Empirical Software Engineering (2021) 26: 114
https://doi.org/10.1007/s10664-021-09996-y

# Predicting unstable software benchmarks using static source code features

**Christoph Laaber**[1] · **Mikael Basmaci**[1] · **Pasquale Salza**[1]

Accepted: 1 June 2021 / Published online: 18 August 2021


**Abstract**

Software benchmarks are only as good as the performance measurements they yield. Unstable benchmarks show high variability among repeated measurements, which causes uncertainty about the actual performance and complicates reliable change assessment. However, if a benchmark is stable or unstable only becomes evident after it has been executed and its results are available. In this paper, we introduce a machine-learning-based approach to predict a benchmark's stability without having to execute it. Our approach relies on 58 statically-computed source code features, extracted for benchmark code and code called by a benchmark, related to (1) meta information, e.g., lines of code (LOC), (2) programming language elements, e.g., conditionals or loops, and (3) potentially performance-impacting standard library calls, e.g., file and network input/output (I/O). To assess our approach's effectiveness, we perform a large-scale experiment on 4,461 Go benchmarks coming from 230 open-source software (OSS) projects. First, we assess the prediction performance of our machine learning models using 11 binary classification algorithms. We find that Random Forest performs best with good prediction performance from 0.79 to 0.90, and 0.43 to 0.68, in terms of AUC and MCC, respectively. Second, we perform feature importance analyses for individual features and feature categories. We find that 7 features related to meta-information, slice usage, nested loops, and synchronization application programming interfaces (APIs) are individually important for good predictions; and that the combination of all features of the called source code is paramount for our model, while the combination of features of the benchmark itself is less important. Our results show that although benchmark stability is affected by more than just the source code, we can effectively utilize machine learning models to predict whether a benchmark will be stable or not ahead of execution. This enables spending precious testing time on reliable benchmarks, supporting developers to identify unstable benchmarks during development, allowing unstable benchmarks to be repeated more often, estimating stability in scenarios where repeated benchmark execution is infeasible or impossible, and warning developers if new benchmarks or existing benchmarks executed in new environments will be unstable.

Communicated by: Meiyappan Nagappan

✉ Christoph Laaber
laaber@ifi.uzh.ch

Extended author information available on the last page of the article.

**Keywords** Performance testing · Software benchmarking · Performance variability · Source code features · Machine learning for software engineering · Go

## 1 Introduction

Software benchmarks are a performance testing technique on the same granularity as unit tests, i.e., they test functions, methods, or statements. Different from unit tests, they measure performance, most often execution time, by executing the benchmark repeatedly to retrieve reliable results. Depending on a myriad of factors, such as the quality of the benchmark, the stability of the execution environment, the programming language, and the source code the benchmark calls, these measurements are close to each other, i.e., the benchmark has low result variability and is "stable", or further apart, i.e., the benchmark has high result variability and is "unstable". Results from unstable benchmarks do not accurately reflect the "true" performance of the software (unit) under test and hinder rigorous and reliable performance change assessment. Unfortunately, this only becomes evident once the benchmark has been executed and its results are available.

Previous research on software performance focussed on performance impact prediction of new code changes on the execution time of software, i.e., whether a code change slows down (or speeds up) the program. They often leverage statically or dynamically determined source code features, such as added loops or method calls, the code change diff, and sometimes profiling data, to predict whether a benchmark or a version is likely to experience a performance change (Jin et al. 2012; Huang et al. 2014; Sandoval Alcocer et al. 2016; de Oliveira et al. 2017; Mostafa et al. 2017; Alshoaibi et al. 2019; Sandoval Alcocer et al. 2020; Chen et al. 2020). This information is then used for selecting which versions to test for performance (Jin et al. 2012; Huang et al. 2014; Sandoval Alcocer et al. 2016, 2020), selecting the benchmarks to execute after a code change (de Oliveira et al. 2017; Alshoaibi et al. 2019; Chen et al. 2020), prioritizing the benchmarks with larger predicted performance changes for execution (Mostafa et al. 2017), or identifying functional tests to use as performance tests (Ding et al. 2020). All of these focus on slowdown/speedup size as the performance property to predict, and none considers measurement variability or benchmark stability. Moreover, the majority employs traditional inference techniques such as rule-based detection (Jin et al. 2012), cost models (Huang et al. 2014; Sandoval Alcocer et al. 2016; Mostafa et al. 2017; Sandoval Alcocer et al. 2020), heuristics (de Oliveira et al. 2017), or genetic algorithms (Alshoaibi et al. 2019), and only two utilize machine learning models (Chen et al. 2020; Ding et al. 2020).

In this paper, we propose an approach that leverages static source code features to predict whether a benchmark will be unstable before executing it. The approach employs 58 source code features extracted with abstract syntax tree (AST) and static call graph (CG) information, each feature once for the benchmark's code and once for the code called by the benchmark, resulting in a total of 116 features. They consist of (1) meta information, e.g., lines of code (LOC), (2) programming language elements, e.g., conditionals or loops, and (3) potentially performance-impacting standard library calls, e.g., file and network input/output (I/O). To assess our approach, we perform a large-scale experiment on 4,461 benchmarks coming from 230 open-source software (OSS) projects written in Go.

In our first research question, we investigate the performance of binary classifiers that predict whether a benchmark is stable on unstable:

**RQ 1** Can we predict benchmark instability with statically-computed source code features?

We build a static model based on the source code features and assess the prediction performance among 11 classification algorithms. For this, we transform the benchmarks' variability into two classes, i.e., stable and unstable, relying on thresholds inspired by previous work (Georges et al. 2007; Curtsinger and Berger 2013). We compare the different machine learning algorithms and find that their models are effective. Random Forest performs best with a prediction performance ranging from 0.79 to 0.90 AUC and from 0.43 to 0.61 MCC.

Our classification model considers benchmarks to be stable or unstable if they fall beneath or above a certain threshold. We study how the threshold value impacts prediction performance with our first sub research question:

**RQ 1.1** How does the prediction performance change when the model is trained with different stability thresholds?

Inspired by previous work (Georges et al. 2007; Curtsinger and Berger 2013), we investigate four stability thresholds $t \in \{1\%, 3\%, 5\%, 10\%\}$, which correspond to $t\%$ benchmark variability. We find that the threshold $t$ impacts prediction performance. Algorithms trained with the largest threshold $t = 10\%$ deliver the best prediction performance in most cases. This shows that our model is better at classifying benchmarks as unstable if they have higher result variability, i.e., they are "more unstable".

The number of measurement iterations, i.e., repetitions, directly influences the benchmark stability, i.e., more iterations lead to narrower confidence intervals and, consequently, to a more stable benchmark. Our second sub research question investigates their impact on the prediction performance of our model:

**RQ 1.2** How does the prediction performance change when the model is trained on benchmark executions with different numbers of iterations?

We study four numbers of repeated iterations $i = \{5, 10, 20, 30\}$ and find that the number of iterations drastically impacts the prediction performance of our model. Measurements from more iterations lead to better prediction performance for the majority of the studied algorithms. This shows that our model is better at predicting benchmark instability if a benchmark remains unstable with an increased number of iterations.

With our third sub research question, we study the impact of pre-processing steps on the prediction performance of our model:

**RQ 1.3** How does the prediction performance change when removing co-linear and multi-co-linear features and applying class-rebalancing?

We apply AutoSpearman (Jiarpakdee et al. 2018), a sophisticated technique to remove co-linear and multi-co-linear features, and SMOTE (Chawla et al. 2002), which performs class-rebalancing. We find that, across all studied classifiers, their application neither improves nor deteriorates the model's prediction performance. In the case of Random Forest, however, AutoSpearman improves MCC and AUC by 0.023 and 0.005, respectively.

Our fourth sub research question studies the impact of different measures of variability on prediction performance:

**RQ 1.4** Do different variability measures have an impact on the prediction performance?

In the previous experiments, we employ the relative confidence interval width (RCIW) based on Maritz-Jarrett's technique (Maritz and Jarrett 1978), i.e., $rciw_{mjhd}$, as variability measure (i.e., dependent variable) to estimate a benchmark's instability. This research question investigates if the prediction performance changes for other variability measures. We compare $rciw_{mjhd}$ to relative confidence interval width (RCIW) based on bootstrap (Kalibera and Jones 2012), i.e., $rciw_{boot}$, and relative median absolute deviation (RMAD) (Arachchige et al. 2020), i.e., $rmad$. The results show that the prediction performance varies considerably. In the case of Random Forest, MCC increases by 0.08 and 0.11, when using $rciw_{boot}$ and $rmad$, respectively.

Finally, with our second research question, we study the importance of individual features and feature categories when building the models:

**RQ 2** Which are the most important individual features and feature categories for good prediction performance?

We consider the best performing model from RQ 1, i.e., Random Forest built with 30 iterations and a 10% threshold. We find that only 7 features are individually important for good predictions. Furthermore, the features concerning the called source code are collectively most important.

Although benchmark stability is affected by more than just source code, our results show that source code features can be effectively used to predict whether a benchmark will be stable or unstable. We envision that this stability prediction can be used in regression benchmarking for selecting only stable benchmarks to be executed, supporting developers to identify unstable benchmarks during development, configuring unstable benchmarks to run more iterations, estimating benchmark stability in scenarios where repeated benchmark execution is infeasible or impossible, or warning developers if new benchmarks or existing benchmarks executed in new environments will be unstable.

**Contributions.** The main contributions of this paper can be summarized as follows:

- an approach to extract statically-computable features from benchmark source code and predict whether the benchmark will be unstable using machine learning algorithms;
- a study comparing the predictive performance of 11 machine learning algorithms, investigating the effects of different stability definitions, benchmark iterations, preprocessing steps, and variability measures;
- a study of the importance of individual features and feature categories for good prediction performance;
- a large data set of 4,461 Go benchmark executions from 230 OSS projects.

We provide all data and scripts to reuse our approach and replicate our study online (Laaber et al. 2021).

**Paper Organization.** In Section 2, we provide an introduction to benchmarking in Go and benchmark stability. Section 3 introduces our approach, how it extracts the features, how we define benchmark stability, what is required for the model creation, and how to use the model. Section 4 describes the study design to assesses the effectiveness of our approach. Sections 5 and 6 report the results for RQ 1 and RQ 2, respectively. In Section 7, we discuss

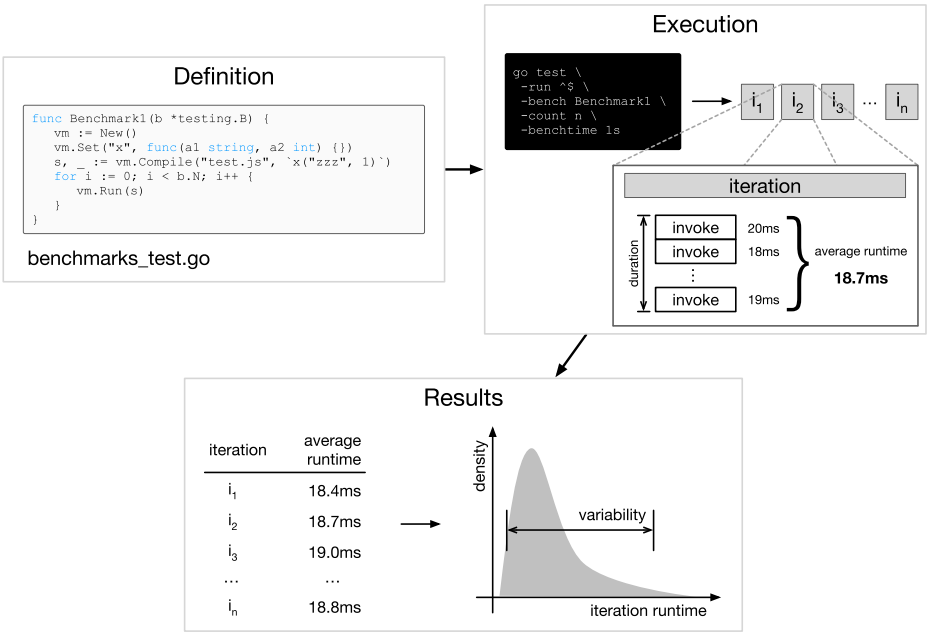


**Fig. 1** Benchmarking workflow in Go. *Benchmark1* corresponds to the benchmark *call_test.go/BenchmarkNativeCallWithString* from the project *robertkrimen/otto*

application scenarios and other aspects of our approach, as well as directions for future research. Section 8 discusses threats to validity, and Section 9 compares to related work. Finally, we conclude the paper in Section 10.

## 2 Software Benchmarks in Go

Software benchmarking—also referred to as microbenchmarking (Laaber and Leitner 2018; Laaber et al. 2019) or performance unit testing (Horký et al. 2015; Stefan et al. 2017; Bulej et al. 2017a)—is a form of measurement-based software performance engineering (SPE) (Woodside et al. 2007) to evaluate the performance, usually execution time, of fine-granular software components such as functions, methods, or statements. They can be considered as the equivalent of unit tests for performance.

The Go programming language[1] comes with a benchmarking framework included in their standard library, as part of the testing framework.[2] Figure 1 depicts a schematic view of Go benchmarks, how they are defined and executed, and what their results look like.

**Definition.** In Go, benchmarks are defined as top-level functions in source code, similar to unit tests, if (1) they are placed in a file ending in `_test.go`, (2) their name starts with

[1] https://golang.org
[2] https://golang.org/pkg/testing

`Benchmark`, and (3) their only function parameter is of type `*testing.B`. A benchmark's body also contains a `for` loop that repeatedly invokes the component that should be benchmarked.

**Execution.** A single measurement is not sufficient to accurately depict a program's performance, because a myriad of factors influences performance measurements, such as the machine they are executed on, the software (versions) installed, and the programming language characteristics. Consequently, performance measurements (and benchmarks) are susceptible to measurement uncertainty (Mytkowicz et al. 2009; Curtsinger and Berger 2013; de Oliveira et al. 2013), which is addressed with repeated measurements.

To execute benchmarks, such as the one previously defined, one uses the *go* command line interface (CLI), specifying which benchmark(s) to execute (through the use of the `-bench` option), and for how many iterations (`-count`). An iteration ($i_n$) is the repeated invocation of the benchmark function, `Benchmark1` in our example, for a defined duration (`-benchtime`). The benchmarking framework measures the runtime (with nanosecond precision) of every benchmark invocation and yields the average runtime among all invocations as the iteration's result.

**Results.** The final result of a benchmark is the distribution of all iteration measurements, which typically does not follow a normal distribution but is often long-tailed or multimodal (Curtsinger and Berger 2013; Maricq et al. 2018). Depending on the variability (or spread) among the individual iteration results, a benchmark is considered more or less stable when it has low or high measurement variability, respectively. Running more iterations leads to narrower confidence intervals of the results and, hence, to more stable benchmark results (see Section 3.2). Figure 2 shows an example of real-world benchmark results with low ("stable") and high ("unstable") variability after 20 iterations.

## 3 Approach

To predict whether a benchmark will be unstable without executing it, we introduce an approach based on machine learning. This approach employs only statically-computed source code features to build a binary classification model used for prediction. Although our

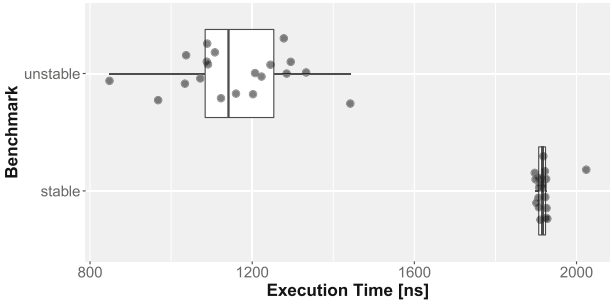


**Fig. 2** Benchmarks with stable and unstable results after 20 iterations (indicated by the dots). The bar indicates the median, the diamond the mean, the box the $IQR$, and the whiskers $[Q1|Q3] + 1.5 * IQR$. The "stable" benchmark is *call_test.go/BenchmarkNativeCallWithString* from *robertkrimen/otto*. The "unstable" benchmark is *frame_pool_b_test.go/BenchmarkFramePoolChannel1000* from *uber/tchannel-go*

approach's general idea is applicable to different programming languages, in particular, the types of source code features employed, this section describes the approach in the context of Go.

Our approach is based on two main phases: (1) training and (2) usage. During the training phase, depicted in Fig. 3, our aim is to build a model based on real benchmark executions. Starting from a possibly large sample of benchmarks, we first extract the statically-computed source code features for each of these. Then, we run the benchmarks for multiple independent executions (iterations) to obtain a measure for their stability, based on the variability among the iterations. To decide whether a benchmark is "stable" or "unstable", we perform "binarization" on the benchmarks' result variability values. Section 7 discusses binary classification versus regression in this context. Based on the extracted source code features and the benchmark stabilities, we train a model using a machine learning algorithm. We investigate the best algorithms for this prediction task in Section 5 and the most important features for the best performing algorithm in Section 6. The trained model can then, for example, be "used" to predict whether a new benchmark executed in the same environment or an existing benchmark to be executed in a different environment will be stable or unstable, before executing it.

### 3.1 Source Code Features

The main idea is to employ source code features that serve as a proxy for performance variability, even if the actual root cause lies outside of the source code, such as the underlying operating system or the network. The choice of features is inspired by previous research that investigated the root causes of performance failures.

To build our approach's prediction model, we statically determine its features by (1) extracting these on a per-function basis through AST parsing, (2) identifying these associated just with the benchmarks' source code, and (3) combining these belonging to all the reachable functions from each benchmark with CG information. Table 1 provides an

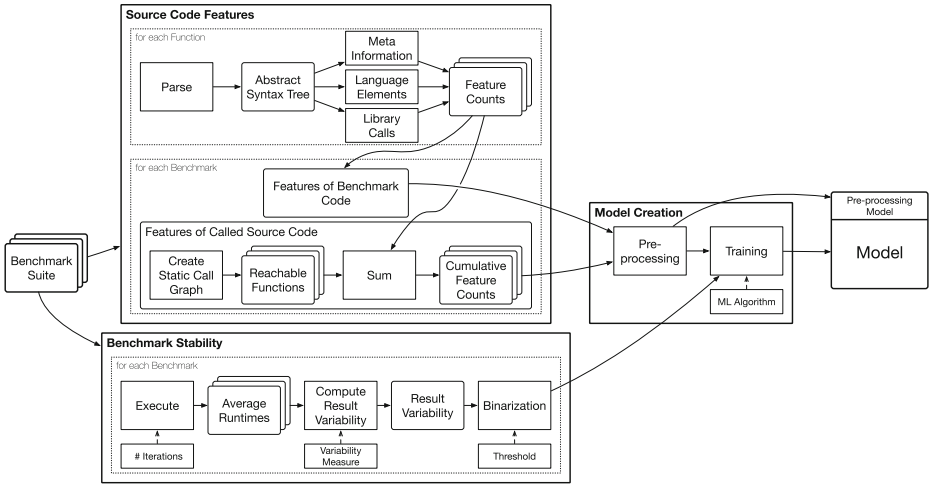


**Fig. 3** Training phase of the approach

**Table 1** Source code features of our approach. The Go code in green indicates the source code elements considered for the corresponding feature. All features are extracted for the benchmarks source code and for the code called by the benchmarks

| Type | Category | Name | Description | Go Code |
|---|---|---|---|---|
| Meta information | file | *fileloc* | LOCs of a file | - |
| | | *pkgfiles* | files in a Go package | - |
| | function | *loc* | LOC of a function | - |
| | | *namelen* | function name length | - |
| Language feature | control flow | *ifs* | if conditionals (potentially with preceding else) | `if ( ) { }` |
| | | *switches* | switch statements | `switch { }` |
| | | *switchcases* | cases in all switch statements | `switch { case X: ; case Y: ; }` |
| | | *loops* | loop statements | `for { }` |
| | | *nestedloops* | nested loop statements (with arbitrary depth) | `for { for { } }` |
| | | *funccalls* | function and method calls | `f()` or `v.f()` |
| | | *rets* | return statements | `return val` |
| | | *defers* | defer statements | `defer f()` |
| | | *panics* | calls to panic (Go's `throw` statement) | `panic( )` |
| | | *recovers* | calls to recover (Go's `catch` mechanism) | `if val := recover(); val != nil { }` |
| | | *cc* | Cyclomatic complexity by McCabe (1976) | - |
| | data | *vars* | variable declarations | `var x T` or `c := val` |
| | | *ptrs* | pointers | `&val` |
| | | *slices* | usage of slice types (e.g., for creation) | `[]T` |
| | | *maps* | usage of map types (e.g., for creation) | `map[T]U` |
| | concurrency | *gos* | creations of new goroutines | `go f()` |
| | | *channels* | usage of channel types (e.g., for creation) | `chan T`, `<-chan T`, or `chan<- T` |
| | | *chsends* | channel sends | `c <- val` |
| | | *chrecs* | channel receives | `<-c` |
| | | *chcloses* | channel closes | `close(c)` |
| | | *chranges* | looping over channels | `for v := range c { }` |

**Table 1** (continued)

| Type | Category | Name | Description | Go Code |
|---|---|---|---|---|
| | | *selects* | select statements | `select { }` |
| | | *selectcases* | cases in all select statements | `select { case X: ; case Y: ; }` |
| Standard library call | - | pkg(/subpkg)* e.g., *sync/atomic* | calls to functions and methods of 31 selected standard library packages | function calls: `pkg.f()` or method calls: `var v pkg.T; v.f()` |

overview of the 58 different features. Note that these 58 features occur once for the benchmarks' source code and once for the code called by the benchmarks, bringing the total number of features to 116.

The benefit of employing static over dynamic features is that it enables our approach to be utilized in scenarios where executing benchmarks is not possible or inopportune, such as during the development of benchmarks in the integrated development environment (IDE) or upon pushing to a version control system (VCS) to provide static analysis warnings to developers. The importance of feature categories and individual features for the prediction model are objects of investigation in Section 6.

### 3.1.1 Feature Extraction

Features are elements of the source code that can be extracted from a project's source code (files). Our approach considers three kinds of source code elements: (1) meta information, e.g., number of LOCs or files; (2) language elements, e.g., loops, conditionals, or variables; and (3) calls to standard library application programming interfaces (APIs) which "might affect" benchmark stability. We encode a feature as the number of occurrences of these source code elements in a function, e.g., the number of `if` branches or calls to a random number generator. We introduce our approach's features in the following and provide the rationale why they are promising for identifying unstable benchmarks.

**Meta Information.** Meta information features are language-agnostic and do not (necessarily) require complicated AST parsing but are potentially useful approximations for performance variability. Our approach considers meta information features on (1) file and (2) function granularity level. File meta information features assume that functions that are part of larger packages (*pkgfiles*) or contained in longer files (*fileloc*) invoke more and more diverse functionality. Similarly, longer functions (*loc*) and longer function names (*namelen*) might be indicative of more complex functionality. The suspected consequence is that benchmarks calling functions which follow these behaviours potentially have higher result variability and, therefore, are less stable. Such meta-information features have recently been successfully employed in prediction models for performance properties (Chen et al. 2020).

**Language Elements.** Similar to other works that build performance impact models based on source code (Huang et al. 2014; Mostafa et al. 2017; de Oliveira et al. 2017; Alshoaibi et al. 2019; Chen et al. 2020; Ding et al. 2020), we extract source code elements that require inter-procedural AST parsing. Our approach considers language element features falling into three main categories: (1) control flow elements, (2) variables and data types, and (3) concurrency elements.

Control flow elements comprise programming language features such as conditionals (*ifs*, *switches*, and *switchcases*), loops (*loops* and *nestedloops*), function lifecycle (*funccalls*, *rets*, and *defers*), exception handling (*panics* and *recovers*), and cyclomatic complexity (*cc*). More of these directly contribute to the increased complexity of a function which in turn might have a negative impact on benchmark variability. Moreover, research has often identified loops as root cause of performance problems (Jin et al. 2012; Sandoval Alcocer and Bergel 2015; Selakovic and Pradel 2016; Chen and Shang 2017; Zhao et al. 2020); and all of these control flow elements have recently been employed in machine-learning-based prediction of performance changes (Chen et al. 2020; Ding et al. 2020).

Data features could have an impact on a function's stack size, because Go stacks are initially 2kB and can dynamically grow (Go Authors 2020a). Benchmarks might encounter more result variability due to dynamically growing stacks. The other potential impact is frequent garbage collector (GC) activity. More variables (*vars*), pointers (*ptrs*), and built-in dynamic data structures (*slices* and *maps*) could increase pressure on the GC, caused by more allocated (and then reclaimed) heap memory objects (Hudson 2018). Also, research found that complex data structures and collections are often expensive to handle and require more resources (Huang et al. 2014; Chen and Shang 2017; Costa et al. 2017), which could impact benchmark variability.

Finally, Go has built-in support for concurrency in form of lightweight user-level threads (goroutines), channels as communication primitives among goroutines, and channel communication through sending and receiving messages as well as non-deterministic selection among multiple channel operations (Go Authors 2020b). These concurrency features are often used in real-life Go programs (Dilley and Lange 2019) and a potential cause for benchmark instability due to the inherent non-determinism of thread schedulers. Also, research found that performance failures often stem from concurrency and synchronization issues (Jin et al. 2012; Alam et al. 2017; Chen and Shang 2017; Zhao et al. 2020).

**Library Calls.** Standard library packages encapsulate behaviour that are essential for all programs. They enable file and network I/O, communication with the underlying operating system (OS), text or string processing, and concurrency primitives. As most of these functionalities rely on or are backed by non-deterministic behaviour; e.g., waiting for locks to become available, blocking on I/O, sending and receiving data over the network; calls to standard library APIs might affect performance variability more than "regular" function calls. Furthermore, inefficient or wrong application programming interface (API) calls as well as concurrency and synchronisation have been identified as root causes of performance bugs (Jin et al. 2012; Sandoval Alcocer and Bergel 2015; Selakovic and Pradel 2016; Alam et al. 2017; Chen and Shang 2017; Zhao et al. 2020). We assume that these calls can serve as proxies for benchmark stability to statically identify unstable benchmarks, without executing them. Hence, we encode standard library calls as individual source code features aggregated on package-level, i.e., all calls to a particular package contribute to the same package-level feature. Table 2 depicts the Go standard library features used by our approach, and the bottom row in Table 1 shows their source code representation.

### 3.1.2 Feature Combination

Source-code-induced benchmark variability is not only determined by the benchmark's body but also the source code it invokes. Consequently, we need to combine the

**Table 2** Standard library call features

| Package | Category | Description |
|---|---|---|
| *bufio* | io | buffered I/O |
| *bytes* | bytes | manipulating byte slices |
| *crypto* | bytes | cryptographic types and constants |
| *database/sql* | io | generic interfaces for SQL databases |
| *encoding* | bytes | interfaces for byte to text conversions |
| *encoding/binary* | bytes | binary encodings |
| *encoding/csv* | bytes | CSV encoding and decoding (RFC 4180) |
| *encoding/json* | bytes | JSON encoding and decoding (RFC 7159) |
| *encoding/xml* | bytes | XML 1.0 parser |
| *io* | io | I/O interfaces and primitives |
| *io/ioutil* | io | I/O utility functions |
| *math* | math | basic mathematical functions |
| *math/rand* | math | pseudo-random number generators |
| *mime* | io | (partial) MIME implementation |
| *net* | io | network I/O including TCP, UDP, domain name resolution, and Unix sockets |
| *net/http* | io | HTTP client and server implementation |
| *net/http/httptest* | io | HTTP testing utility functions |
| *net/http/httptrace*[1] | io | tracing of HTTP requests |
| *net/http/httputil* | io | HTTP utility functions |
| *net/rpc* | io | RPC client and server implementation |
| *net/rpc/jsonrpc* | io | JSON codec for *net/rpc* (JSON-RPC 1.0) |
| *net/smtp* | io | SMTP implementation (RFC 5321) |
| *net/textproto* | io | text-based request/response protocol |
| *os* | os | platform-independent interface to OS functionality |
| *os/exec* | os | running external commands/processes |
| *os/signal* | os | signals from the OS (e.g., SIGKILL) |
| *sort* | bytes | sorting of slices and user-defined collections |
| *strconv* | bytes | string conversions to/from primitives |
| *sync* | concurrency | synchronization primitives (e.g., mutexes) |
| *sync/atomic* | concurrency | low-level atomic memory primitives |
| *syscall* | os | interface to low-level OS primtives |

[1] *net/http/httptrace* is not present in the evaluation of our approach, because no study object uses the package

previously described source code features for *all* functions a benchmark calls. To go from intra-procedural source code features to inter-procedural ones, we employ static CGs rooted at every benchmark and aggregate the source code features of all reachable functions by summing up their values. Note that for the features of the benchmark's source code no combination is necessary, as the final feature values correspond to the extracted feature counts of the benchmark's body. The "Features of Called Source Code" block in Fig. 3 visualizes the feature combination process.

We rely on the *callgraph*[3] tool which is part of the official extended Go distribution. It constructs static CGs through inclusion-based points-to analysis using Andersen's algorithm (Andersen 1994), which is the most precise algorithm available. The algorithm can be classified as (1) inclusion-based, which includes all the potential call targets, e.g., all implementations of an interface; (2) flow-insensitive, which ignores all the control-flow constructs and statement order; (3) field-sensitive, which builds separate points-to sets for distinct fields, e.g., in a struct; (4) context-insensitive for most functions and only context-sensitive for small functions, i.e., which does or does not consider the calling context, respectively; (5) having a context-sensitive heap, where different objects of the same type are distinguished based on the allocation site and calling context; and (6) a whole program analysis, which requires an intermediate representation (IR) for the complete program in static single assignment (SSA) form. Hind (2001) provides a more detailed description of these classification terms. Finally, the algorithm is fully sound unless the program uses reflection (`reflect` package), performs `unsafe.Pointer` conversions, or calls native code.

### 3.2 Benchmark Stability

Our approach uses the benchmarks' result variability as its measure for stability. This requires performance measurements from repeated benchmark executions, i.e., iterations $i$ (see Section 2). Any measure of variability can be plugged into our approach. Such a measure computes the variability across all the iterations of a benchmark. To make the result variabilities comparable across different benchmarks, the measure should normalize the variability. This normalization can be performed by dividing the variability value by an aggregate value, such as the arithmetic mean or the median of the measurement iterations. The normalized variability value is continuous, ranging from 0 (no variability) to theoretically $+\infty$ (infinite variability). Typical measures are the relative confidence interval width (RCIW) or the median absolute deviation (MAD) divided by the median, thereafter called relative median absolute deviation (RMAD). Note, because performance data is usually not normally distributed, i.e., long-tailed or multi-modal (Curtsinger and Berger 2013), we can not use variability measures that assume a normal distribution, such as the coefficient of variation (CV).

The final step for defining the stability of a benchmark is binarization, which transforms the benchmarks' result variabilities into the binary classes *stable* or *unstable* based on a defined threshold. For example, if we consider performance test case (benchmark) selection of only stable benchmarks, the exact variability value is secondary as a binary answer, i.e., whether to select or not, is sufficient. We consider a benchmark to be *stable* if its result variability is below a certain threshold $t$ and *unstable* otherwise. A threshold $t$ is provided in percentages. For example, if $t$ is 3%, binarization assign the class *stable* if the result variability is in [0, 0.03), and *unstable* otherwise.

The variability measure, number of iterations, and threshold are approach parameters and objects of investigation in Section 5.

### 3.3 Model Creation

The final part of the training phase of our approach consists of using the source code features and benchmark stabilities to create a machine learning model.

[3] https://pkg.go.dev/golang.org/x/tools/cmd/callgraph

We first pre-process the data to ease the prediction task, by applying standardization and variance-based feature selection at minimum. These operations are solely based on the independent variables, i.e., the source code features. Additional pre-processing steps can be added if desired or necessary. Note that these operations are based on the data used for training: the variance is computed during the training of the model and needs to be stored together with the model in order to pre-process the data during the usage.

The second step is training a machine learning model. As the benchmark stabilities are binary classes, i.e., *stable* or *unstable*, any binary classification algorithm can be used to create the model. Our approach assigns the class 0 to *stable* and 1 to *unstable* benchmarks. Consequently, the main objective is to identify unstable benchmarks, and the model is trained accordingly. This is similar to defect prediction where defective modules, classes, or functions are the prediction goal, and we expect a similar imbalance between stable and unstable benchmarks (Turhan et al. 2009; Zimmermann et al. 2009).

The classification algorithm and more sophisticated pre-processing steps are objects of investigation in Section 5.

### 3.4 Model Usage

Once the model is created, it can be used to predict a benchmark's (in)stability.

Specifically, the process requires the following steps, also depicted in Fig. 4:

(1) the new benchmarks are processed by using the same source code feature extraction as in the training phase;
(2) we apply the pre-processing operations, i.e., standardization and feature selection, by using the variance values computed and features selected during the training phase; and
(3) we use the pre-processed data as input to the binary classification model to predict the benchmark stability.

As explained above, the model will give an answer for the definition of stability with which it has been trained, therefore depending on the used variability measure, number of iterations, and threshold.

## 4 Study Design

We perform a laboratory experiment (Stol and Fitzgerald 2018) to assess the prediction performance of the approach's machine learning model and study the importance of individual features and feature categories. The experiment utilizes 230 OSS projects written in Go

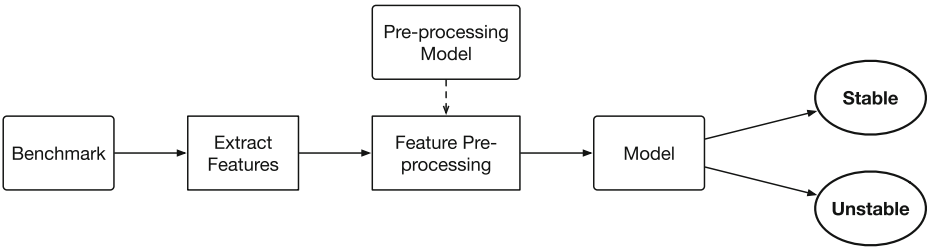


**Fig. 4** Usage phase of our approach

having 4,461 unique benchmarks. A replication package containing all data and scripts as well as additional data not presented in this paper is available online (Laaber et al. 2021).

### 4.1 Research Questions

In the context of the study, we formulate the following research questions.

**RQ 1** Can we predict benchmark instability with statically-computed source code features?

First, we want to assess the performance of a binary classification model to predict whether a benchmark will be stable or unstable. This model is built using the static source code features outlined in Section 3.1.

**RQ 1.1** How does the prediction performance change when the model is trained with different stability thresholds?

The definition of benchmark stability relies on the threshold $t$, which divides the continuous value of the benchmark variability into two binary classes, i.e., *stable* and *unstable*. We want to study whether and to what degree training the model on different stability thresholds $t$ impacts prediction performance.

**RQ 1.2** How does the prediction performance change when the model is trained on benchmark executions with different numbers of iterations?

The benchmark variability is directly affected by the number of repeated measurements, i.e., iterations $i$, a benchmark is executed for; more iterations result in a reduced measured variability (see Section 2). We want to investigate whether and to what degree training the model on different numbers of iterations $i$ impacts prediction performance.

**RQ 1.3** How does the prediction performance change when removing co-linear and multi-co-linear features and applying class-rebalancing?

Our approach performs simple pre-preprocessing steps on the features, i.e., scaling and removing these that have low variance. This research question investigates whether and to what degree more sophisticated pre-processing steps impact the models' prediction performance. In particular, it assesses the impact of two often-used pre-processing steps: (1) removal of co-linear and multi-co-linear features and (2) class-rebalancing.

**RQ 1.4** Do different variability measures have an impact on the prediction performance?

Performance result variability can be computed with different variability measures; different measures might lead to different outcomes whether a benchmark is stable or unstable. As a consequence, our approach's model could experience varying prediction performance. We want to study the impact of different variability measures.

**RQ 2** Which are the most important individual features and feature categories for good prediction performance?

It is unlikely that all of the 116 features are equally important for good predictions. We want to understand which individual features and feature categories contribute most to a model's prediction performance.

### 4.2 Study Objects

We focus our study on benchmarks written in Go. Go statically compiles to machine code, which helps to increase benchmark stability compared to dynamically compiled languages such as Java (Laaber and Leitner 2018). By studying a statically compiled language, our experiment design removes the non-deterministic factor dynamic compilation and, therefore, increases reliability of the benchmark result, which increases internal validity.

We follow an approach outlined by Stefan et al. (2017) to mine GitHub repositories through GitHub's search API. The search considers all projects that use Go as one of their languages and contain at least one benchmark (see Section 2). This results in 10,707 projects.

As executing performance tests is a costly endeavor (Huang et al. 2014), it would be infeasible to execute all projects. Our sampling strategy's goal is to reduce the overall experiment execution time and to filter out "toy" projects. Therefore, we apply the following inclusion criteria to each project: (1) $>$ 50 commits, (2) $>$ 1 authors, (3) not a fork of another repository, (4) $>$ 1 stars, (5) $>$ 1 watchers, (6) $>$ 5 benchmarks, (7) $>$ 1,000 LOCs, and (8) a benchmark suite execution time of at most 2 hours. From the 10,707 projects, 483 adhere to our selection criteria.

We (try to) execute all of these for 2 hours but due to compilation or runtime errors many projects fail, so that we end up with a final data set containing 4,461 individual benchmarks belonging to 230 projects. To the best of our knowledge, this is the most extensive data set of Go OSS projects with benchmarks and their execution results. We provide the full list of projects, including their commit hashes, as part of our replication package (Laaber et al. 2021).

### 4.3 Execution Setup

In order to reduce confounding factors influencing the performance measurements (Mytkowicz et al. 2009; Curtsinger and Berger 2013; de Oliveira et al. 2013) and consequently our approach's definition of benchmark stability, we execute all benchmarks in a controlled bare-metal environment. The setup consists of four machines with a Intel(R) Xeon(R) central processing unit (CPU) E5-2620 v4 @ 2.10GHz with 8 cores, 20MB cache, and 64GB random-access memory (RAM). We disabled hyper-threading, frequency scaling, and Intel's TurboBoost. As OS the machines use Fedora Linux 24, and measurements were taken in the period between November and December 2017. The benchmarks are compiled and run with Go 1.9.2. Every benchmark is executed for a 1 second duration (which is the default in Go at the time of execution) as often as possible and the average runtime is reported, which corresponds to one iteration. 97.4% of the benchmarks in our dataset have an average runtime below 1ms, which correspond to more than 1,000 executions per iteration. The benchmark suite of each project is repeatedly executed for 2 hours, resulting in multiple iterations $i$ per benchmark.

### 4.4 Benchmark Stability Parameterization

The benchmark stability serves as our approach's dependent variable, which comes with a few knobs for parameterization. The concrete parameterization can have an impact on our study and, consequently, on the conclusions drawn from it. In the following, we outline the parameter values used in our study for the (1) number of benchmark iterations, (2) variability measure, and (3) binarization threshold.

### 4.4.1 Number of Benchmark Iterations

To study the impact of the number of iterations $i$ on the prediction performance in RQ 1.2, we perform all analyses with multiple numbers of iterations. Previous research has shown that Go benchmarks are experiencing less result variability than Java benchmarks (Laaber and Leitner 2018). This is likely due to Go being a statically compiled and linked language as opposed to dynamically compiled like Java. By default Go benchmarks are only executed with a single iteration, but in order to follow performance engineering best practice (Georges et al. 2007) and to be able to compute a benchmark's variability (variability can not be computed from a single measurement iteration), we choose $i \in \{5, 10, 20, 30\}$. These values are in line with best practice, i.e., Georges et al. (2007) suggest 30 iterations, and cover a range that previous research used for their performance measurements. For example, 5 iterations are used by Blackburn et al. (2004), Jangda et al. (2019), and Mühlbauer et al. (2020); 10 iterations are used by Selakovic and Pradel (2016), Song and Lu (2017), and Kaltenecker et al. (2019); 20 iterations are used by Laaber and Leitner (2018) and were the default for Java Microbenchmarking Harness (JHM) benchmarks (Shipilev 2018); and 30 iterations are used by Curtsinger and Berger (2013), Blackburn et al. (2016), and Chen et al. (2020).

For RQ 2, we rely on the number of iterations that yields the best prediction performance of the best performing classification algorithm.

### 4.4.2 Variability Measure

The variability measure is central to decide which benchmarks are stable or unstable. Different measures exist which estimate a benchmark's variability from its set of measurement iterations. For RQ 1 and RQ 2, our study mainly relies on a single variability measure, i.e., RCIW of the median estimated with Maritz-Jarrett's technique (Maritz and Jarrett 1978). However, as the variability measure only estimates a benchmark's stability, we also study the impact of two other variability measures in RQ 1.4, i.e., RCIW of the mean with bootstrap (Davison and Hinkley 1997) and the median absolute deviation (MAD) divided by the median (Arachchige et al. 2020), in the following called RMAD.

Formally, let $M^b$ be the set of iterations of a benchmark $b$. Then $M_i^b \subseteq M^b$ is the subset containing the first $i$ iterations so that $M_i^b = \{m_{it}^b | m_{it}^b \in M^b \wedge 1 \leq it \leq i\}$. We compute the variabilities of each benchmark for all numbers of iterations $i$ mentioned in the previous section.

**Maritz-Jarrett Confidence Interval of the Median Estimation.** Maritz and Jarrett (1978) introduced a technique to estimate the confidence interval of the median using the incomplete beta function. This allows to compute the RCIW of the median for distributions that are non-normal, such as performance data, and consist of few samples, which is the case in our experimental setting where the variability is estimated based on 5 to 30 iterations (Akinshin 2020a). We use an adapted version of SCIPY's `scipy.stats.mstats.mquantiles_cimj` function ($cimjhd$), which uses the median estimated with the Harrall-Davis quantile estimator (Harrell and Davis 1982) ($median_{hd}$) instead of the empirical median. $median_{hd}$ performs better for performance data when multiple modes are present (Akinshin 2020b). The RCIW is then defined as the function $rciw_{mjhd}$ in Section 1.

$$rciw_{mjhd}(M^b, cl) = \frac{cimjhd_{cl+\frac{1-cl}{2}}(M^b) - cimjhd_{\frac{1-cl}{2}}(M^b)}{median_{hd}(M^b)} \quad (1)$$

In our study, we use a confidence level of 99% ($cl = 0.99$) and, if not otherwise specified, refer to $rciw_{mjhd}$ as simply RCIW.

**Bootstrap Confidence Interval of the Mean Estimation.** The second variability measure computes the RCIW of the mean of a benchmark, which has been standard practice in performance engineering research (Kalibera and Jones 2012; Bulej et al. 2017a, b). It estimates the benchmark's population confidence interval from an iteration sample with bootstrap (Davison and Hinkley 1997; Kalibera and Jones 2012), a Monte Carlo simulation technique drawing random samples with replacement. Compared to the Maritz-Jarrett technique, the bootstrap technique might under-estimate the confidence interval for small samples.

Formally, the bootstrap sample set $B^s$ is defined as $B^s = \bigcup^s \overline{sample(M^b)}$, where $s$ is the number of bootstrap samples, $sample$ is the function drawing the random sample with replacement from the measurement iterations $M^b$, and overscores indicate the arithmetic mean. The RCIW is then defined as the function $rciw_{boot}$ in Eq. 2.

$$rciw_{boot}(M^b, s, cl) = \frac{quantile_{cl+\frac{1-cl}{2}}(B^s) - quantile_{\frac{1-cl}{2}}(B^s)}{\overline{M^b}} \tag{2}$$

In our study, we use a confidence level of 99% ($cl = 0.99$) and draw 10,000 bootstrap samples ($s = 10,000$) with replacement (Hesterberg 2015).

**Relative median absolute deviation (RMAD).** The final variability measure RMAD is similar to the often used CV, but it is applicable to non-normal distributions (Arachchige et al. 2020; Akinshin 2021). RMAD normalizes the MAD by the median; uses the Harrell-Davis quantile estimator to estimate the median (Harrell and Davis 1982), similar to the Maritz-Jarrett technique; and performs bias-correction to the scale factor $C$ according to Park et al. (2020). RMAD is defined as the function $rmad$ in Eq. 3.

$$rmad(M^b) = C_i * \frac{median_{hd}(\bigcup_{m \in M^b} m - median_{hd}(M^b))}{median_{hd}(M^b)} \tag{3}$$

The scale factor $C_i$ depends on the number of iterations $i$ used for computing RMAD. We use the suggestions by Park et al. (2020): 1.803927 for $i = 5$, 1.624681 for $i = 10$, 1.545705 for $i = 20$, and 1.523031 for $i = 30$.

### 4.4.3 Binarization Threshold

Binary classification requires transforming the benchmark variabilities of the 4,461 benchmarks, as computed by the variability measure, into two classes, i.e., *stable* and *unstable*. For this, we define the threshold $t$ that divides the RCIW values into these classes. In RQ 1.1, our study investigates the model's prediction performance when using different thresholds for binarization. The thresholds under study are $t \in \{1\%, 3\%, 5\%, 10\%\}$, which are informed by previous research: Georges et al. (2007) report that performance measurement variability is often around 3%, Mytkowicz et al. (2009) mention that measurement bias can obfuscate a performance change as large as 10%, and Huang et al. (2014) communicate that in their real-world studies performance regressions between 3% and 20% are considered as relevant. Binarization sets *unstable* as the positive value (1), i.e., the one we are primarily interested in identifying, and *stable* as the negative value (0).

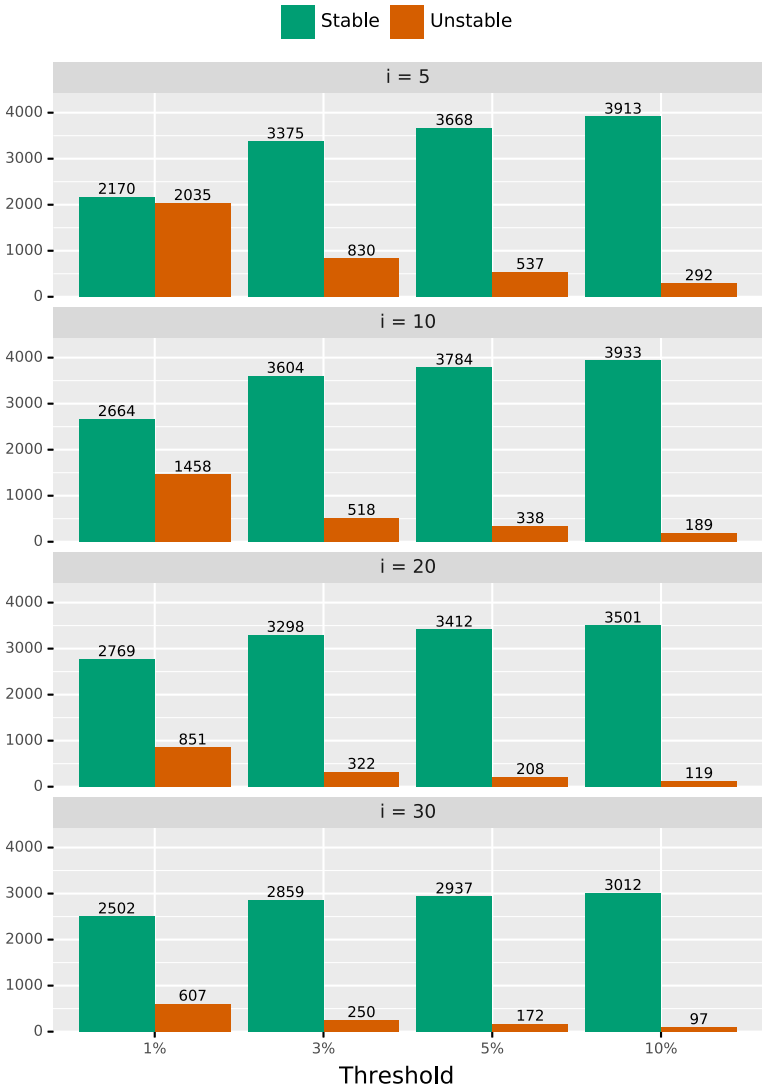


**Fig. 5** Distributions of the data used to train the classifiers after binarization

We apply binarization for all benchmarks after the different numbers of iterations ($i$), i.e., 5, 10, 20, and 30. Figure 5 depicts the resulting distributions. It shows that the data is imbalanced for all iteration-threshold pairs. This phenomenon can also be observed in the defect prediction literature (Tantithamthavorn et al. 2020), where there are much fewer defective methods, classes, or modules than ones without defects. Note that the total number of benchmarks varies depending on the number of executed iterations: 4,205 with 5, 4,122 with 10, 3,620 with 20, and 3,109 with 30. This is due to our study execution design (see Section 4.3) which restricts a full benchmark suite execution to a maximum of 2 hours; hence, our data set does not contain the same number of benchmark iterations, depending on the project (or suite) it belongs to. Figure 6 shows as scatter plot the relation between RCIW and the

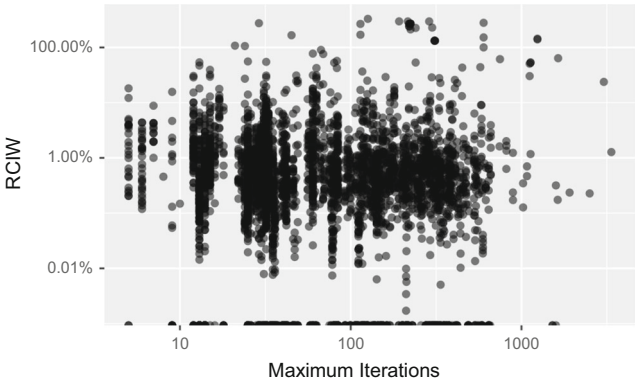


**Fig. 6** Relation between RCIW and maximum number of iterations in our experiment

maximum number of iterations retrieved in our experiment. We observe that benchmarks with highly variable results are evenly distributed; hence, removing benchmarks where only fewer iterations are available than the iteration value requires, i.e., 5, 10, 20, or 30, is unlikely to have a negative impact on our study and its conclusions.

For RQ 2, we rely on the threshold that yields the best prediction performance of the best performing classification algorithm.

### 4.5 Model Creation and Validation

Predicting whether benchmarks are stable or unstable involves creating and validating a machine learning model. In the following, we outline the (1) binary classification algorithms, (2) prediction performance metrics, (3) model validation approach, (4) feature importance approach, and (5) feature pre-processing steps.

#### 4.5.1 Binary Classification Algorithms

In RQ 1, we compare the prediction performance of 11 classification algorithms from the PYTHON library SCIKIT-LEARN (Pedregosa et al. 2011). RQ 2 then investigates the feature importance of the best-performing algorithm from RQ 1. The algorithms are described in the following:

**Naive Bayes (NB)** is an algorithm based on the Bayes' theorem of "naive" assumption of conditional independence between every feature pair (John and Langley 1995). We select the implementation of "Gaussian Naive Bayes" from SCIKIT-LEARN, where the likelihood of the features is assumed to be Gaussian.

**k-Nearest Neighbors (KNN)** is an instance-based learning algorithm, where the classification is computed with the majority vote method (Goldberger et al. 2004). The parameter $k$ indicates the number of neighbors.

**Logistic Regression (LR)** is a linear model used for classification, where the probabilities that describe the possible outcomes are modeled using a logistic function (Hosmer et al. 2013).

**Neural Networks (NN)** are a powerful method for supervised learning, especially effective in estimating functions that can depend on a large number of inputs (Ruck et al. 1990). For our experiments, we select the "Multi-layer Perceptron" implementation from SCIKIT-LEARN, a fully connected neural network with a linear activation function in all neurons.

**Decision Tree (DT)** is a non-parametric classification algorithm where the goal is to predict by learning simple decision rules inferred from the input data (Quinlan 1986).

**Linear Discriminant Analysis (LDA)** is a classification algorithm based on the supervised dimensionality reduction that projects the input data to a linear subspace (Friedman 1991).

**Support Vector Machines (SVMs)** are discriminative classification algorithms, formally defined by a separating hyperplane (Cortes and Vapnik 1995). Given a labeled training data, the algorithm outputs an optimal hyperplane categorizing new examples. Different "kernel" functions can be specified for the decision functions. We report the performance results of two different kernels for SVMs: the **linear kernel (LSVM)** and the **radial kernel (RSVM)**.

**Random Forest (RF)** is a bagging/ensemble method, whose final results depend on the decisions of multiple classifiers, i.e., multiple decision trees in this case (Breiman 2001).

**Boosting** is another ensemble method, where multiple iterations of the same algorithm, e.g., decision tree, are performed. At every new step, it trains the model with a modified version of the input data. We use two boosting algorithms, both based on decision trees, namely **Adaptive Boosting (AB)** (Freund and Schapire 1997), and **Gradient Boosting (GB)** (Friedman 2001). Adaptive Boosting gives weights to the votes of every trained models, adapting them at every step so that a weaker model will have a lower impact on the final decision compared to the stronger ones. Instead, Gradient Boosting trains the models in a gradual, additive, and sequential manner.

We do not apply any tuning to optimize the algorithms' hyper-parameters but use the standard configuration instead, as provided by SCIKIT-LEARN version `0.24.1`. A study on the effects of tuning on benchmark instability prediction is an important subject for future work.

From now on, we will use the names of the classifiers to also indicate the type of trained models.

#### 4.5.2 Prediction Performance Metrics

We evaluate the different models along the following prediction performance metrics:

**Precision** describes the ability of a classifier *not* to label a sample that is negative as positive (Buckland and Gey 1994). It is defined as the ratio $\frac{TP}{TP+FP}$, where TP is the number of true positives and FP the number of false positives. Its values lie between 0.0 and 1.0 ranging from worst to best, respectively.

**Recall** represents the ability of the classifier to find all positive samples (Buckland and Gey 1994). It is defined as $\frac{TP}{TP+FN}$, where FN is the number of false negatives. Its values range from 0.0, i.e., the worst value, to 1.0, i.e., the best one.

**F-measure**, also known as "F1 score" or "F-score", is a weighted harmonic average of precision and recall, defined as $\frac{2*precision*recall}{precision+recall}$ (Chinchor 1992). Its values range from 0.0, i.e., the worst, to 1.0, the best. Intuitively, it is a metric that provides a better intuition about how well the model performs overall, as it is based on both precision and recall.

**Area Under the Curve (AUC)**, precisely area under the receiver operating characteristic curve (AUROC), represents the measure of the area of the ROC curve (Hanley and McNeil 1982). The ROC is a probability curve, plotting the true positives rate vs. false positives rate, showing the performance of a classifier at all classification thresholds. The AUC quantifies, between 0.0, i.e., the worst, and 1.0, i.e., the best, the area that the ROC draws on a plot. This metric represents the probability the model will score a randomly chosen positive class rather than a randomly chosen negative class. It is a form of accuracy measure, but instead of classic accuracy, it is more meaningful in the case of imbalanced data between positive and negative classes, as the distributions of our data set are (see Fig. 5).

**Matthews Correlation Coefficient (MCC)** is a measure of classification quality, which takes into consideration true and false positives and negatives (Matthews 1975). MCC is regarded as a meaningful measure even if the classes are imbalanced. Its values lie between -1.0, i.e., a completely wrong prediction, and +1.0, i.e., a perfect one. 0.0 represents an average random prediction.

In the remainder of the paper, we predominantly discuss the models' prediction performance along the AUC and MCC metrics. AUC (Bradley 1997) and MCC (Chicco and Jurman 2020) have been shown to be reliable metrics, especially for binary classification. We provide insights about the other metrics only in a few cases where more details are of interest. All the details about the above-mentioned metrics are available for reference in our replication package (Laaber et al. 2021).

**Metrics Interpretation.** We provide an interpretation of the metrics in the context of benchmark instability prediction. If precision is high, the classifier is able to recognize unstable benchmarks correctly. A simple classifier that maximizes precision would always answers with a negative value, i.e., "stable". Instead, the recall describes the capability of recognizing all benchmarks that are unstable. To maximize recall, a simple classifier would always answer with "unstable" for all benchmarks.

In case of high precision and low recall, we can be confident that the classifier (mostly) classifies unstable benchmarks as such, i.e., only few benchmarks classified as unstable are indeed stable (low false positive (FP)-rate). However, the classifier might have missed many unstable benchmarks because of the low recall, i.e., many benchmarks that are indeed unstable are classified as stable (high false negative (FN)-rate). Conversely, a classifier with low precision and high recall is able to identify most benchmarks that are unstable as such (low FN-rate), but it also classifies many benchmarks as "unstable" that are indeed "stable" (high FP-rate). Depending on the application scenario of an unstable benchmark classifier, recall, precision, or both should be maximized for. We present more discussion on this trade-off and application scenarios in Section 7. AUC and MCC consider both precision and recall, being more indicative of the general performance of the classifiers.

### 4.5.3 Model Validation

Due to our data set being relatively small ($\geq$ 3,620 instances) for machine learning purposes and highly imbalanced, we employ a repeated k-fold cross validation approach. The data is randomly split into $k$ folds, and the model is trained on $k - 1$ folds and validated on the remaining $k^{\text{th}}$ fold. This process is repeated $k$ times collecting a total of $k$ evaluations, where each time a different $k^{\text{th}}$ fold is used for validation. The k-fold cross validation is then

repeated $m$ times, and the whole data set is shuffled before each repetition. We set $k = 10$ and $m = 30$, collecting a total of $10 \times 30 = 300$ evaluations for each combination of model, number of iterations, and threshold. This way, we can take advantage of a high number of evaluations and then apply statistical tests to investigate the significance and effect size of our results. All analyses in our study employ this cross validation approach.

#### 4.5.4 Feature Importance

In RQ 2, we investigate the importance of individual features and feature categories for good prediction performance of the model. We perform both the analyses on the best-performing combination of classification algorithm, number of iterations, and threshold from RQ 1. The data for the remaining combinations are part of our online appendix (Laaber et al. 2021).

The first part investigates the individual feature importances. We apply the permutation feature importance on the training data of each fold (Altmann et al. 2010), using the `sklearn.inspection.permutation_importance` function. It randomly mutates each feature $n$ times, computes the decrease in a specified prediction performance metric, and returns the mean decrease across all $n$ repetitions. We use $n = 30$ and MCC as metric in our study. This yields a single importance value per feature and fold. Note that due to feature pre-processing (i.e., feature selection) not every feature might receive an importance value for every fold.

The second part investigates the feature category importances. The idea is to identify groups of features that have a similar purpose, e.g., I/O or concurrency, and show their importance. The study investigates the following categories:

(1) all 58 features for the benchmark code (*bench*);
(2) all 58 features for the code called by the benchmark (*code*);
(3) meta information features (*meta*);
(4) all programming language features (*pl*);
(5) control flow features (*pl-cf*);
(6) data features (*pl-data*);
(7) concurrency features of the programming language (*pl-conc*);
(8) all library call features (*lib*);
(9) library call features related to I/O functionality (*lib-io*);
(10) library call features related to string and byte manipulation (*lib-bytes*);
(11) library call features interfacing with the OS (*lib-os*);
(12) library call features related to math and randomization (*lib-math*); and
(13) library call features related to concurrency (*lib-conc*).

Tables 1 and 2 show a mapping of individual features to feature categories. We iteratively remove each category's features and report the decrease in prediction performance, i.e., MCC in our study. This yields a single category importance value per category and fold.

#### 4.5.5 Pre-Processing

Pre-processing of source code features can have an impact on machine learning models. As outlined in Section 3, our approach always applies two basic pre-processing steps: (1) standardization of the features by removing the mean and scaling to unit variance and (2) feature selection to remove all low-variance features.

In our study, we additionally apply more sophisticated pre-processing steps, i.e., removing correlated features and class-rebalancing. Previous research has shown that

removing correlated features is especially important when investigating feature importances (Jiarpakdee et al. 2019) and that class-rebalancing can positively impact prediction importance (Tantithamthavorn et al. 2020). We choose two candidates for each pre-processing step that have been shown to be superior over other techniques (Tantithamthavorn et al. 2020; Jiarpakdee et al. 2020):

(1) AutoSpearman (Jiarpakdee et al. 2018), a feature selection technique to remove all co-linear and multi-co-linear features. It works in two phases: (a) it employs Spearman rank (Hauke and Kossowski 2011) to identify and remove strongly correlated features above a threshold of $|\rho > 0.7|$ (Kraemer et al. 2003); and (b) it uses the variance inflation factor (VIF) to identify and remove features with strong multi-co-linearity to other features, having a VIF threshold above 5 (Fox 2016).
(2) SMOTE (Chawla et al. 2002), a class-rebalancing technique that oversamples the minority class, i.e., in our setting *unstable*.

All pre-processing steps are applied to the training data before creating the models. Standardization and feature selection (based on variance and with AutoSpearman) yield a pre-precessing model (see Figs. 3 and 4) which is later also applied to the test data. This is required to scale the feature values and select the same features according to the training data. Class-rebalancing with SMOTE is *only* applied to the training data and *never* to the test data.

RQ 1.3 investigates the impact of AutoSpearman and SMOTE on the models' prediction performance. For this, we asses the prediction performance of all studied algorithms, numbers of iterations, and thresholds with a combination of the two pre-processing steps: (1) a baseline without AutoSpearman and SMOTE, (2) only AutoSpearman, (3) only SMOTE, and (4) both AutoSpearman and SMOTE.

The study of the feature importances in RQ 2 requires careful selection of pre-processing steps to be applied. When interpreting prediction models, class-rebalancing techniques must not be applied as they change the underlying assumption that training and test data have the same distribution (Tantithamthavorn et al. 2020). If one applies class-rebalancing, this assumption is violated and the feature importances are biased, which is called concept drift. Hence, we do not apply class-rebalancing (i.e., SMOTE) for the analyses in RQ 2. The application of AutoSpearman depends on which feature importance analysis is performed. For the individual feature importances, we apply AutoSpearman before modelling, because co-linear and multi-co-linear features can bias the conclusions of the permutation importance (Jiarpakdee et al. 2018). For the feature category importance, we do not apply AutoSpearman, because we want to keep the categories coherent without removing features that may contribute to the model performance.

## 5 RQ 1 – Classifying Benchmarks as Unstable

To answer RQ 1, we investigate whether the statically-computed features can be used to build effective machine-learning-based predictors. This section describes the statistical tests and the results.

### 5.1 Statistical Tests

To ensure that our reported results are valid from a statistical perspective, we employ statistical hypothesis testing and effect size measures. All reported statistical test

results are significant at a level ($\alpha$) of 0.01. We use the term "group of observations" to indicate the prediction performance scores for all folds of each of the models under study.

To identify the type of statistical tests that are suitable for the distributions of our data, we perform a normality test. For all collected groups of observations, we apply the "D'Agostino's $K^2$" test (D'Agostino et al. 1990), whose null hypothesis states that a group of observations are normally distributed. For the majority of the observations, we can reject the null hypothesis ($p$-value $< 0.01$), thus not allowing to use parametric tests for further investigations. For this reason, we only use non-parametric tests to uniform our discussion. We report the median values of the metrics under analysis, since the non-parametric tests generally refer to the median and to ease understandability of the results. Therefore, when we compare combinations of algorithm, iterations, and threshold, we use the median value as an indicator of the performance over multiple observations, i.e., results from repeated k-fold cross validation. The term "range" thus refers to the minimum and maximum values between median values. We refer the interested reader to our external appendix, which contains the raw data set and enables straight-forward computation of other statistics (Laaber et al. 2021).

To compare the observations, we apply the "Kruskal-Wallis H" test (Kruskal and Wallis 1952), i.e., the non-parametric version of the ANOVA test. The null hypothesis states that the observations' median of all tested groups are equal. The test is applied to multiple groups simultaneously but can not identify exactly *where* and *how much* the groups are statistically different.

When able to reject the null hypothesis, i.e., the median among all the groups is statistically significantly different, we apply a post-hoc pairwise test to identify the pairs of groups of observations that are different. For this, we use the "Dunn's" test (Dunn 1964) where the null hypothesis states that there is no difference.

To measure *how much* two groups are different from each other, we compute the "Vargha-Delaney $\hat{A}_{12}$" test (Vargha and Delaney 2000) for the effect size to characterize the magnitude of such a difference. $\hat{A}_{12} = 0.5$ if two groups (observations) are statistically indistinguishable. $\hat{A}_{12} > 0.5$ means that, on average over all observations, the first group obtains larger values than the one it is compared to, $\hat{A}_{12} < 0.5$ otherwise. The magnitude values can be summarized into 4 nominal categories, which rely on the scaled $\hat{A}_{12}$ defined as $\hat{A}_{12}^{scaled} = (\hat{A}_{12} - 0.5) * 2$ (Hess and Kromrey 2004): "negligible" ($|\hat{A}_{12}^{scaled}| < 0.147$), "small" ($0.147 \leq |\hat{A}_{12}^{scaled}| < 0.33$), "medium" ($0.33 \leq |\hat{A}_{12}^{scaled}| < 0.474$), and "large" ($|\hat{A}_{12}^{scaled}| \geq 0.474$).

### 5.2 Results

We analyze the results along three dimensions: (1) the 11 classification models; (2) the threshold $t$ used for binarization, i.e., which benchmark variability (RCIW) values are considered stable or unstable; and (3) the number of iterations $i$ used for calculating the benchmark result variability.

RQ 1 aims at assessing the effectiveness of machine learning for predicting, i.e., classifying, unstable benchmarks. It gives an overview of the best performing models. RQ 1.1 and RQ 1.2 further investigate whether models trained on different thresholds $t$ and with different numbers of iterations $i$, respectively, exhibit different prediction performance. RQ 1.3 studies the impact of sophisticated pre-processing steps on prediction performance, whereas RQ 1.4 the impact of relying on different variability measures.

### 5.2.1 Overall Comparison

Figure 7 shows the line plots of the median prediction performance (on the y-axis), over 300 observations, for all the evaluation metrics under study. The row facets show the classification models, whereas the column facets represent the number of iterations $i$, i.e., 5, 10, 20, and 30. The x-axis depicts the threshold values $t$, i.e., 1%, 3%, 5%, 10%.

**Generally High-Performance Algorithms.** Referring to MCC as an indicator of the overall effectiveness of the classifiers, Random Forest outperforms the other models for all combinations of iterations and thresholds. Across all combinations, Random Forest's MCC median values range from 0.43 to 0.61. The same applies for AUC, where Random Forest's prediction performance ranges from 0.79 to 0.90. This is in-line with other models, whose MCC median values are over 0.50: (1) Adaptive Boosting (max 0.57), (2) Gradient Boosting (max 0.53), and (3) Decision Tree (max 0.51).

**High Precision Algorithms.** The algorithms with the highest precision are: (1) Radial Support Vector Machines (from 0.70 to 1.0). (2) Linear Support Vector Machines (from 0.5 to 1.0), and (3) Gradient Boosting (from 0.67 to 1.0). It is likely that these algorithms tend to train the classifier to answer mostly with "stable", i.e., the negative value for the metric. However, all of these algorithms suffer from low recall, specifically (1) Radial Support Vector Machines (from 0.06 to 0.48). (2) Linear Support Vector Machines (from 0.06 to 0.42), and (3) Gradient Boosting (from 0.15 to 0.66). arguably rendering them inferior to algorithms with balanced precision and recall.

Further, we observe a growing trend in precision with increasing threshold and iterations (see Fig. 7). We discuss the impact of these values in more detail with RQ 1.1 and RQ 1.2 in Sections 5.2.2 and 5.2.3, respectively. This might be a phenomenon related to the reduced number of unstable benchmarks with an increasing number of iterations and threshold values (see Fig. 5). All of the above-mentioned algorithms are based on subsequent iterations for training, which we set to 100,000 in our experiments. It is likely that these algorithms do not have enough "time" to learn how to recognize such a low number of unstable benchmarks. Instead, the Neural Networks algorithm, which is similarly based on the same concept of iterations, does not fail in the same way. Neural Networks obtain almost 0.5 for the MCC scores. It is likely that their complex structure of layers is more effective in capturing the features that characterize unstable benchmarks, within the given number of iterations. Thus, hyper-parameter tuning is expected to improve the general performance of these classifiers.

**High Recall Algorithm.** The model with, by far, the highest recall is Naive Bayes. The precision is low (often $\approx 0.1$), therefore Naive Bayes is not able to correctly identify "unstable" benchmarks. Having high recall ($\approx 1.0$), it is likely that the algorithm trained the model to answer "unstable" in the majority of the cases. Indeed, precision and recall are not balanced, which is also reflected in a low MCC value (often $\approx 0.06$).

**Comparing to Baselines.** We also trained four basic classification algorithms as baselines, based on different strategies to generate predictions: (1) "stratified", assigning predictions based on the distribution of the data, (2) "most frequent", always predicting the most frequent label in the training data, (3) "uniform", generating predictions uniformly at random, (4) "prior", always predicting the class that maximizes the class prior. We verified that for all

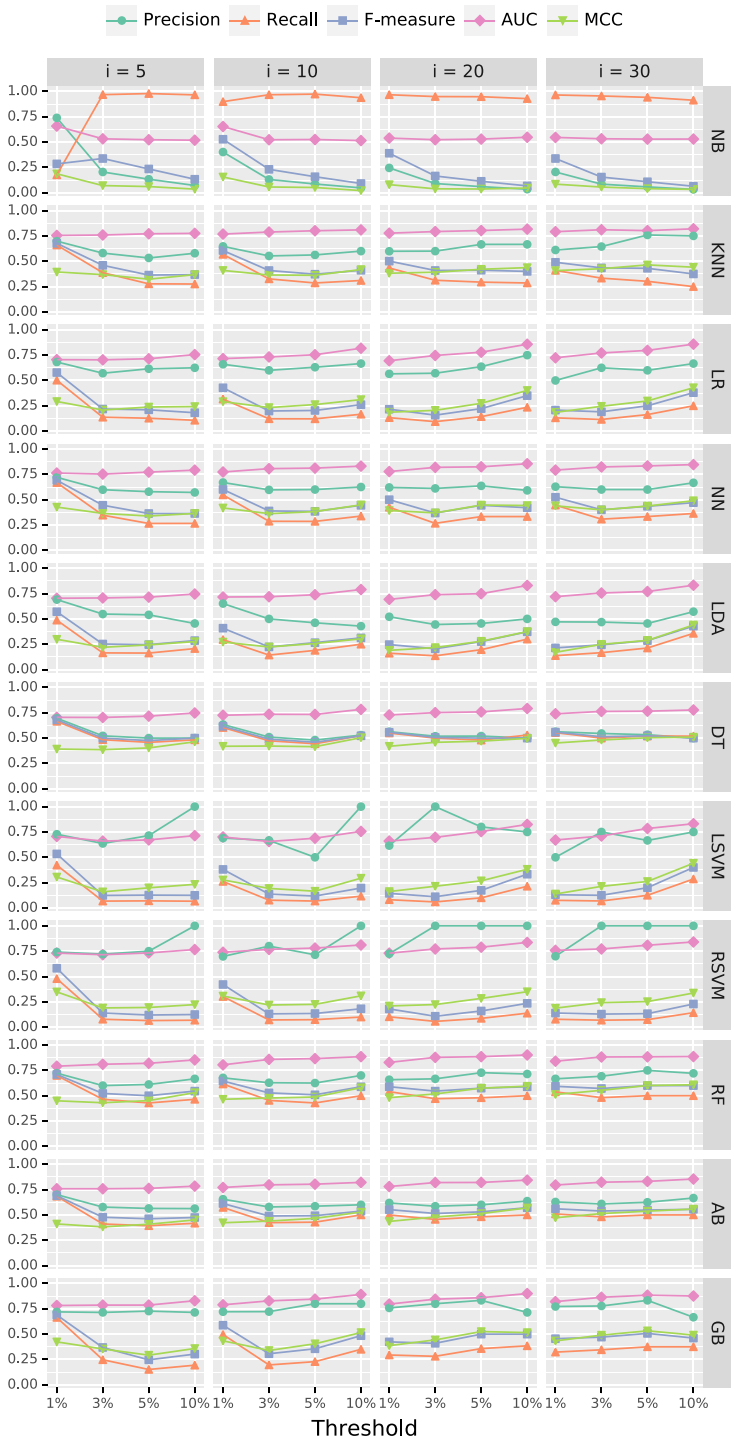


**Fig. 7** Metrics comparison across thresholds $t$, iterations $i$, and algorithms

cases the AUC and MCC values for the baselines are close to $\approx 0.5$ and $\approx 0.0$, respectively, clearly outperformed by all the other classifiers.

Further details are available in our replication package (Laaber et al. 2021).

> **RQ 1 Summary:** Machine learning models can be used to predict benchmark instability based on statically-computed source code features. In particular, Random Forest trained with standard parameters have an AUC value ranging from 0.79 to 0.90 and MCC from 0.43 to 0.61.

#### 5.2.2 RQ 1.1 – Impact of the Threshold

We now investigate wether training the model with different threshold values $t$ has an impact on prediction performance, for all combinations of iteration values $i$ and classification algorithms. More specifically, we fix two of the dimensions and vary the other one, i.e., we compare the evaluation scores of the models with the same values for a specific classification algorithm and iteration value. In this way, we can analyze the impact of a single dimension on the evaluation metrics.

We observe from Fig. 7 that both AUC and MCC values follow a growing trend when the threshold value increases, for the majority of algorithms. Graphically, it is possible to identify this behavior by following the lines of AUC and MCC for each distinct facet box. To validate this, we use the Kruskal-Wallis H test and verify that there is a statistically significant difference between observations whose median value is represented in Fig. 7. After applying the Dunn's post-hoc pairwise test, we can also verify whether there is a significant difference between consecutive threshold values. Moreover, we can measure the magnitude of such a difference with the Vargha-Delaney $\hat{A}_{12}$ test. The $\hat{A}_{12}$ value also indicates whether the first group of observations has values that are greater than the second group, on average. Thus, showing whether there is a statistically growing trend.

The Kruskal-Wallis H test shows that there is always a statistically significant difference between threshold values 1% and 10%, regardless of the model and number of iterations. The Dunn's test rejects the null hypothesis ($p$-value $< 0.01$) and the magnitude computed by the $\hat{A}_{12}$ test ranges from "small" to "large", except for k-Nearest Neighbors, Logistic Regression, Neural Networks, and Linear Support Vector Machines, when $i = 10$. We observe an upwards trend in prediction performance with an increasing threshold value for almost all the algorithms and iterations, except Naive Bayes, which always shows decreasing trends, i.e., both AUC and MCC have decreasing trends for all combinations of models and iteration values. k-Nearest Neighbors, Logistic Regression, Neural Network, Linear Support Vector Machines, Radial Support Vector Machines, and Gradient Boost have some decreasing AUC and MCC values with 5 iterations.

The increase in prediction performance with higher thresholds is likely an effect of measurement uncertainty (see Section 3.2), as reported by Georges et al. (2007) and Mytkowicz et al. (2009). Stable benchmarks (with low result variability) probably have different characteristics, in terms of the source code features they call, compared to unstable benchmarks. If the threshold value is small(er) and measurement uncertainty pushes the benchmark's result variability beyond the threshold, a benchmark is more likely to fall into the "wrong" class, i.e., stable vs. unstable. Consequently, its source code feature characteristics and stability class are misaligned and prediction performance suffers. Higher thresholds tend to increasingly assign benchmarks to their "right" stability class which positively impacts prediction performance.

Regarding Random Forest, i.e., the best overall model (see Section 5.2.1), we notice a clear growing trend for all the iteration values, except for $i = 5$, when the threshold values grows from 1% to 3%. In particular: (1) for 5 iterations, the AUC median values start from 0.79 (threshold at 1%) to reach 0.85 (threshold at 10%), whereas MCC from 0.45 to 0.53; (2) for 10 iterations, AUC ranges from 0.80 to 0.89, and MCC from 0.46 to 0.58; (3) for 20 iterations, AUC ranges from 0.83 to 0.90, and MCC from 0.48 to 0.59; (4) for 30 iterations, AUC ranges from 0.84 to 0.89, and MCC from 0.51 to 0.61. Therefore, considering MCC as the main indicator, for Random Forest the best prediction performance is always with a threshold $t$ of 10%, regardless of the number of iterations $i$.

**RQ 1.1 Summary:** The threshold choice, which defines whether a benchmark is *stable* or *unstable*, has an impact on the prediction performance. In the vast majority of the cases, the models with the highest threshold value $t$, i.e., 10%, reached the highest values for AUC and MCC.

### 5.2.3 RQ 1.2 – Impact of the Number of Benchmark Iterations

We are now interested in whether the number of benchmark iterations $i$ that a model is trained on has an impact on the prediction performance. As for RQ 1.1, we validate if the observations follow a growing trend for AUC and MCC when increasing the number of benchmark iterations. For this, we fix both the models and threshold values and analyze the values of the observations for AUC and MCC, for all iteration values. It is possible to visually observe this in Fig. 7 by looking at the points of a specific metric, e.g., AUC and MCC, and follow the horizontal grid lines of one of the thresholds, e.g., 10%, from left to right.

We can validate the following observations by running the Kruskal-Wallis H test. The test shows statistical significative difference ($p$-value $\geq 0.01$) for all the values of AUC and MCC.

We, again, apply the Dunn's test to pinpoint to the different observations, followed by the Vargha-Delaney's $\hat{A}_{12}$ test for the effect size. In particular, the following models do not show any significant difference between iterations values 5 and 30: (1) Naive Bayes when the threshold is 10%, (2) Neural Network when the threshold is 1%, and (3) Gradient Boosting when the threshold is 1%. Based on the Vargha-Delaney's $\hat{A}_{12}$ test, we find that the following combinations do not improve AUC and MCC values when the number of iterations increases: (1) Logistic Regression, (2) Linear Discriminant, (3) Linear Support Vector Machine, (4) Radial Support Vector Machines, all for a threshold of 1%, and (5) Naive Bayes for all thresholds. However, we observe that in the vast majority of the cases, 30 iterations result in the best prediction performance, considering MCC as the main indicator.

Similar to the prediction performance sensitivity caused by the threshold value (see Section 5.2.2), prediction of benchmark instability is affected by the number of iterations. Recall from Section 2 and Fig. 5, with an increase in number of iterations, a benchmark's variability decreases, i.e., the confidence interval narrows, and the benchmark becomes more stable. This has the effect that characterizing features of unstable benchmarks seem to be better associated with benchmark instability and, hence, prediction performance improves.

As for the previous research question, we analyze the trends of Random Forest, i.e., the best overall model. Also in this case, we notice a growing trend for all the threshold values. In detail: (1) for a threshold of 1%, the AUC median values start from 0.79 (5 iterations) to

reach 0.84 (30 iterations), whereas MCC from 0.45 to 0.51; (2) for a threshold of 3%, AUC ranges from 0.81 to 0.88, and MCC from 0.43 to 0.55; (3) for a threshold of 5%, AUC ranges from 0.82 to 0.88, and MCC from 0.45 to 0.60; and (4) for a threshold of 10%, AUC ranges from 0.85 to 0.89, and MCC from 0.53 to 0.61; Therefore, the best prediction performance with Random Forest is always with 30 iterations, regardless of the threshold value.

> **RQ 1.2 Summary:** The number of benchmark iterations $i$ does considerably impact the prediction performance. In the vast majority of the cases, the models with the largest number of benchmark iterations, i.e., 30, score highest in AUC and MCC.

#### 5.2.4 RQ 1.3 – Impact of Removing Correlated Features and Applying Class Re-Balancing

In this section, we investigate whether more sophisticated pre-processing steps have an impact on the prediction performance. We apply a combination of the following two pre-processing steps: (1) AutoSpearman (Jiarpakdee et al. 2018), a feature selection technique to remove all co-linear and multi-co-linear features; and (2) SMOTE (Chawla et al. 2002), a class-rebalancing technique that oversamples the minority class.

Figure 8 shows the difference in prediction performance for models with the pre-processing steps applied compared to models without the pre-processing steps applied, across all models and for all metrics. Each data point represents a paired difference of one particular model and one particular fold. These differences represent an "increment" when positive, a "decrement" otherwise. Overall, we observe that using pre-processing steps neither improves nor deteriorates the prediction performance for MCC and AUC. The Vargha-Delaney's test shows there is only a negligible statistical difference. In terms of precision, SMOTE increases prediction performance, whereas AutoSpearman does not have an impact. As for recall, SMOTE decreases prediction performance while AutoSpearman's increase is negligible.

Figure 9 focusses on Random Forest, the best model in our study. Employing AutoSpearman improves MCC and AUC by 0.023 and 0.005, respectively, as also confirmed by the statistical tests. As for precision and recall, the overall results are also confirmed for Random Forest: SMOTE improves precision by 0.05 by itself and by 0.07 in conjunction with AutoSpearman, while recall decreases by 0.05. An increase in precision and a simultaneous decrease in recall can be acceptable, depending on the concrete application scenario, as discussed in detail in Section 7. Moreover, reducing the number of features reduces the proneness to overfitting. Considering the MCC value, it is generally advisable to apply AutoSpearman when using Random Forest. However, the model that performs best overall, having MCC 0.62 and AUC 0.89, is Random Forest trained with a threshold $t = 10$ and benchmark executions from 30 iterations ($i = 30$), while applying SMOTE and not applying AutoSpearman.

> **RQ 1.3 Summary:** Across all studied machine learning algorithms, employing AutoSpearman and SMOTE as pre-processing steps neither improves nor deteriorates the models' prediction performance. In case of Random Forest, AutoSpearman slightly improves MCC by 0.023 and AUC by 0.005.

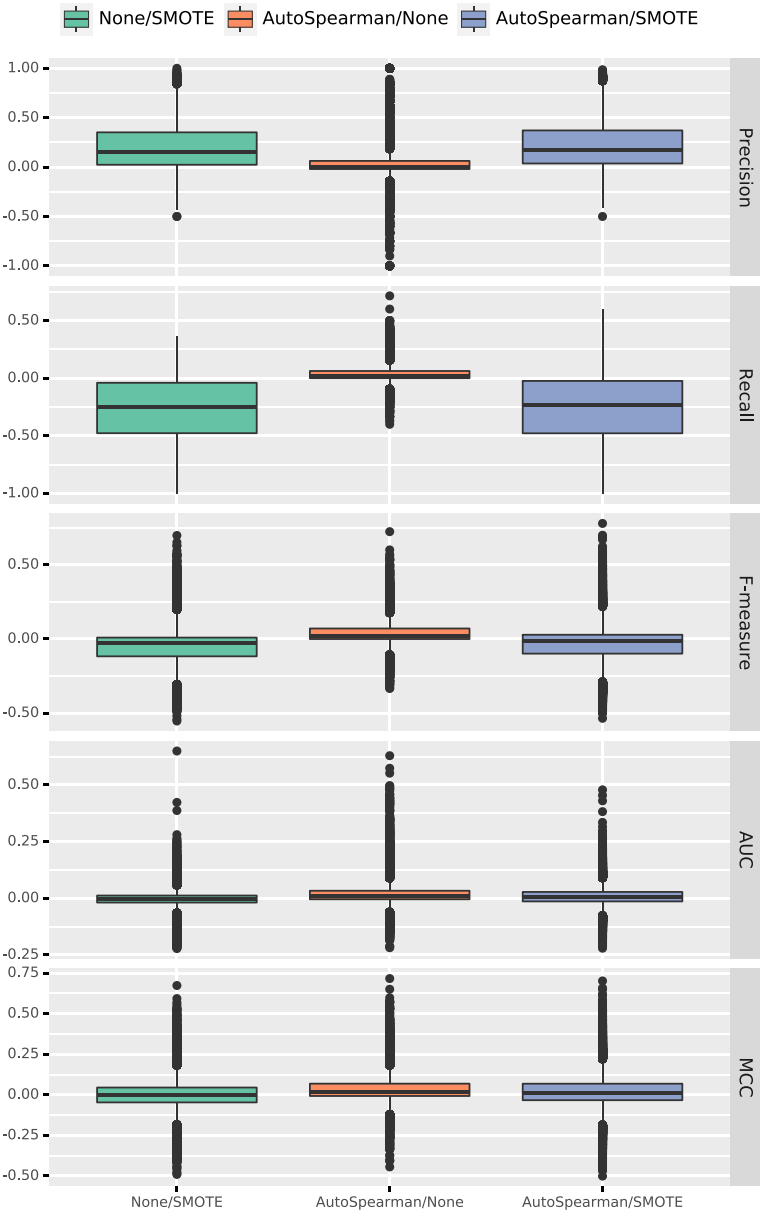


**Fig. 8** Comparison of the increments in prediction performance when using pre-processing steps over all the combinations of thresholds, iterations, models, and folds

### 5.2.5 RQ 1.4 – Impact of Different Measures of Variability

In this section, we investigate whether relying on different measures of variability (i.e., dependent variable) leads to different prediction performance. The previous sections present

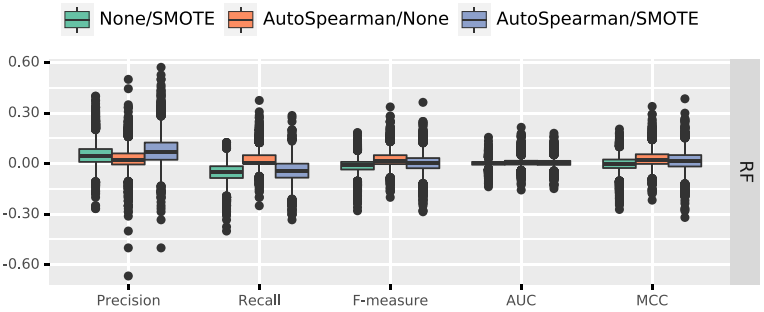


**Fig. 9** Comparison of the increments in prediction performance when using pre-processing steps with Random Forest

the results based on $rciw_{mjhd}$, simply reported as RCIW. We now compare the results $rciw_{mjhd}$ with $rciw_{boot}$ and $rmad$.

Table 3 shows the top 5 models, sorted by MCC in descending order, for each measure of variability. As previously mentioned, Random Forest is the best model with $t = 10$ and $i = 30$. We observe that Random Forest is still a dominant model when considering $rciw_{boot}$ and $rmad$, but also Adaptive Boosting performs well. Interestingly, the use of $rciw_{boot}$ and $rmad$ considerably increases the maximum values for all the prediction performance metrics. In particular, MCC increases by 0.08 with $rciw_{boot}$ and by 0.11 with $rmad$.

It is evident that the use of different variability measures affects the prediction performance. Nonetheless, our study also shows that our model performs well across all three measures of variability under study. Researchers as well as practitioners should consider

**Table 3** The top 5 models ordered by MCC with $rciw_{mjhd}$, $rciw_{boot}$, and $rmad$ as variability measure

| Variability Measure | Model | $t$ | $i$ | Selector | Sampler | MCC | AUC |
|---|---|---|---|---|---|---|---|
| $rciw_{mjhd}$ | RF | 10 | 30 | None | SMOTE | 0.6209 | 0.8863 |
| | RF | 10 | 20 | AutoSpearman | SMOTE | 0.6191 | 0.8838 |
| | RF | 10 | 20 | None | SMOTE | 0.6149 | 0.8991 |
| | RF | 10 | 30 | None | None | 0.6082 | 0.8864 |
| | RF | 5 | 30 | None | None | 0.6006 | 0.8837 |
| $rciw_{boot}$ | RF | 10 | 30 | None | SMOTE | 0.7052 | 0.9280 |
| | RF | 10 | 30 | None | None | 0.7037 | 0.9185 |
| | RF | 10 | 20 | None | None | 0.6741 | 0.9028 |
| | RF | 10 | 30 | AutoSpearman | None | 0.6715 | 0.9224 |
| | AB | 10 | 30 | None | None | 0.6700 | 0.9061 |
| $rmad$ | RF | 10 | 20 | None | SMOTE | 0.7321 | 0.9737 |
| | RF | 10 | 20 | None | None | 0.7107 | 0.9805 |
| | RF | 10 | 20 | AutoSpearman | SMOTE | 0.7107 | 0.9421 |
| | AB | 10 | 20 | None | None | 0.7087 | 0.9535 |
| | AB | 10 | 20 | None | SMOTE | 0.7021 | 0.9499 |

which variability measure is appropriate for their use case and when they evaluate their models.

**RQ 1.4 Summary:** The models' prediction performance varies considerably when different variability measures, other than $rciw_{mjhd}$, are considered for building the models. In case of Random Forest, the MCC values increase for the best model by 0.08 and 0.11 when employing $rciw_{boot}$ and $rmad$, respectively.

## 6 RQ 2 – Important Features for Good Predictions

To answer RQ 2, we investigate which individual features and which feature categories (i.e., groups of features) are important for high prediction performance. Our study considers the best performing model from RQ 1 as the model under investigation in RQ 2, i.e., Random Forest built with 30 iterations and a 10% threshold. This section introduces the statistical tests performed on the feature importances and describes the results.

### 6.1 Statistical Tests

Recall from Section 4.5.4, the study computes a single importance value for each individual feature and fold, if the feature has not been removed by AutoSpearman; in total up to 300 importance values per feature. Similarly, the study computes the feature category importance for each fold; again, 300 importance values per category in total. Importance values correspond to the decrease in MCC prediction performance.

To investigate the individual feature importances, we first replace the missing importance values with multiple imputation (Rubin 1987); specifically, we rely on the `mice` package in R (van Buuren and Groothuis-Oudshoorn 2011), which implements multiple imputation by chained equations (van Buuren 2007). Our study employs `mice` to predict missing values with Bayesian linear regression for 50 iterations and uses the average across these iterations as the final imputed value. Second, we apply the non-parametric version of the Scott-Knott effect size difference (ESD) `v3` test (Tantithamthavorn et al. 2019),[4] which clusters features into statistically distinct groups based on the "Kruskal-Wallis H" test (Kruskal and Wallis 1952) and the "Cliff's Delta" effect size (Cliff 1996). Features in the same cluster have a negligible effect size ($|\delta| < 0.147$), and features in different clusters have a non-negligible effect size ($|\delta| \geq 0.147$) among each other, based on the effect size magnitudes by Romano et al. (2006).

To investigate the feature category importances, we perform "Wilcoxon signed-rank" tests (Wilcoxon 1945) and assess the effect sizes with the "Vargha-Delaney $\hat{A}_{12}$" test (Vargha and Delaney 2000), relying on the same magnitudes as in Section 5.1, between prediction performances of a model trained on all features and one where a feature category has been removed. In our study, we consider a $p$-value below $\alpha = 0.01$ as significant.

[4] https://github.com/klainfo/ScottKnottESD

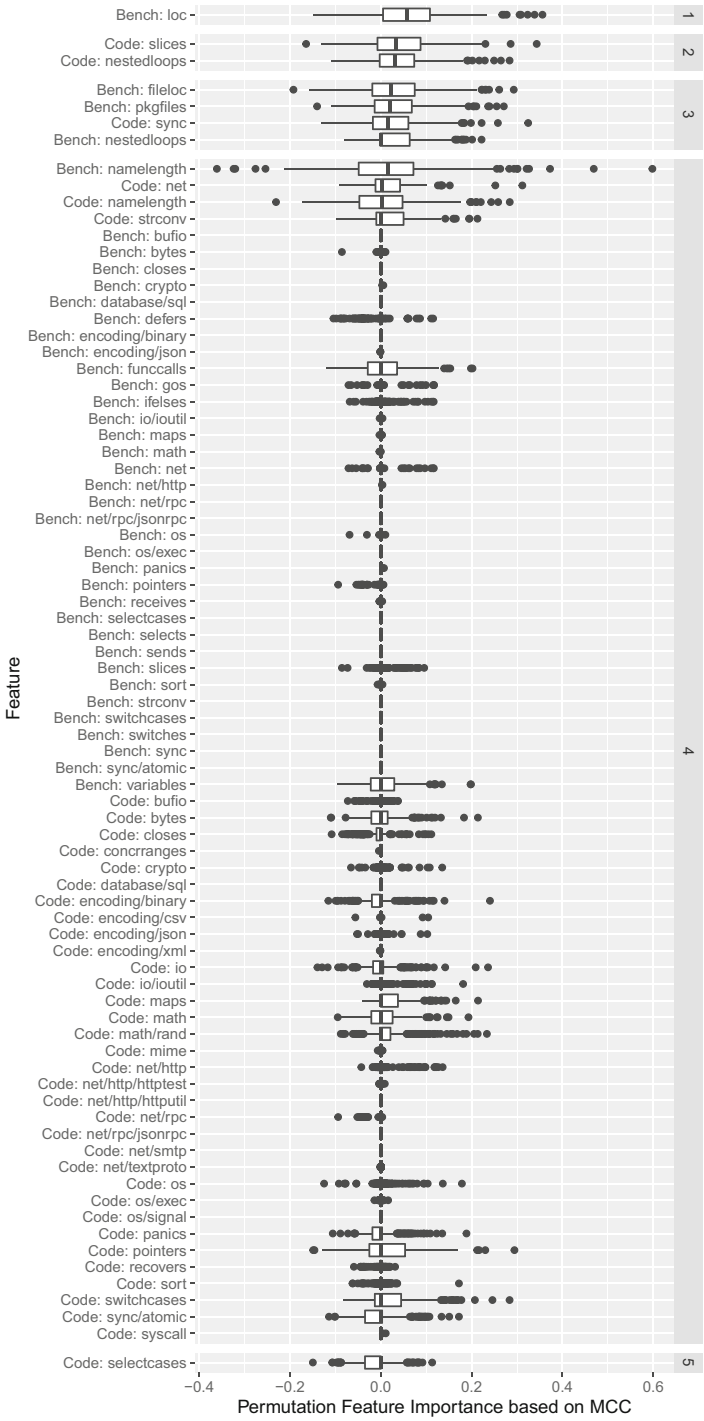


**Fig. 10** Individual feature importances with permutation importance for MCC. Facets indicate the rank as computed by the non-parametric Scott-Knott ESD test

### 6.2 Results

This section presents the feature importance results for (1) individual features and (2) feature categories, which are depicted in Tables 1 and 2.

#### 6.2.1 Individual Features

We assess the importance of all 116 features for good prediction performance. Half of the features belong to the code of the benchmark function (*Bench*), and the other half belongs to the source code called by the benchmark (*Code*). Figure 10 shows the permutation importance scores, i.e., the MCC prediction performance reduction, for each feature. The facets indicate features with negligible effect size among each other, as computed by the Scott-Knott test. Colloquially, features towards the top and to the right are more important for good predictions with Random Forest. Note that the figure does not depict the features that AutoSpearman consistently removes, i.e., for each fold, in our study.

We observe that 7 features, with ranks 1 to 3, are *individually* important for good predictions. The benchmark's LOCs are the most important feature with a median importance of 0.058, followed by the usage of slices (Go's dynamic array implementation) and nested loops in code called by the benchmark, with a median importance of 0.033 and 0.032, respectively. The third-most-important features are again related to the size of the benchmark, i.e., the LOCs of the file (median of 0.023) and the number of files in the package (median of 0.020) in which it is defined; the number of nested loops in the benchmark (median of 0.000 with the $25^{th}$ and $75^{th}$ percentile of -0.001 and 0.064, respectively); and the usage of synchronization APIs, e.g., mutexes, in the code called by the benchmark (median of 0.015). The remaining features, with ranks 4 and 5, have either a median importance of 0.0 or an importance distribution centered around 0.0. Consequently, they are individually unimportant for the model.

The analysis shows that both features related to the benchmark and to the called source code are important for good predictions. It is important to emphasize that this investigation can only capture the importance of individual features in isolation and is likely to miss the collective contribution of multiple features. For example, Siegmund et al. (2015) showed that machine learning models for software performance only thrive if such a combination of features is considered. Hence, a conclusion that the majority of the features are unimportant for good predictions is invalid, especially considering the good prediction performance of the model with all features, i.e., MCC of 0.61.

#### 6.2.2 Feature Categories

Our prediction model relies on different source code features that aim at serving as proxies for typical sources of performance variability. In this section, we investigate whether different feature categories, i.e., groups of features that are related to the same concept or the same part of the software, are collectively important for good prediction performance, as opposed to individual features in the previous section. Section 4.5.4 as well as the Tables 1 and 2 describe the feature categories under investigation and the assignment of individual features to these categories. Table 4 shows the MCC prediction performance of models built without each category as well as their difference compared to a model built with all features.

We observe that only the *code* category has a large effect; the model suffers from a 0.1889 drop in MCC, if features of the called source code are removed, as compared to the

**Table 4** Feature category importance

| Category | MCC | Difference | $p$-value | $\hat{A}_{12}$ | Magnitude |
|---|---|---|---|---|---|
| All features | 0.6091 | – | – | 0.5000 | – |
| *code* | 0.4202 | −0.1889 | **0.0000** | 0.1952 | large |
| *meta* | 0.6029 | −0.0062 | **0.0080** | 0.4733 | negligible |
| *lib-math* | 0.6033 | −0.0058 | 0.0107 | 0.4921 | negligible |
| *pl* | 0.6058 | −0.0033 | 0.4588 | 0.4866 | negligible |
| *lib* | 0.6059 | −0.0032 | **0.0092** | 0.4866 | negligible |
| *lib-io* | 0.6059 | −0.0032 | 0.0319 | 0.4902 | negligible |
| *lib-os* | 0.6059 | −0.0032 | 0.0499 | 0.4912 | negligible |
| *pl-cf* | 0.6061 | −0.0030 | 0.7210 | 0.4943 | negligible |
| *pl-data* | 0.6091 | −0.0000 | 0.5776 | 0.4998 | negligible |
| *pl-conc* | 0.6091 | −0.0000 | 0.5143 | 0.4933 | negligible |
| *lib-conc* | 0.6091 | 0.0000 | 0.7694 | 0.5024 | negligible |
| *lib-str* | 0.6091 | 0.0000 | 0.1091 | 0.4932 | negligible |
| *bench* | 0.6283 | 0.0192 | **0.0000** | 0.5363 | negligible |

Feature categories are individually excluded from the model and compared to the model built with all features. A $p$-value in bold indicates significance at $\alpha = 0.01$

model built with all features. While the benchmark code (*bench*) has a significant impact as well, its effect size is negligible. This indicates that features related to the called code are significantly more important for good predictions than features related to the benchmark code itself, when considering their collective importance.

The other categories that have a significant impact are meta-information features (*meta*), as also observed for individual feature importances where the benchmark's LOCs are shown to be important; and features for standard library calls (*lib*). However, the effect size of both is negligible.

The results show that good prediction performance can not be attributed to the collective impact of individual feature categories, if these categories are based on the same performance-affecting concept, e.g., I/O, concurrency, OS interaction, or programming language constructs. We conjecture that there is an interplay of individual features from different categories, e.g., concurrency constructs in combination with network I/O, which enables good prediction performance. Nevertheless, the analysis also shows that features extracted from the source code called by the benchmark are paramount for our model.

**RQ 2 Summary:** Only 7 features are individually important for good predictions, which are related to meta-information of the benchmark (e.g., LOCs), the usage of slices in the called source code, nested loops in both benchmark and called source code, and calls to synchronization APIs in the called source code. Moreover, the collective impact of the features of the called source code is paramount for our model, while the remaining feature categories, including the benchmark code features, are less important. These results suggest that there is a collective importance of different individual features from different categories for machine learning models with high prediction performance.

## 7 Discussion and Future Research

In this section, we provide insights for researchers to build on and practitioners wishing to apply instability prediction as well as discuss potential directions for future research.

### 7.1 Application Scenarios

We see five potential application scenarios where benchmark instability prediction could be of use for researchers and practitioners: (1) regression benchmarking by selection and reduction, (2) support for developers writing benchmarks, (3) execution configuration of benchmarks, (4) automatic benchmark generation, and (5) stability estimation of new benchmarks or for new environments.

**Regression Benchmarking.** Test suite reduction (also minimization) and regression test selection (RTS) are common techniques in unit testing to reduce the testing effort (Yoo and Harman 2012).

RTS selects a subset of tests that should be executed upon a new version. Our approach can identify benchmarks that yield unstable results and, hence, are potentially less useful for slowdown detection or do not accurately reflect the program's performance. Considering the immense execution times of benchmark suites (Huang et al. 2014; Chen and Shang 2017; Laaber and Leitner 2018, 2020), selecting only benchmarks whose results are of high quality is desirable to reduce overall testing time. Our approach can assist in that with only relatively lightweight, statically-computed metrics.

As RTS techniques often (only) consider the changes made between two versions, our approach would probably need to incorporate change related features, e.g., number of changes to programming language features or API calls. However, we see three reasons why our approach could still work for RTS: (1) recent research found that the metrics related to the source code change are less important for predicting performance properties (Ding et al. 2020); (2) performance variability is complex and the interplay of multiple features are important for accurate predictions (see Section 6); and (3) one is likely interested in the absolute stability of a benchmark and not the change in stability (a change is more applicable for runtime performance).

Note that our approach would not be a *safe* benchmark selection technique, as some (unstable) benchmarks might expose slowdowns and our prediction would remove them. Such a study is out of scope of this paper and subject to future research. We can imagine such a study comparing to state-of-the-art functional RTS research (Gligoric et al. 2015; Zhang 2018; Machalica et al. 2019) and performance test selection (de Oliveira et al. 2017; Alshoaibi et al. 2019; Chen et al. 2020).

Test suite reduction removes redundant tests, usually based on single version information. Our approach would be similar to reduction as it would remove unstable, potentially less meaningful benchmarks. However, the redundancy aspect of traditional reduction is not modeled in our approach. In this regard, we might want to relax the notion of reduction and not remove these benchmarks indefinitely. We argue that unstable benchmarks are similar to flaky tests as their results can not be trusted (Luo et al. 2014), and unstable benchmarks could be quarantined instead of permanently removed for developers to (if possible) fix their source if instability.

**Developer Support.** We envision our benchmark instability prediction to fit well into developer support tooling. For example, as part of the IDE or as a standalone linter, our

approach could raise awareness about the result accuracy of benchmarks where executing them is not an option. This might be especially useful for developers during the development phase of the benchmarks. Based on this early feedback, developers could rethink which parts of their software are benchmarked and which parts are not. Questions like "Do I really need this benchmark?" could arise and optional benchmarks that are also unstable might not be written in the first place. Moreover, developers could mock variability-inducing programming constructs, such as file or network I/O, to improve benchmark stability. This, however, should be done with care as the mocking possibly defeats the purpose of benchmarking and renders the benchmark results unrealistic and useless.

**Configuring Benchmarks.** The number of repeated iterations a benchmark is executed for has a direct impact on its result variability and consequently on its stability. In Section 2, we describe this repetition and its impact. However, finding the right configuration is non-trivial and developers often get it "wrong", as shown by Laaber et al. (2020). Their solution is to dynamically—during the execution of a benchmark—decide when to stop a benchmark; still, this dynamic reconfiguration requires manually setting upper bounds on the iteration parameter. Usually, developers would rely on default parameters imposed by the benchmarking framework or follow best practices from research, e.g., by Georges et al. (2007). Our approach would provide a first step towards solving this problem, by making developers aware of certain benchmarks potentially being unstable, before executing them. This would enable developers to take extra care of these benchmarks and setting the iteration configuration to higher values than set by default or proposed by research. If dynamic reconfiguration by Laaber et al. (2020) is used, the upper bounds for the number of iterations can be pessimistically set to high values for unstable benchmarks.

**Automatic Benchmark Generation.** Writing benchmarks for libraries and frameworks is still a niche technique (Stefan et al. 2017; Leitner and Bezemer 2017). One way to address the challenges of writing benchmarks would be to automatically generate them. Bulej et al. (2012, 2017a) and Rodriguez-Cancio et al. (2016) introduce approaches that generate rigorous benchmarks from developer-defined code segments. However, they do not give an indication whether the results of the generated benchmarks will be reliable. Novel benchmark generation techniques could reuse existing unit tests as performance tests (Ding et al. 2020) or utilize search-based techniques such as EvoSuite (Fraser and Arcuri 2011). Such generation techniques could leverage our approach to identify generated benchmarks that will have stable results and refrain from generating ones that do not.

**Stability Estimation of New Benchmarks or for New Environments.** While for existing benchmarks one could leverage historical execution data to identify potentially unstable benchmarks before testing a new version, this can neither be done for new benchmarks where no historical data exists nor for existing benchmarks that are to be executed in new environments. A model for new benchmarks could be specifically trained on historical execution data of previous version of the software under test and the concrete execution environment the project is executed in. Whereas, a model for new environments could be trained on benchmarks from other projects that have already been executed in said environment. This could especially be useful for continuous integration (CI) providers, such as TravisCI or GitLab, who potentially use different hardware to run builds on, to provide new users with stability estimations for their benchmarks.

### 7.2 Tradeoff between Precision and Recall

As with any binary classification, there is an inherent tradeoff between precision and recall, i.e., whether false positive (FP) or false negative (FNs) are more detrimental to what the approach is trying to ultimately achieve. In our context it depends on the application scenario, as described in Section 7.1, and the positive value for classification. We consider the positive value to be "unstable"; consequently, we can question if it is more important to only select unstable benchmarks (low FP-rate) or not to miss any unstable benchmarks (low FN-rate). Generally, both are desired and our results show that Random Forest performs best among the classification algorithms under study.

Precision is arguable preferred for regression benchmarking and automatic benchmark generation where indeed unstable benchmarks should be removed or not be selected. Similar for tooling support and stability estimation of new benchmarks or for new environments, which should not overwhelm developers with FPs. This can be further optimized for by applying class-rebalancing with SMOTE to increase precision, however, at the expense of recall, as Section 5.2.4 (RQ 1.3) shows. Recall could be preferred for configuring benchmarks to execute all unstable benchmarks with more measurement iterations for them to become more stable.

Our prediction results show that the majority of classifiers perform better in terms of precision than recall (see Section 5.2), supporting the majority of our application scenarios nicely. A definite answer as to whether these application scenarios are indeed well supported, requires follow-up studies employing our approach.

### 7.3 Classification vs. Prediction

We transform benchmark variability into a binary classification problem, i.e., a benchmark being stable or unstable (see Sections 3.2 and 4.4.3). We use four thresholds $t$ inspired by literature to make this distinction between the two classes (Georges et al. 2007; Mytkowicz et al. 2009), and we perform a sensitivity analysis on the threshold value in Section 5.2.2. Ideally, we would predict a benchmark's variability with, e.g., a (linear) regression model; initial experiment were unsuccessful, where the error was exceedingly high. One potential reason for this failure is that the statically-computed features are not precise enough or source code features in general do not offer enough explanatory power for regression-based prediction. Future research should explore other or more precise features to predict the exact performance result variability of a benchmark. Nonetheless, for (at least) two of our outlined application scenarios (see Section 7.1), i.e., regression benchmarking and developer support, we argue that the simplification to a binary classification problem is sufficient. The configuration of a benchmark would greatly benefit from a precise variability prediction, as this would enable suggesting the "right" iteration value to achieve a desired benchmark variability, e.g., smaller than 3% RCIW.

### 7.4 Features

We now discuss the tradeoffs concerning our choice of features and two potential improvements.

**Static vs. Dynamic Features.** Our model is based only on statically-computable source code features, by parsing ASTs (intra-procedural) and combining the features with static CG analyses (inter-procedural). The feature extraction trades faster performance for the

precision of the analyses. However, our prediction results (see Section 5.2) show that our static features perform well; especially Random Forest, our best performing predictor, with a median performance ranging from 0.43 to 0.68 and from 0.79 to 0.90 for MCC and AUC, respectively. To improve feature precision, we identify a switch from static features to dynamic features. In particular, relying on dynamic control flow and CGs to accurately identify which programming languages features are covered and APIs are called by a benchmark. We suppose that more precise analyses lead to better prediction performance; this, however, is subject to future research. Apart from the features we use, dynamic features based on performance profilers, e.g., CPU, memory, locks, or race detector information, might help to further improve prediction performance. An approach employing dynamic features would run a single invocation of a benchmark with the feature extractor injected, gather all necessary information, and feed these features into a predictor to make a decision whether a rigorous benchmark execution with multiple iterations is required.

**Execution Environment Proxies.** Source code is only one of the many factors influencing performance variability; others are, to name a few, dynamic compiler optimizations, memory layout, environment variables, virtualization, and OS-dependant factors (Georges et al. 2007; Mytkowicz et al. 2009; Curtsinger and Berger 2013; de Oliveira et al. 2013; Arif et al. 2018; Maricq et al. 2018; Laaber et al. 2019). These other factors could potentially improve the prediction performance of our model even further. We, therefore, argue that an improved prediction model should consider additional features that approximate the other factors that impact benchmark variability. In particular, we envision to include performance profiles of the execution environment, e.g., based on standardized (system) microbenchmarks. A similar approach to Wang et al. (2018) for predicting the performance of cloud applications based on resource profiles, Scheuner and Leitner (2018) for estimating the application performance from system microbenchmarks, or Jimenez et al. (2018) for inferring the resource profile of a benchmark could offer the desired proxy features for the execution environment.

### 7.5 Machine Learning Approaches on Benchmarks

The two approaches closest to ours are by Chen et al. (2020) and Ding et al. (2020). Both utilize machine learning classifiers to predict performance-related properties of functional unit tests that are used as benchmarks. While the former predicts whether unit tests lend themselves to use as performance test, the latter predicts whether a unit test is affected by a code change in terms of performance.

Both utilize source code metrics as features, inspired by previous research, which is similar to our approach. However, they consider code change diffs which ours does not, and they focus more on "traditional" source code metrics, such as LOC, cyclomatic complexity, coverage, code churn, as well as meta information from issues trackers, such as historical data. Chen et al. (2020) rely on performance-related features as well, such as external function calls, lines added or removed, loops, synchronization, and expensive variables and parameters. Our approach goes a step further and considers different library calls, which could impact performance, as individual features.

Where our approach and study diverges most from theirs is in terms of (1) prediction goal (i.e., dependent variable) and (2) study design. In terms of prediction goal, Ding et al. (2020) identify tests that are able to detect performance changes reported in issue trackers; Chen et al. (2020) predict whether a test will experience a measured performance change between two versions; and our approach identifies benchmarks that are unstable. That is, our approach is concerned with performance variability whereas theirs are

concerned with performance changes. The study design of Ding et al. (2020) and Chen et al. (2020) is somewhat similar: they focus on 2 and 3 study objects across multiple versions, utilize functional unit tests for performance, execute the tests on cloud infrastructure, gather different performance metrics, and additionally emphasize qualitative insights. Our study design is quite different: we focus on 230 study objects in a single version, utilize dedicated performance benchmarks, execute the benchmarks in controlled bare-metal environments to control confounding factors, focus on runtime performance (variability), study a larger range of machine learning classifiers, and concentrate on investigating a large parameter space of the approach.

In terms of prediction performance, the three approaches are similar: they all achieve an AUC around 0.90 for the best performing models. Compared to Chen et al. (2020), our approach's best model performs better regarding precision but worse in terms of recall. Moreover, both traditional and performance-related features are important for our model, whereas performance-related features are less important for theirs. Compared to Ding et al. (2020), features related to the called code are significantly more important than features of the benchmark code for our model.

## 8 Threats to Validity

**Construct Validity.** The central aspect of our paper is predicting benchmark stability. The definition of benchmark stability is consequently subject to validity concerns. We use the performance result variability of a benchmark as the measure for how stable it is. The majority of our study relies on the RCIW of the median estimated with a technique by Maritz and Jarrett (1978), which works well for small samples. To mitigate the threat of bias towards one measure of variability, we study the impact of different measures on our models' prediction performance in RQ 1.4. We provide additional data on the prediction performance and the feature importance for the two other measures, i.e., RCIW of the mean computed with bootstrap (Davison and Hinkley 1997; Kalibera and Jones 2012) and RMAD (Arachchige et al. 2020), as part of our online appendix (Laaber et al. 2021). Nevertheless, a reader should acknowledge that different measures for benchmark stability could change the results of our paper. Moreover, the transformation of result variability to a binary classification problem, i.e., whether a benchmark is stable or unstable, depends on the used threshold $t$ (see Section 4.4.3). We use $t \in \{1\%, 3\%, 5\%, 10\%\}$ informed by previous work (Georges et al. 2007; Mytkowicz et al. 2009; Huang et al. 2014) and perform a sensitivity analysis on the threshold value in Section 5.2.2. Other thresholds and, consequently, a different assignments of benchmarks to the classification classes "stable" and "unstable" might result in different outcomes.

Our approach relies on AST parsing and static CG information and *not* on precise, dynamic coverage information. This imprecision is likely to assign more features in higher numbers (counts) to certain benchmarks, which are actually not executing these features. Hence, the prediction (Section 5.2) and the feature importance (Section 6) results are likely impacted. This was a conscious design decision as we aimed for a purely static model to be applicable in scenarios where benchmark execution is infeasible or impossible. Similarly, different static CG algorithms, e.g., Class Hierarchy Analysis (CHA) (Dean et al. 1995), Rapid Type Analysis (RTA) (Bacon and Sweeney 1996), or control flow analysis (CFA) (Shivers 1988; Grove and Chambers 2001), will yield different feature counts for each benchmark and, therefore, impact the results of this paper. Moreover, the execution environment is likely to impact the performance variability and, hence, the results (also compare to Section 7.4).

In RQ 1, we assess the prediction performance of the 11 classifiers, relying on the 5 evaluation metrics in Section 4.5.2. We aimed for a wide variety of threshold-dependent and threshold-independent metrics to improve construct validity. In particular, we refer to AUC and MCC as main indicators of the prediction performance of the classifiers. The literature shows that AUC (Bradley 1997) and MCC (Chicco and Jurman 2020) are reliable metrics, especially in the case of binary classification.

In RQ 2, we study the importance of individual features and feature categories for the best prediction model from RQ 1. We rely on permutation feature importance for individual features and a simple reduction in MCC prediction performance if a category is removed from the model as the measures of importance. While these are common techniques to assess feature importance, different techniques or prediction performance metrics might highlight other features and categories as important. Consequently, this could change the results and conclusions of RQ 2.

**Internal Validity.** Any performance measurement experiment, which includes benchmark executions, is subject to measurement uncertainty (Mytkowicz et al. 2009; Curtsinger and Berger 2013; de Oliveira et al. 2013; Maricq et al. 2018), which could alter our benchmark result variability and, consequently, the benchmark stability. To reduce measurement uncertainty, we follow a rigorous performance engineering methodology (Georges et al. 2007) and use non-virtualized machines with hyper-threading, frequency scaling, and Intel's TurboBoost turned off (Stefan et al. 2017). Different execution environments are likely changing the stability of individual benchmarks, potentially affecting the correlation and prediction results. We opted for a tightly controlled environment to control for as many confounding factors as possible.

The number of iterations $i$ (see Section 3.2) affects the internal validity of our experiment, as the iterations have a direct impact on the benchmark result variability. To measure this effect, we perform a sensitivity analysis in Section 5.2.3. The exact results, however, might be different for other iteration counts.

Note that we *do not* claim that certain source code features are the cause for benchmark instability in Section 6. RQ 2 is about finding the most important features for the prediction and not showing causality. Such a causal relationship would require a different experiment, which is out of scope of this paper.

Since our dataset is relatively small, i.e., $\geq$ 3,620 instances, we employ a repeated k-fold cross validation approach, for a total of 300 prediction observations for every combination of model, iterations $i$, and thresholds $t$. With such a number of observations, we can apply statistical tests to mitigate the risk of spurious differences. Moreover, the relatively small dataset and the high number of features, i.e., 116, might result in overfitting the models. Creating larger benchmark execution datasets is often infeasible due to the long runtimes. Alternatively, applying AutoSpearman to select a subset of the features reduces the risk of overfitting; our results show that the prediction performance is at worst minimally reduced for the best models.

We also apply pre-processing operations to our data, i.e., standardization, feature selection based on variance, removal of correlated features with AutoSpearman, and class re-balancing with SMOTE, and RQ 1.3 investigates the impact of pre-processing on the prediction performance (see Section 5.2.4). However, a sensitivity analysis of more techniques is infeasible, considering the large number of classifiers used in our study. As in the case of hyper-parameters optimization, which we did not apply in this study, our view is that simple classifiers can already be competitive for this task. However, a sensitivity study on the effects of pre-processing and tuning on benchmark instability prediction is an avenue for future work.

**External Validity.** Generalizability is mostly concerned with regard to the selected study objects. Our study only considered benchmarks written in Go; consequently, our results do not necessarily translate to benchmarks written in other programming languages. The results potentially also do not extend to other Go projects, which are not part of our study. With 230 projects having 4,461 benchmarks, we have an extensive set of benchmarks and projects to draw conclusions from.

We study benchmarks on function/statement granularity, often also called microbenchmarks or performance unit tests (Stefan et al. 2017; Laaber and Leitner 2018). The results presented here do not generalize to other forms of performance tests, such as load tests or system/application-level benchmarks, or functional unit tests used for performance.

The benchmark result variability in our study considers execution time as its performance metric. Execution time is the standard performance metric for benchmarks on this granularity, whereas load tests, system-benchmarks, or profilers often also consider memory performance, I/O, lock contention, or other performance metrics. A careful reader should not assume that our results transfer to these other performance metrics.

Finally, other machine learning algorithms, which are not part of our 11 algorithms under study, might perform differently in terms of prediction as well as sensitivity to the number of iterations $i$, the threshold value $t$, or different measures of variability. We aimed at a diverse set of algorithms to increase generalizability among binary classifiers. Multiclass classification, clustering, and regression algorithms are likely to show different results to binary classifiers and are out of the scope of this work.

## 9 Related Work

Particularly related to our study are works dealing with (1) performance variability, (2) performance bugs, (3) performance testing, and (4) performance impact prediction. We will discuss these four aspects in the following.

### 9.1 Performance Variability

The performance variability of experiment results, such as benchmarks, is a well-known challenge in performance engineering. Georges et al. (2007) outlined a rigorous methodology to measure performance of dynamically compiled languages like Java. They report on effects of dynamic compiler optimizations influencing performance measurements and show that measurement variability is often around 3%. Even if a rigorous methodology is followed, measurement bias is common (Mytkowicz et al. 2009; Curtsinger and Berger 2013; de Oliveira et al. 2013). Mytkowicz et al. (2009) report that different environment variable sizes, such as simply having a longer user name, impacts performance measurements. Curtsinger and Berger (2013) identify that layout of code and memory, i.e., stack frames and heap objects, impacts the results of performance experiments by as much as 10%.

To reduce measurement bias and benchmark variability, it is best practice to repeat measurements on the different levels that introduce measurement uncertainty (kalibera and Jones (2012, 2013) and randomize factors influencing the measurement (Curtsinger and Berger 2013; de Oliveira et al. 2013). Moreover, randomly interleaving benchmarks across multiple trials (Abedi and Brecht 2017; Laaber et al. 2019) or executing benchmarks, that need to be compared with each other, in parallel on different CPUs of the same machine (Bulej et al. 2020) are novel techniques to handle environment-induced variability. The variability can be due to co-located tenants, hardware, OS specifics, or source

code (Maricq et al. 2018; Laaber et al. 2019). In particular, virtualized (Arif et al. 2018) and cloud (Iosup et al. 2011; Gillam et al. 2013; Leitner and Cito 2016) environments suffer from performance variability when used as performance execution environment.

Our work draws inspiration from these works and utilizes source code features, one cause of performance variability, to predict whether a benchmark will be unstable, to ultimately decide whether more repetitions (e.g., iterations) are required. Although our approach's features are solely extracted from source code, the idea is that they act as proxies for performance variability rooted in non-source-code factors, such as I/O, memory access through variables, or non-deterministic pseudo-random generators and concurrency constructs. Both our sensitivity analyses from RQ 1.1 and RQ 1.2 acknowledge the fact that performance measurements are imperfect, and they report on the impact on prediction performance. The sensitivity analysis on the threshold $t$ of what is considered a stable or unstable benchmark bases its values to investigate on the experience reported in the papers mentioned (see RQ 1.1).

Recently, He et al. (2019) and Laaber et al. (2020) introduced techniques for system-benchmarks and microbenchmarks to stop benchmarks once the result variability is unlikely to change with more repetitions. Our approach augments these techniques by identifying the benchmarks that are unstable, before execution.

### 9.2 Performance Bugs

There is an abundance of research on characteristics of performance bugs and how to automatically detect them. In this context, a performance bug can be a slowdown, an increase in memory consumption, reduced throughput, or excessive I/O operations.

Jin et al. (2012) find that performance bugs are often related to function calls, synchronization of concurrent computation, data structures, and API misuses. Selakovic and Pradel (2016) study performance bugs in JavaScript and find that they are often caused by inefficient APIs and loops, as well as unnecessarily repeated executions. Nistor et al. (2013, 2015) identify similar memory access patterns and wasted loops as root causes. Sandoval Alcocer and Bergel (2015) and Sandoval and Alcocer et al. (2016, 2020) discuss performance problems related to programming language features, such as function calls, conditionals, and (heavy) object creation. Often synchronization and concurrency are root causes for performance problems (Alam et al. 2017; Yu and Pradel 2017). Other studies conclude with similar observations of where performance bugs stem from Zhao et al. (2020) and Mazuera-Rozo et al. (2020).

Our approach incorporates the root causes identified in these studies as features in our prediction model. The simple counting of feature occurrences is in line with the finding that unnecessary repetition of source code constructs are causes for performance bugs. Similar to Liu et al. (2014), our approach also checks for potentially performance-variability-inducing API calls in the call graph of a benchmark. However, all these studies characterize and (sometimes) identify performance bugs, whereas our approach and study centers around benchmark (in)stability prediction in terms of its result variability.

### 9.3 Performance Testing

Performance testing is part of measurement-based performance engineering (Woodside et al. 2007), which aims to ensure catching performance degradations of software systems. Literature usually focusses on system-scale or method/statement-scale performance tests, often called load testing (or application/system benchmarking) and microbenchmarking (or performance unit testing), respectively.

Traditionally, research on performance testing focussed mostly on load testing, such as identifying problems and reporting on case studies (Weyuker and Vokolos 2000; Menascé 2002; Jiang and Hassan 2015). More recent work focussed on industrial applicability (Nguyen et al. 2014; Foo et al. 2015; Chen et al. 2019) and reducing the time spent in load testing activities (AlGhamdi et al. 2016; AlGhamdi et al. 2020; He et al. 2019).

Microbenchmarking, which is the focus of our study, has significantly gained traction in recent years. Leitner and Bezemer (2017) and Stefan et al. (2017) empirically study the state in OSS projects and identified gaps that require addressing from research. In particular, the complexity of performance testing activities and lack of tooling seem to be a hurdle to overcome (Bezemer et al. 2019). Horký et al. (2015) utilize benchmarks to increase performance-awareness through documentation, and Bulej et al. (2012, 2017a) introduce a declarative form of specifying performance assumptions without the need for manually writing benchmarks. Sandoval Alcocer and Bergel (2015) and Chen and Shang (2017) explore performance changes in evolving software and find that code changes often introduce performance variation. Laaber and Leitner (2018) and Laaber et al. (2019) assess benchmarks for their applicability in CI and study their results when executed on cloud infrastructure. Damasceno Costa et al. (2019) study bad practices in microbenchmark implementations and show that they significantly impact their results.

Our approach is orthogonal to the research outlined above. It contributes to one of the major challenges of performance testing, i.e., the lack of tooling for performance testing.

### 9.4 Performance Impact Prediction

Previous research investigated the impact of "new situations" on software performance. These can be in regression testing upon a new commit or about the effects of running a performance experiment under different conditions.

Two techniques of regression testing that are extensively studied for unit testing (Yoo and Harman 2012), i.e., RTS and test case prioritization (TCP), have recently become subject of investigation for performance testing. In terms of selection (RTS), they either predict whether a commit potentially introduces a performance regression (Jin et al. 2012; Huang et al. 2014; Sandoval Alcocer et al. 2016; 2020) or whether a benchmark is affected by a code change (de Oliveira et al. 2017; Alshoaibi et al. 2019). Jin et al. (2012) build a rule-based technique to detect these commits, whereas Huang et al. (2014) and Sandoval and Alcocer et al. (2016, 2020) employ a performance cost model. Regarding benchmark selection, de Oliveira et al. (2017) use lightweight static and dynamic source code indicators which are combined with logical operators. Alshoaibi et al. (2019) reuse their indicators, define the selection as an optimization problem, and employ genetic algorithms to predict whether a benchmark will be affected. In terms of prioritization (TCP), Mostafa et al. (2017) rank the execution order of benchmarks according to their predicted performance change size, inferred from a performance cost model. As already mentioned in Section 7.1, regression testing is one context where we foresee our approach to be applied. Different from the works above, our approach (1) targets benchmark stability rather than the performance impact and (2) uses a machine learning model rather than more traditional cost and inference models.

The second area of performance impact prediction focusses on how software (or a benchmark) behaves when executed in different environments. Gao and Jiang (2017) build ensemble models to predict the performance variation of load tests in different environments. Wang et al. (2018) utilize resource profiles of performance tests and cloud performance distributions to estimate how an application will behave when deployed

in cloud environments. Scheuner and Leitner (2018) employ a linear regression model to predict the response time of an application deployed in the cloud based on system microbenchmarks run on the respective cloud instance. All three of these predict the performance based on resource profiles from where the software (or benchmark) is executed. Whereas our approach is fully static and leverages the idea that certain source code features contribute more to performance variability than others, i.e., concurrency, expensive API calls, or randomized algorithms. As discussed in Section 7, dynamic information, such as resource profiles, could enhance our approach and potentially improve prediction performance.

## 10 Conclusions

In this paper, we introduced a static approach to predict whether a software benchmark will have stable or unstable results, without having to execute it. It uses 58 statically-computable source code features, extracted for the benchmark code as well as the code called by the benchmark with AST parsing and static call graph. The features are related to (1) meta information, e.g., LOC, (2) programming language elements, e.g., conditionals or loops, and (3) potentially performance-impacting standard library calls, e.g., file and network I/O.

We assessed the effectiveness of our approach with an empirical experiment on 230 open-source software Go projects that contain a total of 4,461 benchmarks. Their combination and the usage of machine learning classifiers can be used for effective prediction. We built and compared 11 different binary classification models, of which Random Forest performs best, with a median prediction performance of AUC ranging from 0.79 to 0.90 and MCC ranging from 0.43 to 0.61, depending on the concrete approach parameterization. Moreover, we find that 7 features related to meta-information, slice usage, nested loops, and synchronization APIs are individually important for good predictions. The combination of all features of the called source code is paramount for our model, while the combination of features of the benchmark itself is less important.

These results show that predicting benchmark instability with only static features is effective. We envision our approach to enable selecting reliable benchmarks in regression testing scenarios; help developers to spot potentially low-quality benchmarks; improve unstable results of benchmarks by increasing their number of repetitions before execution; be included in scenarios where repeated benchmark execution is infeasible (e.g., within search-based benchmark generation) or impossible (e.g., to statically assess benchmark quality on GitHub), and warn developers if new benchmarks or existing benchmarks executed in new environments will be unstable.

**Acknowledgements** We are grateful for the anonymous reviewers' comments and feedback that helped to significantly improve the paper. Further, we would like to thank Petr Tůma and Vojtěch Horký from the Charles University in Prague, who provided the infrastructure to execute the benchmarks. The research leading to these results has received funding from the Swiss National Science Foundation (SNSF) under project number `165546`.

**Funding** Open Access funding provided by Universität Zürich.

## References

Abedi A, Brecht T (2017) Conducting repeatable experiments in highly variable cloud computing environments. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE, vol 2017. ACM, New York, pp 287–292. https://doi.org/10.1145/3030207.3030229

Akinshin A (2020a) Quantile confidence intervals for weighted samples. https://aakinshin.net/posts/weighted-quantiles-ci/, accessed: 2.2. 2021

Akinshin A (2020b) Quantile-respectful density estimation based on the Harrell-Davis, quantile estimator. https://aakinshin.net/posts/qrde-hd/, accessed: 2.2. 2021

Akinshin A (2021) Unbiased median absolute deviation. https://aakinshin.net/posts/unbiased-mad/, accessed: 9.2.2021

Alam MMu, Liu T, Zeng G, Muzahid A (2017) SyncPerf: Categorizing, detecting, and diagnosing synchronization performance bugs. In: Proceedings of the 12th European Conference on Computer Systems, EuroSys. ACM, New York, pp 298–313. https://doi.org/10.1145/3064176.3064186

AlGhamdi HM, Syer MD, Shang W, Hassan AE (2016) An automated approach for recommending when to stop performance tests. In: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution, ICSME, vol 2016, pp 279–289. https://doi.org/10.1109/ICSME.2016.46

AlGhamdi HM, Bezemer CP, Shang W, Hassan AE, Flora P (2020) Towards reducing the time needed for load testing. Journal of Software, Evolution and Process. https://doi.org/10.1002/smr.2276

Alshoaibi D, Hannigan K, Gupta H, Mkaouer MW (2019) PRICE: Detection of performance regression introducing code changes using static and dynamic metrics. In: Proceedings of the 11th International Symposium on Search Based Software Engineering, Springer Nature, SSBSE 2019, pp 75–88. https://doi.org/10.1007/978-3-030-27455-9_6

Altmann A, Tološi L, Sander O, Lengauer T (2010) Permutation importance: A corrected feature importance measure. Bioinformatics 26(10):1340–1347. https://doi.org/10.1093/bioinformatics/btq134

Andersen LO (1994) Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark

Arachchige CNPG, Prendergast LA, Staudte RG (2020) Robust analogs to the coefficient of variation. J Appl Stat:1–23. https://doi.org/10.1080/02664763.2020.1808599

Arif MM, Shang W, Shihab E (2018) Empirical study on the discrepancy between performance testing results from virtual and physical environments. Empir Softw Eng 23(3):1490–1518. https://doi.org/10.1007/s10664-017-9553-x

Bacon DF, Sweeney PF (1996) Fast static analysis of C++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA, vol 1996. ACM, New York, pp 324–341. https://doi.org/10.1145/236337.236371

Bezemer CP, Eismann S, Ferme V, Grohmann J, Heinrich R, Jamshidi P, Shang W, van Hoorn A, Villavicencio M, Walter J, Willnecker F (2019) How is performance addressed in DevOps? In: Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering, ICPE, vol 2019. ACM, New York, pp 45–50. https://doi.org/10.1145/3297663.3309672

Blackburn SM, Cheng P, McKinley KS (2004) (2004) Myths And realities: The performance impact of garbage collection. In: Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems, ACM, SIGMETRICS/Performance. https://doi.org/10.1145/1005686.1005693

Blackburn SM, Diwan A, Hauswirth M, Sweeney PF, Amaral JN, Brecht T, Bulej L, Click C, Eeckhout L, Fischmeister S et al (2016) The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. ACM Trans Programm Lang Syst 38(4). https://doi.org/10.1145/2983574

Bradley AP (1997) The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recogn 30(7):1145–1159. https://doi.org/10.1016/s0031-3203(96)00142-2

Breiman L (2001) Random forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/a:1010933404324

Buckland M, Gey F (1994) The relationship between Recall and Precision. J Amer Soc Inf Sci 45(1):12–19, https://doi.org/10.1002/(SICI)1097-4571(199401)45:1%3C12::AID-ASI2%3E3.0.CO;2-L

Bulej L, Bureš T, Keznikl J, koubková A, Podzimek A, Tůma P (2012) Capturing performance assumptions using Stochastic Performance Logic. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE, vol 2012. ACM, New York, pp 311–322. https://doi.org/10.1145/2188286.2188345

Bulej L, Bureš T, Horký V, Kotrč J, Marek L, Trojánek T, Tůma P (2017a) Unit testing performance with Stochastic Performance Logic. Autom Softw Eng 24(1):139–187. https://doi.org/10.1007/s10515-015-0188-0

Bulej L, Horký V, Tůma P (2017b) Do we teach useful statistics for performance evaluation? In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE 2017 Companion. ACM, New York, pp 185–189. https://doi.org/10.1145/3053600.3053638

Bulej L, Horký V, Tůma P, Farquet F, Prokopec A (2020) Duet benchmarking: Improving measurement accuracy in the cloud. In: Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering, ICPE. ACM, New York, p 2020. https://doi.org/10.1145/3358960.3379132

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: Synthetic minority over-sampling technique. J Artif Intell Res 16:321–357. https://doi.org/10.1613/jair.953

Chen J, Shang W (2017) An exploratory study of performance regression introducing code changes. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, ISCME. IEEE, New York, p 2017. https://doi.org/10.1109/icsme.2017.13

Chen J, Shang W, Shihab E (2020) PerfJIT: Test-level just-in-time prediction for performance regression introducing commits. IEEE Transactions on Software Engineering pp 1–1. https://doi.org/10.1109/tse.2020.3023955

Chen TH, Syer MD, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2019) Analytics-driven Load testing: An industrial experience report on load testing of large-scale systems. In: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. IEEE, ICSE-SEIP. https://doi.org/10.1109/icse-seip.2017.26

Chicco D, Jurman G (2020) The advantages of the Matthews correlation coefficient (MCC) over F1 score and Accuracy in binary classification evaluation. BMC Genom 21(1). https://doi.org/10.1186/s12864-019-6413-7

Chinchor N (1992) MUC-4 Evaluation metrics. In: Proceedings of the 4th Conference on Message Understanding, Association for Computational Linguistics MUC4. https://doi.org/10.3115/1072064.1072067

Cliff N (1996) Ordinal Methods for Behavioral Data Analysis, 1st edn. Psychology Press

Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297. https://doi.org/10.1007/bf00994018

Costa D, Andrzejak A, Seboek J, Lo D (2017) Empirical study of usage and performance of java collections. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. ACM, ICPE. https://doi.org/10.1145/3030207.3030221

Curtsinger C, Berger ED (2013) STABILIZER: Statistically sound performance evaluation. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS. ACM, New York, pp 219–228. https://doi.org/10.1145/2451116.2451141

D'Agostino RB, Belanger A, D'Agostino RB Jr (1990) A suggestion for using powerful and informative tests of normality. Amer Stat 44(4):316. https://doi.org/10.2307/2684359

Damasceno Costa DE, Bezemer CP, Leitner P, Andrzejak A (2019) What's wrong with my benchmark results? Studying bad practices in JMH benchmarks. IEEE Transactions on Software Engineering, pp 1–1. https://doi.org/10.1109/TSE.2019.2925345

Davison AC, Hinkley D (1997) Bootstrap methods and their application. J Am Stat Assoc:94

Dean J, Grove D, Chambers C (1995) Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the 9th European Conference on Object-Oriented Programming, Springer Berlin Heidelberg, ECOOP 1995, pp 77–101. https://doi.org/10.1007/3-540-49538-x_5

Dilley N, Lange J (2019) An empirical study of messaging passing concurrency in Go projects. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE, SANER. https://doi.org/10.1109/saner.2019.8668036

Ding Z, Chen J, Shang W (2020) Towards the use of the readily available tests from the release pipeline as performance tests. Are we there yet? In: Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering, ICSE. ACM, New York, p 2020. https://doi.org/10.1145/3377811.3380351

Dunn OJ (1964) Multiple comparisons using rank sums. Technometrics 6(3):241–252. https://doi.org/10.1080/00401706.1964.10490181

Foo KC, Jiang ZMJ, Adams B, Hassan AE, Zou Y, Flora P (2015) An industrial case study on the automated detection of performance regressions in heterogeneous environments. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, vol 2. IEEE Press, Piscataway, pp 159–168. https://doi.org/10.1109/icse.2015.144

Fox J (2016) Applied Regression Analysis and Generalized Linear Models, 3rd edn. SAGE Publications, https://us.sagepub.com/en-us/nam/applied-regression-analysis-and-generalized-linear-models/book237254

Fraser G, Arcuri A (2011) EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, ESEC/FSE. https://doi.org/10.1145/2025113.2025179

Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139. https://doi.org/10.1006/jcss.1997.1504

Friedman JH (1991) Multivariate adaptive regression splines. Ann Stat 19(1):1–67. https://doi.org/10.1214/aos/1176347963

Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29(5):1189–1232. https://doi.org/10.1214/aos/1013203451

Gao R, Jiang ZMJ (2017) An exploratory study on assessing the impact of environment variations on the results of load tests. https://doi.org/10.1109/msr.2017.22

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007. ACM, New York, pp 57–76. https://doi.org/10.1145/1297027.1297033

Gillam L, Li B, O'Loughlin J, Tomar APS (2013) Fair benchmarking for cloud computing systems. J Cloud Comput Adv Syst Appl 2(1):6. https://doi.org/10.1186/2192-113X-2-6

Gligoric M, Eloussi L, Marinov D (2015) Practical regression test selection with dynamic file dependencies, Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015. ACM, New York, pp 211–222. https://doi.org/10.1145/2771783.2771784

Go Authors (2020a) Go – frequently asked questions (FAQ). https://golang.org/doc/faq

Go Authors (2020b) The Go programming language specification. https://golang.org/ref/spec

Goldberger J, Roweis S, Hinton GE, Salakhutdinov RR (2004) Neighbourhood components analysis. In: Advances in Neural Information Processing Systems, vol 17. MIT Press, NIPS 2004, vol 17, pp 513–520, https://proceedings.neurips.cc/paper/2004/file/42fe880812925e520249e808937738d2-Paper.pdf

Grove D, Chambers C (2001) A framework for call graph construction algorithms. ACM Trans Programm Lang Syst 23(6):685–746. https://doi.org/10.1145/506315.506316

Hanley JA, McNeil BJ (1982) The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143(1):29–36. https://doi.org/10.1148/radiology.143.1.7063747

Harrell FE, Davis CE (1982) A new distribution-free quantile estimator. Biometrika 69(3):635–640. https://doi.org/10.1093/biomet/69.3.635

Hauke J, Kossowski T (2011) Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. Quaest Geograph 30(2):87–93. https://doi.org/10.2478/v10117-011-0021-1

He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (2019) A statistics-based performance testing methodology for cloud applications. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019. ACM, New York, pp 188–199. https://doi.org/10.1145/3338906.3338912

Hess MR, Kromrey JD (2004) Robust confidence intervals for effect sizes: A comparative study of cohen's d and cliff's delta under non-normality and heterogeneous variances. Annual Meeting of the American Educational Research Association

Hesterberg TC (2015) What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. Amer Stat 69(4):371–386. https://doi.org/10.1080/00031305.2015.1089789

Hind M (2001) Pointer Analysis: Haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, PASTE. https://doi.org/10.1145/379605.379665

Horký V, Libič P, Marek L, Steinhauser A, Tůma P (2015) Utilizing performance unit tests to increase performance awareness. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE 2015. ACM, New York, pp 289–300. https://doi.org/10.1145/2668930.2688051

Hosmer Jr, DW, Lemeshow S, Sturdivant RX (2013) Applied logistic regression, 3rd edn. Wiley

Huang P, Ma X, Shen D, Zhou Y (2014) Performance regression testing target prioritization via performance risk analysis. In: Proceedings of the 36th IEEE/ACM International Conference on Software Engineering, ICSE 2014. ACM, New York, pp 60–71. https://doi.org/10.1145/2568225.2568232

Hudson R (2018) Getting to Go: The journey of Go's garbage collector. https://blog.golang.org/ismmkeynote

Iosup A, Yigitbasi N, Epema D (2011) On the performance variability of production cloud services. In: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2011. IEEE Computer Society, Washington, pp 104–113. https://doi.org/10.1109/CCGrid.2011.22

Jangda A, Powers B, Berger ED, Guha A (2019) Not so fast: Analyzing the performance of WebAssembly vs. native code. In: Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019. USENIX Association, Renton, pp 107–120, https://www.usenix.org/conference/atc19/presentation/jangda

Jiang ZM, Hassan AE (2015) A survey on load testing of large-scale software systems. IEEE Trans Softw Eng 41(11):1091–1118. https://doi.org/10.1109/TSE.2015.2445340

Jiarpakdee J, Tantithamthavorn C, Treude C (2018) AutoSpearman: Automatically mitigating correlated software metrics for interpreting defect models. In: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution. IEEE, ICSME. https://doi.org/10.1109/icsme.2018.00018

Jiarpakdee J, Tantithamthavorn c, Hassan AE (2019) The impact of correlated metrics on the interpretation of defect models. IEEE Transactions on Software Engineering

Jiarpakdee J, Tantithamthavorn C, Treude C (2020) The impact of automated feature selection techniques on the interpretation of defect models. Empir Softw Eng 25(5):3590–3638. https://doi.org/10.1007/s10664-020-09848-1

Jimenez I, Watkins N, Sevilla M, Lofstead J, Maltzahn C (2018) quiho: Automated performance regression testing using inferred resource utilization profiles. In: Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering, ICPE 2018. ACM, New York, pp 273–284. https://doi.org/10.1145/3184407.3184422

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012. ACM, New York, pp 77–88. https://doi.org/10.1145/2254064.2254075

John GH, Langley P (1995) Estimating continuous distributions in bayesian classifiers. In: Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence, UAI 1995. Morgan Kaufmann Publishers Inc., San Francisco, pp 338–345, arXiv:1302.4964

Kalibera T, Jones R (2012) Quantifying performance changes with effect size confidence intervals. Technical Report 4–12, University of Kent, http://www.cs.kent.ac.uk/pubs/2012/3233

Kalibera T, Jones R (2013) Rigorous benchmarking in reasonable time. In: Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management, ISMM 2013. ACM, New York, pp 63–74. https://doi.org/10.1145/2464157.2464160

Kaltenecker C, Grebhahn A, Siegmund N, Guo J, Apel S (2019) Distance-based sampling of software configuration spaces. In: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering. IEEE, ICSE. https://doi.org/10.1109/icse.2019.00112

Kraemer HC, Morgan GA, Leech NL, Gliner JA, Vaske JJ, Harmon RJ (2003) Measures of clinical significance. J Amer Acad Child Adolesc Psych 42(12):1524–1529. https://doi.org/10.1097/00004583-200312000-00022

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621. https://doi.org/10.1080/01621459.1952.10483441

Laaber C, Leitner P (2018) An evaluation of open-source software microbenchmark suites for continuous performance assessment. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018. ACM, New York, pp 119–130. https://doi.org/10.1145/3196398.3196407

Laaber C, Scheuner J, Leitner P (2019) Software microbenchmarking in the cloud. How bad is it really? Empirical Software Engineering. https://doi.org/10.1007/s10664-019-09681-1

Laaber C, Würsten S, Gall HC, Leitner P (2020) Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, ESEC/FSE. https://doi.org/10.1145/3368089.3409683

Laaber C, Basmaci M, Salza P (2021) Replication package "Predicting unstable software benchmarks using static source code features". https://doi.org/10.5281/zenodo.4783139

Leitner P, Bezemer CP (2017) An exploratory study of the state of practice of performance testing in java-based open source projects. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017. ACM, New York, pp 373–384. https://doi.org/10.1145/3030207.3030213

Leitner P, Cito J (2016) Patterns in the chaos – A study of performance variation and predictability in public IaaS clouds. ACM Trans Internet Technol 16(3):15:1–15:23. https://doi.org/10.1145/2885497

Liu Y, Xu C, Cheung SC (2014) Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th IEEE/ACM International Conference on Software Engineering, ICSE 2014. ACM, New York, pp 1013–1024. https://doi.org/10.1145/2568225.2568229

Luo Q, Hariri F, Eloussi L, Marinov D (2014) An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014. ACM Press. https://doi.org/10.1145/2635868.2635920

Machalica M, Samylkin A, Porth M, Chandra S (2019) Predictive test selection. In: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. IEEE, ICSE-SEIP. https://doi.org/10.1109/icse-seip.2019.00018

Maricq A, Duplyakin D, Jimenez I, Maltzahn C, Stutsman R, Ricci R (2018) Taming performance variability. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI 2018. USENIX Association, pp 409–425, https://www.usenix.org/conference/osdi18/presentation/maricq

Maritz JS, Jarrett RG (1978) A note on estimating the variance of the sample median. J Am Stat Assoc 73(361):194–196. https://doi.org/10.1080/01621459.1978.10480027

Matthews BW (1975) Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Bioch Biophys Acta (BBAxs) - Protein Struct 405(2):442–451. https://doi.org/10.1016/0005-2795(75)90109-9

Mazuera-Rozo A, Trubiani C, Linares-Vásquez M, Bavota G (2020) Investigating types and survivability of performance bugs in mobile apps. Empir Softw Eng 25(3):1644–1686. https://doi.org/10.1007/s10664-019-09795-6

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320. https://doi.org/10.1109/tse.1976.233837

Menascé DA (2002) Load testing of web sites. IEEE Internet Comput 6(4):70–74. https://doi.org/10.1109/MIC.2002.1020328

Mostafa S, Wang X, Xie T (2017) PerfRanker: Prioritization of performance regression tests for collection-intensive software. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017. ACM, New York, pp 23–34. https://doi.org/10.1145/3092703.3092725

Mühlbauer S, Apel S, Siegmund N (2020) Identifying software performance changes across variants and versions. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ACM, ASE. https://doi.org/10.1145/3324884.3416573

Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009) Producing wrong data without doing anything obviously wrong! In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009. ACM, New York, pp 265–276. https://doi.org/10.1145/1508244.1508275

Nguyen THD, Nagappan M, Hassan AE, Nasser M, Flora P (2014) An industrial case study of automatically identifying performance regression-causes. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014. ACM, New York, pp 232–241. https://doi.org/10.1145/2597073.2597092

Nistor A, Song L, Marinov D, Lu S (2013) Toddler: Detecting performance problems via similar memory-access patterns. In: Proceedings of the 35th IEEE/ACM International Conference on Software Engineering, ICSE 2013. IEEE Press, Piscataway, pp 562–571. https://doi.org/10.1109/ICSE.2013.6606602

Nistor A, Chang PC, Radoi C, Lu S (2015) Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, vol 1. IEEE Press, Piscataway, pp 902–912. https://doi.org/10.1109/ICSE.2015.100

de Oliveira AB, Petkovich JC, Reidemeister T, Fischmeister S (2013) DataMill: Rigorous performance evaluation made easy. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ICPE 2013. ACM, New York, pp 137–148. https://doi.org/10.1145/2479871.2479892

de Oliveira AB, Fischmeister S, Diwan A, Hauswirth M, Sweeney PF (2017) Perphecy: Performance regression test selection made simple but effective. In: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, pp 103–113. https://doi.org/10.1109/ICST.2017.17

Park C, Kim H, Wang M (2020) Investigation of finite-sample properties of robust location and scale estimators. Commun Stat Simul Comput:1–27. https://doi.org/10.1080/03610918.2019.1699114

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine learning in python. J Mach Learn Res 12(85):2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106. https://doi.org/10.1007/bf00116251

Rodriguez-Cancio M, Combemale B, Baudry B (2016) Automatic microbenchmark generation to prevent dead code elimination and constant folding. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. Association for Computing Machinery, New York, pp 132–143. https://doi.org/10.1145/2970276.2970346

Romano J, Kromrey J, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In: Annual Meeting of the Florida Association of Institutional Research, pp 1–3

Rubin DB (1987) Multiple imputation for nonresponse in surveys. Wiley. https://doi.org/10.1002/9780470316696

Ruck DW, Rogers SK, Kabrisky M, Oxley ME, Suter BW (1990) The multilayer perceptron as an approximation to a bayes optimal discriminant function. IEEE Trans Neural Netw 1(4):296–298. https://doi.org/10.1109/72.80266

Sandoval Alcocer JP, Bergel A (2015) Tracking down performance variation against source code evolution. In: Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015. ACM, New York, pp 129–139. https://doi.org/10.1145/2816707.2816718

Sandoval Alcocer JP, Bergel A, Valente MT (2016) Learning from source code history to identify performance failures. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE 2016. ACM, New York, pp 37–48. https://doi.org/10.1145/2851553.2851571

Sandoval Alcocer JP, Bergel A, Valente MT (2020) Prioritizing versions for performance regression testing: The Pharo case. Sci Comput Program 191:102415. https://doi.org/10.1016/j.scico.2020.102415

Scheuner J, Leitner P (2018) Estimating cloud application performance based on micro-benchmark profiling. In: Proceedings of the 11th IEEE International Conference on Cloud Computing. IEEE, CLOUD 2014. https://doi.org/10.1109/cloud.2018.00019

Selakovic M, Pradel M (2016) Performance issues and optimizations in JavaScript: An empirical study. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering, ICSE 2016. ACM, New York, pp 61–72. https://doi.org/10.1145/2884781.2884829

Shipilev A (2018) Reconsider defaults for warmup and measurement iteration counts, durations. https://bugs.openjdk.java.net/browse/CODETOOLS-7902165

Shivers O (1988) Control flow analysis in scheme. In: Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1988. ACM, New York, pp 164–174. https://doi.org/10.1145/960116.54007

Siegmund N, Grebhahn A, Apel S, Kästner C (2015) Performance-influence models for highly configurable systems. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. ACM, ESEC/FSE. https://doi.org/10.1145/2786805.2786845

Song L, Lu S (2017) Performance diagnosis for inefficient loops. In: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering. IEEE, ICSE. https://doi.org/10.1109/icse.2017.41

Stefan P, Horký V, Bulej L, Tůma P (2017) Unit testing performance in Java projects: Are we there yet? In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017. ACM, New York, pp 401–412. https://doi.org/10.1145/3030207.3030226

Stol KJ, Fitzgerald B (2018) The ABC of software engineering research. ACM Trans Softw Eng Methodol 27(3):1–51. https://doi.org/10.1145/3241743

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2019) The impact of automated parameter optimization on defect prediction models. IEEE Trans Softw Eng 45(7):683–711. https://doi.org/10.1109/TSE.2018.2794977

Tantithamthavorn C, Hassan AE, Matsumoto K (2020) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans Softw Eng 46(11):1200–1219. https://doi.org/10.1109/tse.2018.2876537

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578. https://doi.org/10.1007/s10664-008-9103-7

van Buuren S (2007) Multiple imputation of discrete and continuous data by fully conditional specification. Stat Methods Med Res 16(3):219–242. https://doi.org/10.1177/0962280206074463

van Buuren S, Groothuis-Oudshoorn K (2011) mice: Multivariate imputation by chained equations in R. J Stat Softw 45(3). https://doi.org/10.18637/jss.v045.i03

Vargha A, Delaney HD (2000) A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132. https://doi.org/10.2307/1165329

Wang W, Tian N, Huang S, He S, Srivastava A, Soffa ML, Pollock L (2018) Testing cloud applications under cloud-uncertainty performance effects. In: Proceedings of the 11th IEEE International Conference on Software Testing. Verification and Validation, ICST 2018, pp 81–92. https://doi.org/10.1109/ICST.2018.00018

Weyuker EJ, Vokolos FI (2000) Experience with performance testing of software systems: Issues, an approach, and case study. IEEE Trans Softw Eng 26(12):1147–1156. https://doi.org/10.1109/32.888628
Wilcoxon F (1945) Individual comparisons by ranking methods. Biometr Bullet 1(6):80. https://doi.org/10.2307/3001968
Woodside M, Franks G, Petriu DC (2007) The future of software performance engineering. In: Future of software engineering. IEEE, FOSE. https://doi.org/10.1109/fose.2007.32
Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: A survey. Softw Test Verif Reliab 22(2):67–120. https://doi.org/10.1002/stv.430
Yu T, Pradel M (2017) Pinpointing and repairing performance bottlenecks in concurrent programs. Empir Softw Eng 23(5):3034–3071. https://doi.org/10.1007/s10664-017-9578-1
Zhang L (2018) Hybrid regression test selection. In: Proceedings of the 40th IEEE/ACM International Conference on Software Engineering, ICSE 2018. ACM, New York, pp 199–209. https://doi.org/10.1145/3180155.3180198
Zhao Y, Xiao L, Wang X, Sun L, Chen B, Liu Y, Bondi AB (2020) How are performance issues caused and resolved?–An empirical study from a design perspective. In: Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering. ACM, ICPE. https://doi.org/10.1145/3358960.3379130
Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ACM Press, ESEC/FSE. https://doi.org/10.1145/1595696.1595713



**Christoph Laaber** is a Research Assistant at the Software Evolution and Architecture Lab (s.e.a.l.) at the University of Zurich. He received a PhD degree in Software Engineering from the University of Zurich, Switzerland. His research interests include topics at the intersection of software engineering and performance engineering. Website: laaber.net

**Mikael Basmaci** was part of the Software Evolution and Architecture Lab (s.e.a.l) during his bachelor studies. He received a BSc degree in Software Systems from the University of Zurich, Switzerland. His interests include Data Science, Software Engineering and Full-Stack Software Development. Website: mikaelbasmaci.ch

**Pasquale Salza** is a Senior Research Associate at the Software Evolution and Architecture Lab (s.e.a.l.) at the University of Zurich, Switzerland. He received a PhD degree in Computer Science from the University of Salerno, Italy. His research interests include software engineering, machine learning, cloud computing, and evolutionary computation.

## Affiliations

**Christoph Laaber[1] · Mikael Basmaci[1] · Pasquale Salza[1]**

Mikael Basmaci
mikael.basmaci@uzh.ch

Pasquale Salza
salza@ifi.uzh.ch

[1] Department of Informatics, University of Zurich, Zurich, Switzerland