspotbugs.skip,-Dmaven.site.skip,-Dmpir.skip −−skip=be58b8c85d753b9ce2717b375801c22c60831a63
(x) jmeter-plugins	−−proj="jmeter-plugins" −−commits=800 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Dcobertura.skip,-Ddependency-check.skip,-Dspotbugs.skip,-Dgpg.skip,-DskipNexusStagingDeployMojo,-Dcheckstyle.skip,-Dsonar.skip
(x) jmxtrans	−−proj="jmxtrans" −−commits=300 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Dcobertura.skip,-Dfindbugs.skip,-Danimal.sniffer.skip,-Ddockerfile.skip
(x) jsoup	−−proj="jsoup" −−commits=1000 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Danimal.sniffer.skip,-Dmaven.source.skip,-Djapicmp.skip
(x) logback	−−proj="logback" −−commits=600 −−javaHome="<PATH TO>jdk-21.0.2" −−mvnArgs=-Dfindbugs.skip,-Dmaven.jxr.skip,-Dpgpverify.skip,-Dmaven.source.skip,-Dmaven.site.skip
(x) mustache.java	−−proj="mustache.java" −−commits=200 –mvn="<PATH TO>apache-maven-3.3.3" −−javaHome="<PATH TO>jdk8u402-b06"
(x) openpnp	−−proj="openpnp" −−commits=1000 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Dbuildnumber.plugin.phase=none,-Dcheckstyle.skip
(x) opentsdb	−−proj="opentsdb" −−commits=700 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Dmaven.antrun.skip,-Dgpg.skip,-Dsonar.skip
PDF Box	−−proj="pdfbox"−−−commits=1500 −−javaHome="<PATH TO>jdk-11.0.22+7" −−mvnArgs=-Dcobertura.skip,-Ddependency-check.skip,-Dspotbugs.skip,-Dgpg.skip,-DskipNexusStagingDeployMojo,-Dcheckstyle.skip,-Dsonar.skip,-Dmaven.source.skip,-Drat.skip
(x) Twilio	−−proj="twilio-java" −−commits=700 −−javaHome="<PATH TO>jdk8u402-b06" −−mvnArgs=-Dcobertura.skip,-Ddependency-check.skip,-Dspotbugs.skip,-Dgpg.skip,-DskipNexusStagingDeployMojo,-Dcheckstyle.skip,-Dsonar.skip
XChange	−−-proj="XChange" −−commits=3000 −−javaHome="<PATH TO>jdk-11.0.22+7" −−ignoreTests=.*IntegrationTest$,.*Integration$

Table 8.2: Table of run parameters which the data set generator was run.

In the table 8.1 the respective runtime is measured for the final run of our data collection tool. The tool ran multiple times, until a stable run with most commits reporting a successful build, was reached. During these runs our collection tool was subjects to multiple performance improvements. The runtime is rounded up to the next full hour. If possible we analyze more than the latest 1000 commits. On the next page we will provide a table with all parameters that were used to generate the dataset.

For all projects in the table 8.2, −−path="<PATH TO PROJECTS>" needs to be set to the base directory where the projects are located that the generator uses. All generations can be run with java -jar -Xmx8000m ./datasetparser.jar <PARAMETERS>. The javaHome and mvnHome parameters can be set to use alternative java and maven versions, for building the project and running the tests. If not set, depending on the path variables the installed Java and Maven versions are used. For our setup this defaults to Java 21.0.2 and Maven 3.9.6 is used. More details will be found in the Readme of our generation tool.

Most of the skipped maven phases are chosen to speed up the build process, however some of them will lead to build failures in older commits. Especially when resources they need are not available anymore.

Additionally, for building the project, the following arguments are appended internally: *-U -Denforcer.skip -Dcheckstyle.skip -Dlicense.skip -Dmaven.javadoc.skip -DskipTests*. As well as the following JVM arguments: *-XX:-TieredCompilation -XX:TieredStopAtLevel=1 -Xverify:none*.

For testing the following arguments are appended internally: *-Denforcer.skip=true -Dlicense.skip=true -Dsurefire.forkCount=0 -Dsurefire.useSystemClassLoader=false -Dsurefire.failIfNoSpecifiedTests=false -DfailIfNoTests=false*, as well as the following JVM arguments *-Xmx4048m -Xms1024m*.

8.2 Best Hyperparameter

gamma	max_delta_step	max_depth	min_child_weight	n_estimators	sampler
0.000000	1	2	2	71	Random Over-Sampling

Table 8.3: Best hyperparameters for the XGBoost model, all features

gamma	max_delta_step	max_depth	min_child_weight	n_estimators	sampler
1.000000	0	2	1	64	Random Over-Sampling

Table 8.4: Best hyperparameters for the XGBoost model, after feature selection

activation	alpha	hidden_layer_sizes	learning_rate_init	sampler
relu	0.064604	20	0.000100	SMOTE

Table 8.5: Best hyperparameters for the multi layer perceptron model, all features

activation	alpha	hidden_layer_sizes	learning_rate_init	sampler
tanh	0.066091	10	0.000964	SMOTE

Table 8.6: Best hyperparameters for the multi layer perceptron model, after feature selection

max_depth	min_samples_split	n_estimators	sampler
10	2	78	No sampling

Table 8.7: Best hyperparameters for the random forest model, all features

max_depth	min_samples_split	n_estimators	sampler
10	5	141	No sampling

Table 8.8: Best hyperparameters for the random forest model, after feature selection

kernel___degree	kernel___kernel	C	sampler
5	rbf	4.196023	No sampling

Table 8.9: Best hyperparameters for the support vector machine model

kernel___degree	kernel___kernel	C	sampler
5	rbf	0.351107	No sampling

Table 8.10: Best hyperparameters for the support vector machine model, after feature selection

C	penalty	sampler
0.019327	l1	No sampling

Table 8.11: Best hyperparameters for the logistic regression model

C	penalty	sampler
0.007344	l1	No sampling

Table 8.12: Best hyperparameters for the logistic regression model, after feature selection

8.3 Performance Measures

	train			test			test fitted		
	recall	precision	F1	recall	precision	F1	recall	precision	F1
logistic regression	0.769861	0.910853	0.834443	0.917813	**0.9725**	**0.9444**	0.918750	0.972544	0.944882
multi layer perceptron	0.826431	0.929751	0.875052	0.921875	0.949163	0.935320	**0.9200**	0.953986	0.936685
random forest	0.860776	0.976718	0.915089	**0.9234**	0.955074	0.938990	0.917188	**0.9787**	0.946927
support vector machine	0.799051	0.925318	0.857562	0.920937	0.947588	0.934073	0.919375	0.976435	**0.9470**
XGBoost	**0.9829**	**0.9901**	**0.9864**	0.917500	0.961992	0.939219	0.910000	0.973913	0.940872

Table 8.13: Results of fitted models after hyperparameter tuning on all features

	train			test			test fitted		
	recall	precision	F1	recall	precision	F1	recall	precision	F1
logistic regression	0.768666	0.915691	0.835762	0.917813	**0.9725**	**0.9444**	0.918750	0.972544	0.944882
multi layer perceptron	0.808610	0.904249	0.853760	0.916250	0.961942	0.938540	0.914375	0.972093	0.942351
random forest	**0.8590**	**0.9775**	**0.9144**	**0.9222**	0.959051	0.940258	0.917188	**0.9787**	0.946927
support vector machine	0.799051	0.925318	0.857562	0.920937	0.947588	0.934073	**0.9194**	0.976435	**0.9470**
XGBoost	0.815131	0.890264	0.851043	0.915937	0.963511	0.939122	0.915937	0.969567	0.941989

Table 8.14: Results of fitted models after hyperparameter tuning on selected features

List of Figures

List of Tables

Bibliography

[AD18] Maral Azizi and Hyunsook Do. Retest: A cost effective test case selection technique for modern software development. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 144–154. IEEE, 2018.

[AM22] Fahmy Amin and M Mahmoud. Confusion matrix in binary classification problems: A step-by-step tutorial. *Journal of Engineering Research*, 6(5):0–0, 2022.

[ASD14] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 142–151, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2597073.2597084.

[BB12] Christoph Bergmeir and José M Benítez. On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213, 2012.

[BDM+96] Andrew Brooks, John Daly, James Miller, Marc Roper, and Murray Wood. Replication of experimental results in software engineering. *International Software Engineering Research Network (ISERN) Technical Report ISERN-96-10, University of Strathclyde*, 2, 1996.

[BE12] C Glenn Begley and Lee M Ellis. Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

[BI15] C Glenn Begley and John PA Ioannidis. Reproducibility in science: improving the standard for basic and preclinical research. *Circulation research*, 116(1):116–126, 2015.

[BID+14] Mark John Brandt, H. Ijzerman, A. J. Dijksterhuis, Frank J. Farach, Jason Geller, Roger Giner-Sorolla, James A. Grange, Marco Perugini, Jeffrey R. Spies, and Anna van 't Veer. The replication recipe: What makes for a convincing replication? *Journal of Experimental Social Psychology*, 50:217–224, 2014. Accessed: 2025. URL: https://api.semanticscholar.org/CorpusID:806920.

[BMG20] Andrew L Beam, Arjun K Manrai, and Marzyeh Ghassemi. Challenges to the reproducibility of machine learning models in health care. *Jama*, 323(4):305–306, 2020.

[BMSS11] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3), 2011. Accessed: 2025. URL: https://informatica.si/index.php/informatica/article/viewFile/355/356.

[BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[CBHK02] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[CC20] KR1442 Chowdhary and KR Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[CCZ+11] Songyu Chen, Zhenyu Chen, Zhihong Zhao, Baowen Xu, and Yang Feng. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2011.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2939672.2939785.

[CH20] Marco Castelluccio and Andrew Halberstadt. Testing firefox more efficiently with machine learning – mozilla hacks - the web developer blog. https://hacks.mozilla.org/2020/07/testing-firefox-more-efficiently-with-machine-learning, 2020. Accessed: 2025.

[CHY+23] Jianghui Cai, Jing Hao, Haifeng Yang, Xujun Zhao, and Yuqing Yang. A review on semi-supervised clustering. *Information Sciences*, 2023.

[DS18] Omdev Dahiya and Kamna Solanki. A systematic literature study of regression test case prioritization approaches. *International Journal of Engineering & Technology*, 7(4):2184–2191, 2018. Accessed 2025. URL: https://api.semanticscholar.org/CorpusID:70105269.

[DSSF+14] Fabio QB Da Silva, Marcos Suassuna, A César C França, Alicia M Grubb, Tatiana B Gouveia, Cleviton VF Monteiro, and Igor Ebrahim dos Santos.

Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering*, 19:501–557, 2014.

[EFK+16] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 11–20. IEEE, 2016.

[EIS+20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Jacob Steinhardt, and Aleksander Madry. Identifying statistical bias in dataset replication. In *International Conference on Machine Learning*, pages 2922–2932. PMLR, 2020.

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[GBR12] Jesús M González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17:75–89, 2012.

[GEM15a] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 713–716. IEEE, 2015.

[GEM15b] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2771783.2771784.

[GHK+01] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[GHS92] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *ICSM*, volume 92, pages 299–308. IEEE, 1992.

[GK18] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: reproducibility in artificial intelligence. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[HBGL08] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*, pages 1322–1328. IEEE, 2008.

[HIB+18] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[HJL+01] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. *ACM Sigplan Notices*, 36(11):312–326, 2001.

[HRRW01] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Trans. Softw. Eng.*, 27(3):248–263, March 2001. doi:10.1109/32.910860.

[HS89] Mary Jean Harrold and Mary Lou Soffa. Interprocedual data flow testing. *ACM SIGSOFT software engineering notes*, 14(8):158–167, 1989.

[HS15] Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

[Hut18] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018. Accessed: 2025. URL: https://www.science.org/doi/abs/10.1126/science.359.6377.725, doi:10.1126/science.359.6377.725.

[KAK+25] Adam Khan, Asad Ali, Jahangir Khan, Fasee Ullah, and Muhammad Faheem. Using permutation-based feature importance for improved machine learning model performance at reduced costs. *IEEE Access*, 2025.

[KGH+95] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2):51–65, 1995.

[KK11] Nittaya Kerdprasop and Kittisak Kerdprasop. Discrete decision tree induction to avoid overfitting on categorical data. In *Proceedings of the 13th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems, and 10th WSEAS International Conference*

on Non-Linear Analysis, Non-Linear Systems and Chaos, and 7th WSEAS International Conference on Dynamical Systems and Control, and 11th WSEAS International Conference on Wavelet Analysis and Multirate Systems: Recent Researches in Computational Techniques, Non-Linear Systems and Control, MAMECTIS/NOLASC/CONTROL/WAMUS'11, page 247–252, Stevens Point, Wisconsin, USA, 2011. World Scientific and Engineering Academy and Society (WSEAS).

[LHEM14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635920.

[LL17] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA, 2017. Curran Associates Inc.

[LNA17] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *J. Mach. Learn. Res.*, 18(1):559–563, January 2017.

[LP15] Jeffrey T Leek and Roger D Peng. Reproducible research can still be wrong: adopting a prevention approach. *Proceedings of the National Academy of Sciences*, 112(6):1645–1646, 2015.

[LSM17] Owolabi Legunsen, August Shi, and Darko Marinov. Starts: Static regression test selection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 949–954. IEEE, 2017.

[Lun19] Erik Lundsten. Ealrts: A predictive regression test selection tool. Master's thesis, KTH Royal Institute of Technology, 2019.

[LW90] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[LWW+20] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[LZN+23] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. More precise regression test selection via reasoning about semantics-modifying changes. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 664–676, New York,

NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598086.

[Man09] Christopher D Manning. *An introduction to information retrieval*. Cambridge university press, 2009.

[MC24] Ruchika Malhotra and Madhukar Cherukuri. A systematic review of hyperparameter tuning techniques for software quality prediction models. *Intelligent Data Analysis*, 28(5):1131–1149, 2024.

[MGN+17] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[MK17] Lech Madeyski and Barbara Kitchenham. Would wider adoption of reproducible research be beneficial for empirical software engineering research? *Journal of Intelligent & Fuzzy Systems*, 32(2):1509–1521, 2017.

[MKJ07] Ramal Moonesinghe, Muin J Khoury, and A Cecile J W Janssens. Most published research findings are false—but a little replication goes a long way. *PLoS medicine*, 4(2):e28, 2007.

[Mol20] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[MS13] Michael Mayo and Simon Spacey. Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics. In *Search Based Software Engineering: 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24-26, 2013. Proceedings 5*, pages 158–171. Springer, 2013.

[MSPC19] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE, 2019.

[NSNW13] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, 2013. Accessed: 2025. URL: https://api.semanticscholar.org/CorpusID:14040055.

[PBGB22] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, 27(2):1–43, 2022.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[RGS+18] Adriano Rivolli, Luís Paulo F. Garcia, Carlos Soares, Joaquin Vanschoren, and André Carlos Ponce de Leon Ferreira de Carvalho. Towards reproducible empirical research in meta-learning. *ArXiv*, abs/1808.10406, 2018. Acessed:2025. URL: `https://api.semanticscholar.org/CorpusID:52133902`.

[RH93] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In *1993 Conference on Software Maintenance*, pages 358–367. IEEE, 1993.

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, page 169–184, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/186258.187171`.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[Rob10] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180. IEEE, 2010.

[RSAM18] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. Spiritus: A simple information retrieval regression test selection approach. *Information and Software Technology*, 99:62–80, 2018.

[SBH14] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[SD12] Quinten David Soetens and Serge Demeyer. Cheopsj: Change-based test optimization. In *CSMR*, pages 535–538. IEEE, 2012.

[SDZ13] Quinten David Soetens, Serge Demeyer, and Andy Zaidman. Change-based test selection in the presence of developer tests. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 101–110. IEEE, 2013.

[SGMM17] 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, page 12–22, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3092703.3092709.

[SKPS20] Shantanu Sutar, Rajesh Kumar, Sriram Pai, and BR Shwetha. Regression test cases selection using natural language processing. In *2020 International Conference on Intelligent Engineering and Management (ICIEM)*, pages 301–305. IEEE, 2020.

[SSS00] Robert H Shumway, David S Stoffer, and David S Stoffer. *Time series analysis and its applications*, volume 3. Springer, 2000.

[SWAK12] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2012.

[TTDM15] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[VF97] Filippos I Vokolos and Phyllis G Frankl. Pythia: A regression test selection tool based on textual differencing. In *Reliability, Quality and Safety of Software-Intensive Systems: IFIP TC5 WG5. 4 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems (ENCRESS'97), 29th–30th May 1997, Athens, Greece*, pages 3–21. Springer, 1997.

[VPMG17] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. File-level vs. module-level regression test selection for .net. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 848–853, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3106237.3117763.

[Wah99] Nancy J. Wahl. An overview of regression testing. *ACM SIGSOFT Software Engineering Notes*, 24(1):69–73, January 1999. doi:10.1145/308769.308790.

[WHLA97] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[WS00] Christopher K. I. Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Proceedings of the 14th International Conference on Neural Information Processing Systems*, NIPS'00, page 661–667, Cambridge, MA, USA, 2000. MIT Press.

[YH12] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[Zha18] Lingming Zhang. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 199–209, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180198.

[ZKK12] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2393596.2393642.

[ZLG+22] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. Comparing and combining analysis-based and learning-based regression test selection. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, page 17–28, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524481.3527230.

[ZZL+23] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023. Accessed 2025. URL: https://api.semanticscholar.org/CorpusID:257900969.