Variability-Aware Performance Prediction: A Case Study

by

Pavel Valov

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Configurable software systems allow users to form configurations by selecting and deselecting features. The process of configuration creation may directly affect performance of the system in a non-linear way because of possible complex feature interactions. Understanding the correlation between feature selection and performance is important for stakeholders to acquire a desirable program variant. In this work we try to infer this correlation between system configuration and performance, using small samples of already measured configurations, without additional effort to detect feature interactions. We carry out a case study of several regression methods for solving this problem: regression trees, bagging of regression trees, random forests and support vector machines. All regression methods have their parameters tuned in automatic fashion by using Sobol sampling. To evaluate the prediction accuracy of the regression methods, the case study is performed using six real-world configurable software systems from different application domains and written in different programming languages. We show that bagging outperforms all other regression methods in most of the cases for all configurable systems, sampling sizes and parameter settings. We analyse the sensitivity of different regression methods and show that the most stable ones are regression trees and bagging.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Professor Krzysztof Czarnecki for giving me this great opportunity of studying and performing research in University of Waterloo, for his continuous support, for his patience, motivation and immense knowledge. Without him this thesis would not be possible.

Next, I would also like to thank Dr. Jianmei Guo, for his guidance and help in this project. For his outstanding patience and methodical approach, for his motivation and desire to make me an independent researcher.

I would like to thank Professor Daniel Lizotte and Professor Eric Blais for being reviewers of this thesis and providing a lot of detailed and useful feedback.

I thank my fellow labmates in Generative Software Development Lab that always helped me with advices and made my studies much more joyful: Alexander Murashkin, Ed Zulkoski, Jess Alejandro Padilla Gaeta, Leonardo Passos, Michal Antkiewicz, Rafael Olaechea, Wenbin Ji and Zubair Akhtar.

Last but not the least, I would like to express my deepest gratitude and love to my parents and my sister, for supporting, loving and caring about me my entire life.

Dedication

This work is dedicated to my parents and my sister, people that I will always love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most of the modern software systems nowadays provide configuration options to modify both functional behaviour of the system, i.e. how and what functions system is going to execute, and non-functional properties of the system, like performance and memory consumption.

Configuration options of a software system that are relevant to users are usually called *features*. Certain selection of features together defines a configuration of a software system. Each feature of a system has a direct influence on behaviour and non-functional properties. One of the most important non-functional properties is performance, because it has a direct influence on the user's perception of the system. Performance is influenced by many factors, one of which is the system configuration.

In the current work we try to determine the influence of feature selections on system performance. Understanding this correlation is a non-trivial task. We regard a software system as a black box and try to infer the correlation between feature selection and performance to predict performance based on a given system configuration.

The most straightforward approach to solving this problem is to actually measure the performance of all possible configurations of a system, create a table of correspondence between configurations and performance and give exact value of performance for each configuration. Unfortunately, this approach is feasible only for very small systems because number of possible configurations is exponential in number of features. Moreover, measurement effort of individual configuration is potentially high (e.g., executing a complex benchmark)[8]. Furthermore, in practice only a limited amount of performance measurements might be available, that covers only a small part of all possible configurations.

Another approach is to measure the performance contribution of each feature and predict the performance of the system by summing up the contributions of all selected features in a given configuration. Unfortunately, this approach might not be accurate because feature interactions can lead to unpredictable performance anomalies[18]. Siegmund et al.[18] overcame this hurdle by detecting performance-relevant feature interactions using specific sampling heuristics that meet different feature-coverage criteria. However, in practice, the program variants that we can measure or that we already have at our disposal may not meet any feature-coverage criterion.

Guo et al.[8] addressed this issue by using regression analysis based on small random samples of measured variants. But they used only one regression technique, called Classification And Regression Trees (CART), and they did not perform systematic parameter tuning and sensitivity analysis (i.e., the prediction accuracy varies widely with the parameter settings of the used methods).

In this paper, we aim at an empirical comparison of different regression methods for variability-aware performance prediction. We compare CART to three other popular regression techniques: bagging[4], random forest[5], and support vector machines (SVM)[25]. We evaluate the prediction accuracy of each regression method based on small random samples of measured program variants. Moreover, we analyse the parameter sensitivity of each regression method using a state-of-the-art parameter sampling technique, called Sobol sampling[14], which chooses a representative set of parameter settings that evenly cover the parameter space of each method. We conduct experiments on the same dataset used in[18, 8], which covers six real-world software programs.

In summary, we make the following contributions:

- **Methods.** We extend our previous work[8] and use four regression methods, including CART, bagging, random forest, and SVM, for variability-aware performance prediction based on only small random samples of measured program variants.
- **Evaluation.** We evaluate the prediction accuracy of each regression method by experiments on six real-world software programs. Moreover, we use Sobol sampling to analyse the parameter sensitivity of each regression method.
- **Findings.** Our empirical results show that bagging outperforms all other techniques in terms of prediction accuracy in most cases from the combination the analysed different software programs, different sample sizes, and different parameter settings. Furthermore, bagging is identified as the most stable method with regard to parameter sensitivity.

Chapter 2

Motivating Example

To motivate our work, we use the same example from our previous work[8], a configurable command-line tool x264 for encoding video streams into the H.264/MPEG-4 AVC format. In this example, we consider 16 encoder features of x264, such as encoding with multiple reference frames and parallel encoding on multiple CPUs. The stakeholder can select different features to encode a video. The encoding time is used to indicate the performance of x264 in different configurations. A *configuration* represents a program variant with a certain selection of features. This example with only 16 features gives rise to 1,152 configurations. Intuitively, 16 binary features should provide 2^{16} different configurations, however, in this work we consider only valid configurations i.e. configurations that are allowed by the system under investigation.

In practice, often only a limited set of configurations can be measured, either by simulation or by monitoring in the field. For example, Table 2.1 lists a sample of 16 randomly-selected configurations and their actual performance measurements. How can we determine the performance of other configurations based on a small random sample of measured configurations?

To formulate the above issue, we represent a feature as a binary decision variable x. If a feature is selected in a configuration, then the corresponding variable is set to 1, and 0 otherwise. Assume that there are N features in total, all features of a program are represented as a set $X = \{x_1, x_2, ..., x_N\}$. A configuration is an N-tuple $\mathbf{c}$, assigning 1 or 0 to each variable. For example, each configuration of x264 is represented by 16-tuple, e.g. $\mathbf{c}_1 = (x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, ..., x_{16} = 1)$. All valid configurations of a program are denoted by $\mathbf{C}$.

Each configuration $\mathbf{c}$ of a program has an actual measured performance value y. Per-

Conf.	Features																Perf. (s)
$\mathbf{c}_i$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	y_i
$\mathbf{c}_1$	1	1	0	0	1	1	0	0	1	0	0	1	1	1	0	0	292
$\mathbf{c}_2$	1	0	0	0	1	1	0	1	1	1	0	0	1	0	1	0	571
$\mathbf{c}_3$	1	1	1	0	0	1	0	0	1	0	1	0	1	0	0	1	681
$\mathbf{c}_4$	1	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	263
$\mathbf{c}_5$	1	1	1	0	1	1	0	0	1	0	1	0	1	0	1	0	536
$\mathbf{c}_6$	1	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	305
$\mathbf{c}_7$	1	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0	408
$\mathbf{c}_8$	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	278
$\mathbf{c}_9$	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	0	519
$\mathbf{c}_{10}$	1	1	0	0	1	1	0	1	1	0	1	0	1	0	0	1	781
$\mathbf{c}_{11}$	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	1	822
$\mathbf{c}_{12}$	1	1	0	0	1	0	0	0	1	0	1	0	1	0	0	1	713
$\mathbf{c}_{13}$	1	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	381
$\mathbf{c}_{14}$	1	0	1	1	1	1	0	1	1	1	0	0	1	0	1	0	564
$\mathbf{c}_{15}$	1	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	489
$\mathbf{c}_{16}$	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	275

Table 2.1: A sample of 16 randomly-selected configurations of x264 and corresponding performance measurements (seconds)

formance values are taken from publicly available dataset deployed with SPLConqueror tool[17]. Detailed description of the dataset is given in Section 6.1.

All performance values of all configurations $\mathbf{C}$ form set Y. Suppose that we acquire a random sample of configurations $\mathbf{C}_S \subset \mathbf{C}$ and their actual measured performance values $Y_S \subset Y$, together forming sample S. The problem of variability-aware performance prediction is to predict the performance of other configurations in $\mathbf{C} \backslash \mathbf{C}_S$ based on the measured sample S.

We regard all variables in X as *predictors* and a configuration's actual performance value y as the *response*. In other words, we predict a quantitative response y based on a set of categorical predictors X, which is a typical regression problem[9]. Due to feature interactions[18], the above issue is reduced to a non-linear regression problem, where the response depends non-linearly on one or more predictors[8].

Chapter 3

Regression Methods

To address the above non-linear regression problem, we introduce four popular regression methods, including CART, bagging, random forest, and SVM. We use the motivating example of Chapter 2 to explain how each regression method works for variability-aware performance prediction.

3.1 CART

Decision trees, also referred to as Classification and Regression Trees (or CART), is one of the most widely used machine learning techniques for data analysis[27]. Although CART might not perform as well as other classification or regression methods, it can be displayed graphically and is easily understood by the user[10]. The general idea of the method is to *recursively partition* data under consideration into smaller segments, until a *local prediction model* can be fitted into every segment. After partitioning is performed, all *local prediction models* are organised into a *global prediction model* in the form of a binary decision tree.

Previously, CART has already been used by Guo et al.[8] for variability-aware performance prediction. To illustrate how CART generates regression trees, we are going to continue with our motivating example. We use CART to generate a regression tree from the sample S of x264 data, presented in Table 2.1 on page 4. The resulting regression tree is presented on Figure 3.1 on page 7.

To generate the tree, CART starts with a sample S that contains a set of 16 configurations $\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_{16}$ along with their performance measurements $y_1, y_2, ..., y_{16}$. Then, CART starts *recursive* partitioning of the sample S into smaller sections. At first, S is split into two subsections by value of a selected feature variable x_s. The variable x_s is selected by exhaustively searching the feature variable set X for a variable that provides the best split in terms of reducing the total prediction error in both subsections. As shown in Figure 3.1 the variable that provides the best split for sample S is x_{14}. All configurations that have $x_{14} = 1$ go to the left subsection, while configurations that have $x_{14} = 0$ go to the right subsection. Partitioning into further subsections is performed in the same manner.

The *local prediction model* of a section is defined by the *sample mean* of actual performance values of configurations that are collected in the corresponding section.

$$l_{S_i} = \frac{1}{|S_i|} \sum_{y_j \in S_i} y_j \tag{3.1}$$

The training error of a section is calculated by a common loss function, the sum of *squared error loss*, i.e., the sum of squared differences between the local prediction and actual performance values in section S_i[9]:

$$\sum_{y_j \in S_i} L(y_j, l_{S_i}) = \sum_{y_j \in S_i} (y_j - l_{S_i})^2 \tag{3.2}$$

Therefore, the selection of a feature variable x_s for splitting of a section into subsections, i.e. splitting of S_i into S_{iL} and S_{iR}, is done by minimizing the following sum:

$$\sum_{y_j \in S_{iL}} L(y_j, l_{S_{iL}}) + \sum_{y_j \in S_{iR}} L(y_j, l_{S_{iR}}) \tag{3.3}$$

The partitioning process is controlled by the parameters of CART, as we explain in Section 5.1 on page 20. After the partitioning process is finished, all local estimators are combined into a global prediction model in the form of a binary decision tree, which we call a *regression tree*, denoted as $RT(X, S)$.

The generated regression tree is used to predict the performance of a given configuration. Each non-terminal node of the regression tree represents a feature-selection condition and determines whether the given configuration meets the condition or not. A performance prediction is acquired when a terminal node is reached. For example, given a configuration with feature selection $(x_{14} = 1, x_{10} = 0, x_8 = 1)$, the third leaf from the left of the regression tree of Figure 3.1 is matched. Thus, the predicted performance value of the configuration is 483 seconds.

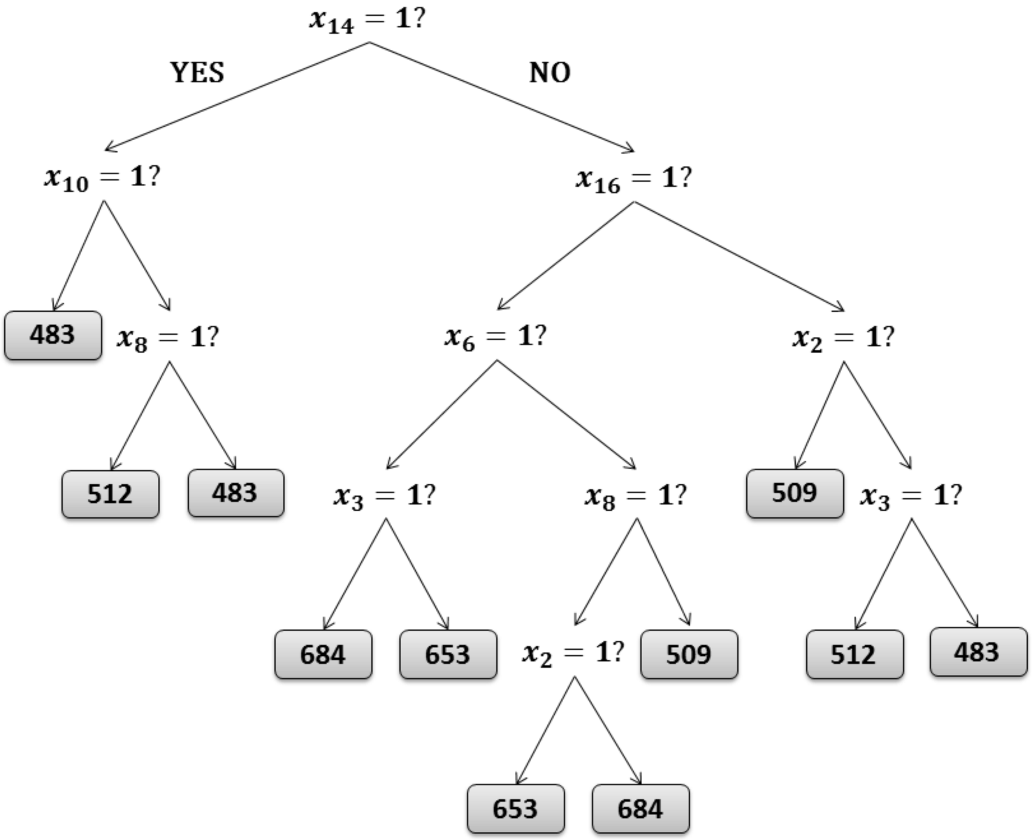

Figure 3.1: Regression tree built from x264 sample

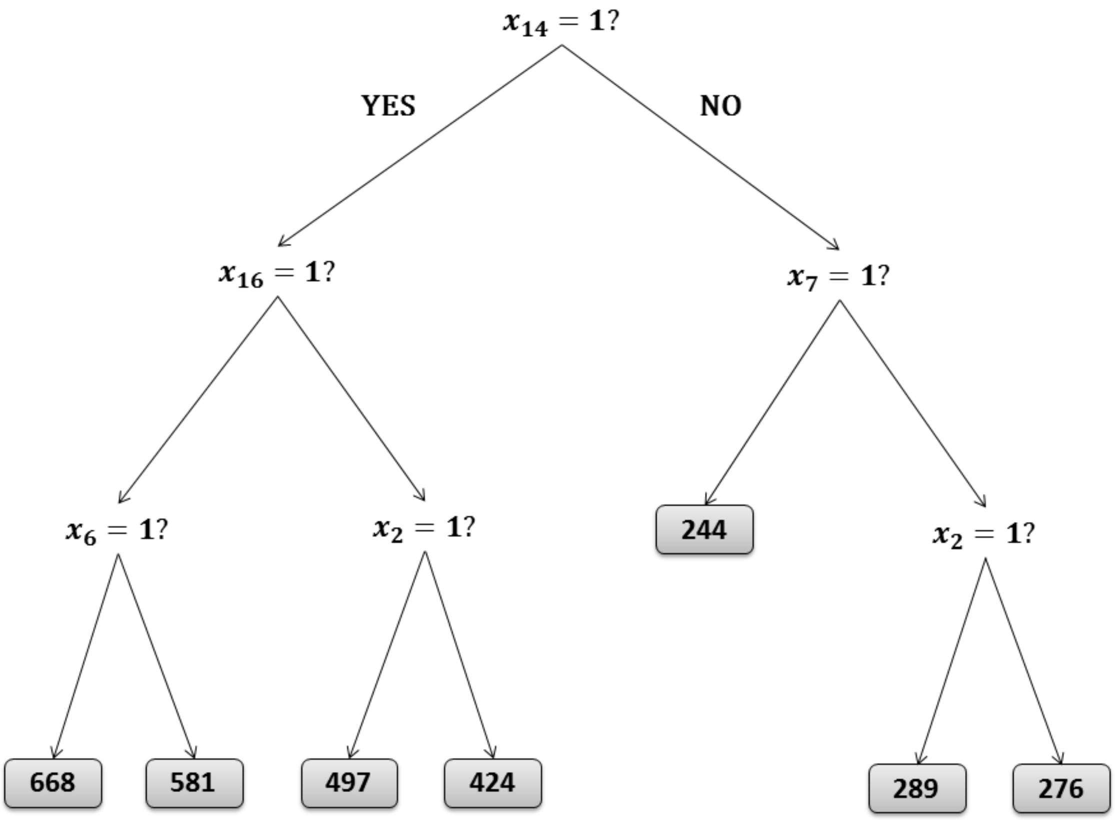

Figure 3.2: Example regression tree 1, generated by bagging from x264 sample

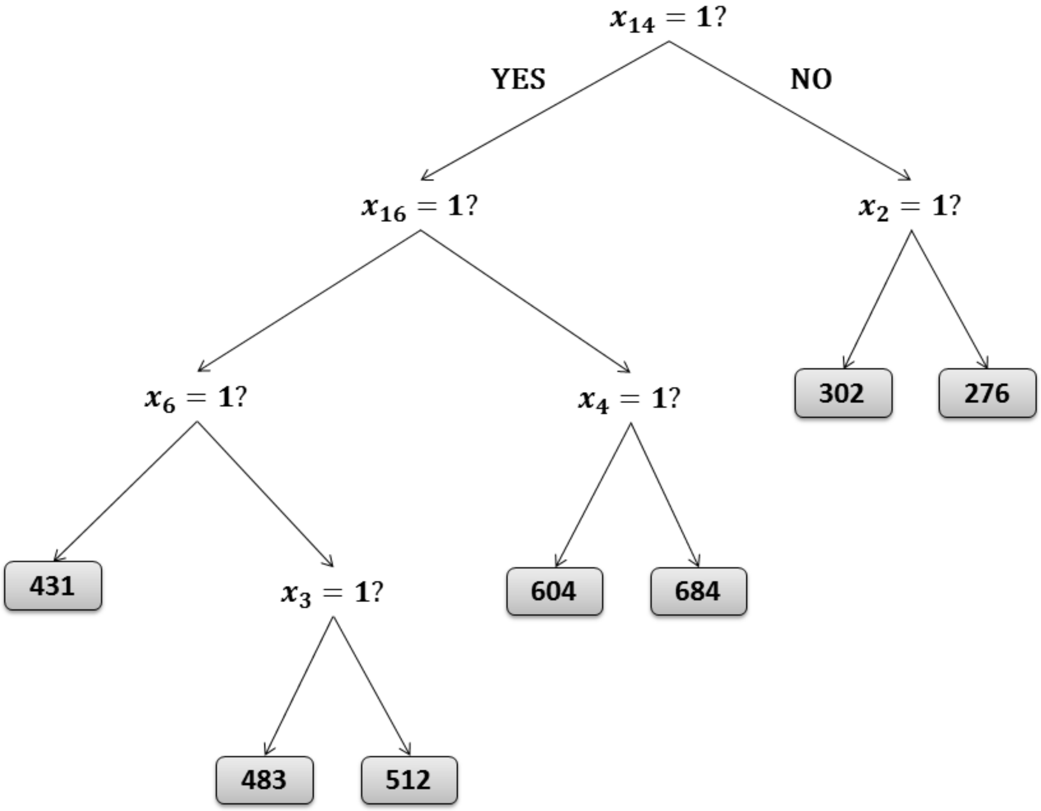

Figure 3.3: Example regression tree 2, generated by bagging from x264 sample

3.2 Bagging

Regression trees discussed in the previous section experience high variance[10], i.e. if we break the initial training set into several training subsets and generate a different regression tree out of each subset, the results provided by those trees might be quite different. Bootstrap aggregation, or Bagging, is a method for generating multiple versions of a given predictor and averaging over these versions when predicting a numerical outcome, in order to get a new, aggregated predictor[4]. Bagging of regression trees generates multiple regression trees and averages over predictions of all regression trees to produce the final result.

Bagging starts with the initial sample S and generates a set of *bootstrap* samples $\{S^B\}$. Each bootstrap sample S^B is produced by random sampling with replacement from the sample S, i.e., configurations along with their performance measurements are randomly chosen from S to form the bootstrap sample S^B and the same configuration may appear more than once in S^B.

Next, following the same procedure of CART, bagging builds a regression tree based on each individual bootstrap sample, and thus generates a set of regression trees $\{RT(X, S^B)\}$. Each regression tree produces a performance prediction for a given configuration. Then, bagging averages the performance predictions of all generated regression trees as the final performance prediction of the configuration.

Bagging of regression trees can be summarized as follows:

1. Prepare a sample $S = (\mathbf{X}_S, Y_S)$ from the configuration set $\mathbf{C} = (\mathbf{X}, Y)$
2. Draw with replacement a sequence of bootstrap samples $\{S^{(B)}\}$ from the sample $S = (\mathbf{X}_S, Y_S)$
3. Generate a sequence of regression trees $\{rt(\mathbf{x}, S^{(B)})\}$ from the sequence of bootstrap samples $\{S^{(B)}\}$
4. Obtain prediction results by averaging over all individual predictions of the sequence of regression trees $\{rt(\mathbf{x}, S^{(B)})\}$

For example, assume that we generate two regression trees in total, presented in Figure 3.2 and Figure 3.3. Given a configuration with feature selection $(x_{14} = 1, x_{16} = 1, x_6 = 0, x_3 = 0)$, we acquire two predictions: 581 seconds and 512 seconds. The final prediction given by bagging is the average of 581 and 512, i.e., 546.5 seconds.

3.3 Random Forest

Random forest[5] is another enhanced version of CART. Like bagging, random forest draws a set of bootstrap samples $\{S^{(B)}\}$ from initial sample S, and generates multiple regression trees from these bootstrap samples. The main difference with Bagging is that Random Forest introduces a subtle tweak to help decorrelate the generated regression trees[10]. This is done by modifying how regression trees are constructed. "Classical" regression trees, described previously, explore all features of a given configurable system at each split, to select the one that provides lowest training error. Random Forest on the contrary, instead of using all features of the configurable system, for each tree and at each split in that tree, uses only a random subset of features.

There is a rationale behind this approach. Let's suppose that out of all features X of a configurable system, feature x_{14} is the most useful for prediction. Because of that, in Bagging, most of the trees might start splitting the data sample using x_{14} feature, since most of the time this feature variable will provide the best reduction in prediction error. Therefore, generated regression trees might be very similar, or correlated, to each other. However, averaging many highly correlated quantities does not lead to reduction in variance as big as when averaging many uncorrelated quantities[10].

We denote regression trees in Random Forest as "randomized" regression trees, or $rrt(\mathbf{x}, S^{(B)})$ since only a random subset of features is used to build each one.

So the actual algorithm of Random Forest has the following structure[12]:

1. Prepare a sample $S = (\mathbf{X}_S, Y_S)$ from configuration set $\mathbf{C} = (\mathbf{X}, Y)$
2. Draw with replacement a sequence of bootstrap samples $\{S^{(B)}\}$ from the sample $S = (\mathbf{X}_S, Y_S)$
3. Generate a sequence of "randomized" regression trees $\{rrt(\mathbf{x}, S^{(B)})\}$ from the sequence of bootstrap samples $\{S^{(B)}\}$
4. Obtain prediction results by averaging over all individual predictions of the sequence of "randomized" regression trees $\{rrt(\mathbf{x}, S^{(B)})\}$

3.4 SVM Regression

Support Vector Machines (SVM) is a family of machine learning techniques used for classification and regression analysis. Different from CART, bagging, and random forest that build regression trees to carry out regression, SVM[25] creates a hyperplane and uses it for value approximation in regression problem. Using special functions called "kernels" to work with non-linear problems, SVM can map input into a higher-dimensional decision space and use hyperplane in that space for solving regression problem. Overall, SVM is considered to perform well on prediction problems that are non-linear and high-dimensional[27], which fits our variability-aware performance prediction problem.

We adopt ε-SV regression[25], which is the simplest SVM for regression. ε-SV regression creates a hyperplane $f(\mathbf{c})$ which has maximum deviation from each actual measured performance y_i specified by parameter ε, i.e. deviation of hyperplane from each performance value is not greater than ε.

The hyperplane in the current context is defined as flat affine subspace of the target space, where regression problem is being solved. If the target space is 2-dimensional one, then the hyperplane in that target space would be 1-dimensional flat affine subspace, i.e. a line. If the target space is 3-dimensional one, then the hyperplane in that target space would be 2-dimensional flat affine subspace, i.e. a plane. In our motivational example, x264 has 16 configurable features, therefore the target space is 17-dimensional one. Then the hyperplane in the target space would be 16-dimensional flat affine subspace.

The hyperplane used by ε-SV regression is defined as follows:

$$f(\mathbf{c}) = \langle \mathbf{w}, \mathbf{c} \rangle + b \tag{3.4}$$

where $\langle \cdot, \cdot \rangle$ denotes a dot product, $\mathbf{w}$ denotes a normal to the hyperplane, and b denotes a scalar. The hyperplane defined above is linear one and can be used for linear regression, as shown in Figure 3.4.

In Figure 3.4 actual regression hyperplane $f(\mathbf{c})$, that is used for value approximation, is surrounded by hyperplanes $f(\mathbf{c}) + \varepsilon$ and $f(\mathbf{c}) - \varepsilon$, that form ε-insensitive region, in which all performance measurements should fall. Performance measurements are represented by black crosses.

One way to use linear hyperplane $f(\mathbf{c})$ for a non-linear regression problem is to preprocess all original predictors in X using a mapping function $\Phi : X \to D$ into a decision space D, in which the linear hyperplane can be used. Moreover, since ε-SV regression depends only on dot products of the values in space D, it is enough to know only a kernel function $k(\mathbf{c_i}, \mathbf{c_j})$ for calculating dot products in space D, and to use it to build hyperplane in the decision space. For our experiments a general purpose Radial Basis kernel (or Gaussian kernel) is used:

$$k(\mathbf{c_i}, \mathbf{c_j}) = \exp(-\sigma \|\mathbf{c_i} - \mathbf{c_j}\|^2) \tag{3.5}$$

where σ is a tuning parameter called inverse kernel width.

Sometimes it is necessary to allow some measurements to lie beyond ε-boundary. In order to account for those measurements, the coefficient C is introduced to specify the cost of constraint violation, i.e., how far beyond ε-boundary measurements can lie.

Unfortunately it is not possible to visualize SVM for our motivational example, since x264 has 16 configurable features and SVM would build a hyperplane in a 17-dimensional space. But for the sake of illustration, let us suppose that using a special mapping function $\Psi : X \to X'$ we can convert our observations from 16-dimensional space to 1-dimensional space. In this case our regression problem will look similar to Figure 3.5. In order to carry out performance prediction using example regression hyperplane, one should take system configuration $\mathbf{c}$, map it to $\mathbf{c}'$ using function Ψ, and get performance prediction value from hyperplane, corresponding to $\mathbf{c}'$.

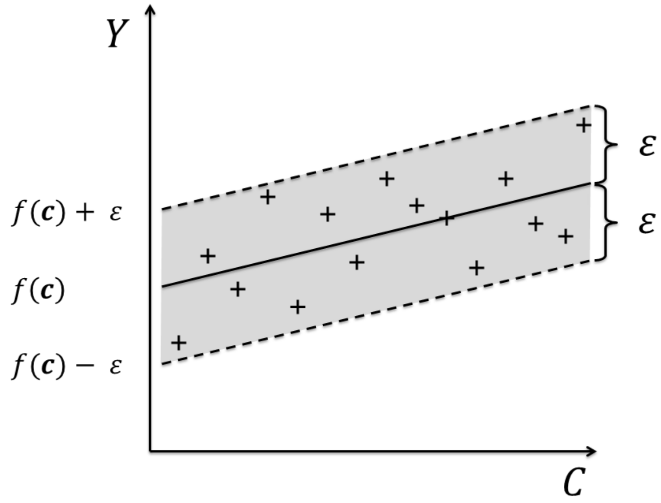
Y
$f(\boldsymbol{c}) + \varepsilon$
$f(\boldsymbol{c})$
$f(\boldsymbol{c}) - \varepsilon$
ε
ε
C

Figure 3.4: Example of linear ε-SV regression

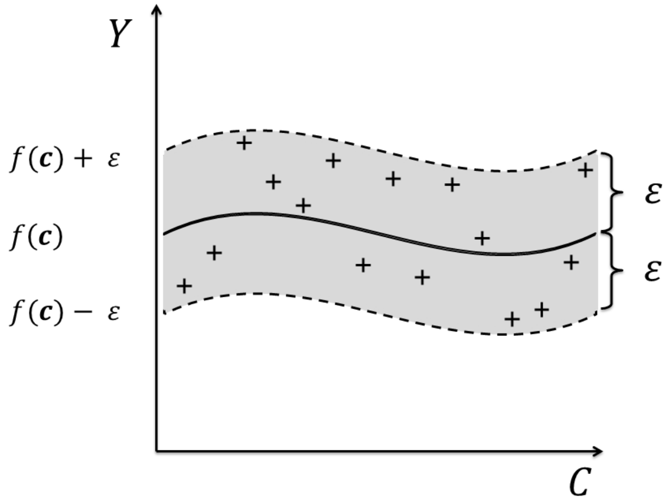
Y
$f(\boldsymbol{c}) + \varepsilon$
$f(\boldsymbol{c})$
$f(\boldsymbol{c}) - \varepsilon$
ε
ε
C

Figure 3.5: Example of non-linear ε-SV regression

Chapter 4

Parameter Tuning

Most of machine-learning algorithms encompass parameters - flags, arguments and other configuration data - that guide execution process of the algorithms[3]. For CART these parameters include minimal number of observations in terminal nodes and maximal depth of generated regression tree, for Bagging and Random Forest these parameters include number of regression trees to be generated and number of features to be considered in each split, while for SVMs these parameters include width of ε-insensitive region and cost of constraint violation by outliers. Usually, the parameters have a strong influence on the prediction accuracy of the algorithms. However in many studies parameters of machine-learning algorithms are chosen by researches empirically, relying on experience with particular algorithms, datasets or research problems.

There are many different techniques proposed for systematic parameter tuning. The simplest technique is called *grid search*[3]. Grid search works in the most straightforward way possible. It generates all possible combinations of parameters for a given algorithm. If parameters are continuous, grid search carries out discretisation of parameters with specified intervals. After generation of combinations is complete, grid search executes the algorithm using each and every combination, assessing algorithm's performance. When all combinations are assessed, grid search selects the one that produces the best prediction accuracy.

Different from grid search, *random search* generates a set of parameter combinations that are randomly selected according to sequences of random numbers[3]. There are different types of random numbers available for use. By definition, standard random numbers (SRNs) are independent observed values of a random variable γ that is uniformly distributed in the interval $0 < x < 1$[22]. Unfortunately, variable γ is an ideal mathematical abstraction that is not practically available[21], therefore for actual computer modelling pseudo-random and quasi-random numbers[28] can be used.

Numbers $\gamma_1, \gamma_2, \cdots$ are called pseudo-random if they are generated in terms of a specified formula, i.e. they are not actual SRNs, but satisfy certain properties of random numbers[22]. Different methods were created for generation of sequences of pseudo-random numbers like: general linear congruential generators, multiple recursive generators and generalised feedback shift registers.

Numbers $\delta_1, \delta_2, \cdots$ are called quasi-random if, while also being generated in terms of a specified formula or algorithm, they fill the space under consideration more uniformly[14], therefore quasi-random numbers are preferred to pseudo-random numbers in some applications. Therefore, quasi-random numbers are often recommended and used in systematic parameter tuning. Moreover, empirical studies have demonstrated that random search is more efficient than grid search for parameter tuning[3]. Various methods were created for generation of sequences of quasi-random numbers, for example: Halton sequences, Latin Hypercube Sampling and Sobol sequences.

Pseudo-random sequences are presented on Figure 4.1, Figure 4.2 and Figure 4.3 on page 17. Quasi-random sequences are presented on Figure 4.4, Figure 4.5, and Figure 4.6 on page 18. As we can see from these figures, pseudo-random sequences have areas of low and high density, while quasi-random sequences fill the space more evenly.

We use random search with quasi-random numbers for the parameter tuning of regression methods. Among many techniques that generate quasi-random numbers, we choose Sobol sampling[14], because it explicitly minimizes the density differences across samples and covers the parameter space more evenly. Furthermore, Sobol sampling has been recommended for parameter sensitivity analysis[16].

For implementation of Sobol sequences in our code, an R package called randtoolbox was used[7]. In order to generate different Sobol sequences, Owen type scrambling with a specified random seed was used.

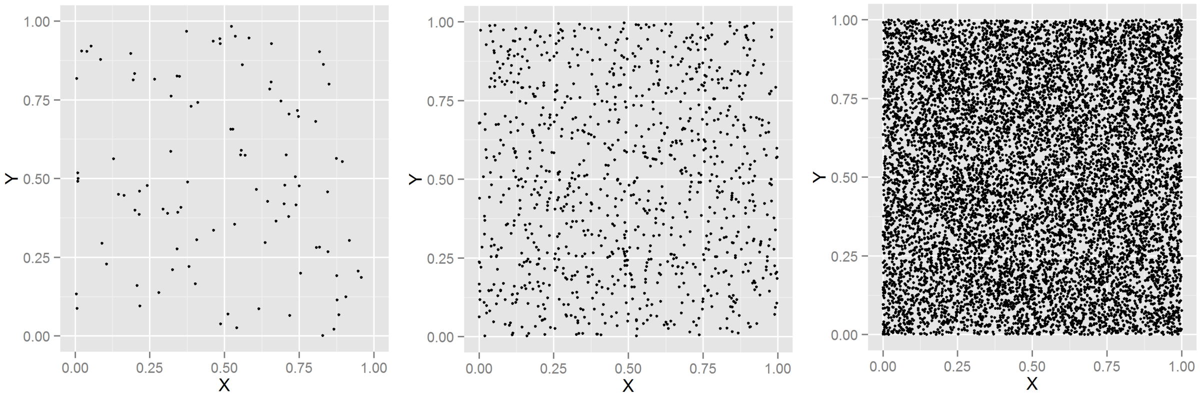

Figure 4.1: Example pseudo-random sequences, generated using linear congruential generator, for 10^2, 10^3 and 10^4 samples

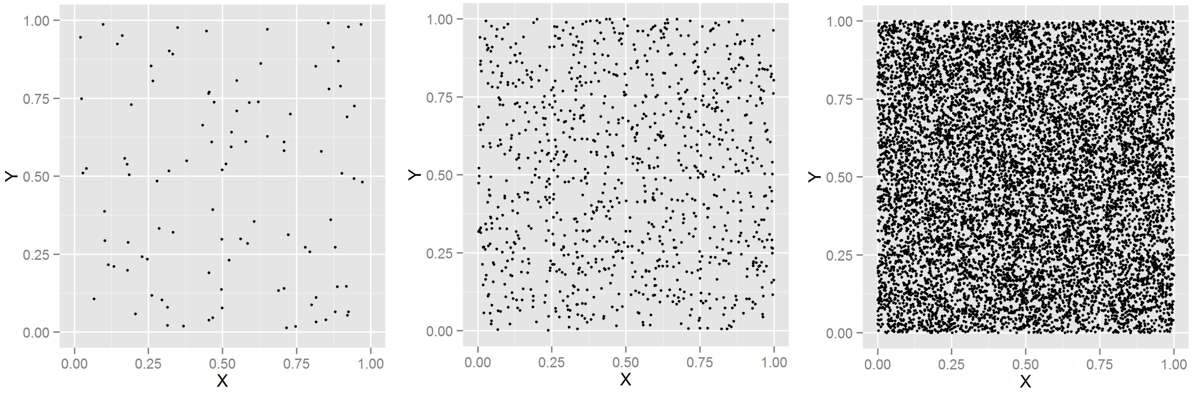

Figure 4.2: Example pseudo-random sequences, generated using Knuth multiple recursive generator, for 10^2, 10^3 and 10^4 samples

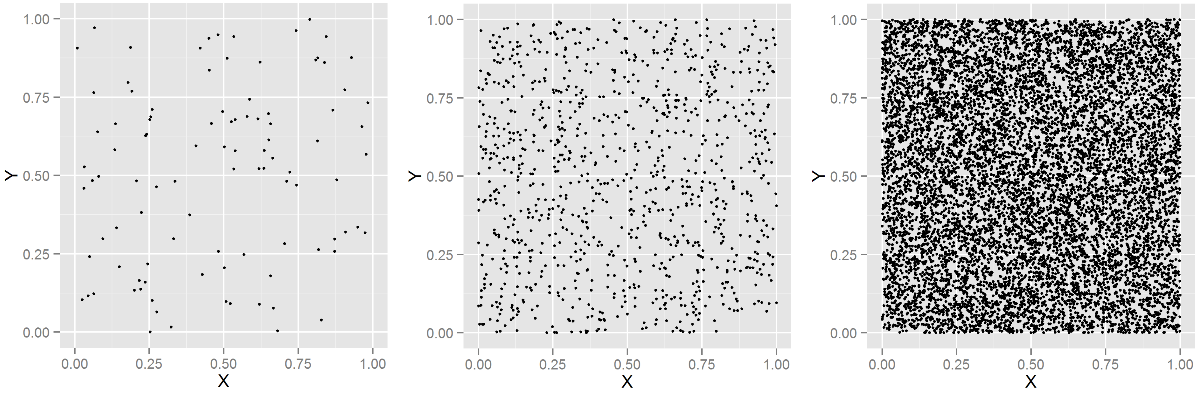

Figure 4.3: Example pseudo-random sequences, generated using Mersenne Twister 2002 generator, for 10^2, 10^3 and 10^4 samples

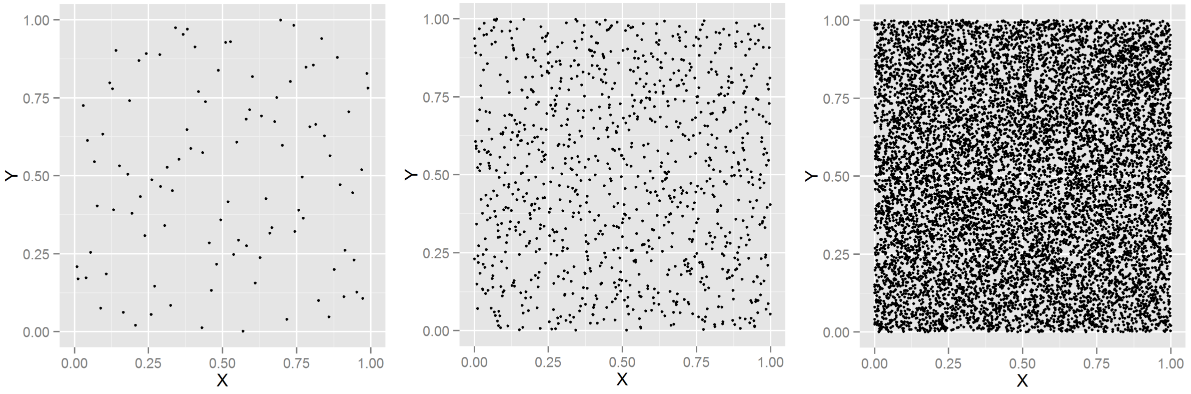

Figure 4.4: Example quasi-random sequences, generated using Latin Hypercube algorithm, for 10^2, 10^3 and 10^4 samples

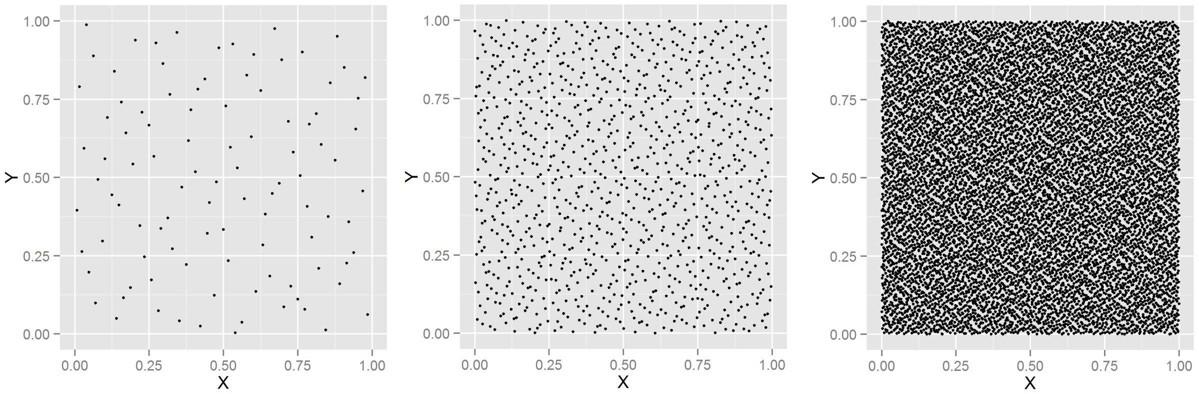

Figure 4.5: Example quasi-random sequences, generated using Halton algorithm, for 10^2, 10^3 and 10^4 samples

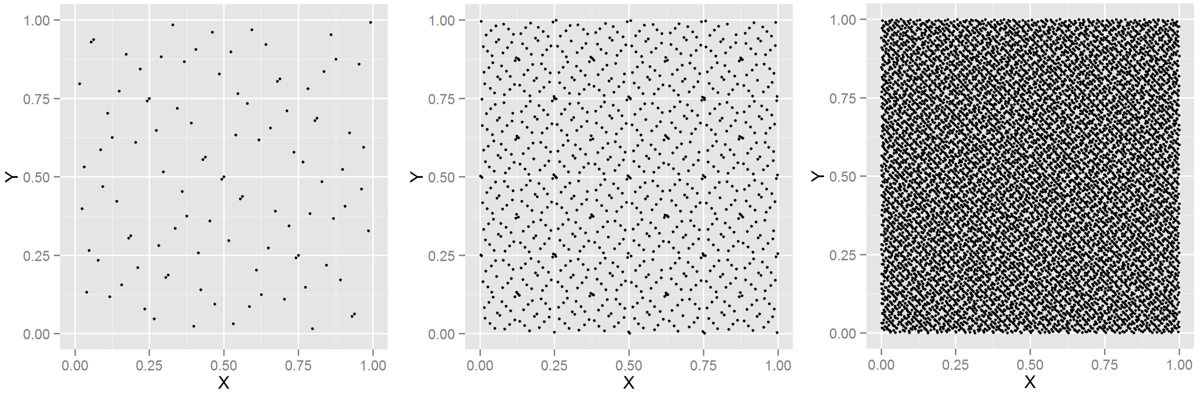

Figure 4.6: Example quasi-random sequences, generated using Sobol algorithm, for 10^2, 10^3 and 10^4 samples

Chapter 5

Implementation and Parameter Settings

We implement the regression methods for variability-aware performance prediction in R 3.0.1[15]. R is an open-source language and environment that provides many packages for implementation of statistical learning (classification and regression, linear and non-linear modelling, clustering and time-series analysis, etc.) and graphical (well-designed publication quality plots, mathematical symbols and formulas, etc.) techniques.

We use packages RPART[24], RANDOMFOREST[12] and KERNLAB[11] for regression analysis, and adopt package RANDTOOLBOX[7] for Sobol sampling. To implement each regression method, we consider a number of key parameters for parameter tuning and sensitivity analysis, which will be explained in the following sections.

Almost every parameter of all regression methods, described in this work, has extreme values that for certain lower predictions accuracy of the method. Therefore, when performing parameter tuning or sensitivity analysis of our regression methods, we perform initial empirical testing of parameter values to identify reasonable parameter range of values for tuning.

5.1 CART

We implement CART using R package RPART[24]. RPART provides functionality for recursive partitioning of data and creation of regression trees. For parameter tuning and parameter sensitivity analysis we consider the following key parameters of CART:

- **minsplit** - controls the minimum amount of observations that must exist in a tree node in order for further partitioning. In our experiments, *minsplit* varies in the integer value range $[2, 10]$. Lower bound is set to the lowest possible value for this parameter - 2. Upper bound is set to 10 to guarantee creation of non-trivial regression trees, i.e. trees that have more than one node. Setting upper bound any higher is not desirable since we are using a small sample of observations.
- **minbucket** - specifies the minimum amount of observations that must be present in any leaf of a regression tree. In our experiments, *minbucket* varies in the integer value range $[2, 10]$. Lower bound is set to 2 in order to allow complex trees with many terminal nodes to be grown. However, setting lower bound to 1 might cause overfitting of our regression models to training samples. Upper bound is set to 10, because setting it any higher won't make any difference due to a small number of training observations.
- **maxdepth** controls the maximum depth of a regression tree. In our experiments, *maxdepth* varies in the integer value range $[2, 20]$. Lower bound is set to 2 in order to allow non-trivial regression trees to be grown. Setting upper bound higher than 20 is not needed due to a small number of training observations.
- **cp** is complexity parameter that controls optimal size of a regression tree. *cp* specifies the minimum improvement in prediction accuracy that each split in a regression tree should provide. If splitting a node does not produce prediction improvement higher than *cp*, splitting will not happen. In our experiment, *cp* varies in the real value range $[0.0001, 0.01]$, which is chosen by preliminary empirical tests to provide best values.

5.2 Bagging and Random Forest

We implement Bagging and Random Forest using R package RANDOMFOREST[12]. We consider the following key parameters for tuning and sensitivity analysis:

- **ntree** - specifies the number of trees that will be created. In our experiments, *ntree* varies in the integer value range $[2, 20]$. Lower bound is set to 2 to allow averaging over different regression trees. Upper bound is set to 20, because, as empirical estimation has shown, setting it any higher doesn't provide any improvement to prediction accuracy.
- **nodesize** - controls the minimum amount of observations in any terminal node of a regression tree. This parameter is analogous to the *minbucket* parameter of CART. In our experiments, *nodesize* varies in the integer value range $[2, 10]$. *Minsplit*

parameter is not directly accessible in RANDOMFOREST package and is calculated automatically.

- **mtry** - specifies the number of feature-selection variables that are considered when creating a regression tree. This parameter indicates the key difference between bagging and random forest. For bagging, *mtry* is fixed to the number of all feature-selection variables, i.e., $|X|$. For random forest, *mtry* varies in the integer value range $[|X|/3, |X|/2]$. The lower bound is set to default *mtry* value for regression. The upper bound is set to a value chosen by our preliminary experiments. Setting upper bound to a larger number will make Random Forest results very similar to Bagging.

5.3 SVM Regression

We implement SVM using R package KERNLAB[11]. The parameters we consider for tuning and sensitivity analysis are as follows:

- **epsilon**, i.e., ε - specifies the width of error-insensitive boundary of a SVM hyperplane for regression. In our experiments, the parameter varies in the real value range $[0.01, 1]$. Those values were chosen by our preliminary experiments.
- **sigma**, i.e. σ specifies inverse kernel width for Radial Basis Kernel. KERNLAB provides a special function to automatically tune *sigma* parameter. Nevertheless, in our experiment, we manually vary *sigma* in the real value range $[0.01, 1]$. This range is based on our empirical observations of best values for *sigma*.
- **C** represents the cost of constraint violation. In our experiments, C varies in the real value range $[1, 100]$. This interval is also based on preliminary experiments.

Chapter 6

Evaluation

We conducted a series of experiments to evaluate the four regression methods for variability-aware performance prediction. We compared the prediction accuracy of the regression methods and analysed their parameter sensitivity.

6.1 Subject Systems

We chose a publicly available dataset deployed with the SPLConqueror tool[17]. This dataset contains six real-world configurable systems with different sizes, implementation languages, configuration mechanisms and application domains. The dataset includes all configurations of each system along with their performance measurements (the exception is SQLite, for which the dataset contains 4,553 configurations for prediction modelling and 100 additional random configurations for prediction evaluation[18]). Table 6.1 presents a summary of configurable systems included in the dataset.

Performance of these subject systems was measured using standard benchmarks, either provided by vendors or used widely in the corresponding application domains. For example, to measure the performance of Berkeley DB C, Oracle's standard benchmark was used, while autobench and httperf were used for measuring the performance of Apache Web Server[18].

System	Lang.	LOC	C	N
Apache	C	230,277	192	9
LLVM	C++	47,549	1,024	11
x264	C	45,743	1,152	16
Berkeley DB	C	219,811	2,560	18
Berkeley DB	Java	42,596	400	26
SQLite	C	312,625	3,932,160	39

Table 6.1: Overview of investigated systems; Lang. - Language; LOC - Lines of code; C - number of all valid configurations; N - number of all features

6.2 Experimental Setup

Our experiment can be divided into three main steps: data preparation, regression method preparation, and regression analysis. During data preparation phase, data is loaded and random samples for analysis are generated. To assess effectiveness of our approach we use small random samples of different sizes to train our regression methods. We generate samples that are linear in the number of system features, which is reasonable in practice if cost of performance measurement is high. Throughout method preparation, method parameters are analysed and parameter combinations for analysis are selected. Finally, using generated random samples and parameter combinations, regression analysis is performed.

In detail the whole experiment has the following structure:

1. Data preparation.
   (a) We load data for each subject system and calculate the number of involved features N.
   (b) As described before, we generate five sampling sizes in the form of $T * N$, where T is the training coefficient with value from 1 to 5 and N is the number of features in each system. For example x264 has 16 features, therefore 5 different sampling sizes for x264 will be generated: 16, 32, 48, 64, 80.
   (c) For each configurable system (e.g. x264) we generate 10 random samples of each of the generated sampling sizes (e.g. 16, 32, 48, 64, 80) from the original dataset.
2. Regression method preparation.
   (a) We identify a set of parameters for each regression method. Thorough description of parameters for each method under study is provided in Chapter 5. For example, for CART method, **minsplit**, **minbucket**, **maxdepth** and **cp** are

selected.

   (b) For each parameter of a certain regression method, we set a range of values for parameter tuning; For example, for CART method, value range for **minsplit** parameter will be $[2, 10]$;
   (c) We use Sobol sampling to generate 1000 quasi-random numbers and convert them to the corresponding random parameter settings. For example, for CART method, for **minsplit** parameter, Sobol sampling will provide value 0.875 which will be converted to 8, a value from **minsplit** range $[2, 10]$.

3. Regression analysis.
   (a) From the 1000 randomly-generated parameter settings, we choose one to initialise each regression method. For example, all parameters of CART method will be initialised with values that Sobol sampling provided, e.g.: **minsplit** - 8, **minbucket** - 9, **maxdepth** - 17, **cp** - 0.002;
   (b) For a certain parameter setting of each regression method, we run the regression method 10 times and calculate the relative error of each run using the following formula:

$$Relative\ Error = \left| \frac{actual - predicted}{actual} \right| \times 100\%$$

   The relative error is the relative difference between the actual measured performance value and the predicted performance value. For example, if the actual measured performance of a given configuration is 536 and predicted performance is 512, then relative error will be calculated as follows:

$$Relative\ Error = \left| \frac{536 - 512}{536} \right| \times 100\% = 4.5\%$$

   (c) We average over 10 different relative errors of all runs of selected regression method (e.g. CART), selected parameter settings, and produce the average relative error.
   (d) We repeat steps (a)-(c) for all studied configurable systems, sampling sizes, regression methods and generated random parameters of each regression method. We aggregate all acquired relative errors into a single results Table 6.2 on page 31.
   (e) We analyse prediction accuracy and sensitivity to parameter settings of each regression method.

6.3 Results and Discussion

In this section, we present our experimental results. We compare the prediction relative errors of the regression methods and analyse their parameter sensitivity.

6.3.1 Prediction Relative Errors

Table 6.2 lists experimental results, showing prediction relative errors for different configurable systems, sampling sizes, regression methods and parameter settings. For each configurable system, we present prediction results for five different sampling sizes: N, $2 * N$, $3 * N$, $4 * N$, and $5 * N$; where N is the total number of features in the system. For each regression method, we perform Sobol sampling to choose 1000 parameter settings, and then present the prediction results using the best, average, and worst parameter settings. Best parameter settings are those that provided the smallest relative error for given regression method and sampling size, and therefore the best result. Worst parameter settings are those that provided the biggest relative error for regression method and sampling size, and therefore the worst result. Average parameter settings are those that provided the closest relative error to the mean relative error of all 1000 parameter settings for given regression method and sampling size. We regard a relative error below 10%, i.e., an accuracy above 90%, as an acceptable prediction.

CART

CART overall produces good prediction results. For Apache system, CART reaches a relative error of 8.2% using a sample of $3 * N$ measured configurations with the best parameter setting, and 9.8% using $5 * N$ measured configurations with the average parameter setting. For Berkeley DB C system, CART is not able to produce an acceptable relative error using any experimental setting. For Berkeley DB Java system, CART reaches 3% relative error using only N measured configurations with the best parameter setting, 8.1% using $2 * N$ configurations for the average setting, and 3% using $3 * N$ configurations for the worst setting. For LLVM and SQLite, CART gains acceptable relative errors using a small sample of only N measured configurations with either the best, the average, or the worst parameter setting. For x264, CART reaches relative error 7.3%, 8.4%, and 9.4% using $2 * N$, $2 * N$, $3 * N$ measured configurations with the best, the average, and the worst parameter setting, respectively.

CART results are presented in the Table 6.2 and in the Figure 6.1.

Bagging

Bagging overall provides the best prediction results. For Apache system, bagging produces relative error 7.5%, 8.5%, and 8.5% using $3 * N$, $4 * N$, and $5 * N$ measured configurations with the best, the average, and the worst parameter setting, respectively. For Berkeley DB C system, bagging reaches a relative error of 6.1% using $5*N$ measured configurations with the best parameter setting. For Berkeley DB Java system, bagging reaches relative error 6.9%, 3.3%, and 2.6% using N, $2 * N$ and $3 * N$ measured configurations with the best, the average, and the worst parameter setting, respectively. For LLVM and SQLite, bagging reaches acceptable relative errors using only N measured configurations with either the best, the average, or the worst parameter setting. For x264 system, bagging reaches relative error 9.5%, 8%, and 8.4% using N, $2 * N$ and $3 * N$ measured configurations with the best, the average, and the worst parameter setting, respectively.

Bagging results are presented in the Table 6.2 and in the Figure 6.2.

Random Forest

Random forest overall provides decent prediction results. For Apache system, random forest reaches a relative error of 8.8% using $5 * N$ measured configurations with the best parameter setting. For Berkeley DB C system, random forest is not able to produce an acceptable relative error with any experimental setting. For Berkeley DB Java system, random forest reaches a relative error of 3.1% using $2 * N$ measured configurations with the best parameter setting. For LLVM and SQLite, random forest reaches acceptable relative errors using only N measured configurations with either the best, the average, or the worst parameter setting. For x264 system, random forest reaches a relative error of 7.7% using $2 * N$ measured configuration with the best parameter setting.

Random Forest results are presented in the Table 6.2 and in the Figure 6.3

SVM Regression

SVM overall provides the worst prediction results. For Apache, Berkeley DB C and Berkeley DB Java systems, SVM is not able to achieve an acceptable relative error using any experimental setting. For LLVM and SQLite, SVM reaches acceptable relative errors using only N measured configurations with either the best, the average, or the worst parameter setting. For x264 system, SVM produces a relative error of 9.2% using $3 * N$ measured configurations with the best parameter setting.

SVM results are presented in the Table 6.2 and in the Figure 6.4.

Summary

According to the experimental results, different subject systems may make or break variability-aware performance prediction using regression analysis. For example, all regression methods can produce acceptable prediction results for systems LLVM and SQLite. However, for Berkeley DB C, only bagging is able to provide acceptable prediction results when using samples of size less than $5 * N$. We suspect that these anomalies can be explained by investigating feature interactions that are present in different configurable systems and their influence on performance prediction.

We average the prediction relative errors of each regression method on six subject systems, and present the results at the bottom of Table 6.2. Overall, bagging provides the lowest prediction relative errors in most experimental settings. With the best parameter setting, the smallest sample size needed for bagging to produce acceptable predictions is $3*N$; but with the average parameter setting, the smallest sample size is $4*N$. Only using a sample of N and $2*N$ measured configurations with the best parameter setting, CART outperforms other regression methods. Moreover, we observe a stable decreasing relative error with the increasing sample size of each regression method. That is, acquiring more measurements always helps improving the prediction accuracy of regression methods.

6.3.2 Sensitivity Analysis

Figures 6.6, 6.7, 6.8, 6.9, 6.10 present the distributions of prediction relative errors of each regression method with 1000 parameter settings for different subject systems and different sample sizes. The 1000 parameter settings are generated by Sobol sampling to cover the parameter space of each regression method evenly. We put the experimental results of five subject systems together for convenient comparison. Experimental results for Berkeley DB C are presented in an independent Figure 6.10, as the relative errors for Berkeley DB C have a far larger ranges than those for other subject systems.

To analyse the parameter sensitivity of each regression method, we intuitively observe the distributions of relative errors of each regression method for each system and for each sample size, and we analyse the change trend of the distribution ranges, such as the range between the best case and the worst case and the range between the second quartile and the third quartile, with the increasing sample sizes. As the sample sizes increase, for CART and bagging, we observe a rapid decrease in the distribution ranges of relative errors for Berkeley DB C, Berkeley DB Java and x264 systems, while the ranges remain stable for Apache, LLVM and SQLite systems. However, bagging reduces the distribution ranges faster than CART for Apache and Berkeley DB Java. For random forest, we do not see a substantial decrease in the distribution ranges of relative errors for all subject systems. For SVM, we observe an increase in the distribution ranges for Apache, Berkeley DB Java, LLVM and x264 systems. In general, if a regression method has a larger distribution range of relative errors, then it is more sensitive to parameters, because a random parameter change may cause a significant change in prediction accuracy. Therefore, according to the experimental results and intuitive analysis, bagging is identified as the most "stable" method that is quite insensitive to parameter settings when using a sample with enough measured configurations. If the input sample is very small, such as N, each regression method may present non-negligible parameter sensitivity.

6.3.3 Threats to Validity

To increase internal validity of our work, we performed automated sampling. All training samples (from size N to $5 * N$) were selected randomly from the whole population of configurations for each subject system. The rest of the whole population was taken as testing samples. For each configurable system and each sampling size, we repeated random sampling ten times.

We are aware that more parameter settings would cover the parameter space more evenly. However, finding more parameter setting comes at a cost: more prediction effort. Hence, we use Sobol sampling to generate 1000 parameter settings for each regression method. Moreover, more parameter tuning methods, such as Bayesian optimization, have been proposed. We may compare different parameter tuning methods, which will be explored as future work.

To enhance external validity, we performed our experiments using six real-world software systems of different sizes, different implementation languages, and different application domains. However, we are aware that the results of our experiments are not automatically transferable to all other software systems.

6.3.4 Related Work

Thereska et al.[23] proposed a model-based performance prediction approach AppModel and applied it for performance modelling of popular client applications, such as Microsoft Office Suite, Visual Studio, and Media Player. Systems under consideration were instrumented with performance monitoring capabilities, and all performance data were collected from real-world deployments. AppModel analyses the collected performance data, and creates a performance prediction model using CART. However, AppModel works only with hardware deployment parameters, such as CPU speed, memory size, network configuration; it does not consider software configuration options, i.e., features. In contrast, we consider software features and use different regression methods to build performance models.

Westermann et al.[26] proposed an approach that builds prediction models of expected accuracy using the least possible number of measurements. Their approach uses three measurement-point-selection algorithms for exploring the parameter space and two validation strategies for assessing the performance of prediction model. Their approach assumes that all features involved in the prediction models are performance-relevant, while our work considers all features of a software system.

Siegmund et al.[19] proposed SPLConqueror that automatically detects performance-relevant feature interactions using specific sampling heuristics that meet different feature-coverage criteria. On the contrary, the regression methods use random samples as basis and avoid the effort of detecting feature interactions.

This paper extends our previous work[8] using CART for variability-aware performance prediction. We add three popular regression methods and compare the prediction accuracy of all regression methods.

Arcuri et al.[1] investigated parameter tuning in search-based software engineering. They conducted empirical case studies on test data generation and developed a tool called EvoSuite. They identified five parameters of EvoSuite for tuning. For each parameter, a set of representative parameter values were chosen empirically. Different from their approach, we define a value range for each parameter and then use Sobol sampling to randomly choose a representative sample of parameters that cover the entire parameter space evenly.

System	$\|S\|$	Regression Methods											
		CART			Bagging			Random Forest			SVM		
		Best	Average	Worst	Best	Average	Worst	Best	Average	Worst	Best	Average	Worst
Apache	N	21 ± 6.3	28.5 ± 5.7	30.5 ± 5.7	**20.9 ± 5.2**	**23.3 ± 4**	**24.2 ± 4.5**	22.6 ± 2.4	28.2 ± 5	30.7 ± 5.5	24.2 ± 3.1	25.1 ± 2.9	25.7 ± 2.8
	2 * N	13.6 ± 6.1	19.9 ± 3.6	29.6 ± 3.3	**13.1 ± 4.5**	**17.7 ± 3.3**	**21.8 ± 3.3**	15.2 ± 4.4	24.6 ± 2.7	30.7 ± 3.8	20.8 ± 1.1	22.5 ± 1.2	23.8 ± 1.2
	3 * N	8.2 ± 1	14.3 ± 2.4	18.9 ± 0.5	**7.5 ± 1.4**	**10.5 ± 1.8**	**18.2 ± 2.9**	11.4 ± 2.2	21.5 ± 4.2	28.3 ± 2.8	19.6 ± 1.3	22.2 ± 1.4	24.1 ± 1.6
	4 * N	7.9 ± 1.1	11.8 ± 3	18.6 ± 1.1	**7 ± 1.3**	**8.5 ± 1.3**	**12.9 ± 3**	10.3 ± 1.8	21.1 ± 2.1	29.6 ± 3.3	18.3 ± 1.4	21.6 ± 1.3	24 ± 1.6
	5 * N	7.1 ± 0.6	9.8 ± 3.5	17.4 ± 1.9	**6.2 ± 0.6**	**7.2 ± 0.6**	**8.5 ± 1.1**	8.8 ± 1	19.5 ± 2	28.5 ± 4.5	17 ± 1	20.8 ± 1	23.9 ± 1.1
BerkeleyC	N	**135.8 ± 90.5**	296.5 ± 464.4	700 ± 294.7	183.9 ± 122.3	**276.5 ± 134.6**	**434.4 ± 339.7**	278.2 ± 134.3	521 ± 175.7	700.2 ± 293.4	254.4 ± 353.1	496.1 ± 209.2	680.8 ± 274.7
	2 * N	**48.9 ± 67.7**	136 ± 93.6	169.8 ± 92.1	65.3 ± 45.7	**117.3 ± 78.2**	**159.6 ± 83.8**	124.3 ± 49.8	390.4 ± 141.9	645.1 ± 176.5	123 ± 34.8	409.1 ± 115.4	646.4 ± 177.5
	3 * N	40.9 ± 68.7	138.9 ± 90.6	188.1 ± 85.7	**28.9 ± 22.4**	**66.2 ± 46.6**	**121.7 ± 79.4**	100.4 ± 44.8	329.9 ± 109.5	612.1 ± 177.8	117.5 ± 42.4	361.6 ± 70.4	609.2 ± 134.8
	4 * N	15.7 ± 4.3	84.4 ± 79.9	144.3 ± 79.8	**10.7 ± 2.3**	**32 ± 24.3**	**84.5 ± 69.2**	62.5 ± 18.1	292 ± 94.5	632 ± 149.1	100.3 ± 20.7	329.6 ± 57.9	576.2 ± 98.1
	5 * N	14.6 ± 4.2	65.3 ± 82.2	130.9 ± 71	**6.1 ± 2.4**	**21.8 ± 13**	**48.7 ± 63.6**	50.8 ± 22.6	260.7 ± 52	619.2 ± 130.6	89.9 ± 16.1	297.1 ± 30.1	541.5 ± 50
BerkeleyJ	N	**3 ± 0.3**	24.4 ± 2.3	35.3 ± 5	6.9 ± 6.2	**19.3 ± 6.7**	**30.4 ± 6.4**	12.6 ± 7.4	30.9 ± 5.9	42.2 ± 9.7	30.3 ± 1.1	31.8 ± 1.2	32.5 ± 1.2
	2 * N	2.2 ± 0.4	8.1 ± 10.2	25.6 ± 2.9	**1.9 ± 0.2**	**3.3 ± 2.1**	**10 ± 5.7**	3.1 ± 1.3	25 ± 3.4	49.2 ± 10.5	28.1 ± 1.2	31.2 ± 1.3	32.7 ± 1.4
	3 * N	2 ± 0.2	2.7 ± 0.5	3 ± 0.2	**1.6 ± 0.2**	**2 ± 0.3**	**2.6 ± 0.2**	2.7 ± 0.7	19.3 ± 6.1	46.2 ± 5	25.9 ± 1.4	30.3 ± 1.3	32.4 ± 1.2
	4 * N	1.9 ± 0.2	2.6 ± 0.5	3 ± 0.2	**1.4 ± 0.2**	**1.7 ± 0.2**	**2.1 ± 0.4**	2.2 ± 0.4	16.6 ± 2.2	48.9 ± 7.6	23.9 ± 1.9	29.3 ± 1.9	32 ± 2
	5 * N	2 ± 0.3	2.6 ± 0.5	3 ± 0.2	**1.4 ± 0.3**	**1.6 ± 0.4**	**1.8 ± 0.4**	1.9 ± 0.3	15.4 ± 3.1	46.7 ± 4.7	22 ± 3.4	28.5 ± 3.5	31.9 ± 3.6
LLVM	N	5.7 ± 0.6	6.1 ± 0.5	6.3 ± 0.2	4.9 ± 0.9	**5.3 ± 0.8**	**5.9 ± 0.9**	4.9 ± 0.7	5.9 ± 0.6	6.4 ± 0.2	**4.3 ± 0.6**	**5.3 ± 0.5**	6.4 ± 0.3
	2 * N	4.1 ± 0.8	5.4 ± 0.7	6.1 ± 0.7	4 ± 0.6	4.6 ± 0.6	**5.8 ± 0.4**	4 ± 0.4	5.4 ± 0.5	6.3 ± 0.2	**2.8 ± 0.5**	**4.4 ± 0.4**	6.3 ± 0.3
	3 * N	3.1 ± 0.7	4.6 ± 0.4	5.4 ± 0.2	2.9 ± 0.3	**3.6 ± 0.6**	**4.5 ± 0.5**	3.2 ± 0.3	4.8 ± 0.3	5.9 ± 0.4	**2.2 ± 0.1**	3.7 ± 0.2	6.2 ± 0.2
	4 * N	2.7 ± 0.4	3.9 ± 0.6	4.9 ± 0.5	2.5 ± 0.3	**3.1 ± 0.4**	**3.9 ± 0.5**	3 ± 0.3	4.5 ± 0.6	6 ± 0.4	**2.1 ± 0.1**	3.4 ± 0.3	6.1 ± 0.2
	5 * N	2.3 ± 0.4	3.5 ± 0.4	4.5 ± 0.5	2.2 ± 0.3	**2.7 ± 0.5**	**3.5 ± 0.5**	2.7 ± 0.4	4.3 ± 0.3	5.9 ± 0.4	**1.9 ± 0.1**	3.1 ± 0.2	6 ± 0.1
Sqlite	N	4.5 ± 0.3	4.7 ± 0.2	5.1 ± 0.6	4.4 ± 0.2	4.6 ± 0.2	5.2 ± 0.2	**4.3 ± 0.2**	**4.5 ± 0.3**	**5 ± 0.3**	**4.3 ± 0.2**	**4.5 ± 0.1**	5.3 ± 0.4
	2 * N	4.4 ± 0.3	4.6 ± 0.3	4.9 ± 0.5	4.2 ± 0.3	**4.3 ± 0.3**	4.8 ± 0.3	**4.1 ± 0.2**	**4.3 ± 0.2**	**4.7 ± 0.2**	4.2 ± 0.2	4.4 ± 0.1	**4.7 ± 0.3**
	3 * N	4.1 ± 0.1	4.3 ± 0.3	**4.5 ± 0.2**	3.9 ± 0.1	**4 ± 0.1**	**4.5 ± 0.3**	**3.8 ± 0.1**	4.1 ± 0.1	**4.5 ± 0.2**	4 ± 0.1	4.3 ± 0.2	4.6 ± 0.1
	4 * N	4 ± 0.2	4.1 ± 0.3	**4.4 ± 0.2**	3.8 ± 0.1	**3.9 ± 0.2**	4.5 ± 0.3	**3.7 ± 0.1**	4 ± 0.1	4.5 ± 0.1	3.9 ± 0.1	4.3 ± 0.2	4.6 ± 0.1
	5 * N	3.9 ± 0.2	4 ± 0.3	4.4 ± 0.2	**3.6 ± 0.1**	**3.8 ± 0.1**	**4.3 ± 0.2**	**3.6 ± 0.1**	3.9 ± 0.2	4.4 ± 0.1	3.9 ± 0.1	4.3 ± 0.3	4.5 ± 0.1
x264	N	11.6 ± 3.3	21.8 ± 9.6	34 ± 2.8	**9.5 ± 1.3**	**12.3 ± 3.5**	**17 ± 4.7**	11.4 ± 2.4	24.6 ± 3.1	33.8 ± 2.3	21.8 ± 2.8	30.1 ± 2.9	34.1 ± 3.1
	2 * N	7.3 ± 1.1	10.2 ± 2.3	14.8 ± 4.6	**6.5 ± 1.1**	**8 ± 0.9**	**10 ± 2.1**	7.7 ± 1.2	20 ± 2.1	32.5 ± 3.7	13.8 ± 1.8	26.9 ± 2	34.4 ± 2.1
	3 * N	6 ± 1.5	8.4 ± 0.8	9.4 ± 0.4	**5.2 ± 1.3**	**6.7 ± 1**	**8.4 ± 0.7**	5.9 ± 1	17.1 ± 3.5	32.3 ± 3.6	9.2 ± 1	24.7 ± 2.8	34.9 ± 2.5
	4 * N	4.3 ± 0.8	7.6 ± 0.5	9.1 ± 0.4	**3.8 ± 0.5**	**5.3 ± 0.6**	**6.9 ± 0.7**	4.9 ± 0.9	15.4 ± 2.6	34.1 ± 3.1	6.7 ± 0.5	22.2 ± 2.4	35 ± 2.1
	5 * N	3.3 ± 0.7	6.9 ± 0.6	8.7 ± 0.5	**3 ± 0.8**	**4.4 ± 0.6**	**6.4 ± 0.9**	4.1 ± 0.7	13.4 ± 4.1	30.4 ± 1.5	5.1 ± 0.5	20 ± 2.4	34.8 ± 1.9
Average	N	**30.3 ± 16.9**	63.7 ± 80.5	135.2 ± 51.5	38.4 ± 22.7	**56.9 ± 25**	**86.2 ± 59.4**	55.7 ± 24.6	102.5 ± 31.8	136.4 ± 51.9	56.5 ± 60.2	98.8 ± 36.1	130.8 ± 47.1
	2 * N	**13.4 ± 12.7**	30.7 ± 18.5	41.8 ± 17.3	15.8 ± 8.7	**25.9 ± 14.2**	**35.3 ± 15.9**	26.4 ± 9.6	78.3 ± 25.1	128.1 ± 32.5	32.1 ± 6.6	83.1 ± 20.1	124.7 ± 30.5
	3 * N	10.7 ± 12	28.9 ± 15.8	38.2 ± 14.5	**8.3 ± 4.3**	**15.5 ± 8.4**	**26.7 ± 14**	21.2 ± 8.2	66.1 ± 20.6	121.6 ± 31.6	29.8 ± 7.7	74.5 ± 12.7	118.5 ± 23.4
	4 * N	6.1 ± 1.2	19.1 ± 14.1	30.7 ± 13.7	**4.8 ± 0.8**	**9.1 ± 4.5**	**19.1 ± 12.4**	14.4 ± 3.6	58.9 ± 17	125.8 ± 27.3	25.9 ± 4.1	68.4 ± 10.7	113 ± 17.3
	5 * N	5.5 ± 1.1	15.3 ± 14.6	28.2 ± 12.4	**3.8 ± 0.7**	**6.9 ± 2.5**	**12.2 ± 11.1**	12 ± 4.2	52.9 ± 10.3	122.5 ± 23.6	23.3 ± 3.5	62.3 ± 6.3	107.1 ± 9.5

Table 6.2: Relative errors of different regression methods for different subject systems, different sample sizes (i.e., $|S|$), and different parameter settings including the best, average, and worst cases. Bold font indicates the smallest relative error for specified system and sampling size across the same parameter settings of different regression methods

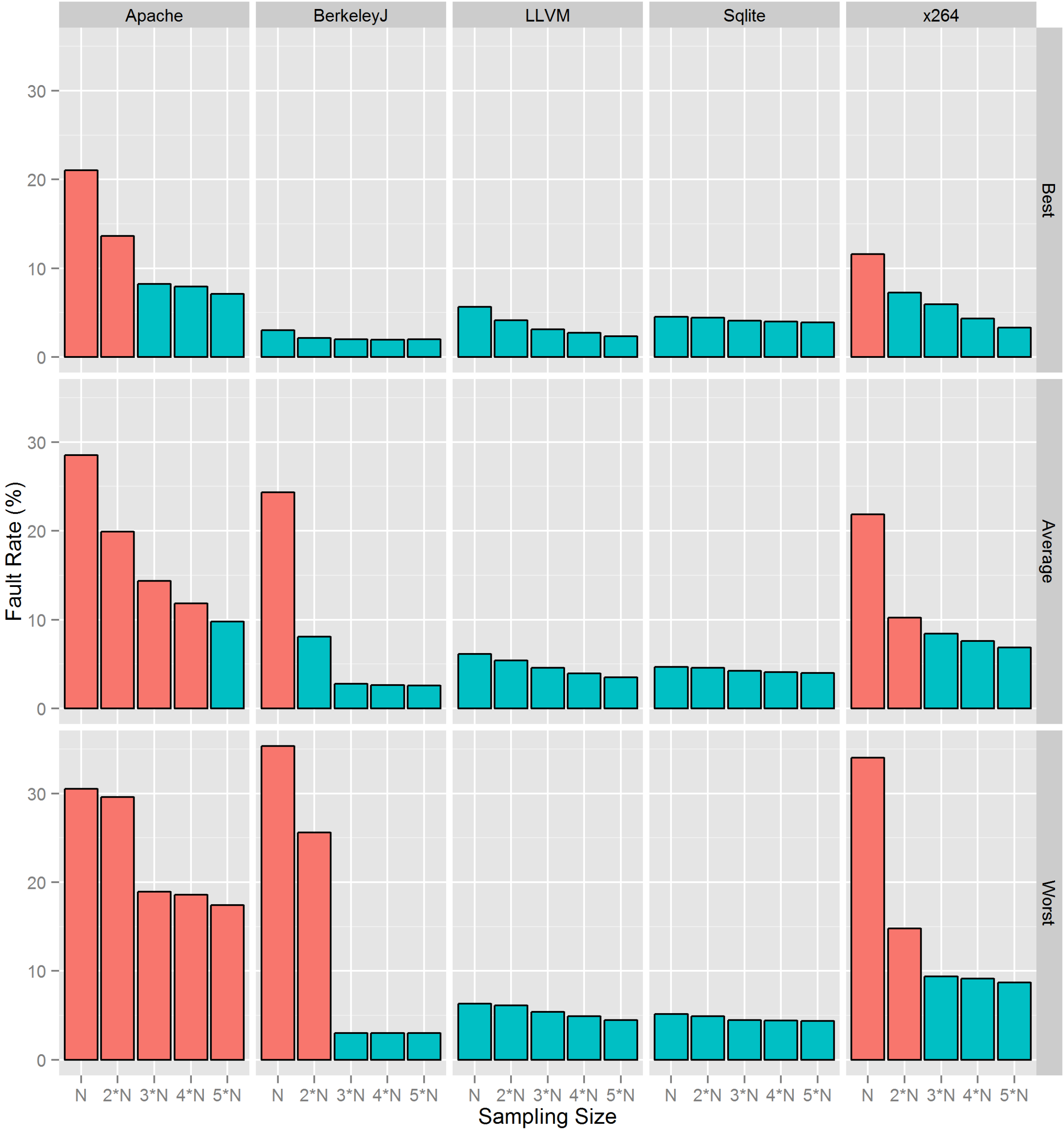

Figure 6.1: CART relative errors for different configurable systems, sampling sizes and parameter settings. Relative errors that are greater than 10% threshold are shown in red.

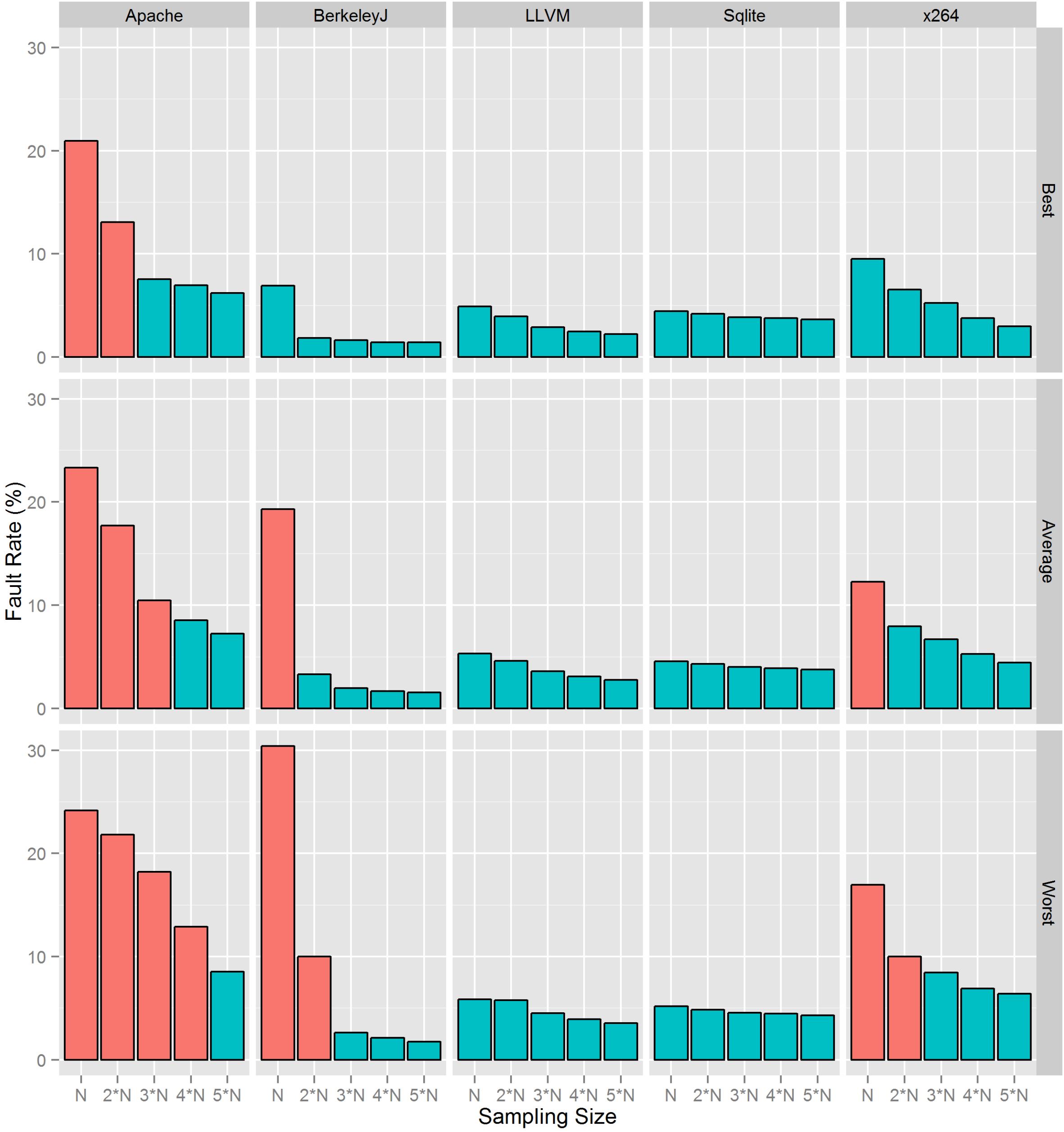

Figure 6.2: Bagging relative errors for different configurable systems, sampling sizes and parameter settings. Relative errors that are greater than 10% threshold are shown in red.

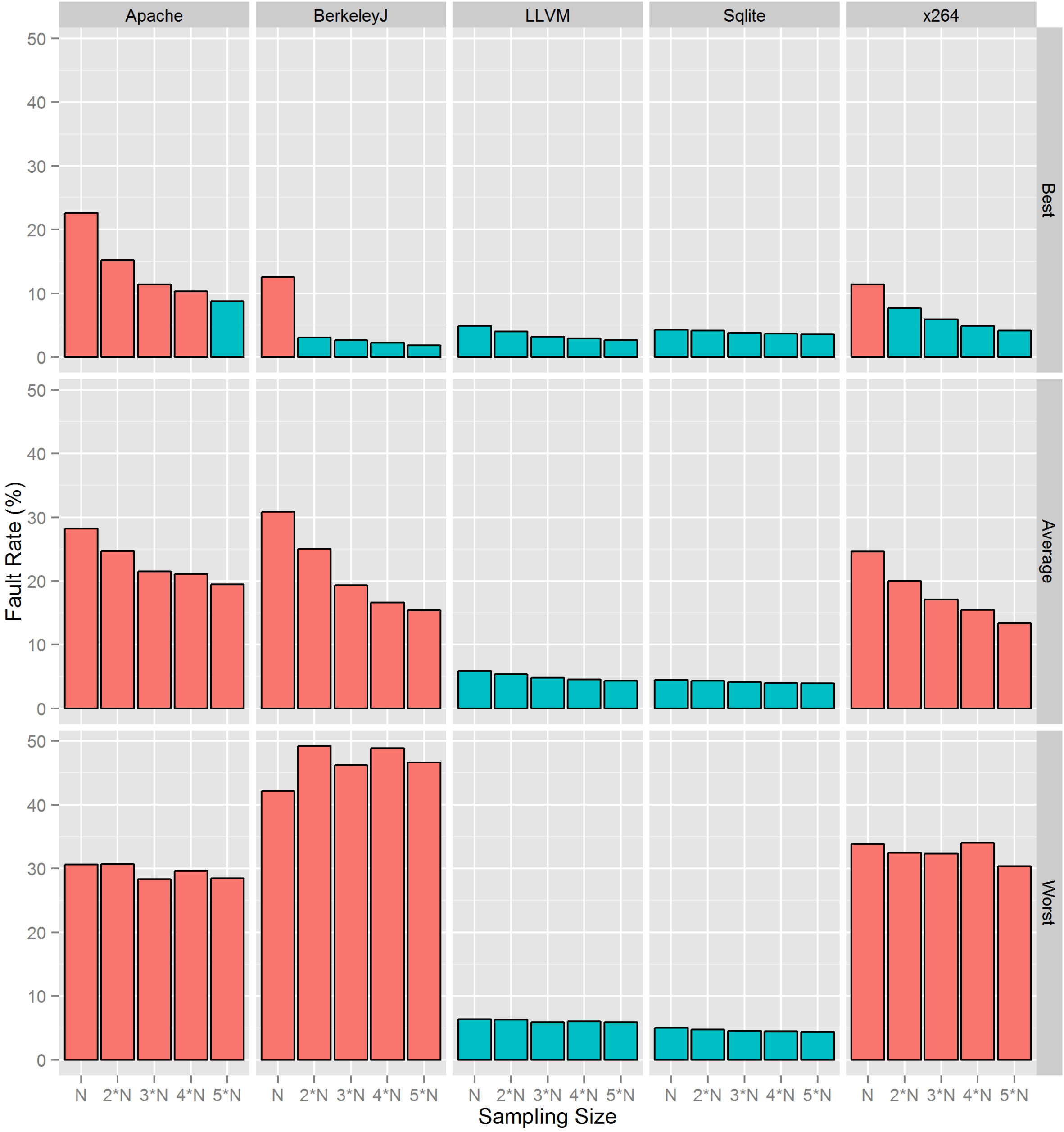

Figure 6.3: Random Forest relative errors for different configurable systems, sampling sizes and parameter settings. Relative errors that are greater than 10% threshold are shown in red.

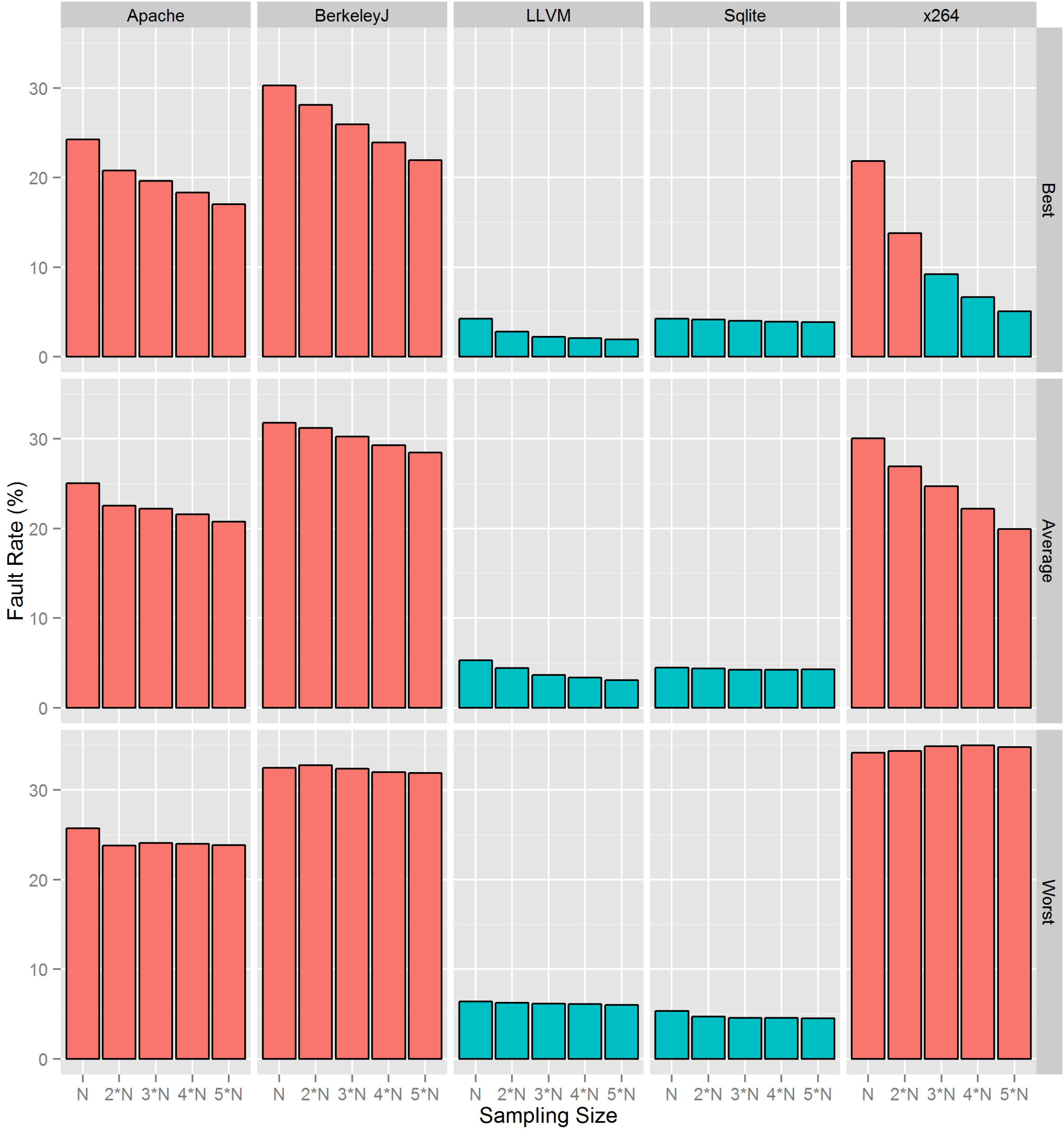

Figure 6.4: SVM relative errors for different configurable systems, sampling sizes and parameter settings. Relative errors that are greater than 10% threshold are shown in red.

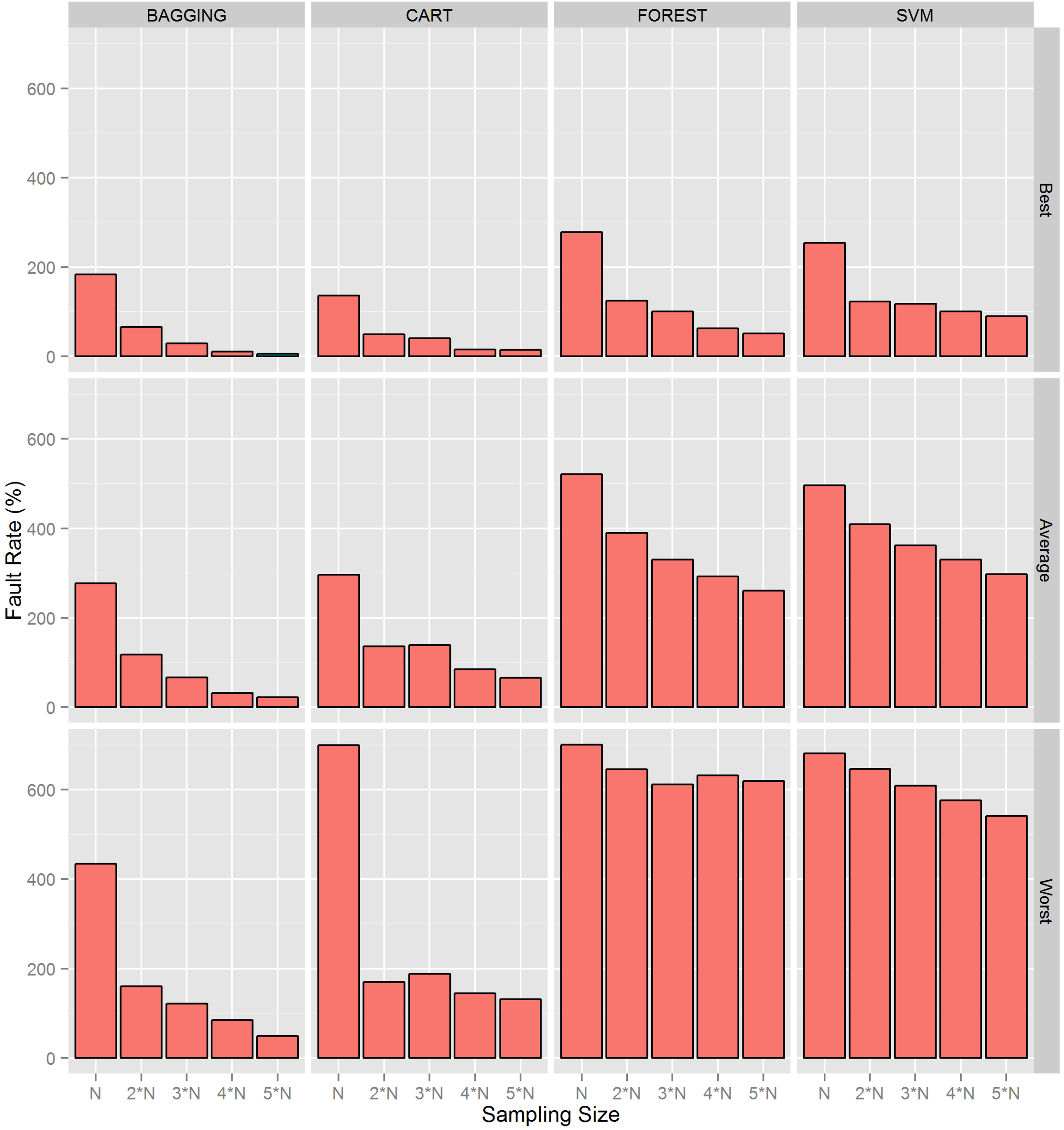

Figure 6.5: Relative errors of regression methods for BerkeleyC system, using different sampling sizes and parameter settings. Relative errors that are greater than 10% threshold are shown in red.

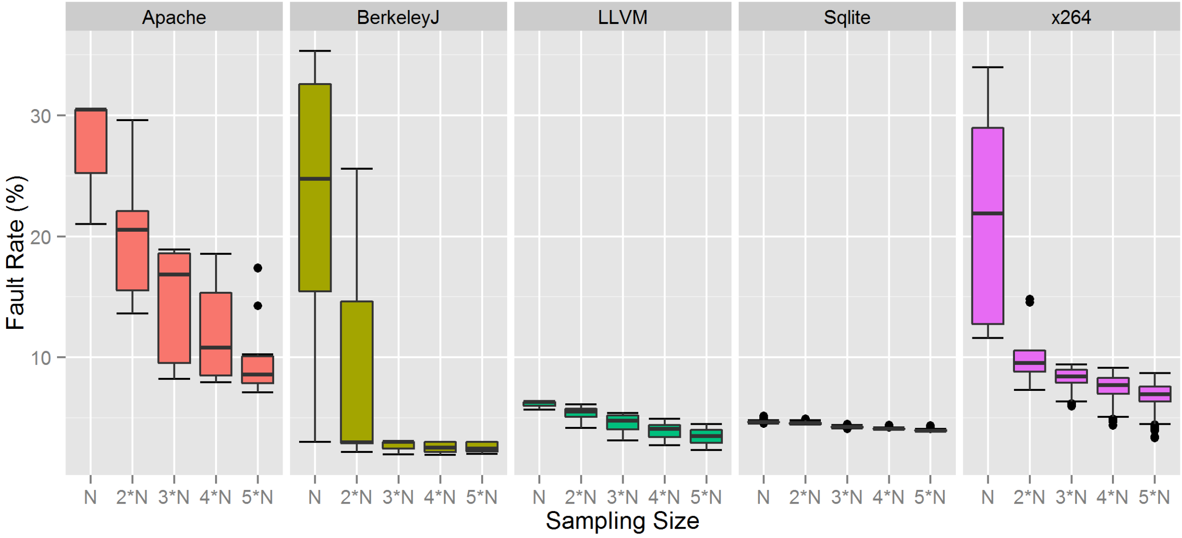

Figure 6.6: Relative error distributions of CART for different systems and different sampling sizes

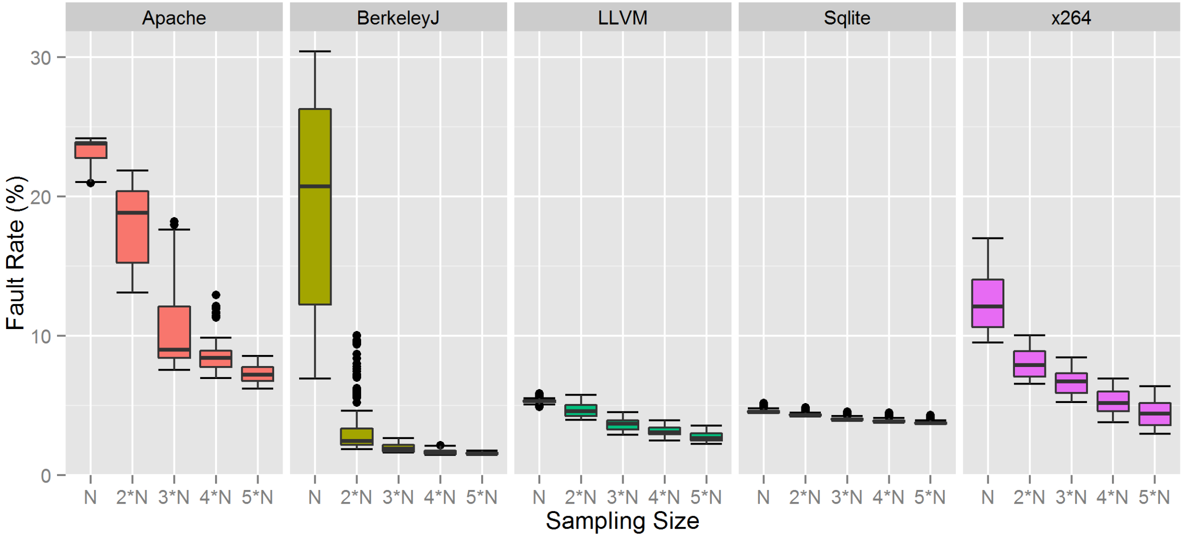

Figure 6.7: Relative error distributions of bagging for different systems and different sampling sizes

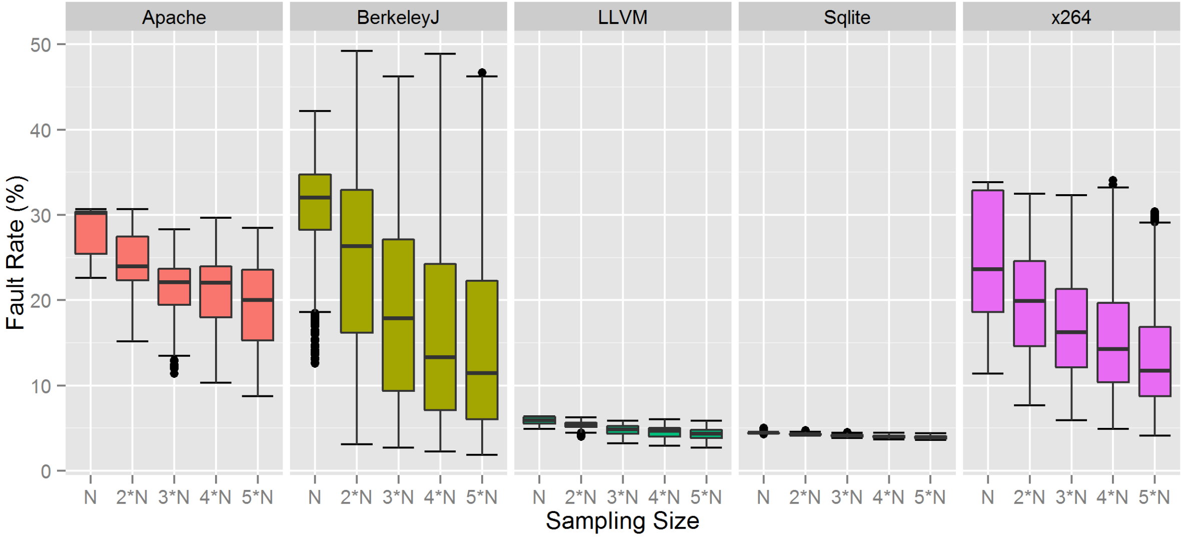

Figure 6.8: Relative error distributions of random forest for different systems and different sampling sizes

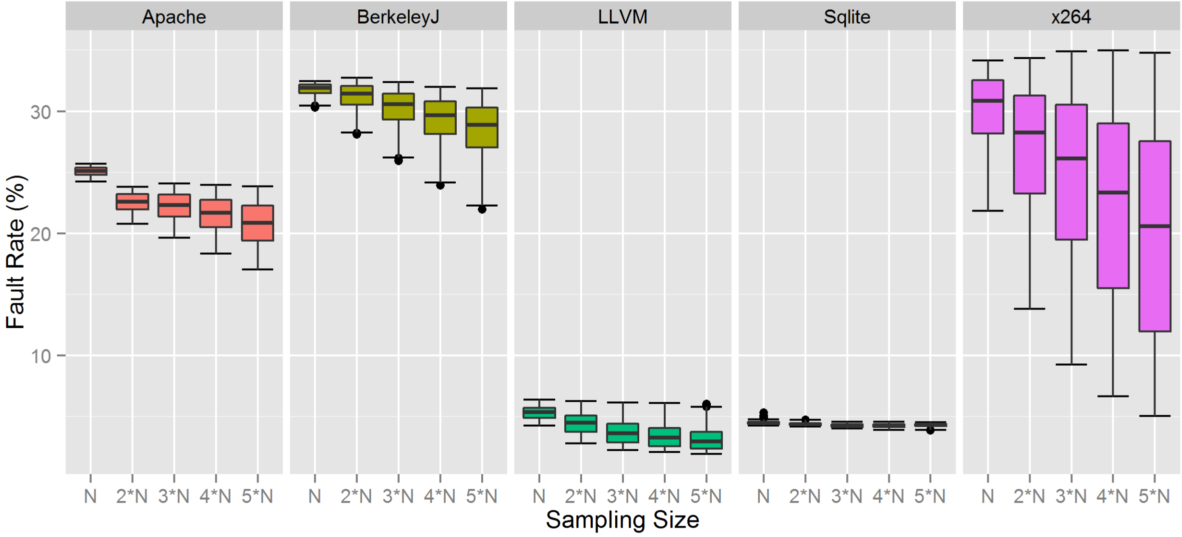

Figure 6.9: Relative error distributions of SVM for different systems and different sampling sizes

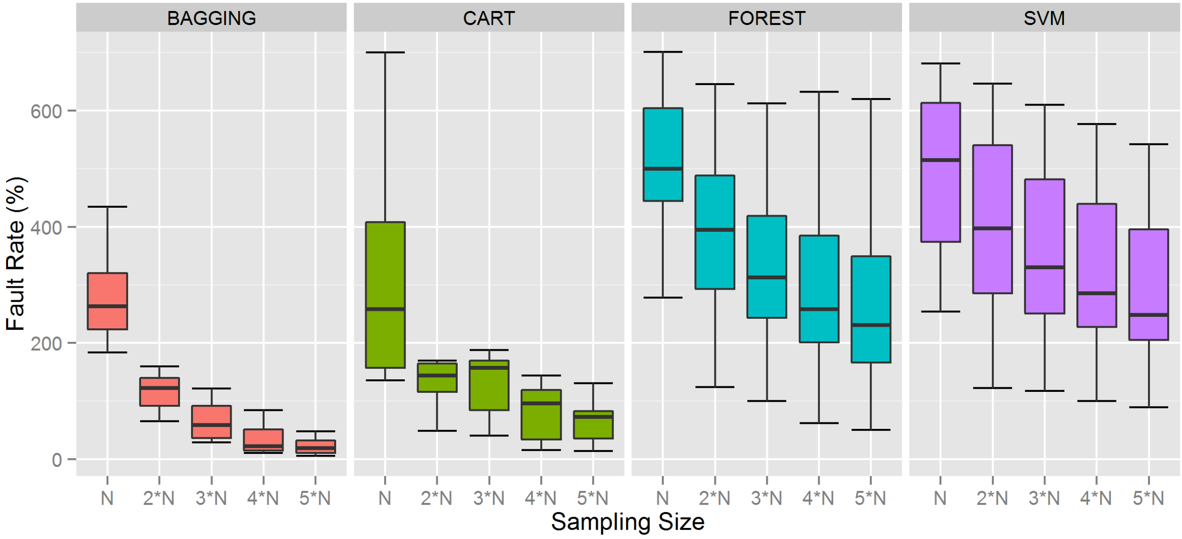

Figure 6.10: Relative error distributions of different regression methods for Berkeley DB C system and different sample sizes

Chapter 7

Conclusion and Future Work

Throughout our research we tried to deduce correlation between feature selection of a configurable software system and performance of the system using non-linear regression analysis. During the analysis we regarded software systems under investigation as black boxes without decomposing or analysing their structure. We performed a case study of different regression methods (regression trees, bagging, random forest, SVM) for our problem of variability-aware performance modelling, using only small random samples of measured configuration variants. In order to avoid manual setting of parameters for regression methods and to explore parameter space more evenly, we implemented automatic parameter tuning of regression methods by using Sobol sampling.

As a result of our research, we assessed accuracy and sensitivity of aforementioned regression methods for the problem of of variability-aware performance prediction. We showed that in term of accuracy Bagging outperforms all other regression methods in most of the cases for all configurable systems, sampling sizes and parameter settings. In terms of sensitivity the most stable regression methods turned out to be Regression Trees and Bagging.

To make our experiment more sound, for our analysis we used random sampling from six real-world configurable software systems of different sizes, configuration mechanisms, application domains and written in different programming languages. However, we understand that results of our work are not automatically transferable to other software systems.

In our future work, we plan to explore additional regression methods for solving the problem of variability-aware performance prediction. For automatic parameter tuning of regression methods we are going to attempt more parameter tuning techniques like Bayesian optimisation and other general optimisation methods. We intend to analyse complex feature interactions in configurable software systems under study and explain which types of feature interactions influence performance prediction and how. We also plan on investigating different experimental design policies[13] for performance prediction.

APPENDICES

Chapter 8

Experimental R code

```
ConvertFromSobol <- function(lower, upper, sobol, dgts){
    # Given parameter range and number from Sobol sequence,
    # function converts Sobol number to parameter value from
    # the range with specified accuracy
    #
    # Args:
    #   lower: lower bound of the parameter range
    #   upper: upper bound of the parameter range
    #   sobol: number from Sobol sequence that needs to be
    #          converted
    #   dgts: number of digits after decimal point

    return (lower + round((upper - lower) * sobol, dgts))
}

PredictUsingCART <- function(){
    # Generates Regression Tree (CART) model for the
    # specified configurable software system and uses it to
    # predict system performance

    # Initialize CART parameter ranges
    minSplitLower <- 2
    minSplitUpper <- 10
```

```
minBucketLower <- 2
minBucketUpper <- 10

maxDepthLower <- 2
maxDepthUpper <- 20

complexLower <- 0.0001
complexUpper <- 0.01

# Initialize utility variables for working with data
crs$dataset <- NULL
errorDataset  <- NULL
resultDataset <- NULL

# Load experimental data for configurable software system
dataAddr <- paste("file:///", "D:\\ExpData.csv", sep="")
dataset <- read.csv(dataAddr, strip.white=TRUE,
                    na.strings=c(".", "NA", "", "?"),
                    encoding="UTF-8")

# Calculate number of features and observations in the
# experimental dataset
featureCount <- ncol(dataset) - 1
obsCount <- nrow(dataset)

# Number of times experiment with the same parameters
# should be repeated in order to reduce bias
expReps <- 10

# Calculate sizes of random samples that will be used
# for training regression trees
samplingSizes <- 1:5
samplingSizes <- samplingSizes * featureCount

# Generate 4-dimensional (4 is the number of parameters
# that CART has) Sobol sequence that will be used for
# parameter tuning
sobolN <- 1000  # length of Sobol sequence
```

```
sobolDim <- 4    # number of dimensions in Sobol sequences
sobolParams <- sobol(n = sobolN, dim = sobolDim,
                     init = TRUE, seed = 1)

# Iterate over all different sampling sizes
for(samplingSize in samplingSizes){

    minErrorRate <- 1000

    # Initialize list for storing data samples of the same
    # sampling size
    dataSamples <- list()
    dataSamples <- c(dataSamples, 1:expReps)

    # Populate list with random data samples of the same
    # sampling size for each experiment repetition
    for(expRep in 1:expReps){
        set.seed(expRep)
        dataSamples[[expRep]] <- sample(nrow(crs$dataset),
                                        samplingSize)
    }

    # Iterate over Sobol sequence in order to tune method
    # parameters
    for(sobolNIter in 1:sobolN){

        # Convert dimensions of Sobol sequence into CART
        # parameters
        sobolMinSplit <- convertParam(
                            minSplitLower,
                            minSplitUpper,
                            sobolParams[sobolNIter, 1],
                            0)
        sobolMinBucket <- convertParam(
                            minBucketLower,
                            minBucketUpper,
                            sobolParams[sobolNIter, 2],
                            0)
```

```
sobolMaxDepth <- convertParam(
                    maxDepthLower,
                    maxDepthUpper,
                    sobolParams[sobolNIter, 3],
                    0)
sobolComplex <- convertParam(
                    complexLower,
                    complexUpper,
                    sobolParams[sobolNIter, 4],
                    4)

# Perform experiment several times to reduce bias
for(expRep in 1:expReps){

    # Build training and testing datasets
    set.seed(expRep)
    crs$nobs <- nrow(crs$dataset)
    crs$train <- dataSamples[[expRep]]
    crs$test <- setdiff(seq_len(nrow(crs$dataset)),
                        crs$train)

    # Select input and target variables
    crs$input <- setdiff(colnames(crs$dataset),
                         "PERF")
    crs$target  <- "PERF"

    # Build CART regression model
    require(rpart, quietly = TRUE)
    set.seed(expRep)
    crs$rpart <-
        rpart(PERF ~ .,
            data = crs$dataset[crs$train,
                               c(crs$input,
                               crs$target)],
            method = "anova",
            parms = list(split = "information"),
            control = rpart.control(
                        minsplit = sobolMinSplit,
                            minbucket = sobolMinBucket,
                            maxdepth = sobolMaxDepth,
                            cp = sobolComplex,
                            usesurrogate = 0,
                            maxsurrogate = 0))

        # Obtain prediction from built regression tree
        crs$pr <-
            predict(crs$rpart,
                    newdata = crs$dataset
                        [crs$test, c(crs$input)])

        # Retrieve actual performance values from the
        # initial data
        perfs <- subset(crs$dataset[crs$test,],
                        select=c("PERF"))

        # Calculate prediction relative error
        faultRate <- abs(perfs - crs$pr) / perfs * 100

        # Aggregate prediction relative errors
        if(is.null(errorDataset)){
            errorDataset <- faultRate
        }else{
            errorDataset <- cbind(errorDataset,
                                  faultRate)
        }

    } # for(expRep in 1:expReps){

# Process all results
meanRowsErrorRate <- mean(rowMeans(errorDataset))
meanColsErrorRate <- mean(colMeans(errorDataset))
sdRowsErrorRate <- sd(rowMeans(errorDataset))
sdColsErrorRate <- sd(colMeans(errorDataset))
minErrorRate <- min(minErrorRate, meanRowsErrorRate)

resultDataset <- rbind(resultDataset,
                                c(samplingSize,
                                sobolNIter,
                                sobolMinSplit,
                                sobolMinBucket,
                                sobolMaxDepth,
                                sobolComplex,
                                minErrorRate,
                                meanRowsErrorRate,
                                meanColsErrorRate,
                                sdRowsErrorRate,
                                sdColsErrorRate))

        errorDataset <- NULL

        }
    }

    # Output the combined data
    write.csv(resultDataset,
              file=”D:\\Results.csv”,
              row.names=FALSE)
}
```

References

[1] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Proc.*, SSBSE. Springer-Verlag, 2011.

[2] James Bergstra, Rémy Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proc. NIPS*, 2011.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, February 2012.

[4] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[5] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] Christophe Dutang and Petr Savicky. *randtoolbox: Generating and Testing Random Numbers*, 2013.

[8] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. ASE*. IEEE, 2013.

[9] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer, 2009.

[10] G. James, T. Hastie, D. Witten, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer London, Limited, 2013.

[11] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004.

[12] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[13] Douglas C. Montgomery. *Design and Analysis of Experiments.* John Wiley & Sons, 2006.

[14] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing.* Cambridge University Press, 1992.

[15] R Core Team. *R: A Language and Environment for Statistical Computing*, 2013.

[16] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer.* Wiley, 2008.

[17] Norbert Siegmund. Splconqueror dataset, 2012.

[18] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proc. ICSE.* IEEE Press, 2012.

[19] Norbert Siegmund, Marko Rosenmller, Martin Kuhlemann, Christian Kstner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[20] Jasper Snoek, Hugo Larochelle, and Ryan Prescott Adams. Practical bayesian optimization of machine learning algorithms. In *Proc. NIPS.* Curran Associates, Inc., 2012.

[21] I.M Sobol'. Calculation of improper integrals using uniformly distributed sequences. *Soviet Math Doklady*, 14(3):734–738, 1973.

[22] I.M Sobol' and Yu.L Levitan. A pseudo-random number generator for personal computers. *Computers and Mathematics with Applications*, 37(45):33–40, 1999.

[23] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical performance models for complex, popular applications. In *Proc. SIGMETRICS.* ACM, 2010.

[24] Terry Therneau, Beth Atkinson, and Brian Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2014.

[25] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag New York, Inc., 1995.

[26] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proc. ASE.* ACM, 2012.

[27] Graham J. Williams. *Data Mining with Rattle and R: The art of excavating data for knowledge discovery.* Springer, 2011.

[28] S. G. Rozin I. M. Sobol' Yu. L. Levitan, N. I. Markovich. On quasirandom sequences for numerical computations. *USSR Computational Mathematics and Mathematical Physics*, 28(3):88–92, 1988.