Using Bad Learners to Find Good Configurations

Vivek Nair
North Carolina State University
Raleigh, North Carolina, USA

Tim Menzies
North Carolina State University
Raleigh, North Carolina, USA

Norbert Siegmund
Bauhaus-University
Weimar, Germany

Sven Apel
University of Passau
Passau, Germany

ABSTRACT

Finding the optimally performing configuration of a software system for a given setting is often challenging. Recent approaches address this challenge by learning performance models based on a sample set of configurations. However, building an accurate performance model can be very expensive (and is often infeasible in practice). The central insight of this paper is that exact performance values (e.g., the response time of a software system) are not required to rank configurations and to identify the optimal one. As shown by our experiments, performance models that are cheap to learn but inaccurate (with respect to the difference between actual and predicted performance) can still be used rank configurations and hence find the optimal configuration. This novel *rank-based approach* allows us to significantly reduce the cost (in terms of number of measurements of sample configuration) as well as the time required to build performance models. We evaluate our approach with 21 scenarios based on 9 software systems and demonstrate that our approach is beneficial in 16 scenarios; for the remaining 5 scenarios, an accurate model can be built by using very few samples anyway, without the need for a rank-based approach.

CCS CONCEPTS

• **Computing methodologies** → **Ranking**; **Classification and regression trees**; • **Software and its engineering** → *Model-driven software engineering*; Feature interaction; Software performance;

KEYWORDS

Performance Prediction, SBSE, Sampling, Rank-based method

ACM Reference format:
Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using Bad Learners to Find Good Configurations. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 04-08, 2017,* 11 pages.
https://doi.org/10.1145/3106237.3106238

1 INTRODUCTION

This paper proposes an improvement of recent papers presented at ICSE'12, ASE'13, and ASE'15, which predict system performance based on learning influences of individual configuration options and combinations of thereof [5, 18, 21]. The idea is to measure a few configurations of a configurable software system and to make statements about the performance of its other configurations. Thereby, the goal is to predict the performance of a given configuration as accurate as possible. We show that, if we (slightly) relax the question we ask, we can build useful predictors using very small sample sets. Specifically, instead of asking "How long will this configuration run?", we ask instead "Will this configuration run faster than that configuration?" or "Which is the fastest configuration?".

This is an important area of research since understanding system configurations has become a major problem in modern software systems. In their recent paper, Xu et al. documented the difficulties developers face with understanding the configuration spaces of their systems [24]. As a result, developers tend to ignore over 83% of configuration options, which leaves considerable optimization potential untapped and induces major economic cost [24].

With many configurations available for today's software systems, it is challenging to optimize for functional and non-functional properties. For functional properties, Chen et. al [1] and Sayyad et. al [19] developed fast techniques to find near-optimal configurations by solving a five-goal optimization problem. Henard et. al [6] used a SAT solver along with Multi-Objective Evolutionary Algorithms to repair invalid mutants found during the search process.

For non-functional properties, researchers have also developed a number of approaches. For example, it has been shown that the runtime of a configuration can be predicted with high accuracy by sampling and learning performance models [5, 18, 21]. State-of-the-art techniques rely on configuration data from which it is possible to build very accurate models. For example, prior work [15] has used sub-sampling to build predictors for configuration runtimes using predictors with error rates less than 5% (quantified in terms of *residual-based* measures such as Mean Magnitude of Relative Error, or MMRE, $(\sum_i^n(|a_i - p_i|/a_i))/n$ where a_i, p_i are the *actual* and *predicted values*). Figure 1 shows in green a number of real-world systems whose performance behavior can be modelled with high accuracy using state-of-the-art techniques.

Recently, we have come across software systems whose configuration spaces are far more complicated and hard to model. For example, when the state-of-the-art technique of Guo at al. [5] is applied to these software systems, the error rates of the generated predictor is up to 80%—see the yellow and **red** systems of Figure 1. The existence of these harder-to-model systems raises a serious validity question for all prior research in this area:

- Was prior research merely solving easy problems?

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106238

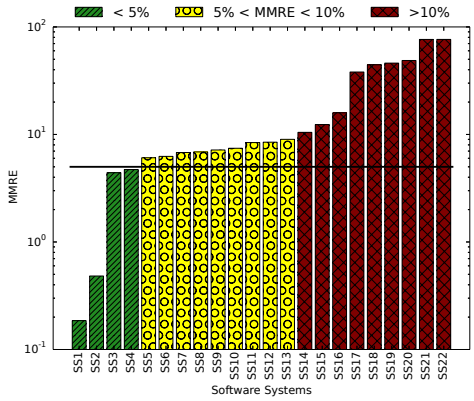

Figure 1: Errors of the predictions made by using CART, a machine learning technique (refer to Section 5), to model different software systems. Due to the results of Figure 2, we use 30%/70% of the valid configurations (chosen randomly) to train/test the model.

- Can we learn predictors for non-functional properties of more complex systems?

One pragmatic issue that complicates answering these two questions is the *minimal sampling problem*. It can be prohibitively expensive to run and test all configurations of modern software systems since their configuration spaces are very large. For example, to obtain the data used in our experiments, we required over a month of CPU time for measuring (and much longer, if we also count the time required for compiling the code prior to execution). Other researchers have commented that, in real-world scenarios, the cost of acquiring the optimal configuration is overly expensive and time-consuming [23]. Hence, the goal of this paper must be:

(1) Find predictors for non-functional properties for the hard-to-model systems of Figure 1, where learning accurate performance models is expensive.
(2) Use as few sample configurations as possible.

The key result of this paper is that, even when *residual-based* performance models are inaccurate, *ranking* performance models can still be very useful for configuration optimization. Note that:

- Predictive models return a value for a configuration;
- Ranking models rank N configurations from "best" to "worst".

There are two very practical cases where such ranking models would suffice:

- Developers want to know the fastest configuration;
- Developers are debating alternate configurations and want to know which might run faster.

In this paper, we explore two research questions about constructing ranking models.

RQ1: *Can inaccurate predictive models still accurately rank configurations?*

We show below that, even if a model has, overall, a low predictive accuracy (i.e., a high MMRE), the predictions can still be used to effectively rank configurations. The rankings are heuristic in nature and hence may be slightly inaccurate (w.r.t the actual performance value). That said, overall, our rankings are surprisingly accurate.

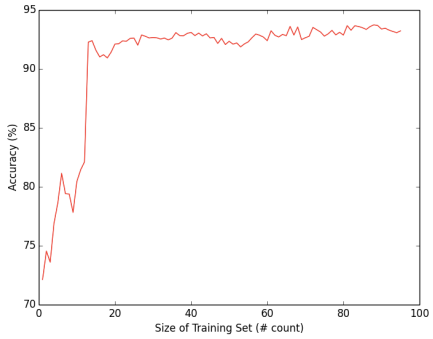

Figure 2: The relationship between the accuracy (in terms of MMRE) and the number of samples used to train the performance model of the Apache Web Server. Note that the accuracy does not improve substantially after 20 sample configurations.

For example, when exploring the configuration space of SQLite, our rankings are usually wrong only by less than 6 neighboring configurations – which is a very small number considering that SQLite has almost 4 million configurations.

RQ2: *How expensive is a rank-based approach (in terms of how many configurations must be executed)?*

To answer this question, we studied the configurations of 21 scenarios based on 9 open-source systems. We measure the benefit of our rank-based approach as the percentage of required measurements needed by state-of-the-art techniques in this field (see Sarkar et al. [18] presented at ASE'15). Those percentages were as follows – Note that *lower* values are *better* and values under 100% denote an improvement over the state-of-the-art (from Figure 9):

{5,5,5,5,5,10,20,20,20,20,30,30,35,35,40,40,50,50,70,80,80,110}%

That is, the novel rank-based approach described in this paper is rarely worse than the state of the art and often far better. For example, as shown later in Figure 9, for one of the scenarios of Apache Storm, SS11, the rank-based approach uses only 5% of the measurements used by a residual-based approach.

The rest of this paper is structured as follows: we first formally describe the prediction problem. Then, we describe the state-of-the-art approach proposed by Sarkar et al. [18], henceforth referred to as residual-based approach, followed by the description of our rank-based approach. Then, the subject systems used in the paper are described followed by our evaluation. The paper ends with a discussion on why a rank-based approach works; finally, we conclude. To assist other researchers, a reproduction package with all our scripts and data are available on GitHub.[1]

2 PROBLEM FORMALIZATION

A configurable software system has a set X of configurations $x \in X$. Let x_i indicate the *ith* configuration option of configuration x, which takes the values from a finite domain $Dom(x_i)$. In general, x_i indicates either an (i) integer variable or a (ii) Boolean variable. The configuration space is thus $Dom(x_1) \times Dom(x_2) \times ... \times Dom(x_n)$, which is the Cartesian product of the domains, where $n = |x|$ is the number of configuration options of the system. Each configuration

[1] https://github.com/ai-se/Reimplement/tree/cleaned_version

```
1   # Progressive Sampling
2   def progressive(training, testing, lives=3):
3     # For stopping criterion
4     last_score = -1
5     independent_vals = list()
6     dependent_vals = list()
7     for count in range(1, len(training)):
8       # Add one configuration to the training set
9       independent_vals += training[count]
10      # Measure the performance value for the newly
11      # added configuration
12      dependent_vals += measure(training_set[count])
13      # Build model
14      model = build_model(independent_vals, ↩
            dependent_vals)
15      # Test Model
16      perf_score = test_model(model, testing, measure(↩
            testing))
17      # If current accuracy score is not better than
18      # the previous accuracy score, then loose life
19      if perf_score <= last_score:
20        lives -= 1
21      last_score = perf_score
22      # If all lives are lost, exit loop
23      if lives == 0: break
24    return model
```

Figure 3: Pseudocode of progressive sampling.

```
# Projective Sampling                                          1
def projective(training, testing, thres_freq=3):               2
  collector = list()                                           3
  independent_vals = list()                                    4
  dependent_vals = list()                                      5
  for count in range(1, len(training)):                        6
    # Add one configuration to the training set                7
    independent_vals += training[count]                        8
    # Measure the performance value for the newly              9
    # added configuration                                      10
    dependent_vals += measure(training_set[count])             11
    # update feature frequency table                           12
    T = update_frequency_table(training[count])                13
    # Build model                                              14
    model = build_model(independent_vals, ↩                    15
        dependent_vals)
    # Test Model                                               16
    perf_score = test_model(model, testing, measure(↩         17
        testing))
    # Collect the the pair of |training set|                   18
    # and performance score                                    19
    collector += [count, perf_score]                           20
    # minimum values of the feature frequency table            21
    if min(T) >= thresh_freq: break                            22
  return model                                                 23
```

Figure 4: Pseudocode of projective sampling.

(x) has a corresponding performance measure $y \in Y$ associated with it. The performance measure is also referred to as dependent variable. We denote the performance measure associated with a given configuration by $y = f(x)$. We consider the problem of ranking configurations (x^*) that such that $f(x)$ is less than other configurations in the configuration space of X with few measurements.

$$f(x^*) \leq f(x),\ \forall x \in X \setminus x^* \tag{1}$$

Our goal is to find the (near) optimal configuration of a system where it is not possible to build an accurate performance model as prescribed in earlier work.

3 RESIDUAL-BASED APPROACHES

In this section, we discuss the residual-based approaches for building performance models for configurable software systems. For further details, we refer to Sarkar et. al [18].

3.1 Progressive Sampling

When the cost of collecting data is higher than the cost of building a performance model, it is imperative to minimize the number of measurements required for model building. A learning curve shows the relationship between the size of the training set and the accuracy of the model. In Figure 2, the horizontal axis represents the number of samples used to create the performance model, whereas the vertical axis represents the accuracy (measured in terms of MMRE) of the model learned. Learning curves typically have a steep sloping portion early in the curve followed by a plateau late in the curve. The plateau occurs when adding data does not improve the accuracy of the model. As engineers, we would like to stop sampling as soon as the learning curve starts to flatten.

Figure 3 is a generic algorithm that defines the process of progressive sampling. *Progressive* sampling starts by clearly defining the data used in the training set, called training pool, from which the samples would be selected (randomly, in this case) and then tested against the testing set. At each iteration, a (set of) data instance(s) of the training pool is added to the training set (Line 9). Once the data instances are selected from the training pool, they are evaluated, which in our setting means measuring the performance of the selected configuration (Line 12). The configurations and the associated performance scores are used to build the model (Line 14). The model is validated using the testing set[2], then, the accuracy is then computed. The accuracy can be quantified by any measure, such as MMRE, MBRE, Absolute Error, etc. In our setting, we assume that the measure is accuracy (higher is better). Once the accuracy score is calculated, it is compared with the accuracy score obtained before adding the new set of configurations to the training set. If the accuracy of the model (with more data) does not improve the accuracy when compared to the previous iteration (lesser data), then a life is lost. This termination criterion is widely used in the field of multi-objective optimization to determine degree of convergence [9].

3.2 Projective Sampling

One of the shortcomings of progressive sampling is that the resulting performance model achieves an acceptable accuracy only after a large number of iterations, which implies high modelling cost. There is no way to actually determine the cost of modelling until the performance model is already built, which defeats its purpose, as there is a risk of over-shooting the modelling budget and still not obtain an accurate model. *Projective* sampling addresses this

[2]The testing data consist of the configurations as well as the corresponding performance scores.

problem by approximating the learning curve using a minimal set of initial sampling points (configurations), thus providing the stakeholders with an estimate of the modelling cost. Sarkar et. al [18] used projective sampling to predict the number of samples required to build a performance model. The initial data points are selected by randomly adding a constant number of samples (configurations) to the training set from the training pool. In each iteration, the model is built, and the accuracy of the model is calculated using the testing data. A feature-frequency heuristic is used as the termination criterion. The feature-frequency heuristic counts the number of times a feature has been selected and deselected. Sampling stops when the counts of features selected and deselected is, at least, at a predefined threshold (*thresh_freq*).

Figure 4 provides a generic algorithm for projective sampling. Similar to progressive sampling, projective sampling starts with selecting samples from the training pool and adding them to the training set (Line 8). Once the samples are selected, the corresponding configurations are evaluated (Line 11). The feature-frequency table T is then updated by calculating the number of features that are selected and deselected in *independent_vals* (Line 13). The configurations and the associated performance values are then used to build a performance model, and the accuracy is calculated (Lines 15–17). The number of configurations and the accuracy score are stored in the collector, since our objective is to estimate the learning curve. $min(T)$ holds the minimum value of the feature selection and deselection frequencies in T. Once the value of $min(T)$ is greater than *thresh_freq*, the sampled points are used to estimate the learning curve. These points are used to search for a best-fit function that can be used to extrapolate the learning curve (there are several available, including Logarithmic, Weiss and Tian, Power Law and Exponential [18]). Once the best-fit function is found, it is used to determine the point of convergence.

4 RANK-BASED APPROACH

Typically, performance models are evaluated based on the accuracy or error. The error can be computed using[3]:

$$MMRE = \frac{|\ predicted - actual\ |}{actual} \cdot 100 \tag{2}$$

The key idea in this paper is to use ranking as an approach for building regression models. There are a number of advantages of using a rank-based approach:

- For the use cases listed in the introduction, *ranking is the ultimate goal*. A user may just want to identify the top-ranked configurations rather than to rank the whole space of configurations. For example, a practitioner trying to optimize an Apache Web server is searching for a set of configurations that can handle maximum load, and is not interested in the whole configuration space.
- *Ranking is extremely robust* since it is only mildly affected by errors or outliers [8, 17]. Even though measures such as Mean Absolute Error are robust, in the configuration setting, a practitioner is often more interested in knowing the rank rather than the predicted performance scores.

[3] Aside: There has been a lot of criticism regarding MMRE, which shows that MMRE along with other accuracy statistics such as MMRE, MBRE has been shown to cause conclusion instability [3, 13, 14].

```
# rank–based approach                                          1
def rank_based(training, testing, lives=3):                    2
  last_score = −1                                              3
  independent_vals = list()                                    4
  dependent_vals = list()                                      5
  for count in range(1, len(training)):                        6
    # Add one configuration to the training set                7
    independent_vals += training[count]                        8
    # Measure the performance value for the newly              9
    # added configuration                                      10
    dependent_vals += measure(training_set[count])             11
    # Build model                                              12
    model = build_model(independent_vals, ↩                    13
        dependent_vals)
    # Predicted performance values                             14
    predicted_performance = model(testing)                     15
    # Compare the ranks of the actual performance              16
    # scores to ranks of predicted performance scores          17
    actual_ranks = ranks(measure(testing))                     18
    predicted_ranks = ranks(predicted_performance)             19
    mean_RD = RD(actual_ranks, predicted_ranks)                20
    # If current rank difference is not better than            21
    # the previous rank difference, then loose life            22
    if mean_rank_difference <= last_rank_difference:           23
      lives −= 1                                               24
    last_rank_difference = mean_RD                             25
    # If all lives are lost, exit loop                         26
    if lives == 0: break                                       27
  return model                                                 28
```

Figure 5: Psuedocode of rank-based approach.

- *Ranking reduces the number of training samples required to train a model.* We will demonstrate that the number of training samples required to find the optimal configuration using a rank-based approach is reduced considerably, compared to residual-based approaches which use MMRE.

It is important to note that we aim at building a performance model similar to the accurate performance model building process used by prior work as described in Section 3. But instead of using residual measures of errors, as described in Equation 2, which depend on residuals ($r = y - f(x)$), [4] we use a rank-based measure. While training the performance model ($f(x)$), the configuration space is iteratively sampled (from the training pool) to train the performance model. Once the model is trained, the accuracy of the model is measured by sorting the values of $y = f(x)$ from 'small' to 'large', that is:

$$f(x_1) \leq f(x_2) \leq f(x_3) \leq \ldots \leq f(x_n). \tag{3}$$

The predicted rank order is then compared to the actual rank order. The accuracy is calculated using the mean rank difference:

$$accuracy = \frac{1}{n} \cdot \sum_{i=1}^{n} \Big| rank(y_i) - rank(f(x_i)) \Big| \tag{4}$$

This measure simply counts how many of the pairs in the test data were ordered incorrectly by the performance model $f(x)$ and measures the average of magnitude of the ranking difference.

In Figure 5, we list a generic algorithm for our rank-based approach. Sampling starts by selecting samples randomly from the

[4] Refer to Section 2 for definitions.

training pool and by adding them to the training set (Line 8). The collected sample configurations are then evaluated (Line 11). The configurations and the associated performance measure are used to build a performance model (Line 13). The generated model (CART, in our case) is used to predict the performance measure of the configurations in the testing pool (Line 16). Since the performance value of the testing pool is already measured, hence known, the ranks of the actual performance measures, and predicted performance measure are calculated. (Lines 18–19). The actual and predicted performance measure is then used to calculate the rank difference using Equation 4. If the rank difference of the model (with more data) does not decrease when compared to the previous generation (lesser data), then a life is lost (Lines 23–24). When all lives are expired, sampling terminates (Line 27).

The motivation behind using the parameter *lives* is: to detect convergence of the model building process. If adding more data does not improve the accuracy of the model (for example, in Figure 2 the accuracy of the model generated does not improve after 20 samples configuration), the sampling process should terminate to avoid resource wastage; see also Section 8.4.

5 SUBJECT SYSTEMS

To compare residual-based approaches with our rank-based approach, we evaluate it using 21 test cases collected in 9 open-source software systems[5].

(1) SS1 x264 is a video-encoding library that encodes video streams to H.264/MPEG-4 AVC format. We consider 16 features, which results in 1152 valid configurations.
(2) SS2 Berkeley DB (C) is an embedded key-value-based database library that provides scalable high performance database management services to applications. We consider 18 features resulting in 2560 valid configurations.
(3) SS3 SQLite is the most popular lightweight relational database management system. It is used by several browsers and operating systems as an embedded database. In our experiments, we consider 39 features that give rise to more than 3 million valid configurations.
(4) SS4 wget is a software package for retrieving files using HTTP, HTTPS, and FTP. It is a non-interactive command line tool. In our experiments, we consider 16 features, which result in 188 valid configurations.
(5) SS5 lrzip or Long Range ZIP is a compression program optimized for large files, consisting mainly of an extended rzip step for long distance redundant reduction and a normal compressor step. We consider 19 features, which results in 432 valid configurations.
(6) SS6 Dune or the Distributed and Unified Numerics Environment, is a modular C++ library for solving partial differential equations using grid-based methods. We consider 11 feature resulting in 2305 valid configurations.
(7) SS7 HSMGP or Highly Scalable MG Prototype is a prototype code for benchmarking Hierarchical Hybrid Grids data structures, algorithms, and concepts. It was designed to run on super computers. We consider 14 features resulting in 3456 valid configurations.
(8) SS8 Apache HTTP Server is a Web Server; we consider 9 features resulting in 192 valid configurations.

In addition to these 8 subject systems, we also consider Apache Storm, a distributed system, in several scenarios. The datasets were obtained from the paper by Jamshidi et al. [7]. The experiment considers three benchmarks namely:

- WordCount (wc) counts the number of occurences of the words in a text file.
- RollingSort (rs) implements a common pattern in real-time analysis that performs rolling counts of messages.
- SOL (sol) is a network intensive topology, where the message is routed through an inter-worker network.

The experiments were conducted with all the above mentioned benchmarks on 5 cloud clusters. The experiments also contain measurement variabilities, the wc experiments were also carried out on multi-tenant cluster, which were shared with other jobs. For example, wc+rs means wc was deployed in a multi-tenant cluster with rs running on the same cluster. As a result, not only latency increased but also variability became greater. The environments considered in our experiments are:

(9) SS9 wc-6d-throughput is an environment configuration where wc is executed by varying 6 features resulting in 2879 configurations; throughput is calculated.
(10) SS10 rs-6d-throughput is an environment configuration where rs is run by varying 6 features which results in 3839 configurations; the throughput is measured.
(11) SS11 wc-6d-latency is an environment configuration where wc is executed by varying 6 features resulting in 2879 configurations; latency is calculated.
(12) SS12 rs-6d-latency is an environment configuration where rs is executed by varying 6 features, which results in 3839 configurations; latency is measured.
(13) SS13 wc+rs-3d-throughput is an environment configuration where wc is run in a multi-tenant cluster along with rs. wc is executed by varying 3 features resulting in 195 configurations; throughput is measured.
(14) SS14 wc+sol-3d-throughput is an environment configuration where wc is run in a multi-tenant cluster along with sol. wc is executed by varying 3 features resulting in 195 configurations; throughput is measured.
(15) SS15 wc+wc-3d-throughput is an environment configuration where wc is run in a multi-tenant cluster along with wc. wc is executed by varying 3 features resulting in 195 configurations; throughput is measured.
(16) SS16 sol-6d-throughput is an environment configuration where sol is executed by varying 6 features resulting in 2865 configurations; throughput is measured.
(17) SS17 wc-wc-3d-throughput is an environment configuration where wc is executed by varying 3 features resulting in 755 configurations; throughput is calculated.
(18) SS18 wc+sol-3d-latency is an environment configuration where wc is run in a multi-tenant cluster along with sol. The wc is executed by varying 3 features resulting in 195 configurations; latency is measured.
(19) SS19 wc+wc-3d-latency is an environment configuration where wc is run in a multi-tenant cluster along with wc.

[5]For more details on the subject systems and configurations options refer to http://tiny.cc/3wpwly

The wc is executed by varying 3 features resulting in 195 configurations; latency is measured.

(20) **SS20** sol-6d-latency is an environment configuration where sol is executed by varying 6 features resulting in 2861 configuration setting; latency is measured.

(21) **SS21** wc+rs-3d-latency is an environment configuration where wc is run in a multi-tenant cluster along with rs. wc is executed by varying 3 features resulting in 195 configurations; latency is measured.

6 EVALUATION

6.1 Research Questions

In the past, configuration ranking required an accurate model of the configuration space, since an inaccurate model implicitly indicates that the model has missed the trends of the configuration space. Such accurate models require the evaluation/measurement of hundreds of configuration options for training [5, 18, 21]. There are also cases where building an accurate model is not possible, as shown in Figure 1 (right side).

Our research questions are geared towards finding optimal configurations when building an accurate model of a given software system is not possible. As our approach relies on ranking, our hypothesis is that we would be able to find the (near) optimal configuration using our rank-based approach while using fewer measurements, as compared to an accurate model learnt using residual-based approaches[6].

Our proposal is to embrace rank preservation but with inaccurate models and to use these models to guide configuration rankings. Therefore, to assess the feasibility and usefulness of the inaccurate model in configuration rankings, we consider the following:

- Accurate rankings found by inaccurate models using a rank-based approach, and
- the effort (number of measurements) required to build an inaccurate model.

The above considerations lead to two research questions:

RQ1: *Can inaccurate models accurately rank configurations?* Here, the optimal configurations found using an inaccurate model are compared to the more accurate models generated using residual-based approaches. The accuracy of the models is calculated using MMRE (from Equation 2).

RQ2: *How expensive is a rank-based approach (in terms of how many configurations must be executed)?* It is expensive to build accurate models, and our goal is to minimize the number of measurements. It is important to demonstrate that we can find optimal configurations of a system using inaccurate models as well as reducing the number of measurements.

6.2 Experimental Rig

For each subject system, we build a table of data, one row per valid configuration. We then run all configurations of all systems and record the performance scores (i.e., that are invoked by a benchmark). The exception is SQLite, for which we measure only 4400 configurations corresponding to feature-wise and pair-wise sampling and additionally 100 random configurations. (This is because SQLite has 3,932,160 possible configurations, which is an impractically large number of configurations to test.) To this table, we added a column showing the performance score obtained from the actual measurements for each configuration. Note that, while answering the research questions, we ensure that we never test any prediction model on the data that we used to learn the model. Next, we repeat the following procedure 20 times.

To answer the research questions, we split the datasets into training pool (40%), testing pool (20%), and validation pool (40%). The experiment is conducted in the following way:

- Randomize the order of rows in the training data
- **Do**
  - Select one configuration (by sampling with replacement) and add it to the training set
  - Determine the performance scores associated with the configuration. This corresponds to a table look-up, but would entail compiling or configuring and executing a system configuration in a practical setting.
  - Using the training set and the accuracy, build a performance model using CART.
  - Using the data from the testing pool, assess the accuracy either using MMRE (as described in Equation 2) or the rank difference (as described in Equation 4).
- **While** the accuracy is greater or equal to the threshold determined by the practitioner (rank difference in the case of our rank-based approach and MMRE in the case of the residual-based approaches).

Once the model has been iteratively trained, it is used on the data in the validation pool. Please note, the learner has not been trained on the validation pool. RQ1 relates the results found by the inaccurate performance models (rank-based) to more accurate models (residual-based). We use the absolute difference between the ranks of the configurations predicted to be the optimal configuration and the actual optimal configuration. We call this measure rank difference (*RD*).

$$RD = |rank(actual_{optimal}) - rank(predicted_{optimal})| \quad (5)$$

Ranks are calculated by sorting the configurations based on their performance scores. The configuration with the least performance score, $rank(actual_{optimal})$, is ranked 1 and the one with highest score is ranked as N, where N is the number of configurations.

7 RESULTS

7.1 RQ1: *Can inaccurate models accurately rank configurations?*

Figure 6 shows the *RD* of the predictions built using the rank-based approach and residual-based approaches [7] by learning from 40% of the training set and iteratively adding data to the training set (from the training pool), while testing against the testing set (20%). The model is then used to find the optimal configuration among the configurations in the validation dataset (40%). The horizontal axis

[6] Aside: It is worth keeping in mind that the approximation error in a model does not always harm. A model capable to smoothing the complex landscape of a problem can be beneficial for the search process. This sentiment has been echoed in the evolutionary algorithm literature as well [10].

[7] The MMRE scores for the models http://tiny.cc/bu14iy

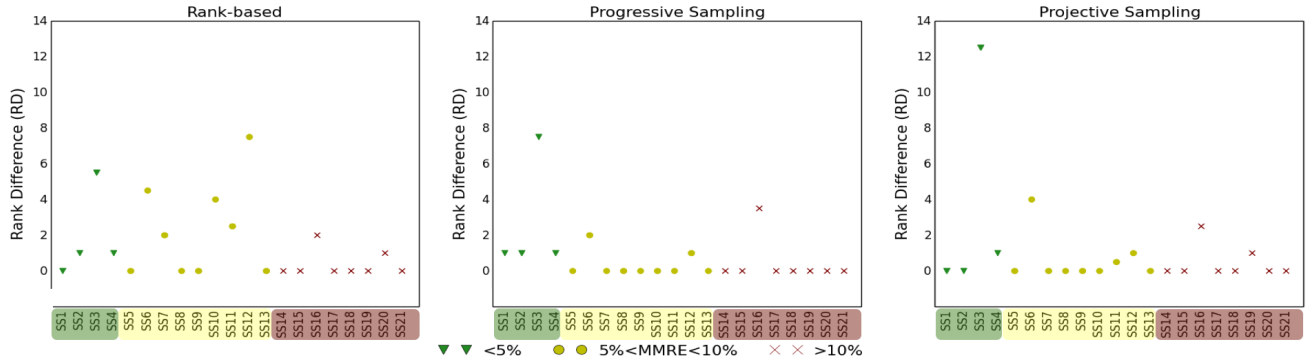

Figure 6: The rank difference of the prediction made by the model built using residual-based and rank-based approaches. Note that the y-axis of this chart rises to some very large values; e.g., SS3 has over three million possible configurations. Hence, the above charts could be summarised as follows: "the rank-based approach is surprisingly accurate since the rank difference is usually close to 0% of the total number of possible configurations". In this figure, ▼, ●, and ✕ represent the subject systems (using the technique mentioned at the top of the figure), in which we could build a prediction model, where accuracy is $< 5\%$, $5\% < MMRE < 10\%$, and $> 10\%$ respectively. This is based on Figure 1.

Rank	Treatment	Median	IQR	Median and IQR chart
SS1				
1	Projective	0.0	0.0	
1	Progressive	0.0	1.0	
2	Rank-based	2.0	8.0	
SS2				
1	Projective	0.0	1.0	
2	Rank-based	1.0	6.0	
2	Progressive	2.0	18.0	
SS3				
1	Progressive	10.0	17.0	
2	Projective	15.0	139.0	
2	Rank-based	21.0	40.0	
SS11				
1	Progressive	0.0	1.0	
2	Rank-based	1.0	3.0	
2	Projective	1.0	2.0	
SS20				
1	Projective	0.0	1.0	
1	Progressive	0.0	1.0	
2	Rank-based	5.0	19.0	

Figure 7: Median rank difference of 20 repeats. Median ranks is the rank difference as described in Equation 5, and IQR the difference between 75th percentile and 25th percentile found during multiple repeats. Lines with a dot in the middle (—●—), show the median as a round dot within the IQR. All the results are sorted by the median rank difference: a lower median value is better. The left-hand column (*Rank*) ranks the various techniques for example, when comparing various techniques for SS1, a rank-based approach has a different rank since their median rank difference is statistically different.

shows subject systems. The vertical axis shows the rank difference (*RD*) from Equation 5. In this figure:

- The perfect performance model would be able to find the optimal configuration. Hence, the ideal result of this figure would be if all the points lie on the $y = 0$ or the horizontal axis. That is, the model was able to find the optimal configuration for all the subject systems ($RD = 0$).
- The markers ▼, ●, and ✕ represent the software systems where a model with a certain accuracy can be built, measured in MMRE is $< 5\%$, $5\% <$MMRE$< 10\%$, and $> 10\%$ respectively.

Overall, in Figure 6, we find that:

- The ✕ represents software systems where the performance models are inaccurate ($> 10\%$ MMRE) and still can be used for ranking configurations, since the rank difference of these systems is always less than 4. Hence, even an inaccurate performance model can rank configurations.
- All three models built using both rank-based and residual-based approaches are able to find near optimal configurations. For example, progressive sampling for SQLite predicted the configuration whose performance score is ranked 9th in the testing set. This is good enough since progressive sampling is able to find the 9th most performant configuration among 1861 configurations[8].
- The mean rank difference of the $predicted_{optimal}$ is 1.4, 0.77, and 0.93[9] for the rank-based approach, progressive sampling, and projective sampling respectively. Thus, a performance model can be used to rank configurations.

We claim that the rank of the optimal configuration found by the residual and rank-based approaches is the same. To verify that the similarity is statistically significant, we further studied the results using non-parametric tests, which were used by Arcuri and Briand at ICSE'11 [12]. For testing statistical significance, we used a non-parametric bootstrap test with 95% confidence [2], followed by an A12 test to check that any observed differences were not trivially small effects; that is, given two lists X and Y, count how often there are larger numbers in the former list (and if there are ties, add a half mark): $a = \forall x \in X, y \in Y \frac{\#(x>y)+0.5\cdot\#(x=y)}{|X|\cdot|Y|}$ (as per Vargha [22], we say that a "small" effect has $a < 0.6$). Lastly, to generate succinct reports, we use the Scott-Knott test to recursively divide our approaches. This recursion used A12 and bootstrapping

[8] Since we test only on 40% of the possible configuration space (40% of 4653).
[9] The median rank difference is 0 for all the approaches.

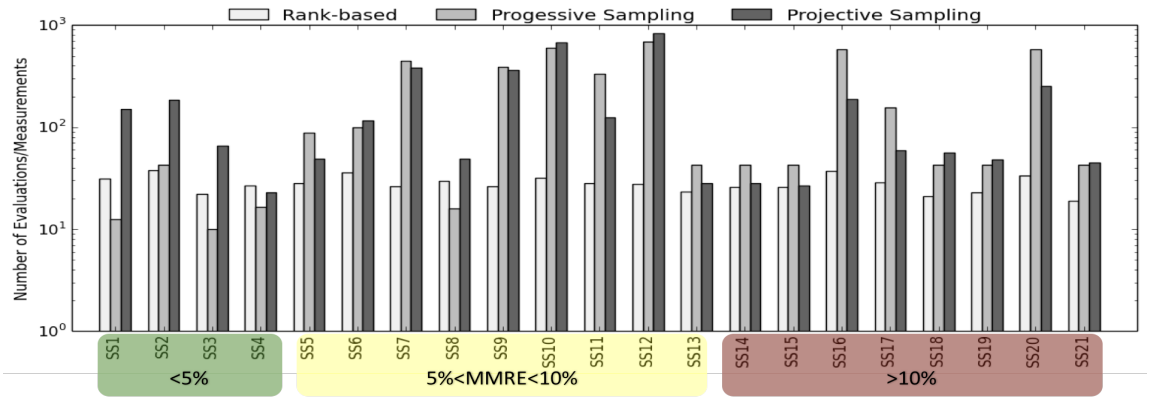

Figure 8: Number of measurements required to train models by different approaches. The software systems are ordered based on the accuracy scores of Figure 1.

to group together subsets that are (a) not significantly different and are (b) not just a small effect different to each other. This use of the Scott-Knott test is endorsed by Mittas and Angelis [12] and by Hassan et al. [4].

In Figure 7, the table shows the Scott-Knott ranks for the three approaches. The quartile charts are the Scott-Knott results for our subject systems, where the rank-based approach did not do as well as the residual-based approaches [10]. For example, the statistic test for SS1 shows that the ranks of the optimal configuration by the rank-based approach was statistically different from the ones found by the residual-based approaches. We think this is reasonably close since the median rank found by the rank-based approach is 2 out of 460 configurations, whereas residual-based approaches find the optimal configurations with a median rank of 0. As our motivation was to find optimal configurations for software systems for which performance models were difficult or infeasible to build, we look at SS20. If we look at the Skott-Knott chart for SS20, the median rank found by the rank-based approach is 5, whereas the residual-based approaches could find the optimal configurations very consistently (IQR=1). But as engineers, we feel that this is close because we are able to find the 5th best configuration using 33 measurements compared to 251 and 576 measurements used for progressive and projective sampling, respectively. Overall, our results indicate that:

> A rank preserving (probably inaccurate) model can be useful in finding (near) optimal configurations of a software system using a rank-based approach.

7.2 RQ2: *How expensive is a rank-based approach?*

To answer the question of whether we can recommend the rank-based approach as a cheap method for finding the optimal configuration, it is important to demonstrate that rank-based models are indeed cheap to build. In our setting, the cost of a model is determined by the number of measurements required to train the model. Figure 8 demonstrates this relationship. The vertical axis denotes the number of measurements in log scale and horizontal axis represents the subject systems.

In the systems SS1–SS4 (green band), the number of measurements required by the rank-based approach is less than for projective sampling and more than for progressive sampling. This is because the subject systems are easy to model. For the systems SS4–SS13 (yellow band), the number of measurements required to build models using the rank-based approach is less than residual-based approaches, with the exception of SS8. Note that, as building accurate models becomes difficult, the difference between the number of measurements required by the rank-based approach and residual-based approaches increases. For the systems SS14–SS21 (red band), the number of measurements required by the rank-based approach to build a model is always less than for residual-based approaches, with significant gains for SS19–SS21.

In Figure 9, the ratio of the measurements of different approaches are represented as the percentage of number of measurements required by projective sampling – since it uses the most measurements in 50% of the subject systems. For example, in SS5, the number of measurements used by progressive sampling is twice as much as used by projective sampling, whereas the rank-based approach uses half of the total number of measurements used by projective sampling. We observe that the number of measurements required by the rank-based approach is much lower than for the residual-based approaches, with the only exceptions of SS4 and SS8. We argue that such outliers are not of a big concern since the motivation of rank-based approach is to find optimal configurations for software systems, where an accurate model is infeasible.

To summarize, the number of samples required by the rank-based approach is much smaller than for residual-based approaches. There are $\frac{4}{21}$ cases where residual-based approaches (progressive sampling) use fewer measurements. The subject systems where residual-based approaches use fewer measurements are systems where accurate models are easier to build (green and yellow band):

> Models built using the rank-based approach require fewer measurements than residual-based approaches. In $\frac{8}{21}$ of the

[10] For complete Skott-Knott charts, refer to http://geekpic.net/pm-1GUTPZ.html.

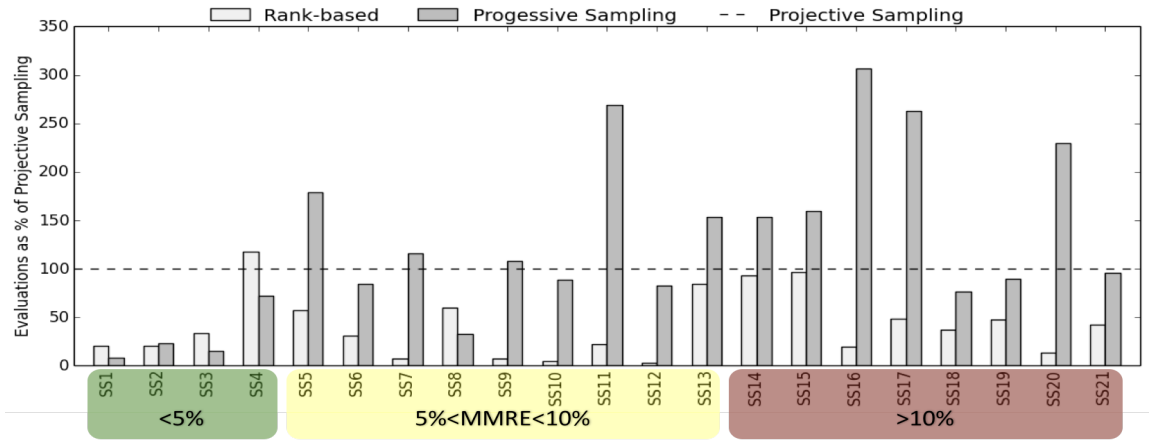

Figure 9: The percentage of measurement used for training models with respect to the number of measurements used by projective sampling (dashed line). The rank-based approach uses almost 10 times less measurements than the residual-based approaches. The subject systems are ordered based on the accuracy scores of Figure 1.

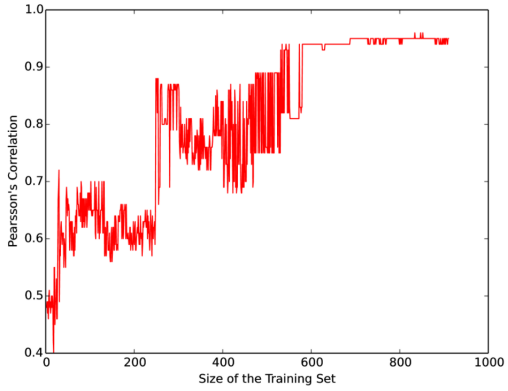

Figure 10: The correlation between actual and predicted performance values (not ranks) increases as the training set increases. This is evidence that the model does learn as training progresses.

cases, the number of measurements is an order of magnitude smaller than residual-based approaches.

8 DISCUSSION

8.1 How is rank difference useful in configuration optimization?

The objective of the modelling task considered here is to assist practitioners to find optimal configuration/s. We use a performance model, like traditional approaches, to solve this problem, but rather than using a residual-based accuracy measure, we use rank difference. Rank difference can also be thought as a correlation-based metric, since rank difference essentially tries to preserve the ordering of performance scores.

In our rank-based approach, we do not train the model to predict the dependent values of the testing set. But rather, we attempt to train the model to predict the dependent value that is correlated to the actual values of the testing set. So, during iterative sampling based on the rank-based approach, we should see an increase in the correlation coefficient as the training progresses. Figure 10 shows how the correlation between actual and the predicted values increases as the size of the training set grows. From the combination of Figure 1 and Figure 6, we see that even an inaccurate model can be used to find an optimal configuration.[11]

8.2 Can inaccurate models be built using residual-based approaches?

We have already shown that a rank preserving (probably inaccurate) model is sufficient to find the optimal configuration of a given system. MMRE can be used as a stopping criterion, but as we have seen with residual-based approaches, they require a larger training set and hence are not cost effective. This is because, with residual-based approaches, unlike our rank-based approach, it is not possible to know when to terminate sampling. It may be noted that rank difference can be easily replaced with a correlation-based approach such as Pearson's or Spearman's correlation.

8.3 Can we predict the complexity of a system to determine which approach to use?

From our results, we observe that a rank-based approach is not as effective as the residual-based approaches for software systems that can be modelled accurately (green band). Hence, it is important to distinguish between software systems where the rank-based approach is suitable and software systems where residual-based approaches are suitable. This is relatively straight-forward since both rank-based and residual-based approaches use random sampling to select the samples. The primary difference between the approaches is the termination criterion. The rank-based approach uses rank difference as the termination criterion whereas residual-based approaches use criterion based on MMRE, etc. Hence, it is possible to use both techniques simultaneously. If a practitioner observes

[11] This also shows how a correlation-based measure can be used as a stopping criterion.

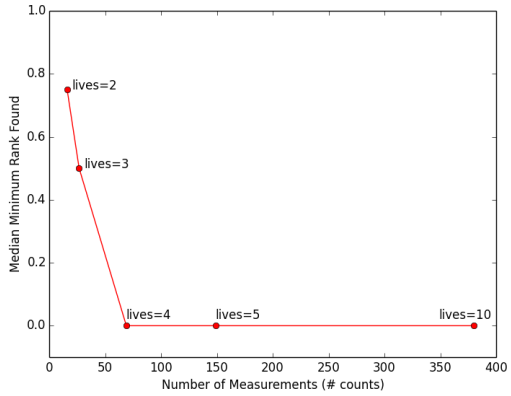

Figure 11: The trade-off between the number of measurements or size of the training set and the number of *lives*.

that the accuracy of the model during the building process is high (as in case of SS2), residual-based approaches would be preferred. Conversely, if the accuracy of the model is low (as in the case of , SS21), the rank-based approach would be preferred.

8.4 What is the trade-off between the number of lives and the number of measurements?

Our rank-based approach requires that the practitioner defines a termination criterion (*lives* in our setting) before the sampling process commences, which is a similar to progressive sampling. The termination criterion preempts the process of model building based on an accuracy measure. The rank-based approach uses rank difference as the termination criterion, whereas residual-based approaches use residual-based measures. In our experiments, the number of measurements or the size of the training set depends on the termination criterion (*lives*). An early termination of the sampling process would lead to a sub-optimal configuration, while late termination would result in resource wastage. Hence, it is important to discuss the trade-off between the number of *lives* and the number of measurements. In Figure 11, we show the trade-off between the median minimum ranks found and the number of measurements (size of training set). The markers of the figure are annotated with the values of *lives*. The trade-off characterizes the relationship between two conflicting objectives, for example, point (*lives*=2) requires very few measurements but the minimum rank found is the highest, whereas point (*lives*=10) requires a large number of measurements but is able to find the best performing configuration. Note, this curve is an aggregate of the trade-off curves of all the software systems discussed in this paper[12]. Since our objective is to minimize the number of measurements while reducing rank difference, we assign the value of 3 to *lives* for the purposes of our experiments.

9 RELIABILITY AND VALIDITY

Reliability refers to the consistency of the results obtained from the research. For example, how well can independent researchers reproduce the study? To increase external reliability, we took care to either clearly define our algorithms or use implementations from the public domain (scikit-learn) [16]. Also, all data and code used in this work are available on-line.[13]

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate [20]. *Internal validity* is concerned with whether the differences found in the treatments can be ascribed to the treatments under study.

For SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual values. We are aware that this evaluation leaves room for outliers and that measurement bias can cause false interpretations [11]. Since we limit our attention to predicting performance for a given workload, we did not vary benchmarks.

We aimed at increasing *external validity* by choosing subject systems from different domains with different configuration mechanisms. Furthermore, our subject systems are deployed and used in the real world.

10 CONCLUSION

Configurable systems are widely used in practice, but it requires careful tuning to adapt them to a particular setting. State-of-the-art approaches use a residual-based technique to guide the search for optimal configurations. The model-building process involves iterative sampling used along with a residual-based accuracy measure to determine the termination point. These approaches require too many measurements and hence are expensive. To overcome the requirement of a highly accurate model, we propose a rank-based approach, which requires a lesser number of measurements and finds the optimal configuration just by using the ranks of configurations as an evaluation criterion.

Our key findings are the following. First, a highly accurate model is not required for configuration optimization of a software system. We demonstrated how a rank-preserving (possibly even inaccurate) model can still be useful for ranking configurations, whereas a model with accuracy as low as 26% can be useful for configuration ranking. Second, we introduce a new rank-based approach that can be used to decide when to stop iterative sampling. We show how a rank-based approach is not trained to predict the raw performance score but rather learns the model, so that the predicted values are correlated to actual performance scores.

To compare the rank-based approach to the state-of-the-art residual-based approaches (projective and progressive sampling), we conducted a number of experiments on 9 real-world configurable systems to demonstrate the effectiveness of our approach. We observed that the rank-based approach is effective to find the optimal configurations for most subject systems while using fewer measurements than residual-based approaches. The only exceptions are subject systems for which building an accurate model is easy, anyway.

ACKNOWLEDGEMENT

The work is partially funded by an NSF award #1302169. Siegmund's work is supported by the DFG under the contract SI 2171/2. Apel's work is supported by the DFG under the contract AP 206/6 and AP 206/7.

[12] Complete trade-off curves can be found at http://tiny.cc/kgs2iy or http://tiny.cc/rank-param.

[13] https://github.com/ai-se/Reimplement/tree/cleaned_version

REFERENCES

[1] J. Chen, V. Nair, R. Krishna, and T. Menzies. 2016. Is Sampling better than Evolution for Search-based Software Engineering? *arXiv* (2016).

[2] B. Efron and R. J. Tibshirani. 1993. *An Introduction to the Bootstrap.* CRC.

[3] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. 2003. A Simulation Study of the Model Evaluation Criterion MMRE. *IEEE Transactions on Software Engineering (TSE)* 29 (2003), 985–995.

[4] B. Ghotra, S. McIntosh, and A. E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *Proc. of International Conference on Software Engineering (ICSE)*. IEEE, 789–800.

[5] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. of International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[6] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proc. of International Conference on Software Engineering (ICSE)*. IEEE, 517–528.

[7] P. Jamshidi and G. Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *Proc. of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 39–48.

[8] J.D. Kloke and J. W. McKean. 2012. Rfit: Rank-based Estimation for Linear Models. *The R Journal* 4 (2012), 57–64.

[9] J. Krall, T. Menzies, and M. Davies. 2015. GALE: Geometric Active Learning for Search-Based Software Engineering. *IEEE Transactions on Software Engineering (TSE)* 41 (2015), 1001–1018.

[10] D. Lim, Y. Jin, Y. S. Ong, and B. Sendhoff. 2010. Generalizing Surrogate-Assisted Evolutionary Computation. *IEEE Transactions on Evolutionary Computation* 14 (2010), 329–355.

[11] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. 2013. Local versus Global Lessons for Defect Prediction and Effort Estimation. *IEEE Transactions on Software Engineering (TSE)* 39 (2013), 822–834.

[12] N. Mittas and L. Angelis. 2013. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Transactions on Software Engineering (TSE)* 39 (2013), 537–551.

[13] I. Myrtveit and E. Stensrud. 2012. Validity and Reliability of Evaluation Procedures in Comparative Studies of Effort Prediction Models. *Empirical Software Engineering (ESE)* 17 (2012), 23–33.

[14] I. Myrtveit, E. Stensrud, and M. Shepperd. 2005. Reliability and Validity in Comparative Studies of Software Prediction Models. *IEEE Transactions on Software Engineering (TSE)* 31 (2005), 380–391.

[15] V. Nair, T. Menzies, N. Siegmund, and S. Apel. 2017. Faster Discovery of Faster System Configurations with Spectral Learning. *arXiv* (2017).

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and others. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[17] S. Rosset, C. Perlich, and B. Zadrozny. 2005. Ranking-based Evaluation of Regression Models. In *Proc. of International Conference on Data Mining (ICDM)*. IEEE.

[18] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *Proc. of International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[19] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. of International Conference on Automated Software Engineering (ASE)*. IEEE, 465–474.

[20] J. Siegmund, N. Siegmund, and S. Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proc. of International Conference on Software Engineering (ICSE)*. IEEE, 9–19.

[21] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. of International Conference on Software Engineering (ICSE)*. IEEE, 167–177.

[22] A. Vargha and H. D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25 (2000), 101–132.

[23] G. M. Weiss and Y. Tian. 2008. Maximizing Classifier Utility when there are Data Acquisition and Modeling Costs. *Journal of Data Mining and Knowledge Discovery* (2008), 253–282.

[24] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. 2015. Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software. In *Proc. of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, ACM, 307–319.