Proceedings of the 55th Hawaii International Conference on System Sciences | 2022

A Rule-Learning Approach for Detecting Faults in Highly Configurable Software Systems from Uniform Random Samples

Ruben Heradio
Universidad Nacional de Educacion a Distancia (UNED), Madrid, Spain
rheradio@issi.uned.es

David Fernandez-Amoros
UNED, Madrid, Spain
david@issi.uned.es

Victoria Ruiz
Universidad Rey Juan Carlos, Madrid, Spain
victoria.ruiz.parrado@urjc.es

Manuel J. Cobo
University of Cadiz, Cadiz, Spain
manueljesus.cobo@uca.es

Abstract

Software systems tend to become more and more configurable to satisfy the demands of their increasingly varied customers. Exhaustively testing the correctness of highly configurable software is infeasible in most cases because the space of possible configurations is typically colossal. This paper proposes addressing this challenge by (i) working with a representative sample of the configurations, i.e., a "uniform" random sample, and (ii) processing the results of testing the sample with a rule induction system that extracts the faults that cause the tests to fail. The paper (i) gives a concrete implementation of the approach, (ii) compares the performance of the rule learning algorithms AQ, CN2, LEM2, PART, and RIPPER, and (iii) provides empirical evidence supporting our procedure.

1. Introduction

According to Britton et al. [1], the cost of detecting and fixing defects in software systems is about 312 billion dollars annually. Moreover, fault repairing consumes on average half of the development costs in a regular software project. Consequently, considerable research is being undertaken to enable testing software efficiently [2, 3, 4, 5, 6].

In particular, testing highly configurable software systems is especially challenging because, in order to satisfy a wide range of customer necessities, these systems provide numerous selectable options that originate a combinatorial explosion of configurations to test [7, 8, 9]. For instance, the tool BusyBox[1], popularly known as "the Swiss Army knife of embedded Linux", substitutes several standard GNU/Linux utilities with a single executable. To support optimizing the size of this executable file, BusyBox is notably modular, supporting the selection of 613 options at compile time. Options cannot be combined in any manner, though. They need to fulfill a set of constraints specified with the Kconfig language [10, 11]. Even so, the number of valid configurations is still huge[2]: 7.428e146 [13]. Obviously, it is not feasible to test all of these configurations to guarantee BusyBox correctness.

To overcome this problem, this paper proposes testing just a small sample of configurations that allows inferring what defects are causing the configurations to fail for the most part. A fundamental condition that any sample must satisfy to be representative of the population is that it is chosen at random [14]. In the context of this paper, this means that every configuration must be *equally likely* to be included in the sample. In the *software product line* [15, 16, 17] and *SAT* literature [18, 19, 20], this idea is typically stressed using the term *uniform* random sampling. It is worth remarking that getting a uniform random sample is not trivial due to the option inter-dependencies. Fortunately, there are methods to do it efficiently nowadays [13, 21].

As a result of testing a sample, its configurations become partitioned into two categories: those that passed the tests (OK) and those that failed (KO). Then, we propose to use *rule induction* techniques [22, 23] for uncovering the hidden relationship between the selectable options and the KO results. Thus, the learned rules assist the tester in understanding what faults are producing the system failures[3].

In particular, this paper examines the application of the following rule learning algorithms: AQ [25, 26], CN2 [27], LEM2 [28], PART [29], and RIPPER [30]. These algorithms have been profoundly studied by the rule induction community [23] and are variated enough to validate our approach from different perspectives (i.e., AQ is a covering algorithm, CN2 combines decision trees with ideas from AQ, LEM2 is built upon rough set theory, PART uses partitioning trees, and RIPPER

[1] https://busybox.net/

[2] To appreciate the vastness of this number, it is worth noting that 1e82 is the current estimate of the number of atoms in the universe [12].

[3] In software testing terminology, a *failure* happens when the system's external behavior fails to meet its specification. A failure is caused by a *fault*, also called *defect*, that is placed somewhere in the software [24].

URI: https://hdl.handle.net/10125/79595
978-0-9981331-5-7
(CC BY-NC-ND 4.0)

applies pruning techniques to prevent rule overfitting).

Also, this paper reports the empirical validation of our approach by testing the prominent web application generator JHipster[4], which is an open-source project that in June 2021 had 18,475 stars, 620 contributors, and 40,987 users in GitHub. According to the experimental results, the rules induced from random samples of 100 configurations are easily interpretable and can identify, on average, the whole population defects with *accuracy*=0.91, κ=0.81, *precision*=0.90, *recall*=0.86, and *specificity*=0.95.

The remainder of this paper is organized as follows. Section 2 introduces the JHipster case study, which will be used throughout the paper to motivate, explain and validate our approach. Section 3 describes our rule-based method in detail. Section 4 summarizes the experimental validation of our procedure. Finally, Section 5 provides some concluding remarks.

2. Case study

Halin et al. [31, 32, 33] have developed an exhaustive dataset[5] where the generation, compilation, and build of every possible configuration of JHipster v3.6.1 is tested. To the extent of our knowledge, this is the most complete dataset to date for configurable software testing, and thus it will be the one used in this article.

Figures 1 and 2 show the grammar (in *Extended Backus–Naur Form* [34]) and constraints that rule options' combination. For example, according to the *production rule* in Lines 1-5 of Figure 1, every JHipster configuration must include Authentication (i.e., a security mechanism to authenticate the web application users), but not DataBase mandatorily. According to Lines 10-13, JHipster offers four alternative authentication modalities: HTTPSession (a traditional stateful authentication mechanism based on HTTP sessions), JWT (an authentication method with a signed secure token that holds user's and authorities' credentials), OAuth2 (a stateless protection procedure with a secret key), or Uaa (an OAuth2 version for microservices). In addition, Line 11 of Figure 2 obligates that if the generated application is Monolithic, then its authentication mechanism must be HTTPSession, JWT, or OAuth2.

3. Analyzing random samples with rules

Halin et al.'s dataset encompasses a total of 26,256 configurations, from which 9,376 involve failures.

[4]https://www.jhipster.tech/

[5]https://github.com/xdevroey/jhipster-dataset

```
1  JHipster = Application Authentication BackEnd
2      ClusteredSession [DataBase] [Docker]
3      [InternationalizationSupport] [LibSass]
4      [SocialLogin] [SpringWebSockets]
5      TestingFrameworks;
6  Application = Gateway
7      | Microservice
8      | Monolithic
9      | UaaServer;
10 Authentication = HTTPSession
11     | JWT
12     | OAuth2
13     | Uaa;
14 BackEnd = Gradle
15     | Maven;
16 ClusteredSession = NoCS
17     | HazelCastCS;
18 DataBase = DataBaseType DataBaseDev DataBaseProd;
19 DataBaseType = Cassandra
20     | MongoDB
21     | [ElasticSearch] [Hibernate2ndLviCache];
22 Hibernate2ndLviCache = EhCache
23     | HazelCast;
24 DataBaseDev = CassandraDev
25     | H2
26     | MariaDBDev
27     | MongoDBDev
28     | MySQLDev
29     | PostgreSQLDev;
30 H2 = DiskBased
31     | InMemory;
32 DataBaseProd = CassandraProd
33     | MariaDBProd
34     | MongoDBProd
35     | MySQLProd
36     | PostgreSQLProd;
37 TestingFrameworks = Cucumber Gatling [Protractor];
```

Figure 1. JHipster grammar.

```
1  OAuth2 ∧ ¬SocialLogin ∧ ¬Microservice ⇒ SQL ∨
2      MongoDB
3  SocialLogin ⇒ (HTTPSession ∨ JWT) ∧ Monolithic ∧
4      (SQL ∨ MongoDB)
5  UaaServer ⇒ Uaa
6  ¬OAuth2 ∧ ¬SocialLogin ∧ ¬Microservice ⇒ SQL ∨
7      MongoDB ∨ Cassandra
8  Microservice ∨ UaaServer ⇒ ¬Protractor
9  Gateway ∨ Monolithic ⇒ Protractor
10 Microservice ∨ Gateway ⇒ JWT ∨ Uaa
11 Monolithic ⇒ HTTPSession ∨ JWT ∨ OAuth2
12 SpringWebSockets ∨ HazelCastCS ⇒ Gateway ∨
13     Monolithic
14 SQL ⇔ MariaDBProd ∨ MySQLProd ∨ PostgreSQLProd
15 MariaDBProd ⇒ H2 ∨ MariaDBDev
16 MySQLProd ⇒ H2 ∨ MySQLDev
17 PostgreSQLProd ⇒ H2 ∨ PostgreSQLDev
18 MongoDB ⇔ MongoDBProd
19 MongoDBProd ⇔ MongoDBDev
20 Cassandra ⇔ CassandraProd
21 CassandraProd ⇔ CassandraDev
22 LibSass ⇒ Gateway ∨ Monolithic
23 HazelCastCS ⇒ Gateway ∨ Monolithic
24 H2 ⇒ SQL
```

Figure 2. Additional constraints that JHipster configurations must satisfy.

For testing all configurations, the INRIA Grid'5000[6] consumed 4,376 hours of CPU time (182 days approximately), and 5.2 terabytes of disk space.

To identify the faults that caused the failures, Halin et al. used the arules[7] environment [35], which mines *association rules*. In contrast to the rule learners we recommend to use, *arules* not only looks for how the testing results depend on the selected options, but also on *how the options depend on themselves*. As a result, most of the rules that *arules* induces correspond to dependencies already specified by Figures 1 and 2. As that information is known before running the tests and it is useless for detecting the defects, it needs to be filtered manually.

In contrast, our method tests a reduced sample of configurations, and the obtained rules do not need any further post-processing.

3.1. Getting a uniform random sample

A straightforward strategy to get a uniform random sample is (i) producing a random configuration set without considering any option dependencies, and then (ii) verifying with a logic engine (e.g., a SAT-solver [36]) whether the configurations meet the dependencies. Unfortunately, option dependencies typically narrow the configuration space notably and thus getting a valid configuration at sheer random is very unlikely. As a result, more complex algorithms have been proposed to generate uniform random samples much more efficiently. An in-depth comparative study of state-of-the-art random samplers is presented in [13, 21].

Coming back to our case study, Table 1 summarizes the results of testing a uniform random sample with 20 configurations of JHipster. Each row represents a configuration, which is identified by a number (*Id. n*). The first 17 columns account for JHipster selectable options (Application, ..., SpringWebSockets). The last column shows if the configuration passed the test or failed.

It is worth noting that the KO's proportion in the sample is $\frac{8}{20} = 0.4$, which resembles quite fairly the population KO's proportion ($\frac{9{,}376}{26{,}256} = 0.36$).

3.2. Identifying faults

To infer which option combinations are causing the configurations to fail, we propose utilizing a rule induction system.

For example, Table 2 shows the rules learned with JRip, which is a RIPPER [30] implementation provided by the RWeka package[8]. There are three rules. The first two have the form conditions $\Rightarrow$ test result, where (i) conditions are a conjunction of option=value pairs, and (ii) the symbol "$\Rightarrow$" stands for the classical logic implication. The last rule, typically called the *default rule* [23], is triggered when none of the previous ones is matched. Table 2 includes two additional columns that show how many configurations in the sample are covered and misclassified by a rule. For instance, the second rule covers Configurations 8, 14, and 15; and the default rule misclassifies Configuration 16.

One of our approach advantages is that rules are usually highly interpretable, which happens because learning algorithms strive to obtain a rule set with the following properties:

- *Generality*, i.e., each rule should cover the maximum number of configurations in the sample. To do that, rule conditions try to impose as few constraints as possible. As a result, they tend to have few option=value pairs, and hence the rules become easy to understand.

- *Completeness*, i.e., every configuration in the sample should be covered by some rule.

- *Consistency*, i.e., each configuration's test result should be correctly classified by the rule set.

It is worth noting that the rules in Table 2 identify the two most important faults in the entire JHipster configuration population:

- Regarding the first rule, there was a *GitHub issue* in JHipster v3.6.1 stating that Uaa was in Beta version. In fact, Uaa did not work at all: in total, 4,488 of the 26,256 configurations include Uaa selected, and 4,114 of them failed (i.e., 91.67% of the configurations with Uaa).

- Concerning the second rule, all of the 4,248 configurations with MariaDB and Gradle failed.

The next section deepens on how to evaluate rule-set quality.

3.3. Evaluating rule-set quality

Several metrics can be used to account for the rule-set performance from different perspectives. These metrics analyze a *confusion matrix* that classifies predictions depending on whether they met the actual values or not.

[6]https://www.grid5000.fr
[7]https://cran.r-project.org/web/packages/arules
[8]https://cran.r-project.org/web/packages/RWeka/

Table 1. A JHipster uniform random sample of size 20.

Id.	**Applicat.**	**Authentic.**	**Back End**	**Clustered Session**	**Cucum.**	**DataBase Dev**	**DataBase Prod**	**DataBase Type**	**Docker**
1	Gateway	Uaa	Maven	No	TRUE	DiskBased	PostgreSQL	SQL	FALSE
2	Monolithic	JWT	Gradle	No	TRUE	DiskBased	MySQL	SQL	FALSE
3	Gateway	JWT	Maven	No	TRUE	MySQL	MySQL	SQL	TRUE
4	Monolithic	OAuth2	Maven	HazelCast	TRUE	PostgreSQL	PostgreSQL	SQL	TRUE
5	Gateway	JWT	Gradle	HazelCast	TRUE	DiskBased	MySQL	SQL	TRUE
6	Monolithic	JWT	Maven	HazelCast	TRUE	InMemory	PostgreSQL	SQL	FALSE
7	Gateway	Uaa	Maven	HazelCast	TRUE	Cassandra	Cassandra	Cassandra	TRUE
8	Monolithic	HTTPSession	Gradle	HazelCast	TRUE	DiskBased	MariaDB	SQL	FALSE
9	Gateway	JWT	Maven	HazelCast	TRUE	InMemory	MySQL	SQL	FALSE
10	Monolithic	HTTPSession	Maven	No	TRUE	PostgreSQL	PostgreSQL	SQL	TRUE
11	Gateway	Uaa	Maven	No	TRUE	PostgreSQL	PostgreSQL	SQL	TRUE
12	Monolithic	HTTPSession	Maven	HazelCast	TRUE	InMemory	PostgreSQL	SQL	FALSE
13	Monolithic	OAuth2	Maven	No	TRUE	MariaDB	MariaDB	SQL	FALSE
14	Monolithic	HTTPSession	Gradle	HazelCast	TRUE	InMemory	MariaDB	SQL	FALSE
15	Monolithic	OAuth2	Gradle	HazelCast	TRUE	InMemory	MariaDB	SQL	TRUE
16	Monolithic	OAuth2	Gradle	HazelCast	TRUE	InMemory	MySQL	SQL	TRUE
17	Gateway	Uaa	Maven	No	TRUE	InMemory	PostgreSQL	SQL	FALSE
18	Monolithic	OAuth2	Gradle	No	TRUE	InMemory	MySQL	SQL	TRUE
19	Monolithic	JWT	Maven	No	TRUE	InMemory	PostgreSQL	SQL	TRUE
20	Monolithic	OAuth2	Maven	HazelCast	TRUE	InMemory	MariaDB	SQL	TRUE

Id.	**Elastic Search**	**Gatling**	**Hibernate 2ndLvl Cache**	**Internat. Support**	**LibSass**	**Protractor**	**Social Login**	**Spring Web Sockets**	**Test Result**
1	FALSE	TRUE	No	FALSE	FALSE	TRUE	FALSE	TRUE	**KO**
2	FALSE	TRUE	HazelCast	TRUE	FALSE	TRUE	FALSE	TRUE	**OK**
3	TRUE	TRUE	EhCache	FALSE	FALSE	TRUE	FALSE	FALSE	**OK**
4	FALSE	TRUE	EhCache	FALSE	TRUE	TRUE	FALSE	FALSE	**OK**
5	TRUE	TRUE	HazelCast	FALSE	TRUE	TRUE	FALSE	FALSE	**OK**
6	TRUE	TRUE	EhCache	TRUE	TRUE	TRUE	TRUE	TRUE	**OK**
7	FALSE	TRUE	No	FALSE	FALSE	TRUE	FALSE	TRUE	**KO**
8	FALSE	TRUE	No	FALSE	FALSE	TRUE	FALSE	FALSE	**KO**
9	TRUE	TRUE	No	FALSE	FALSE	TRUE	FALSE	FALSE	**OK**
10	FALSE	TRUE	No	FALSE	TRUE	TRUE	FALSE	TRUE	**OK**
11	FALSE	TRUE	No	TRUE	TRUE	TRUE	FALSE	TRUE	**KO**
12	FALSE	TRUE	EhCache	FALSE	TRUE	TRUE	TRUE	FALSE	**OK**
13	TRUE	TRUE	HazelCast	FALSE	TRUE	TRUE	FALSE	TRUE	**OK**
14	TRUE	TRUE	No	FALSE	TRUE	TRUE	TRUE	TRUE	**KO**
15	TRUE	TRUE	HazelCast	FALSE	FALSE	TRUE	FALSE	FALSE	**KO**
16	FALSE	TRUE	No	TRUE	TRUE	TRUE	FALSE	TRUE	**KO**
17	TRUE	TRUE	EhCache	TRUE	TRUE	TRUE	FALSE	TRUE	**KO**
18	FALSE	TRUE	No	FALSE	FALSE	TRUE	FALSE	FALSE	**OK**
19	TRUE	TRUE	No	FALSE	TRUE	TRUE	TRUE	TRUE	**OK**
20	TRUE	TRUE	EhCache	TRUE	FALSE	TRUE	FALSE	FALSE	**OK**

Table 2. Rules induced with JRip from the sample in Table 1.

Induced Rule	#Covered configurations	#Misclassified configurations
Authentication = Uaa ⇒ Test Result = **KO**	4	0
(DataBaseProd = MariaDB) ∧ (BackEnd = Gradle) ⇒ Test Result = **KO**	3	0
DEFAULT ⇒ Test Result = **OK**	13	1

For instance, Figure 3 shows the confusion matrix resulting from applying the rules in Table 2 to the whole population of configurations. These rules:

- Correctly predicted that 16,506 configurations did not present any failures. As the *testing goal* is to demonstrate that a program has faults[9], these configurations are considered as *True Negatives* (TNs).
- Correctly predicted that 7,642 configurations raised failures. These configurations are considered *True Positives* (TPs).
- Mistakenly predicted that 374 configurations produced failures. These configurations are considered *False Positives* (FPs).
- Mistakenly predicted that 1,734 configurations did not present any failure. These configurations are considered *False Negatives* (FNs).

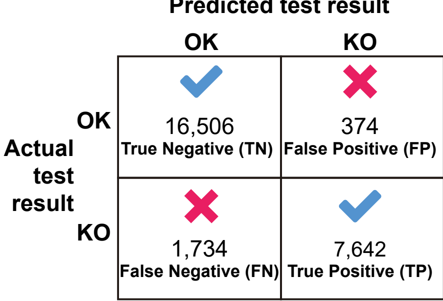

Figure 3. Confusion matrix resulting from applying the rules in Table 2 to all possible JHipster configurations.

We propose measuring rule-set performance with the following indicators, all of them ranging from 0 to 1:

1. *Accuracy*, also known as *success rate*, is calculated with Equation 1 [38] as the ratio between correct predictions and the total number of predictions. Note that Equations 1-6 also show the measures for the numerical values corresponding to the confusion matrix in Figure 3.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 0.92 \tag{1}$$

2. *Cohen's kappa coefficient*. The population of configurations may sometimes be considerably imbalanced, with an overwhelming proportion of OKs (or KOs). In those situations, a rule induction algorithm could "cheat" to accomplish high accuracy by constantly picking the most frequent class. The kappa statistic κ adjusts accuracy with Equations 2 and 3 [39] by considering the probability of a correct prediction just by chance.

$$\kappa = \frac{\text{Accuracy} - \text{Pr}}{1 - \text{Pr}} = 0.82 \tag{2}$$

$$\begin{aligned}\text{Pr} = & \frac{(\text{TN} + \text{FP})(\text{TN} + \text{FN})}{(\text{TP} + \text{TN} + \text{FP} + \text{FN})^2} \\ & + \frac{(\text{FN} + \text{TP})(\text{FP} + \text{TP})}{(\text{TP} + \text{TN} + \text{FP} + \text{FN})^2} = 0.56\end{aligned} \tag{3}$$

3. *Precision* and *recall* are typically used in *information retrieval* problems [40]. Precision, known as the *positive predictive value* as well, calculates with Equation 4 how frequently the predicted KOs are actually KO. Recall, which is also called *sensitivity* or *true positive rate*, calculates with Equation 5 the proportion of actual KOs that were correctly predicted.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = 0.95 \tag{4}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = 0.82 \tag{5}$$

4. *Specificity*, also known as *true negative rate*, calculates with Equation 6 [38] the proportion of actual OKs that were correctly predicted.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} = 0.98 \tag{6}$$

According to the measures obtained with Equations 1-6 from Figure 3, the rules in Table 2 worked surprisingly well for a small sample with just twenty configurations. The following section analyzes in-depth the effect on rule-set performance of increasing the sample size and selecting the proper rule induction algorithm.

[9] As Myers et al. [37] highlight, the testing goal has a critical psychological effect on the testers. When the objective is to demonstrate that a program has no faults, testers are influenced subconsciously to this end and tend to write test sets with a low probability of producing failures. Accordingly, it is recommended that testers pursue the contrary, i.e., demonstrating that the program fails.

4. Experimental validation

This section reports the experiments we undertook to answer the following Research Questions (RQs):

- **RQ1: Sample size and rule learning influence on rule-set quality**. *To what extent do the number of configurations in the sample and the chosen rule induction system affect the rule-set quality?*
- **RQ2: Best rule learning algorithm**. *What rule induction algorithm does predict faults the best?*
- **RQ3: Expected results from a small sample**. *What performance can achieve the rules induced from a reduced sample (particularly, from a sample with 100 configurations)?*

4.1. Experimental setup

Five algorithms were validated empirically to verify their value for identifying faults in the JHipster dataset:

1. *Algorithm Quasi-optimal (AQ)* [25, 26]
2. *CN2* [27]
3. *Repeated Incremental Pruning to Produce Error Reduction (RIPPER)* [30]
4. *Learning from Examples Module, version 2 (LEM2)* [28]
5. *PART* [29]

In particular, the following R packages were used to perform the validation:

1. *RoughSets*[10] [41] for testing the algorithms AQ, CN2, and LEM2.
2. *RWeka*[11] [42, 29] for testing the algorithms PART and a RIPPER implementation called JRip.
3. *Caret*[12] [43] to evaluate rules' quality.

The data and code scripts to replicate our experiments are freely available at:

https://github.com/rheradio/hicss2022-experiments

[10]https://cran.r-project.org/package=RoughSets
[11]https://CRAN.R-project.org/package=RWeka
[12]https://CRAN.R-project.org/package=caret

4.2. Results

The scatter plot in Figure 4 shows the measurements (accuracy, κ, precision, recall, and specificity) corresponding to the rules induced with the learning algorithms (AQ, CN2, JRip, LEM2, and PART) from samples of increasing size (from 20 to 200 configurations). Each blue point represents a measure. To facilitate the trend visualization, each plot includes a red regression curve.

According to Figure 4, measures improve as the sample size increases. Also, algorithms affect the measures. To quantify precisely the influence of both factors on each performance metric, Table 3 summarizes five *multivariate regression models* [14]. Each model has a metric as *response variable*, and the rule algorithms and sample size as *explanatory variables*. For example, the first subtable corresponds to the model:

$$\text{Accuracy} \sim \text{AQ} + \text{CN2} + \text{JRip} + \text{LEM2} + \text{PART} + \text{sample size}$$

Therefore, the expected accuracy for a rule set induced with AQ from a sample with 100 configurations of the JHipster dataset would be on average:

$$0.75{\cdot}\text{AQ} + 0{\cdot}\text{CN2} + 0{\cdot}\text{JRip} + 0{\cdot}\text{LEM2} + 0{\cdot}\text{PART} + 100{\cdot}0.0008 = 0.83$$

According to Table 3, the influence of the rule algorithms and sample size on every metric is statistically significant. Moreover, the coefficients highlighted in red show that (i) JRip obtains the best results in terms of accuracy, κ, precision, and recall, and (ii) LEM2 gets the highest specificity.

For validating to what extent our approach works with small samples, we generated 1,000 random samples with 100 configurations, induced their corresponding rules with JRip, and computed their performance measures on the entire configuration population. The histograms in Figure 5 show the obtained results, which are complementarily summarized in Table 4.

4.3. Discussion

The analysis of the JHipster dataset provides experimental evidence supporting the following answers to the research questions:

- **RQ1 (Sample size and rule learning influence)**: Both the sample size and the chosen rule learning algorithm influence the quality of the induced rules, i.e., the rule set capacity to point relevant faults from a sample.
- **RQ2 (Best rule learning algorithm)**: The rule induction algorithm that obtained the best results is the RIPPER implementation JRip.

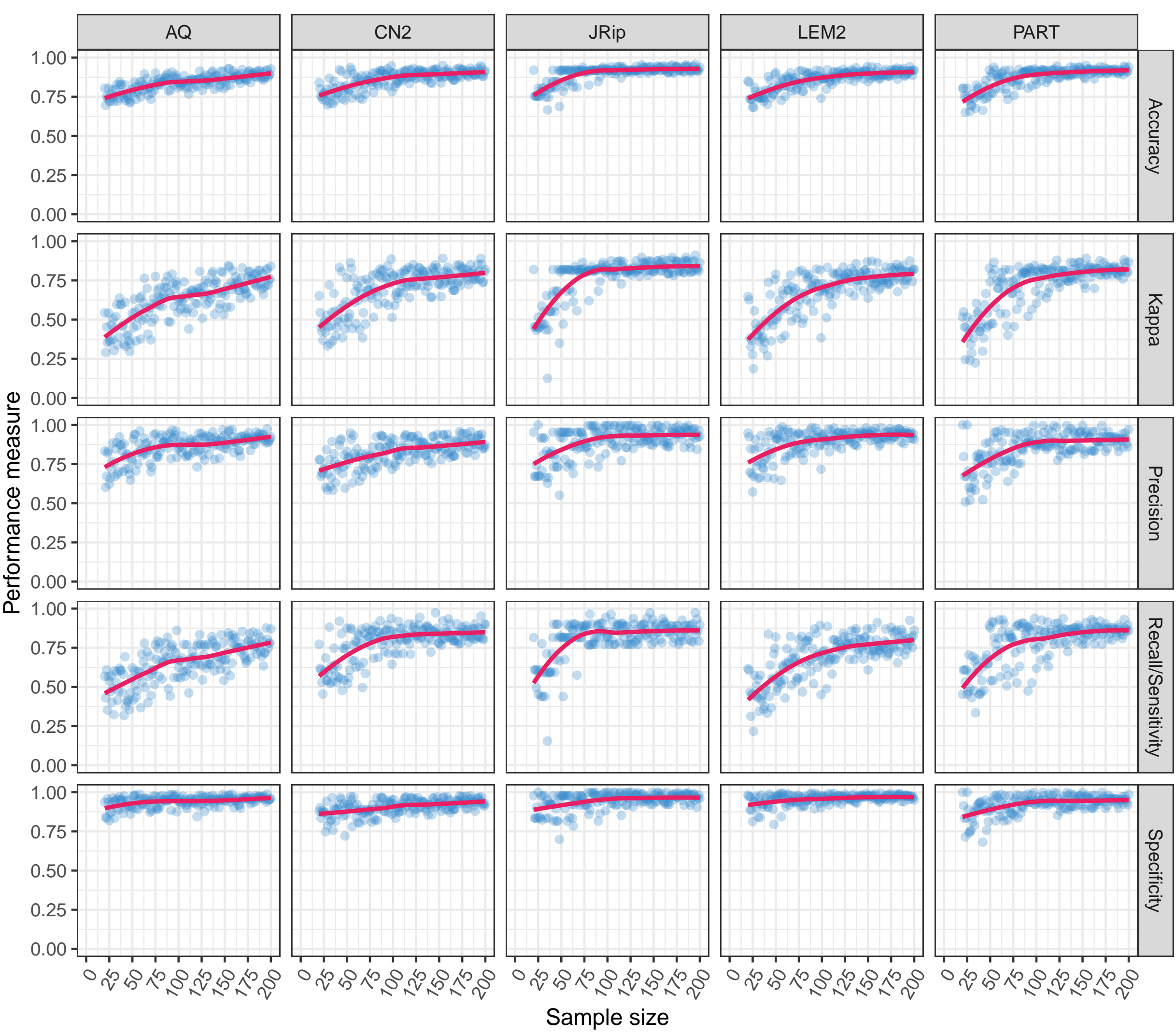

Figure 4. Performance measures depending on the sample size and the used rule induction algorithm.

Table 3. Summary of the linear models that account for the influence of the rule induction algorithms and the sample size on the performance metrics.

Coeff.	Estimate	Std. Error	t-value	Pr(>|t|)
Accuracy (adj. $R^2 = 0.99$, p-value<2.2e-16)				
AQ	0.7527	0.0043	174.51	<2e-16
CN2	0.7770	0.0043	180.14	<2e-16
JRip	**0.8116**	0.0043	188.16	<2e-16
LEM2	0.7744	0.0043	179.54	<2e-16
PART	0.7862	0.0043	182.27	<2e-16
sample size	0.0008	<2e-16	29.30	<2e-16
Kappa (adj. $R^2 = 0.98$, p-value<2.2e-16)				
AQ	0.4227	0.0102	41.48	<2e-16
CN2	0.4939	0.0102	48.47	<2e-16
JRip	**0.5674**	0.0102	55.68	<2e-16
LEM2	0.4724	0.0102	46.36	<2e-16
PART	0.5110	0.0102	50.14	<2e-16
sample size	0.0019	0.0001	29.32	<2e-16
Precision (adj. $R^2 = 0.99$, p-value<2.2e-16)				
AQ	0.7615	0.0075	101.89	<2e-16
CN2	0.7260	0.0075	97.14	<2e-16
JRip	**0.7991**	0.0075	106.93	<2e-16
LEM2	0.7959	0.0075	106.50	<2e-16
PART	0.7566	0.0075	101.24	<2e-16
sample size	0.0009	<2e-16	19.46	<2e-16
Recall / Sensitivity (adj. $R^2 = 0.98$, p-value<2.2e-16)				
AQ	0.4873	0.0101	48.07	<2e-16
CN2	0.6181	0.0101	60.96	<2e-16
JRip	**0.6441**	0.0101	63.53	<2e-16
LEM2	0.5214	0.0101	51.43	<2e-16
PART	0.6114	0.0101	60.30	<2e-16
sample size	0.0015	0.0001	24.14	<2e-16
Specificity (adj. $R^2 = 0.99$, p-value<2.2e-16)				
AQ	0.9001	0.0043	211.38	<2e-16
CN2	0.8653	0.0043	203.19	<2e-16
JRip	0.9046	0.0043	212.42	<2e-16
LEM2	**0.9149**	0.0043	214.86	<2e-16
PART	0.8833	0.0043	207.43	<2e-16
sample size	0.0004	<2e-16	14.23	<2e-16

Table 4. Performance measure of the JRip rules learned from 1,000 samples with 100 configurations (descriptive statistics).

Metric	Mean	Std. dev.	Median	Min.	Max.
Accuracy	0.91	0.02	0.92	0.81	0.96
Kappa	0.81	0.04	0.82	0.56	0.90
Precision	0.90	0.06	0.91	0.69	1.00
Recall	0.86	0.05	0.86	0.65	0.98
Specificity	0.95	0.04	0.95	0.76	1.00

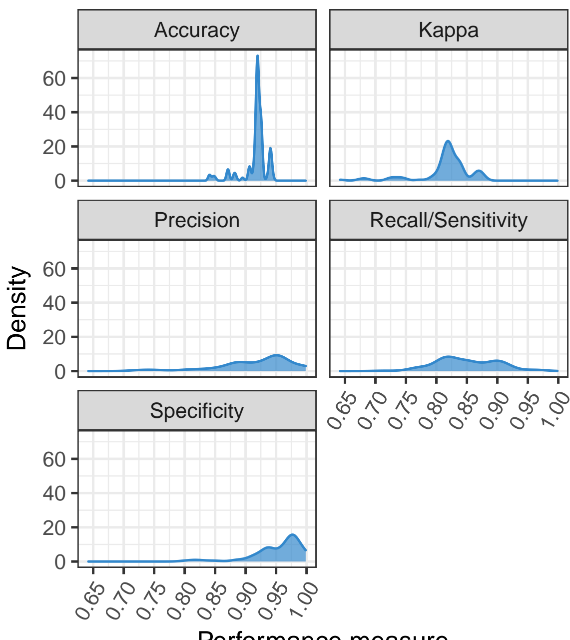

Figure 5. Performance measures of the JRip rules induced from 1,000 samples with 100 configurations (density plots).

- **RQ3 (Rules' quality from small samples)**: A sample of size 100 typically supports extracting a rule set with enough quality to identify the most significant faults.

5. Conclusions and future work

Classic literature on software testing emphasizes that the testers' primary goal should be triggering the maximum number of failures that subsequently lead to uncovering software defects. In the context of highly configurable software, this strategy would look for a configuration sample that *maximizes* the number of failures. In contrast, we think it is preferable collecting a sample that *represents* the population adequately and thus supports the automated determination of the faults. That is, a sample that includes information concerning what option combinations fail but also what others work correctly.

In this paper, we have provided a concrete implementation of our approach built on top of a uniform random SAT-sampler and a rule-learning engine. As the input of the random sampler is a propositional logic formula, our prototype supports testing any software whose configurable options are expressed in a language translatable into Boolean logic, such as KConfig [10, 11], CDL [44], and Feature

Diagrams [45, 46].

The approach has been empirically validated with the JHipster dataset, which encompasses a considerable configuration space with 26,256 possible variants (36% of them involving failures). This is the largest dataset available in the literature [31, 32, 33], and due to its size and complexity, we believe our results are generalizable to other cases. Nevertheless, this should be tested in future work. Also, we plan to examine the capability of our approach to find and debug faults compared to other testing strategies, such as *combinatorial testing* [47] and *dissimilarity sampling* [48]. This comparison would definitely prove which strategy is better to fix faults: processing a representative sample, or working with a distorted sample that maximizes the number of failures.

Acknowledgements

This work has been supported by (i) the Universidad Nacional de Educacion a Distancia under grant 096-034091 2021V/PUNED/008 (OPTIVAC), (ii) the Spanish Ministry of Science, Innovation and Universities, under grant PID2019-105381GA-I00 (iScience), and (ii) the Community of Madrid, under the research network CAM ROBOCITY2030-DIH-CM S2018/NMT-4331.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, , and T. Katzenellenbogen, "Reversible debugging software," tech. rep., University of Cambridge, Judge Business School, 2013.

[2] I. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology*, vol. 58, pp. 388–402, 2015.

[3] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *International Conference on Software Engineering (ICSE)*, (Austin, Texas, USA), 2016.

[4] R. Heradio, D. Fernandez-Amoros, C. Cerrada, and M. J. Cobo, "Machine Learning for Software Engineering: a Bibliometric Analysis from 2015 to 2019," in *54th Hawaii International Conference on System Sciences (HICSS)*, (Hawaii, USA), 2021.

[5] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[6] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," *Empirical Software Engineering*, vol. 21, no. 1, pp. 43–71, 2016.

[7] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–45, 2014.

[8] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications," in *36th International Conference on Software Engineering (ICSE)*, (Hyderabad, India), p. 907–918, 2014.

[9] R. Heradio, H. Perez-Morago, D. Fernandez-Amoros, F. Javier Cabrerizo, and E. Herrera-Viedma, "A bibliometric analysis of 20 years of research on software product lines," *Information and Software Technology*, vol. 72, pp. 1–15, 2016.

[10] D. Fernandez-Amoros, R. Heradio, C. Mayr-Dorn, and A. Egyed, "A Kconfig Translation to Logic with One-Way Validation System," in *23rd International Systems and Software Product Line Conference (SPLC)*, (Paris, Feance), pp. 303–308, 2019.

[11] D. Fernandez-Amoros, S. Bra, E. Aranda-Escolastico, and R. Heradio, "Using Extended Logical Primitives for Efficient BDD Building," *Mathematics*, vol. 8, no. 8, p. 1253, 2020.

[12] J. C. Villanueva, "How many atoms are there in the universe?." Universe Today, 2009.

[13] R. Heradio, D. Fernandez-Amoros, J. A. Galindo, and D. Benavides, "Uniform and scalable SAT-sampling for configurable systems," in *24th Systems and Software Product Line Conference (SPLC)*, (Montréal, Canada), pp. 1–11, 2020.

[14] D. Kaplan, *Statistical Modeling: A Fresh Approach*. Project Mosaic, 2012.

[15] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?," in *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, (Xian, China, China), pp. 240–251, 2019.

[16] J. Oh, P. Gazzillo, and D. Batory, "t-wise Coverage by Uniform Sampling," in *23rd International Systems and Software Product Line Conference (SPLC)*, (Paris, France), pp. 84–87, 2019.

[17] D. Batory, J. Oh, R. Heradio, and D. Benavides, *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, ch. Product Optimization in Stepwise Design, pp. 63–81. Springer International Publishing, 2021.

[18] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge Compilation meets Uniform Sampling," in *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, (Awassa, Ethiopia), pp. 620–636, 2018.

[19] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, (Oxford, UK), pp. 135–147, 2018.

[20] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, (London, UK), pp. 304–319, 2015.

[21] R. Heradio, D. Fernandez-Amoros, J. Galindo, D. Benavides, and D. Batory, "Uniform and Scalable Sampling of Highly Congurable Systems," *Submitted to Empirical Software Engineering (currently under review)*, 2021.

[22] J. W. Grzymala-Busse, *Data Mining and Knowledge Discovery Handbook*, ch. Rule Induction, pp. 249–265. Springer US, 2010.

[23] J. Fürnkranz, D. Gamberger, and N. Lavrac, *Foundations of Rule Learning*. Springer, 2012.

[24] A. M. Hass, *Guide to Advanced Software Testing, 2nd Edition*. Artech House, 2014.

[25] R. S. Michalski, I. Mozetic, J. Hong, and N. Lavrac, "The Multi-Purpose Incremental Learning System AQ15 and Its Testing Application to Three Medical Domains," in *5th National Conference on Artificial Intelligence (AAAI)*, (Philadelphia, Pennsylvania, USA), pp. 1041–1045, 1986.

[26] R. S. Michalski, K. A. Kaufman, and J. Wnek, "The AQ Family of Learning Programs: A Review of Recent Developments and an Exemplary Application," tech. rep., Machine Learning and Inference Laboratory, George Mason University, 1991.

[27] P. Clark and T. Niblett, "The CN2 induction algorithm," *Machine Learning*, vol. 3, no. 4, pp. 261–283, 1989.

[28] J. W. Grzymala-Busse, "A New Version of the Rule Induction System LERS," *Fundamenta Informaticae*, vol. 31, no. 1, pp. 2–9, 1997.

[29] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd ed., 2005.

[30] W. W. Cohen, "Fast effective rule induction," in *12th International Conference on Machine Learning (ICML)*, (Tahoe City, California, USA), 1995.

[31] A. Halin and A. Nuttinck, "Sampling & Testing all configurations: The JHipster case study," Master's thesis, Universite de Namur. Faculty of Computer Science, 2017.

[32] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and P. Heymans, "Yo Variability! JHipster: A Playground for Web-Apps Analyses," in *11th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, (Eindhoven, Netherlands), 2017.

[33] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack," *Empirical Software Engineering*, vol. 24, no. 2, pp. 674–717, 2019.

[34] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?," *Communications of the ACM*, vol. 20, no. 11, p. 822–823, 1977.

[35] M. Hahsler, B. Grün, and K. Hornik, "arules - A Computational Environment for Mining Association Rules and Frequent Item Sets," *Journal of Statistical Software*, vol. 14, no. 15, pp. 1–25, 2005.

[36] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*. IOS Press, 2009.

[37] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing (3rd edition)*. John Wiley & Sons Inc., 2011.

[38] B. Lantz, *Machine Learning with R: Expert techniques for predictive modeling*. Packt Publishing, 2019.

[39] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[40] D. Powers, "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[41] L. S. Riza, *Data Science and Big Data Processing in R: Representations and Software*. PhD thesis, Dept. de Ciencias de la Computación e Inteligencia Artificial. Universidad de Granada, 2015.

[42] K. Hornik, C. Buchta, and A. Zeileis, "Open-source machine learning: R meets Weka," *Computational Statistics*, vol. 24, no. 2, pp. 225–232, 2009.

[43] M. Kuhn, "Building Predictive Models in R Using the caret Package," *Computational Statistics*, vol. 28, no. 5, pp. 1–26, 2008.

[44] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[45] R. Heradio, D. Fernandez-Amoros, J. A. Cerrada, and I. Abad, "A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 8, pp. 1177–1204, 2013.

[46] D. Fernandez-Amoros, R. Heradio, J. A. Cerrada, and C. Cerrada, "A scalable approach to exact model and commonality counting for extended feature models," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 895–910, 2014.

[47] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. Chapman and Hall/CRC, 2013.

[48] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "PLEDGE: a product line editor and test generation tool," in *17th International Software Product Line Conference (SPLC)*, (Tokyo, Japan), 2013.