Frugal: Cheaper methods for Search Based Software Engineering

Vivek Nair
North Carolina State University, USA
vivekaxl@gmail.com

ABSTRACT

Prior work on predicting the performance of software configurations suffered from either (a) requiring far too many sample configurations or (b) large variances in their predictions. Both these problems can be avoided using the WHAT spectral learner. WHAT's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors — less than 10 % prediction error, with a standard deviation of less than 2 %. When compared to the state of the art, our approach (a) requires 2 to 10 times fewer samples to achieve similar error rates and (b) its predictions are more stable (i.e., have lower standard deviation). Furthermore, predictive models generated by WHAT can be used by optimizers to discover system configurations that closely approach the optimal performance, as we will demonstrate.

Categories/Subject Descriptors: D.2 [**Software Engineering**]; I.2 [**Artificial Intelligence**];

Keywords: Performance Prediction, Spectral Learning, Decision Trees, Search-Based Software Engineering, Sampling.

1. INTRODUCTION

Search based software engineering (SBSE) involves utilizing the rich literature of search based optimization to solve software engineering problems. Using SBSE, a software engineering task can be solved by finding solutions along with an associated fitness function. The automatic evaluation of the solutions open up whole range of possibilities with Evolutionary algorithms(EA) (which we would talk in great depth later in the paper). In the last few years, EA has been used widely to solve problems in software engineering like requirement selection [1] and resource allocation in project scheduling [2] to name a few. While using EA is very useful and easy there is a caveat. We mentioned before the EA uses a fitness function and a normal run of an EAs takes about tens of thousands of evaluations. If the cost/time involved in evaluating the fitness function is high, it would hinder the acceptance of EAs as a defacto solution to real-world problems. In this work (and in future), we work predominantly on exploring ways to use the search techniques to solve the hard problems with a 'budget'in mind. The problem we choose to explore is the problem of choosing the optimal configurations for a software.

DOI: XX.XXX/XXX+X

1.1 Description of our problem

Most software systems today are configurable. Despite the undeniable benefits of configurability, large configuration spaces challenge developers, maintainers, and users. In the face of hundreds of configuration options, it is difficult to keep track of the effects of individual configuration options and their mutual interactions. So, predicting the performance of individual system configurations or determining the optimal configuration is often more guess work than engineering.

Addressing the challenge of performance prediction and optimization in the face of large configuration spaces, researchers have developed a number of approaches that rely on sampling and machine learning [3–5]. While gaining some ground, state-of-the-art approaches face two problems: (a) they require far too many sample configurations for learning or (b) they are prone to large variances in their predictions. For example, prior work on predicting performance scores using regression-trees had to compile and execute hundreds to thousands of specific system configurations [4]. A more balanced approach by Siegmund et al. is able to learn predictors for configurable systems [3] with low mean errors, but with large variances of prediction accuracy (e.g. in half of the results, the performance predictions for the Apache Web server were up to 50 % wrong).

Guo et al. [4] also proposed an incremental method to build a predictor model, which uses incremental random samples with steps equal to the number of configuration options (features) of the system. This approach also suffered from unstable predictions (e.g., predictions had a mean error of up to 22 %, with a standard deviation of up 46 %). Finally, Sarkar et al. [5] proposed a projective-learning approach (using fewer measurements than Guo at al. and Siegmund et al.) to quickly compute the number of sample configurations for learning a stable predictor. However, as we will discuss, after making that prediction, the total number of samples required for learning the predictor is comparatively high (up to hundreds of samples).

The problems of large sample sets and large variances in prediction can be avoided using the **WHAT** spectral learner, which is our main contribution. **WHAT** 's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated config-

urations can be studied by measuring only a few samples. In a number of experiments, we compared **WHAT** against the state-of-the-art approaches of Siegmund et al. [3], Guo et al. [4], and Sarkar et al. [5] by means of six real-world configurable systems: Berkeley DB, the Apache Web server, SQLite, the LLVM compiler, and the x264 video encoder. We found that **WHAT** performs as well or better than prior approaches, while requiring far fewer samples (just a few dozen). This is significant and most surprising, since some of the systems explored here have up to millions of possible configurations.

1.2 Research Questions

We formulate our research questions in terms of the challenges of exploring large complex configuration spaces. Since our model explores the spectral space, our hypothesis is that only a small number of samples is required to explore the whole space. However, a prediction model built from a very small sample of the configuration space might be very inaccurate and unstable, that is, it may exhibit very large mean prediction errors and variances on the prediction error.

Also, if we learn models from small regions of the training data, it is possible that a learner will miss *trends* in the data between the sample points. Such trends are useful when building *optimizers* (i.e., systems that input one configuration and propose an alternate configuration that has, for instance, a better performance). Such optimizers might need to evaluate hundreds to millions of alternate configurations. To speed up that process, optimizers can use a *surrogate model* [1] that mimics the outputs of a system of interest, while being computationally cheap(er) to evaluate [6]. For example, when optimizing performance scores, we might ask a CART for a performance prediction (rather than compile and execute the corresponding configuration). Note that such surrogate-based reasoning critically depends on how well the surrogate can guide optimization.

Therefore, to assess feasibility of our sampling policies, we must consider:

- Performance scores generated from our minimal sampling policy;
- The variance of the error rates when comparing predicted performance scores with actual ones;
- The optimization support offered by the performance predictor (i.e., can the model work in tandem with other off-the-shelf optimizers to generate useful solutions).

The above considerations lead to four research questions:

RQ1: *Can* **WHAT** *generate good predictions after executing only a small number of configurations?*

Here, by "good" we mean that the predictions made by models that were trained using sampling with **WHAT** are as accurate, or more accurate, as those generated from models supplied with more samples.

RQ2: *Does less data used in building the models cause larger variances in the predicted values?*

RQ3: *Can "good" surrogate models (to be used in optimizers) be built from minimal samples?*

Note that **RQ2** and **RQ3** are of particular concern with our approach, since our goal is to sample as little as possible from the configuration space.

[1] Also known as response surface methods, meta models, or emulators.

RQ4: *How good is* **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

To answer RQ4, we will compare **WHAT** against approaches presented by Siegmund et al. [3], Guo et al. [4], and Sarkar et al. [5].

Berkeley DB C Edition (BDBC) is an embedded database system written in C. It is one of the most deployed databases in the world, due to its low binary footprint and its configuration abilities. We used the benchmark provided by the vendor to measure response time.
Berkeley DB Java Edition (BDBJ) is a complete re-development in Java with full SQL support. Again, we used a benchmark provided by the vendor measuring response time.
Apache is a prominent open-source Web server that comes with various configuration options. To measure performance, we used the tools autobench and httperf to generate load on the Web server. We increased the load until the server could not handle any further requests and marked the maximum load as the performance value.
SQLite is an embedded database system deployed over several millions of devices. It supports a vast number of configuration options in terms of compiler flags. As benchmark, we used the benchmark provided by the vendor and measured the response time.
LLVM is a compiler infrastructure written in C++. It provides various configuration options to tailor the compilation process. As benchmark, we measured the time to compile LLVM's test suite.
x264 is a video encoder in C that provides configuration options to adjust output quality of encoded video files. As benchmark, we encoded the Sintel trailer (735 MB) from AVI to the xH.264 codec and measured encoding time.

System	LOC	Features	Configurations
BDBC	219,811	18	2,560
BDBJ	42,596	32	400
Apache	230,277	9	192
SQLite	312,625	39	3,932,160
LLVM	47,549	11	1,024
x264	45,743	16	1,152

Figure 1: Subject systems used in the experiments.

1.3 Contributions of this thesis

We take the first step in introducing Surrogate modelling into Software Engineering literature by presenting a efficient sampling method, which can choose only the informative samples given all possible samples. Overall we make the following contributions:

- We present a novel sampling and learning approach for predicting the performance of software configurations in the face of large configuration spaces. The approach is based on a *spectral learner* that uses an approximation to the first principal component of the configuration space to recursively cluster it, relying only on a few points as representatives of each cluster.
- We demonstrate the practicality and generality of our approach by conducting experiments on six real-world configurable software systems (see Figure 1). The results show that our approach is more accurate (lower mean error) and more stable (lower standard deviation) than state-of-the-art approaches.
- We report on a comparative analysis of our approach and three state-of-the-art approaches, demonstrating that our approach outperforms previous approaches in terms of sample size and prediction stability. A key finding is the utility of the principal component of a configuration space to find informative samples from a large configuration space.

1.4 Statement of Thesis

Top-Down Recusive binary division of the decision space based on the approximated first principal component can be utilized to find candidates to (i) build efficient, (ii) accurate, and (iii) highly stable surrogate models that can be used by an off-the-shelf optimizer to find configurations with lower runtimes.

1.5 Publications

- Vivek Nair, Tim Menzies, Norbert Siegmund, Sven Apel; Faster Discovery of Faster System Configurations with Spectral Learning; Submitted to Foundations of Software Engineering 2016

2. MOTIVATION

In this section, we try to answer questions on why we make the following choices in our work and why we think it is important.

Why is the problem of choosing correct configurations so important?

In cloud computing, applications are provided on-demand as a self-service [7]. Commonly, a software as a service (SaaS) application is hosted by a provider in the cloud, rented to multiple customers (called tenants) and in turn by the tenants'users over the Internet. In the provisioning of a SaaS application, various stakeholders with different objectives are involved, i.e., providers of all cloud stack layers as well as tenants and their users [18, 26]. Thus, providing highly configurable SaaS applications for a large number of tenants and their associated users in a shared cloud environment demands for a dynamic, yet scalable configuration management to support multiple stakeholders.

Tenancy contracts define the provisioned application functionality as well as quality of service (QoS) guarantees. To add another layer of complexity, some configuration steps, e.g., performed by tenants, are independent from each other while others are dependent, e.g., a tenant's configuration choices depend on the preconfiguration of the provider. Therefore, a structured configuration process is needed.

In addition, stakeholders'objectives may change over time, e.g., if a tenant decides to change the tenancy contract. Thus, the configuration process needs to support reconfiguration of stakeholder preconfigurations and subsequent ones from being further affected. In contrast to conventional methods, multiple tenancy contracts and user variants are derived, which are then integrated into a single application instance in the solution space. Thus, in a shared cloud environment, variants are independent in the problem space, but become dependent in the solution space.

In recent years, we have witnessed that a number of large Internet and cloud-service providers (eg. Google, Microsoft, Amazon, LinkedIn) experienced misconfiguration-induced outages, affecting millions of their customers [**?**, **?**, 9–11]. Typically, a large, complex software system exposes hundreds of configuration parameters, and each parameter has its setting constraints (i.e., correctness rules or requirements) [12, 13]; many configuration constraints are neither intuitively designed nor well documented [14, 15]. Moreover, many configuration parameters have correlations and dependencies with other parameters or the system's runtime environment, which further increases the complexity. In their recent paper, Xu et al. documented the difficulties developers face with understanding the configuration spaces of their systems [8]. As a result, developers tend to ignore over 5/6ths of the configuration options, which leaves considerable optimization potential untapped and induces major economic cost [8]. Figure 2 depicts how the number of configuration parameters increases as projects mature.

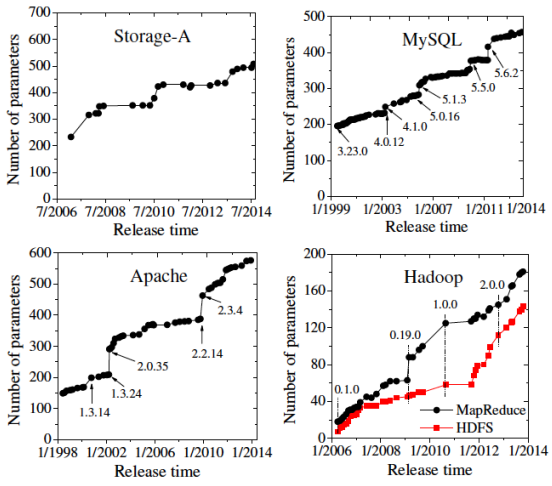

Figure 2: The increasing number of configuration parameters with software evolution. Storage-A is a commercial storage system from a major storage company in the U.S. (from [8])

.

As a service provider, their customers come with different software and specific performance requirements. The general tendency of the service providers is to be conservative and over-provision the system to provide a particular performance which results in resource wastage. Though there has been numerous attempts by the system software groups around the world to come up with dynamic resource scaling [2], administrators (skeptics) would still like to start the system with the optimal configurations. This means hours spent in running various benchmarks to build an accurate performance model. Various runs of the benchmarks translate into wasted time and resources and this task is expatiated if the configurations have to be optimized for multiple performance measure such as energy, infrastructure cost etc. [17]. Since there is no concrete mathematical model available for such problems, it renders all the numerical methods useless.

> The growing popularity of SaaS has resulted in demand of highly configurable software. The task of selecting optimal configuration to satisfy QoS guarantees (sometimes multiple) has become more challenging. Manually setting configurations is potentially dangerous since big systems generally have large number of configurations.

Why use Evolutionary Algorithm?

Performing a exhaustive search of the possible options is not feasible, so researchers over the years have come up with various strategies like stochastic search, evolutionary search etc.

Stochastic and evolutionary algorithms have been primarily used for problems which are :

- Multi-Modality: A problem is called multi-modal if there are more than one optimum solution
- High Dimensionality: The number of points required for sample a search space grows exponentially [3]

[2] detailed survey in [16]

[3] This can be illustrated on a simple example: first, place 100 points in a real interval

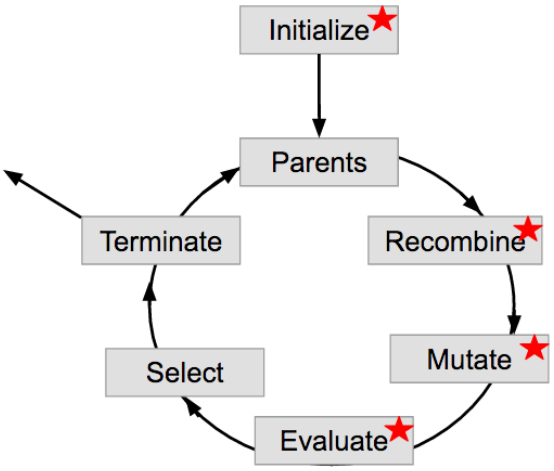

Figure 3: . A diagram for an evolutionary algorithm. A star denotes an evolutionary operation where a surrogate can be helpful.

.

- Non-Separability: A problem is called non-separable if the optimum of one of the objectives is not the optimum for the other objective/s.
- Expensive: A problem is called expensive if the cost of evaluating a problem (or a function) is more costly with respect to any measures e.g. time or money with respect to the cost of optimization.

In this report we would be biased towards evolutionary algorithm because of the general nature of the algorithm and effectiveness in many real world problems. Figure 3 describes the basic steps of an evolutionary algorithm. Evolutionary algorithm starts of with randomly generated points across the search space called Initial Population. The population is increased by using various 'reproduction 'operators like recombination (also known as crossover) and mutation. Once the mutants are generated, they are evaluated using itsassociated fitness function. A selection operator is applied to the population (parent population and mutation) and the best (fittest) candidates are selected. This process continues till a termination criteria is reached also called stopping criteria. Evolutionary Algorithms (EAs) have been thoroughly investigated in this context due to their ability to solve complex optimization problems by coupling problem-specific variation operators and selection operators

Firstly, a search directed by a population of candidate solutions is quite robust with respect to a moderate noise and multi-modality of the objective function, in contrast to some classical optimization methods such as quasi-Newton methods.

Secondly, the role of variation operators is to explore the search space of potential solutions, taking into account already retrieved information about the problem. Usually using randomized variation operators, together with the representation of the candidate solutions, encapsulate extensive prior knowledge about the problem domain.

Finally, selection is responsible for directing the search toward more promising regions on the basis of the current solutions, controlling the exploration versus exploitation trade-off. The lessons learned from multi-objective optimization are that, in order to build a dense Pareto front, (i) one must compulsorily preserve the population diversity, and (ii) avoid discarding too easily the solutions which are dominated w.r.t. the current objectives. Despite their shortcomings, some solutions are found to pave the way toward truly non-dominated solutions in unfrequented regions of the Pareto front. In other words, the celebrated Exploitation versus Exploration trade-off should be considered at the level of algorithm design too.

The real-world problems are very difficult to solve since they don't have the properties assumed by the numerical methods [a]. EAs are very handy to solve these problems because it organically finds solutions to problem and does that faster than deterministic algorithms.

[a](i) Concave vs Convex, (ii) Differentiability, (iii) Linear vs Non Linear etc

How can we run EA on budget?

We have seen how EAs can be very useful in solving problems with relative ease and bare minimum knowledge about the domain. But EAs have been plagued with the problems of slow convergence and requiring a lot of function evaluations. When function evaluations are 'expensive'as explained earlier it is infeasible to run thousands of function evaluations to get the desired results. There are various ways used to solve this problem:

- Active learning based techniques: GALE, an optimizer that identifies and evaluates just those most informative examples. GALE does not use clustering as a post-processor to some other optimizer. Rather, GALE replaces the need for a post-processor with a tool called WHERE, which performs only two evaluations per recursive split of the data. Hence, this algorithm performs at most $2log2N$ evaluations per generation, and often less. The figure 5 shows how GALE can perform as good as, if not better, other algorithms like SPEA-2 and NSGA-II.
- Surrogate based techniques: Using approximation models also known as surrogates or metamodels, the computational burden can be greatly reduced since the efforts required to build the surrogates and to use them are much lower than those in the standard approach. Figure 3 describes the steps where surrogate models can be used to solve problem (for more information refer to [20]).

There has been numerous work on applying evolutionary techniques on the problems in software engineering but it all faces problems of a long runtimes which isn't appreciated by the industry. Zuluaga et al. [21] comment on the cost of such an analysis for software/hardware co-design: "synthesis of only one design can take hours or even days ". Harman [22] comments on the problems of evolving a test suite for software if every candidate solution requires a time-consuming execution of the entire system: such test suite generation can take weeks of execution time. Though these concerns have been raised by the community, to the best of our knowledge, there is no considerable research on reducing the time required to reduce the time required for multiobjective optimization. The only known approach was introduced by Krall et. al. [19] who proposed a novel approach called GALE (Geometric Active Learner), which used recursive binary splits and a pruning mechanism to find the most promising regions of the search space. Their experiments shows that GALE, on a expensive model (CDA-An Aviation Safety Model), can get the similar results in a few hours compared to NSGA-II which took 7 days to complete. Unlike, Krall et al., we would like to explore the surrogate modelling aspect of solving a problem. We have a definite bias towards the

[0, 1]; then to have a similar coverage in terms of distance between points in 10-dimensional space requires $100^{10} = 10^{20}$ points.

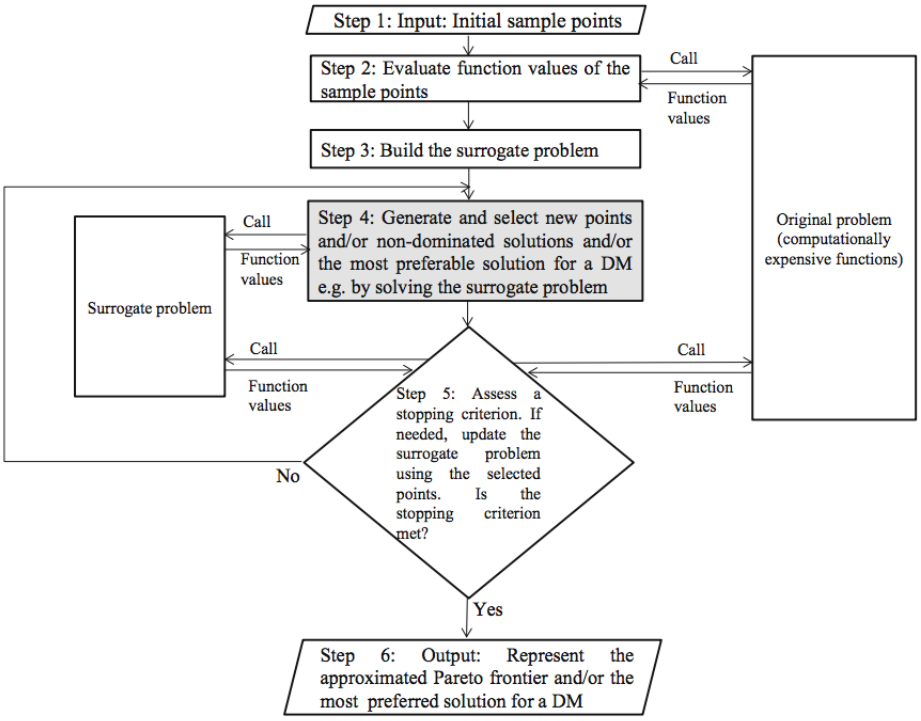

Figure 4: Flowchart of Surrogate Modelled Evolutionary Algorithm (from [18])

.

surrogate literate because of the following reasons:

- It is a harmonious marriage between machine learning and evolutionary algorithm. The machine literature has been rigorously proven and have been known to work for a pleothra of applications
- Classic evolutionary algorithms chooses the candidates for the next generation using problem specific operators (for eg. specific crossover or mutation strategies [23]). In this process it looses all the knowledge learnt in the process. But, in contrast to classical methods Surrogate assisted Evolutionary Algorithms (SAEA) has memory and can be used to learn during the process of evolution

The basic idea in a surrogate based method is to replace a computationally expensive problem with a computationally less expensive surrogate problem. The work flow of surrogate model has been be depicted in the figure 4. The first step involved initial points that are sampled, and then evaluated with the computationally expensive functions in step 2. After this, function values of the sampling points are available. In Step 3, the initial surrogate problem is constructed. In order to capture the Pareto frontier or to provide the most preferred solution based on the preference of a user (DM-Decision Maker), new samples are generated in Step 4. These points can be obtained by solving the surrogate problem and/or sampling unexplored regions in the decision and/or objective space depending on a sampling process. In accordance with a method-dependent criterion, a subset of points among the generated points is selected. In Step 5, accessing a stop criterion may required to evaluated the selected sample points with computationally expensive functions and/or to update the surrogate problem. If a stopping criterion is met, the set of non-dominated solutions or the most preferred solution of the last surrogate problem which maybe have been evaluated with the computationally expensive functions are considered as the approximated Pareto Frontier of the original problem or the most preferred solution for a DM in Step 6. When the stopping criterion is not met, if the surrogate problem has already been updated, Step 4-5 are repeated. If not, first the surrogate problem is updated with the evaluated sample points and then Step4-5 are repeated.

Expensive evaluation of fitness functions limits the application of EAs wrt. to the real-world problems. Surrogate models are the cheaper, low fidelity replacement of the expensive fitness functions which can work around 'curse of dimensionality 'and also use 'blessing of uncertainty '.

3. LITERATURE REVIEW

Work presented in the paper is an adaptation of the machine learning techniques along with existing works of surrogate modelling (primarily from engineering design literature) to software engineering. So we divide our literature survey into two main parts of (i) Sampling software configuration data (Software Engineering) and (ii) Surrogate Modelling.

3.1 Sampling in Software Configuration Data

A configurable software system has a set X of Boolean configuration options,[4] also referred to as features or independent variables in our setting. We denote the number of features of system S as n. The configuration space of S can be represented by a Boolean space $\mathbb{Z}_2^n$, which is denoted by F. All valid configurations of S belong to a set V, which is represented by vectors $\vec{C}_i$ (with $1 \leq i \leq |V|$) in $\mathbb{Z}_2^n$.

[4]In this paper, we concentrate on Boolean options, as they make up the majority of all options; see Siegmund et al., for how to incorporate numeric options [24].

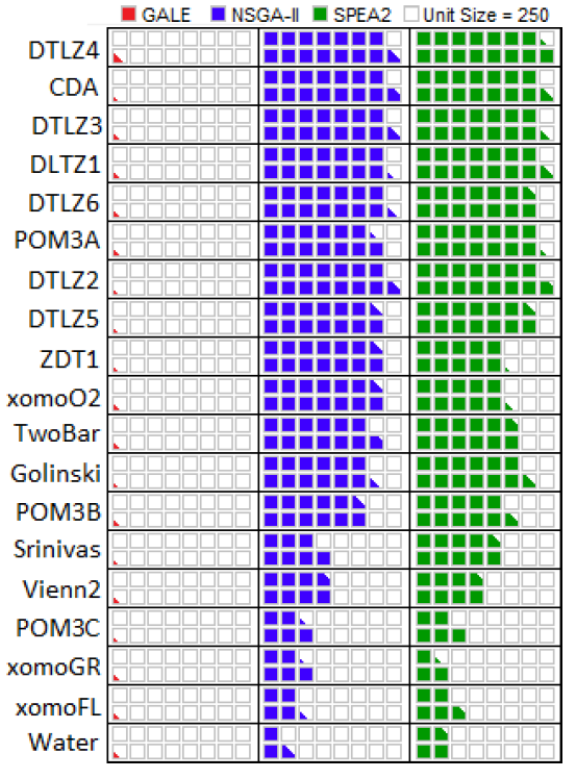

Figure 5: The figure shows how a surrogate method like GALE can achieve almost the same performance as the other (then) state-of-the-art algorithms like NSGA-2 and SPEA-2. Number of evaluations in units of 250. (from [19])

.

Each element of a configuration represents a feature, which can either be *True* or *False*, based on whether the feature is selected or not. Each valid instance of a vector (i.e., a configuration) has a corresponding performance score associated to it.

The literature offers two approaches to performance prediction of software configurations: a *maximal sampling* and a *minimal sampling* approach: With *maximal sampling*, we compile all possible configurations and record the associated performance scores. Maximal sampling can be impractically slow. For example, the performance data used in this paper required 26 days 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 [25].

If collecting performance scores of all configurations is impractical, *minimal sampling* can be used to intelligently select and execute just enough configurations (i.e., samples) to build a predictive model. For example, Zhang et al. [26] approximate the configuration space as a Fourier series, after which they can derive an expression showing how many configurations must be studied to build predictive models with a given error. While a theoretically satisfying result, that approach still needs thousands to hundreds of thousands of executions of sample configurations.

Another set of approaches are the four "additive" *minimal sampling* methods of Siegmund et al. [3]. Their first method, called feature-wise sampling (*FW*), is their basic method. To explain *FW*, we note that, from a configurable software system, it is theoretically possible to enumerate many or all of the valid configurations[5]. Since each configuration ($\vec{C}_i$) is a vector of n Booleans, it is possible to use this information to isolate examples of how much each feature individually contributes to the total run time:

1. Find a pair of configurations $\vec{C}_i$ and $\vec{C}_2$, where $\vec{C}_2$ uses exactly the same features as $\vec{C}_i$, plus one extra feature f_i.

2. Set the run time $\Pi(f_i)$ for feature f_i to be the difference in the performance scores between $\vec{C}_2$ and $\vec{C}_i$.

3. The run time for a new configuration $\vec{C}_i$ (with $1 \leq i \leq |V|$) that has not been sampled before is then the sum of the run time of its features, as determined before:

$$\Pi(C_i) = \sum_{f_j \in C_i} \Pi(f_j) \quad (1)$$

When many pairs, such as $\vec{C}_1, \vec{C}_2$, satisfy the criteria of point 1, Siegmund et al. used the pair that covers the *smallest* number of features. Their minimal sampling method, *FW*, compiles and executes only these smallest C_1 and C_2 configurations. Siegmund et al. also offers three extensions to the basic method, which are based on sampling not just the smallest $\vec{C}_i$,$\vec{C}_2$ pairs, but also any configurations with *interactions* between features. All the following minimal sampling policies compile and execute valid configurations selected via one of three heuristics:

PW (pair-wise): For each pair of features, try to find a configuration that contains the pair and has a minimal number of features selected.

HO (higher-order): Select extra configurations, in which three features, f_1, f_2, f_3, are selected if two of the following pair-wise interactions exist: (f_1, f_2) and (f_2, f_3) and (f_1, f_3).

HS (hot-spot): Select extra configurations that contain features that are frequently interacting with other features.

Guo et al. [4] proposed a progressive random sampling approach, which samples in steps of the number of features of the software system in question. They used the sampled configurations to train a regression tree, which is then used to predict the performance scores of other system configurations. The termination criterion of this approach is based on a heuristic, similar to the *PW* heuristics of Siegmund et al.

Sarkar et al. [5] proposed a cost model for predicting the effort (or cost) required to generate an accurate predictive model. The user can use this model to decide whether to go ahead and build the predictive model. This method randomly samples configurations and uses a heuristic based on feature frequencies as termination criterion. The samples are then used to train a regression tree; the accuracy of the model is measured by using a test set (where the size of the training set is equal to size of the test set). One of four projective functions (e.g., exponential) is selected based on how correlated they are to accuracy measures. The projective function is used to approximate the accuracy-measure curve, and the elbow point of the curve is then used as the optimal sample size. Once the optimal size is known, Sarkar et al. uses the approach of Guo et al. to build the actual prediction model.

The advantage of these previous approaches is that, unlike the results of Zhang et al., they require only dozens to hundreds of samples. Also, like our approach, they do not require to enumerate all

[5]Though, in practice, this can be very difficult. For example, in models like the Linux Kernel such an enumeration is practically impossible [27].

configurations, which is important for highly configurable software systems. That said, as shown by our experiments (see Section 5), these approaches produce estimates with larger mean errors and partially larger variances than our approach. While sometimes the approach by Sarkar et al. results in models with (slightly) lower mean error rates, it still requires a considerably larger number of samples (up to hundreds, while **WHAT** requires only few dozen).

3.2 Surrogate Modelling

We divide the background literature related to Surrogate Models into two logical sections namely (i) Components of a Surrogate Models, (ii) Classifications in Surrogate Models.

3.2.1 Componets of a Surrogate Model

Surrogate Models can be build primarily using a combination of two techniques: Machine Learning techniques which uses techniques like polynomial functions [28], Kriging models [29], Radial Basis Functions (RBFs) [30], Multivariate Adaptive Regression Splines (MARS) [31], Neural Networks [32] and Support Vector Regression (SVR) [33]. To learn from the given data points the learners need samples which is provided by a host of sampling algorithms. Sampling techniques primarily used in the literature are:

- Latin Hypercube Sampling (LHS) [34]: Latin hypercube sampling operates in the following manner:
  - Ranges of all the variables (dependent) are divided into n samples
  - For each sample, each variable is selected at random without replacement
  - This is performed till all the features are not selected.
- Full Factorial Sampling (FFS) [35]: A common experimental design is one with all input factors set at two levels each. These levels are called 'high 'and 'low 'or +1 and -1, respectively. A design with all possible high/low combinations of all the input factors is called a full factorial design in two levels.
- Central Composite Design (CCD) [35]: Central Composite desgin is used with Response Surface method to generate a second order model. CCD uses the full factorial samples along with the axial and the central points.
- Orthogonal Array Sampling (OAS) [35]: Latin hypercube sampling stratifies the components of Xi one at a time. OAS stratifies all pairs of components. By randomizing an orthogonal array and embedding it into the unit cube [0, 1]d we obtain a generalization of Latin hypercube sampling.

For more details refer to [18].

3.2.2 Classifications in Surrogate Models

Surrogate methods can be divided into two general frameworks based on the classiciation described in [36] and [37]. The classification is based on when the surrogate model is updated. The two main classes are

- Sequential: The key point in sequential framework is to build the surrogate model first and then apply the optimizer. Sequential mode of surrogates are used in [38–42] among many others
- Adaptive: The points sampled in the initial pass (before optimization) may not have points which are representative of Pareto Front. To overcome this shortcoming this mode of Surrogates are updated in each generation by selecting points such that accuracy of the model stays in admissible range. Adaptive mode of surrogates were used in [43–46] among many others

The general methods to assess the quality of the models are: RMSE, PRedicted Error Sum of Squares (PRESS) [47], cross-validation [48] and R^2 [49]. Other papers worth mentioning are work by Lim et al., which proposed online weighted average ensemble of surrogate methods [50], Zhou et al. which proposed to use a global model to find promising points and then use local models to guide memetic search [51], Ong et al. which makes a statement how uncertainty of a surrogate model can be a blessing in disguise and why highly accurate model is not required.

4. APPROACH

4.1 Spectral Learning

The minimal sampling method proposed in this paper is based on a spectral-learning algorithm that explores the spectrum (eigenvalues) of the distance matrix between configurations. In theory, such spectral learners are an appropriate method to handle noisy, redundant, and tightly inter-connected variables, for the following reasons: When data sets have many irrelevancies or closely associated data parameters d, then only a few eigenvectors e, $e \ll d$ are required to characterize the data. In that reduced space:

- Multiple inter-connected variables $i, j, k \subseteq d$ can be represented by a single eigenvector;
- Noisy variables from d are ignored, because they do not contribute to the signal in the data;
- Variables become (approximately) parallel lines in e space. For redundancies $i, j \in d$, we can ignore j since effects that change over j also change in the same way over i;

That is, in theory, samples of configurations drawn via an eigenspace sampling method would not get confused by noisy, redundant, or tightly inter-connected variables. Accordingly, we expect predictions built from that sample to have lower mean errors and lower variances on that error.

Spectral methods have been used before for a variety of data mining applications [52]. Algorithms, such as PDDP [53], use spectral methods, such as principle component analysis (PCA), to recursively divide data into smaller regions. Software-analytics researchers use spectral methods (again, PCA) as a pre-processor prior to data mining to reduce noise in software-related data sets [54]. However, to the best of our knowledge, spectral methods have not been used before in software engineering as a basis of a minimal sampling method.

WHAT is somewhat different from other spectral learners explored in, for instance, image processing applications [55]. Work on image processing does not address defining a minimal sampling policy to predict performance scores. Also, a standard spectral method requires an $O(N^2)$ matrix multiplication to compute the components of PCA [56]. Worse, in the case of hierarchical division methods, such as PDDP, the polynomial-time inference must be repeated at every level of the hierarchy. Competitive results can be achieved using an $O(2N)$ analysis that we have developed previously [57], which is based on a heuristic proposed by Faloutsos and Lin [58] (which Platt has shown computes a Nyström approximation to the first component of PCA [59]).

Our approach inputs N (with $1 \leq |N| \leq |V|$) valid configurations $(\vec{C}), N_1, N_2, ...$, and then:

1. Picks any point N_i ($1 \leq i \leq |N|$) at random;

2. Finds the point *West* $\in N$ that is furthest away from N_i;

3. Finds the point *East* $\in N$ that is furthest from *West*.

The line joining *East* and *West* is our approximation for the first principal component. Using the distance calculation shown in Equation 2, we define δ to be the distance between *East* and *West*. **WHAT** uses this distance (δ) to divide all the configurations as follows: The value x_i is the projection of N_i on the line running from *East* to *West*[6]. We divide the examples based on the median value of the projection of x_i. Now, we have two clusters of data divided based on the projection values (of N_i) on the line joining *East* and *West*. This process is applied recursively on these clusters until a predefined stopping condition. In our study, the recursive splitting of the N_i's stops when a sub-region contains less than $\sqrt{|N|}$ examples.

$$dist(x,y) = \begin{cases} \sqrt{\sum_i (x_i - y_i)^2} & \text{if } x_i \text{ and } y_i \text{ is numeric} \\ \begin{cases} 0, & \text{if } x_i = y_i \\ 1, & \text{otherwise} \end{cases} & \text{if } x_i \text{ and } y_i \text{ is Boolean} \end{cases} \quad (2)$$

We explore this approach for three reasons:

- *It is very fast*: This process requires only $2|n|$ distance comparisons per level of recursion, which is far less than the $O(N^2)$ required by PCA [60] or other algorithms such as K-Means [61].
- *It is not domain-specific*: Unlike traditional PCA, our approach is general in that it does not assume that all the variables are numeric. As shown in Equation 2,[7] we can approximate distances for both numeric and non-numeric data (e.g., Boolean).
- *It reduces the dimensionality problem*: This technique explores the underlying dimension (first principal component) without getting confused by noisy, related, and highly associated variables.

4.2 Spectral Sampling

When the above clustering method terminates, our sampling policy (which we will call S_1:Random) is then applied:

Random sampling ***(S_1):*** compile and execute one configuration, picked at random, from each leaf cluster;

We use this sampling policy, because (as we will show later) it performs better than:

East-West sampling ***(S_2):*** compile and execute the *East* and *West* poles of the leaf clusters;

Exemplar sampling ***(S_3):*** compile and execute all items in all leaves and return the one with lowest performance score.

Note that S_3 is *not* a *minimal* sampling policy (since it executes all configurations). We use it here as one baseline against which we can compare the other, more minimal, sampling policies. In the results that follow, we also compare our sampling methods against another baseline using information gathered after executing all configurations.

[6]The projection of N_i can be calculated in the following way:
$a = dist(East, N_i); b = dist(West, N_i); x_i = \sqrt{\frac{a^2 - b^2 + \delta^2}{2\delta}}$.

[7]In our study, *dist* accepts configurations ($\vec{C}$) and returns the distance between them. If x_i and $y_i \in \mathbb{R}^n$, then the distance function would be same as the standard Euclidean distance.

4.3 Regression-Tree Learning

After collecting the data using one of the sampling policies (S_1, S_2, or S_3), as described in Section 4.2, we use a CART regression-tree learner [62] to build a performance predictor. Regression-tree learners seek the attribute-range split that most increases our ability to make accurate predictions. CART explores splits that divide N samples into two sets A and B, where each set has a standard deviation on the target variable of σ_1 and σ_2. CART finds the "best" split defined as the split that minimizes $\frac{A}{N}\sigma_1 + \frac{B}{N}\sigma_2$. Using this best split, CART divides the data recursively.

In summary, **WHAT** combines:

- The FASTMAP method of Faloutsos and Lin [58];
- A spectral-learning algorithm initially inspired by Boley's PDDP system [53], which we modify by replacing PCA with FASTMAP (called "WHERE" in prior work [57]);
- The sampling policy that explores the leaf clusters found by this recursive division;
- The CART regression-tree learner that converts the data from the samples collected by sampling policy into a run-time prediction model [62].

That is,

$$\begin{aligned} \text{WHERE} &= \text{PDDP} - \text{PCA} + \text{FASTMAP} \\ \textbf{WHAT} &= \text{WHERE} + \text{SamplingPolicy} + \text{CART} \end{aligned}$$

This unique combination of methods has not been previously explored in the software-engineering literature.

5. EXPERIMENTS

All materials required for reproducing this work are available at https://goo.gl/689Dve.

5.1 Subject Systems

The configurable systems we used in our experiments are described in Figure 1. Note, with "predicting performance", we mean predicting performance scores of the subject systems while executing test suites provided by the developers or the community, as described in Figure 1. To compare the predictions of our and prior approaches with actual performance measures, we use data sets that have been obtained by measuring *nearly all* configurations[8]. We say *nearly all* configurations, for the following reasoning: For all except one of our subject systems, the total number of valid configurations was tractable (192 to 2560). However, SQLite has 3,932,160 possible configurations, which is an impractically large number of configurations to test whether our predictions are accurate and stable. Hence, for SQLite, we use the 4500 samples for testing prediction accuracy and stability, which we could collect in one day of CPU time. Taking this into account, we will pay particular attention to the variance of the SQLite results.

5.2 Experimental Rig

RQ1 and **RQ2** require the construction and assessment of numerous runtime predictors from small samples of the data. The following rig implements that construction process.

For each configurable software system, we built a table of data, one row per valid configuration. We then ran all configurations of all software systems and recorded the performance scores (i.e., that are invoked by a benchmark). The exception is SQLite for which we measured only the configurations needed to detect interactions and additionally 100 random configurations to evaluate the accuracy of predictions. To this table, we added a column showing the

[8]http://openscience.us/repo/performance-predict/cpm.html

performance score obtained from the actual measurements for each configuration.

Note that the following procedure ensures that we **never** test any prediction model on the data used to learn that model. Next, we repeated the following procedure 20 times (the figure of 20 repetitions was selected using the Central Limit Theorem): For each system in {BDBC, BDBJ, Apache, SQLite, LLVM, x264}

- Randomize the order of the rows in their table of data;
- For X in {10, 20, 30, ... , 90};
  - Let *Train* be the first X % of the data
  - Let *Test* be the rest of the data;
  - Pass *Train* to **WHAT** to select sample configurations;
  - Determine the performance scores associated with these configurations. This corresponds to a table lookup, but would entail compiling and executing a system configuration in a practical setting.
  - Using the *Train* data and their performance scores, build a performance predictor using CART.
  - Using the *Test* data, assess the accuracy of the predictor using the error measure of Equation 3 (see below).

The validity of the predictors built by the regression tree is verified on testing data. For each test item, we determine how long it *actually* takes to run the corresponding system configuration and compare the actual measured performance to the *prediction* from CART. The resulting prediction error is then computed using:

$$error = \frac{|\, predicted - actual \,|}{actual} * 100 \quad (3)$$

RQ2 requires testing the standard deviation of the prediction error rate. To support that test, we:

- Determine the X-th point in the above experiments, where all predictions stop improving (elbow point);
- Measure the standard deviation of the error at this point, across our 20 repeats.

As shown in Figure 6, all our results plateaued after studying $X = 40\,\%$ of the valid configurations[9]. Hence to answer **RQ2**, we will compare all 20 predictions at $X = 40\,\%$.

RQ3 uses the learned regression tree as a *surrogate model* within an optimizer;

- Take $X = 40\,\%$ of the configurations;
- Apply **WHAT** to build a CART model using some minimal sample taken from that 40 %;
- Use that CART model within some standard optimizer while searching for configurations with least runtime;
- Compare the faster configurations found in this manner with the fastest configuration known for that system.

[9]Just to clarify one frequently asked question about this work, we note that our rig "studies" 40 % of the data. We do not mean that our predictive models require accessing the performance scores from the 40 % of the data. Rather, by "study" we mean reflect on a sample of configurations to determine what minimal subset of that sample deserves to be compiled and executed.

This last item requires access to a ground truth of performance scores for a large number of configurations. For this experiment, we have access to that ground truth (since we have access to all system configurations, except for SQLite). Note that such a ground truth would not be needed when practitioners choose to use **WHAT** in their own work (it is only for our empirical investigation).

For the sake of completeness, we explored a range of optimizers seen in the literature in this second experiment: DE [63], NSGA-II [64], and our own GALE [19, 21] system. Normally, it would be reasonable to ask why we used those three, and not the hundreds of other optimizers described in the literature [65, 66]. However, as shown below, all these optimizers in this domain exhibited very similar behavior (all found configurations close to the best case performance). Hence, the specific choice of optimizer is not a critical variable in our analysis.

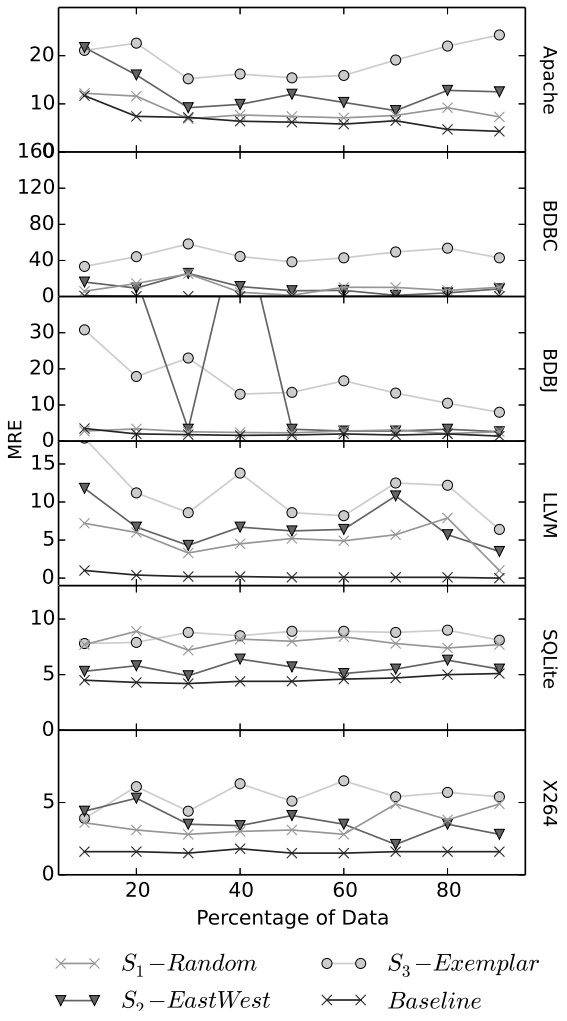

Figure 6: Errors of the predictions made by WHAT with four different sampling policies. Note that, on the y-axis, *lower* errors are *better*.

6. RESULTS

6.1 RQ1

Can **WHAT** *generate good predictions after executing only a small number of configurations?*

Figure 6 shows the mean errors of the predictors learned after taking X % of the configurations, then asking **WHAT** and some sampling method (S_1, S_2, and S_3) to (a) find what configurations to measure; then (b) asking CART to build a predictor using these measurements. The horizontal axis of the plots shows what X % of the configurations are studied; the vertical axis shows the mean relative error (from Equation 3). In that figure:

- The $\times$—$\times$ lines in Figure 6 show a *baseline* result where data from the performance scores of 100 % of configurations were used by CART to build a runtime predictor.
- The other lines show the results using the sampling methods defined in Section 4.2. Note that these sampling methods used runtime data only from a subset of 100 % of the performance scores seen in configurations from 0 to X %.

In Figure 6, *lower* y-axis values are *better* since this means lower prediction errors. Overall, we find that:

- Some software systems exhibit large variances in their error rate, below $X = 40$ % (e.g., BDBC and BDBJ).
- Above $X = 40$ %, there is little effect on the overall change of the sampling methods.
- Mostly, S_3 shows the highest overall error, so that it cannot be recommended.
- Always, the $\times$—$\times$ baseline shows the lowest errors, which is to be expected since predictors built on the baseline have access to all data.
- We see a trend that the error of S_1 and S_2 are within 5 % of the *baseline* results. Hence, we can recommend these two minimal sampling methods.

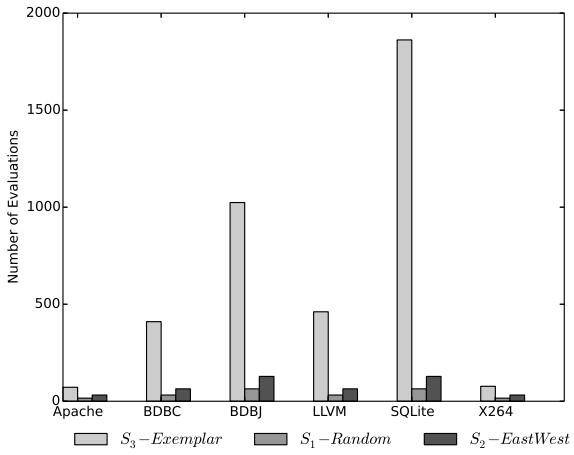

Figure 7: Comparing evaluations of different sampling policies. We see that the number of configurations evaluated for S_2 is twice as high as S_1, as it selects 2 points from each cluster, where as S_1 selects only 1 point.

Figure 7 provides information about which of S_1 or S_2 we should recommend. This figure displays data taken from the $X = 40$ % point of Figure 6 and displays how many performance scores of configurations are needed by our sub-sampling methods (while reflecting on the configurations seen in the range $0 \leq X \leq 40$). Note that:

- S_3 needs up to thousands of performance-score points, so it cannot be recommended as minimal-sampling policy;
- S_2 needs twice as much performance-score information as S_1 (S_2 uses *two* samples per leaf cluster while S_1 uses only *one*).
- S_1 needs performance-score information on only a few dozen (or less) configurations to generate the predictions with the lower errors seen in Figure 6.

Combining the results of Figure 6 and Figure 7, we conclude that:

> S_1 is our preferred spectral sampling method. Furthermore, the answer to **RQ1** is "yes", because applying **WHAT** , we can (a) generate runtime predictors using just a few dozens of sample performance scores; and (b) these predictions have error rates within 5 % of the error rates seen if predictors are built from information about all performance scores.

6.2 RQ2

Do less data used in building the models cause larger variances in the predicted values?

Two competing effects can cause increased or decreased variances in runtime predictions. The less we sample the configuration space, the less we constrain model generation in that space. Hence, one effect that can be expected is that models learned from too few samples exhibit large variances. But, a compensating effect can be introduced by sampling from the spectral space since that space contains fewer confusing or correlated variables than the raw configuration space.

Figure 8 reports which one of these two competing effects are dominant. Figure 6 shows that after some initial fluctuations, after seeing $X = 40$ % of the configurations, the variances in prediction errors reduces to nearly zero.

> Hence, we answer **RQ2** with "no": Selecting a small number of samples does not necessarily increase variance (at least to say, not in this domain).

6.3 RQ3

Can "good" surrogate models (to be used in optimizers) be built from minimal samples?

The results of answering **RQ1** and **RQ2** suggest to use **WHAT** (with S_1) to build runtime predictors from a small sample of data. **RQ3** asks if that predictor can be used by an optimizer to infer what *other* configurations correspond to system configurations with fast performance scores. To answer this question, we ran a random set of 100 configurations, 20 times, and related that baseline to three optimizers (GALE [19], DE [63] and NSGA-II [64]) using their default parameters.

When these three optimizers mutated existing configurations to suggest new ones, these mutations were checked for validity. Any mutants that violated the system's constraints (e.g., a feature excluding another feature) were rejected and the survivors were "evaluated" by asking the CART surrogate model. These evaluations either rejected the mutant or used it in generation $i+1$, as the basis for a search for more, possibly better mutants.

Figure 9 shows the configurations found by three optimizers projected onto the ground truth of the performance scores of nearly all configurations (see Section 5.1). Again note that, while we use that ground truth for the validation of these results, our optimizers used

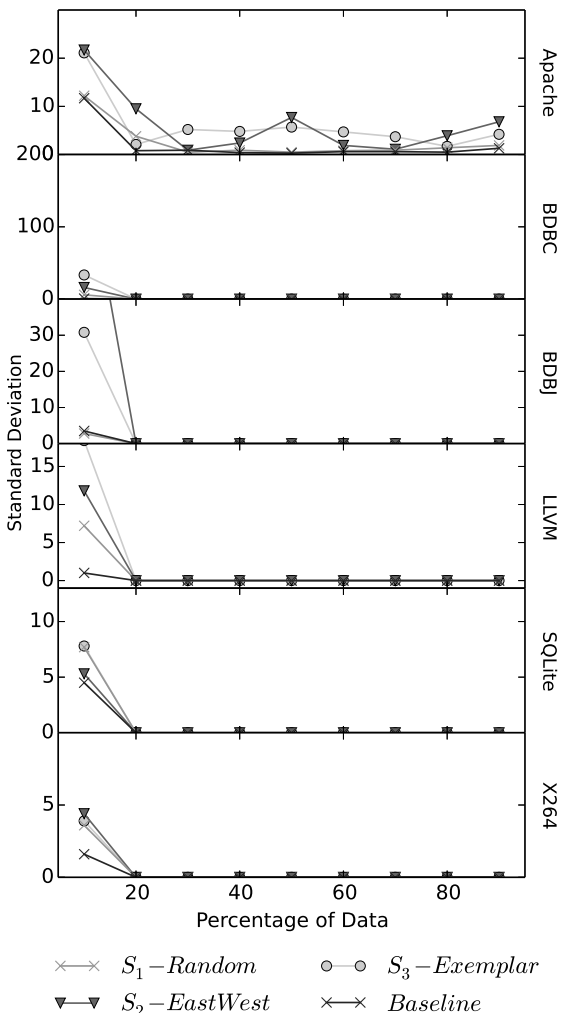

Figure 8: Standard deviations seen at various points of Figure 6.

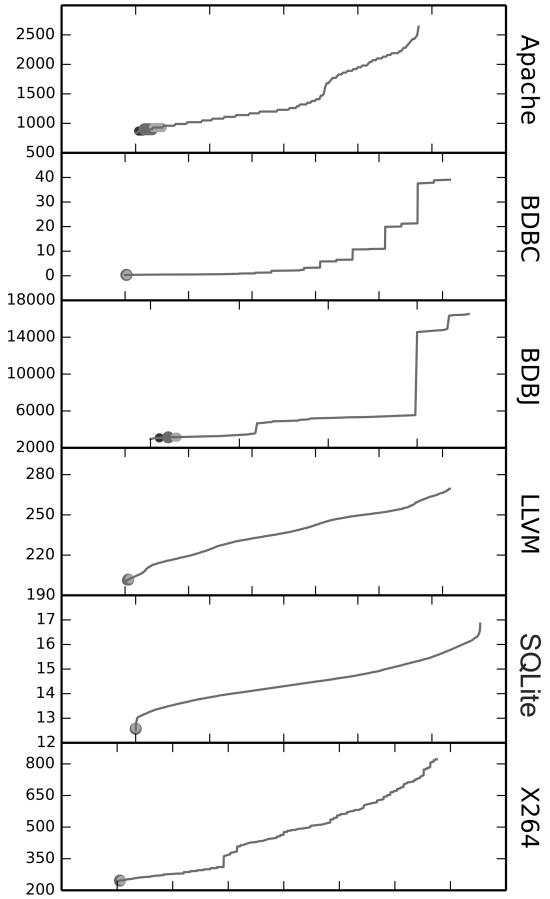

Figure 9: Solutions found by GALE, NSGA-II and DE (shown as points) laid against the ground truth (all known configuration performance scores). It can be observed that all the optimizers can find the configuration with lower performance scores.

only a small part of that ground-truth data in their search for the fastest configurations (see the **WHAT** + S_1 results of Figure 7).

The important feature of Figure 9 is that all the optimized configurations fall within 1 % of the fastest configuration according to the ground truth (see all the left-hand-side dots on each plot). Table 1 compares the performance of the optimizers used in this study. Note that the performances are nearly identical, which leads to the following conclusions:

> The answer to **RQ3** is "yes": For optimizing performance scores, we can use surrogates built from few runtime samples. The choice of the optimizer does not critically effect this conclusion.

6.4 RQ4

How good is **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

We compare **WHAT** with the three state-of-the-art predictors proposed in the literature [3], [4], [5], as discussed in Section **??**.

Note that all approaches use regression-trees as predictors, except Siegmund's approach, which uses a regression function derived using linear programming.

The ▭ bars of Figure 10 show the mean error rate, the standard deviation of the error rate, and the mean percentage of total configurations used in 30 repeats of the different approaches. Note that the y-axis of that figure is a logarithmic scale so, within each plot:

- Differences near the bottom are very small differences;
- Differences near the top are very large differences;

As seen in the left and middle plots of Figure 10, the *FW* approach of Siegmund et al. (i.e., the sampling approach using the fewest number of configurations) often has the highest error rate and the highest standard deviation on that error rate. Hence, we cannot recommend this method or, if one wishes to use this method, we recommend using the other sampling heuristics (e.g., HO, HS) to make more accurate predictions (but at the cost of much more measurements). Moreover, the size of the standard deviation of this method causes further difficulties in estimating which configurations are those exhibiting a large prediction error.

As to the approach of Guo et al. (with PW), this does not standout on any of our measurements. Its error results are within 1 % of

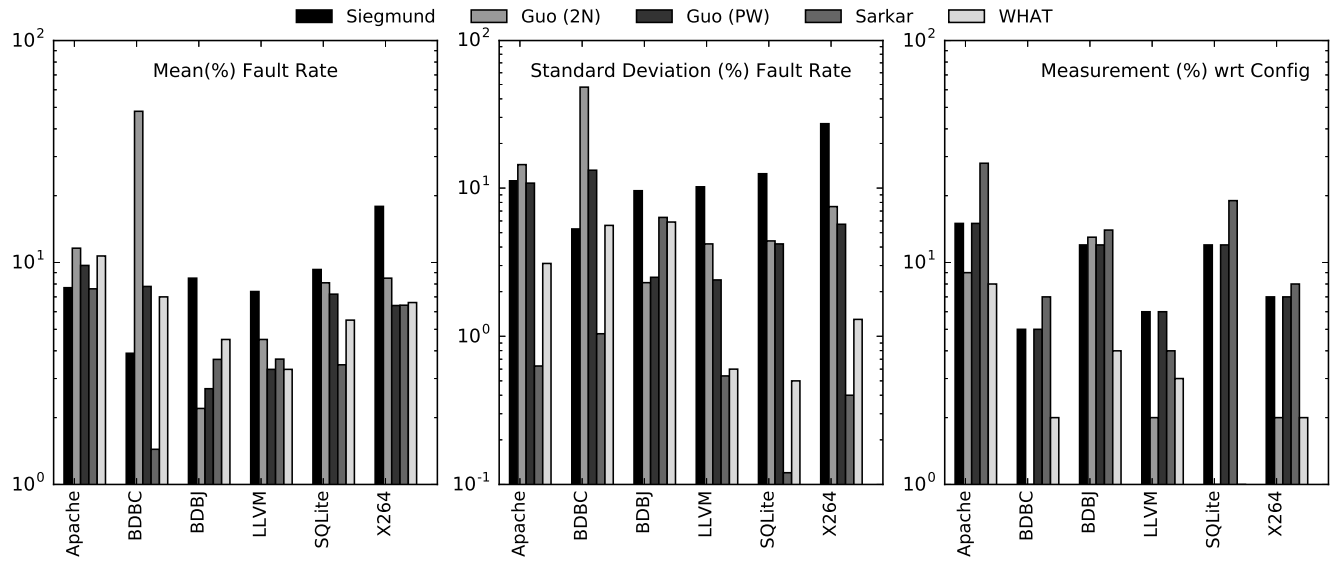

Figure 10: Comparison between WHAT and the state-of-the-art approaches regarding mean error, standard deviation, and the percentage of configurations used for training the model.

Table 1: The table shows how the minimum performance scores as found by the learners GALE, NSGA-II, and DE, vary over 20 repeated runs. Mean values are denoted μ and IQR denotes the 25th–75th percentile. A low IQR suggests that the surrogate model build by WHAT is stable and can be utilized by off the shelf optimizers to find performance-optimal configurations.

Dataset	Searcher					
	GALE		DE		NSGAII	
	Mean	IQR	Mean	IQR	Mean	IQR
Apache	870	0	840	0	840	0
BDBC	0.363	0.004	0.359	0.002	0.354	0.005
BDBJ	3139	70	3139	70	3139	70
LLVM	202	3.98	200	0	200	0
SQLite	13.1	0.241	13.1	0	13.1	0.406
X264	248	3.3	244	0.003	244	0.05

WHAT ; its standard deviation are usually larger; and it requires much more data than **WHAT** .

In terms of the number of measured samples required to build a model, the right-hand-side plot of Figure 10 shows that **WHAT** requires the fewest samples except for two cases: the approach of Guo et al. (with 2N) working on BDBC and LLVM. In both these cases, the mean error and standard deviation on the error estimate is larger than **WHAT** (see the ▭ bars in the left and middle plots of Figure 10). Furthermore, in the case of BDBC, the error values are $\mu = 14\%$, $\sigma = 13\%$, which are much larger than **WHAT** 's error scores of $\mu = 6\%$, $\sigma = 5\%$.

Although the approach of Sarkar et al. produces an error rate that is sometimes less than the one of **WHAT** (see the left-hand-side of Figure 10), it requires the most number of measurements. Moreover, **WHAT** 's accuracy is close to Sarkar's approach (1 % to 2 % difference). Hence, we cannot recommend this approach, too.

The right-hand-side of Figure 10 shows the *percent* of required measurements. Table 2 shows the same expressed as absolute values. We see that most state-of-the-art approaches often require many more samples than **WHAT** . Using those fewest numbers of samples, **WHAT** has within 1 to 2 % of the lowest standard deviation rates and within 1 to 2 % of lowest error rates. The exception is Sarkar's approach, which has 5 % lower mean error rates (in BDBC, see the left-hand-side plot of Figure 10). However, as shown in right-hand-side of Table 2, Sarkar's approach needs nearly three times more measurements than **WHAT** (191 vs 64 samples). Given the overall reduction of the error is small (5 % difference between Sarkar and **WHAT** in mean error), the overall cost of tripling the data-collection cost is often not feasible in a practical context and might not justify the small additional benefit in accuracy.

Table 2: Comparison of the number of the samples required with the state of the art. The grey colored cells indicate the approach which has the lowest number of samples. We notice that WHAT and Guo (2N) uses less data compared to other approaches. The high fault rate of Guo (2N) accompanied with high variability in the predictions makes WHAT our preferred method.

Dataset	Samples				
	Siegmund	Guo (2N)	Guo (PW)	Sarkar	WHAT
Apache	29	181	29	55	16
BDBC	139	36	139	191	64
BDBJ	48	52	48	57	16
LLVM	62	22	64	43	32
SQLite	566	78	566	925	64
X264	81	32	81	93	32

Hence, we answer **RQ4** with "yes", since **WHAT** yields predictions that are similar to or more accurate than prior work, while requiring fewer samples.

7. RELIABILITY AND VALIDITY

Reliability refers to the consistency of the results obtained from the research. For example, how well independent researchers could reproduce the study? To increase external reliability, this paper has taken care to either clearly define our algorithms or use implementations from the public domain (SciKitLearn) [67]. Also, all the data used in this work is available on-line in the PROMISE code repository and all our algorithms are on-line at github.com/ai-se/where.

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate [68]. *Internal validity* checks if the differences found in the treatments can be ascribed to the treatments under study.

One internal validity issue with our experiments is the choice of *training and testing* data sets discussed in Figure 1. Recall that while all our learners used the same *testing* data set, our untuned learners were only given access to *training* data.

Another internal validity issues is *instrumentation*. The very low μ and σ error values reported in this study are so small that it is reasonable to ask whether they are due to some instrumentation quirk, rather than due to using a clever sample strategy:

- Our low μ values are consistent with prior work (e.g. [5]);
- As to our low σ values, we note that, when the error values are so close to 0 %, the standard deviation of the error is "squeezed" between zero and those errors. Hence, we would expect that experimental rigs that generate error values on the order of 5 % and Equation 3 should have σ values of $0 \leq \sigma \leq 5$ (e.g., like those seen in our introduction).

Regarding SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual performance values. We are aware that this evaluation leaves room for outliers. Also, we are aware that measurement bias can cause false interpretations [57]. Since we aim at predicting performance for a special workload, we do not have to vary benchmarks.

We aimed at increasing the *external validity* by choosing software systems from different domains with different configuration mechanisms and implemented with different programming languages. Furthermore, the systems used are deployed and used in the real world. Nevertheless, assuming the evaluations to be automatically transferable to all configurable software systems is not fair. To further strengthen external validity, we run the model (generated by **WHAT** + S_1) against other optimizers, such as NSGA-II and differential evolution [63]. That is, we validated whether the learned models are not only applicable for GALE style of perturbation. In Table 1, we see that the models developed are valid for all optimizers, as all optimizers are able to find the near optimal solutions.

8. CONCLUSIONS

Configurable software systems today are widely used in practice, but expose challenges regarding finding performance-optimal configurations. State-of-the-art approaches require too many measurements or are prone to large variances in their performance predictions. To avoid these shortcomings, we have proposed a fast spectral learner, called **WHAT** , along with three new sampling techniques. The key idea of **WHAT** is to explore the configuration space with eigenvalues of the features used in a configuration to determine exactly those configurations for measurement that reveal key performance characteristics. This way, we can study many closely associated configurations with only a few measurements.

We evaluated our approach on six real-world configurable software systems borrowed from the literature. Our approach achieves similar to lower error rates, while being stable when compared to the state of the art. In particular, with the exception of Berkeley DB, our approach is more accurate than the state-of-the-art approaches by Siegmund et al. [3] and Guo et al. [4]. Furthermore, we achieve a similar prediction accuracy and stability as the approach by Sarkar et al [5], while requiring a far smaller number of configurations to be measured. We also demonstrated that our approach can be used to build cheap and stable surrogate prediction models, which can be used by off-the-shelf optimizers to find the performance-optimal configuration.

9. FUTURE WORK

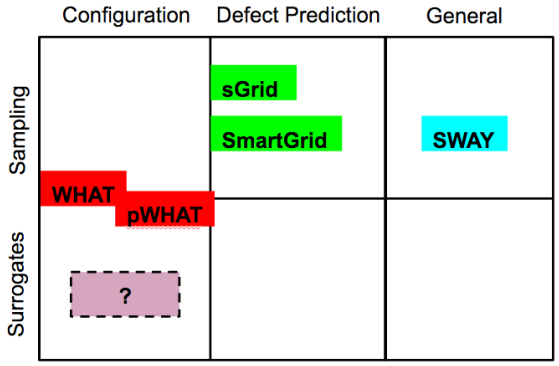

Figure 11: Planned work for the rest of my Ph.D.

.

The high level idea we are running with is: "How can we introduce surrogate modelling to SBSE ". Through the survey of the current body of work in SBSE we don't find any mention of surrogate based methods. We breakdown our research into main techniques: Sampling and Surrogate Modelling and we want to apply the techniques developed in two case studies: Configurations and Defect Prediction. As a first attempt to introduce surrogate models to SBSE, we have tried exploring sampling techniques on software engineering data. The following are our future plans

- EAs start with a random set of initial population and a good sampling algorithm can be used to get a good estimate of near optimal solution and if the problem isn't expensive this can be used to 'SuperCharge 'other state of the optimizers. We call this **SWAY**.
- One of the main drawbacks of the current version of WHAT is its inflexible nature. We would like to modify WHAT in such a way that (i) it takes the software system (to model) as well as the budget allocated for the experiment, (ii) for the purpose of saving resource make the process progressive. We call the system **pWhat** (progressive WHAT).
- In the defect prediction world, tuning the defect predictors is a challenging problem, which is very similar to the problem of hyper-parameter optimization. There has been numerous attempts to tune the defect predictors using grid search. Spectral GridSearch (**sGrid**) takes into account the principal component of the search space and construct grids.
- Grid Search starts with regular grids and does not take into account how important the features are. A good sampling

algorithm can be used to build a stable surrogate which can be then used to find the important feature using any feature ranking algorithm. This can be then used to create **SmartGrid**, which would be irregular grid and would be far more useful.

The figure 11 is visual explanation our future work. The box in the pink suggests how we can use surrogate models on configuration problem to optimize for multiple goals. We choose configuration problem because of the real-world impact of the solutions we provide.

10. REFERENCES

[1] Juan J Durillo, Yuanyuan Zhang, Enrique Alba, Mark Harman, and Antonio J Nebro. A study of the bi-objective next release problem. *Empirical Software Engineering*, 16(1):29–60, 2011.

[2] Francisco Luna, David L González-Álvarez, Francisco Chicano, and Miguel A Vega-Rodríguez. The software project scheduling problem: A scalability analysis of multi-objective metaheuristics. *Applied Soft Computing*, 15:136–148, 2014.

[3] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.

[4] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 301–311. IEEE, 2013.

[5] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 342–352. IEEE, 2015.

[6] Ilya Gennadyevich Loshchilov. *Surrogate-assisted evolutionary algorithms*. PhD thesis, 2013.

[7] Keith Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay, and Malcolm Munro. Service-based software: the future for flexible software. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 214–221. IEEE, 2000.

[8] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.

[9] K Thomas. Thanks, amazon: The cloud crash reveals your importance. 2002.

[10] Why gmail went down: Google misconfigured load balancing servers. http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server/. Accessed: 2016-03-25.

[11] Microsoft: Misconfigured network device caused azure outage. http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/. Accessed: 2016-03-25.

[12] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 28–35. IEEE, 2004.

[13] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151. ACM, 2014.

[14] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 131–140. IEEE, 2011.

[15] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

[16] Athanasios Naskos, Anastasios Gounaris, and Spyros Sioutas. Cloud elasticity: A survey. In *Algorithmic Aspects of Cloud Computing*, pages 151–167. Springer, 2016.

[17] Neha Golkar. A power-aware cost model for hpc procurement, 2016.

[18] Mohammad Tabatabaei, Jussi Hakanen, Markus Hartikainen, Kaisa Miettinen, and Karthik Sindhya. A survey on handling computationally expensive multiobjective optimization problems using surrogates: non-nature inspired methods. *Structural and Multidisciplinary Optimization*, 52(1):1–25, 2015.

[19] Joseph Krall, Tim Menzies, and Misty Davies. Gale: Geometric active learning for search-based software engineering. *IEEE Transactions on Software Engineering*, 41(10):1001–1018, 2015.

[20] Khaled Rasheed and Haym Hirsh. Informed operators: Speeding up genetic-algorithm-based design optimization using reduced models. In *GECCO*, pages 628–635, 2000.

[21] Marcela Zuluaga, Guillaume Sergent, Andreas Krause, and Markus Püschel. Active learning for multi-objective optimization. In *Proceedings of the 30th International Conference on Machine Learning*, pages 462–470, 2013.

[22] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[23] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 33–42. ACM, 2011.

[24] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM, 2015.

[25] Gary M Weiss and Ye Tian. Maximizing classifier utility when there are data acquisition and modeling costs. *Data Mining and Knowledge Discovery*, 17(2):253–282, 2008.

[26] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by

fourier learning. In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 365–373. IEEE, 2015.

[27] Abdel Salam Sayyad, Joe Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel's back. In *IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 465–474. IEEE, 2013.

[28] Jens I Madsen, Wei Shyy, and Raphael T Haftka. Response surface techniques for diffuser shape optimization. *AIAA journal*, 38(9):1512–1518, 2000.

[29] Jack PC Kleijnen. Kriging metamodeling in simulation: A review. *European Journal of Operational Research*, 192(3):707–716, 2009.

[30] Martin D Buhmann. Radial basis functions: theory and implementations. *Cambridge monographs on applied and computational mathematics*, 12:147–165, 2004.

[31] Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

[32] Martin T Hagan, Howard B Demuth, Mark H Beale, and Orlando De Jesús. *Neural network design*, volume 20. PWS publishing company Boston, 1996.

[33] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[34] Jon C Helton, Jay Dean Johnson, Cedric J Sallaberry, and Curt B Storlie. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Engineering & System Safety*, 91(10):1175–1209, 2006.

[35] Ruichen Jin, Wei Chen, and Timothy W Simpson. Comparative studies of metamodelling techniques under multiple modelling criteria. *Structural and Multidisciplinary Optimization*, 23(1):1–13, 2001.

[36] G Gary Wang and Songqing Shan. Review of metamodeling techniques in support of engineering design optimization. *Journal of Mechanical design*, 129(4):370–380, 2007.

[37] GP Liu, X Han, and C Jiang. A novel multi-objective optimization method based on an approximation model management technique. *Computer Methods in Applied Mechanics and Engineering*, 197(33):2719–2731, 2008.

[38] Tushar Goel, Rajkumar Vaidyanathan, Raphael T Haftka, Wei Shyy, Nestor V Queipo, and Kevin Tucker. Response surface approximation of pareto optimal front in multi-objective optimization. *Computer methods in applied mechanics and engineering*, 196(4):879–893, 2007.

[39] Nestor V Queipo, Raphael T Haftka, Wei Shyy, Tushar Goel, Rajkumar Vaidyanathan, and P Kevin Tucker. Surrogate-based analysis and optimization. *Progress in aerospace sciences*, 41(1):1–28, 2005.

[40] Benjamin Wilson, David Cappelleri, Timothy W Simpson, and Mary Frecker. Efficient pareto frontier exploration using surrogate approximations. *Optimization and Engineering*, 2(1):31–50, 2001.

[41] Xingtao Liao, Qing Li, Xujing Yang, Weigang Zhang, and Wei Li. Multiobjective optimization for crash safety design of vehicles using stepwise regression model. *Structural and Multidisciplinary Optimization*, 35(6):561–569, 2008.

[42] Ruiyi Su, Liangjin Gui, and Zijie Fan. Multi-objective optimization for bus body with strength and rollover safety constraints based on surrogate models. *Structural and Multidisciplinary Optimization*, 44(3):431–441, 2011.

[43] Jian Zhou, Lih-Sheng Turng, et al. Adaptive multiobjective optimization of process conditions for injection molding using a gaussian process approach. *Advances in Polymer Technology*, 26(2):71–85, 2007.

[44] Massimiliano Gobbi, Paolo Guarneri, Leonardo Scala, and Lorenzo Scotti. A local approximation based multi-objective optimization algorithm with applications. *Optimization and Engineering*, 15(3):619–641, 2014.

[45] Zeeshan Omer Khokhar, Hengameh Vahabzadeh, Amirreza Ziai, G Gary Wang, and Carlo Menon. On the performance of the psp method for mixed-variable multi-objective design optimization. *Journal of Mechanical Design*, 132(7):071009, 2010.

[46] Guodong Chen, Xu Han, Guiping Liu, Chao Jiang, and Ziheng Zhao. An efficient multi-objective optimization method for black-box functions using sequential approximate technique. *Applied Soft Computing*, 12(1):14–27, 2012.

[47] Michael H Kutner, Chris Nachtsheim, and John Neter. *Applied linear regression models*. McGraw-Hill/Irwin, 2004.

[48] Songqing Shan and G Gary Wang. Survey of modeling and optimization strategies to solve high-dimensional design problems with computationally-expensive black-box functions. *Structural and Multidisciplinary Optimization*, 41(2):219–241, 2010.

[49] YF Li, Szu Hui Ng, Min Xie, and TN Goh. A systematic comparison of metamodeling techniques for simulation optimization in decision support systems. *Applied Soft Computing*, 10(4):1257–1273, 2010.

[50] Dudy Lim, Yaochu Jin, Yew-Soon Ong, and Bernhard Sendhoff. Generalizing surrogate-assisted evolutionary computation. *Evolutionary Computation, IEEE Transactions on*, 14(3):329–355, 2010.

[51] Zongzhao Zhou, Yew Soon Ong, Prasanth B Nair, Andy J Keane, and Kai Yew Lum. Combining global and local surrogate models to accelerate evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(1):66–76, 2007.

[52] Kamvar Kamvar, Sepandar Sepandar, Klein Klein, Dan Dan, Manning Manning, and Christopher Christopher. Spectral learning. In *International Joint Conference of Artificial Intelligence*. Stanford InfoLab, 2003.

[53] Daniel Boley. Principal direction divisive partitioning. *Data mining and knowledge discovery*, 2(4):325–344, 1998.

[54] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering*, pages 199–208. IEEE Press, 2015.

[55] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[56] Alexander Ilin and Tapani Raiko. Practical approaches to principal component analysis in the presence of missing values. *The Journal of Machine Learning Research*, 11:1957–2000, 2010.

[57] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, 2013.

[58] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. volume 24. ACM, 1995.

[59] John Platt. Fastmap, Metricmap, and Landmark MDS are all nystrom algorithms. pages 261–268. Society for Artificial Intelligence and Statistics, 2005.

[60] Qian Du and James E Fowler. Low-complexity principal component analysis for hyperspectral image compression. *International Journal of High Performance Computing Applications*, 22(4):438–448, 2008.

[61] Greg Hamerly. Making k-means even faster. Society for Industrial and Applied Mathematics.

[62] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[63] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[64] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[65] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[66] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.

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

[68] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *Proceedings of the 37th International Conference on Software Engineering*, pages 9–19. IEEE, 2015.