Black-box Learning of Parametric Dependencies for Performance Models

Vanessa Ackermann
University of Würzburg
Würzburg, Germany
vanessa.ackermann@stud-mail.uni-wuerzburg.de

Johannes Grohmann
University of Würzburg
Würzburg, Germany
johannes.grohmann@uni-wuerzburg.de

Simon Eismann
University of Würzburg
Würzburg, Germany
simon.eismann@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Modeling parametric dependencies in architectural performance models increases performance prediction accuracy. However, manually modeling parametric dependencies is time-intensive and requires expert knowledge. Existing automated extraction approaches require dedicated performance tests, which is often infeasible. In this paper, we propose to characterize parametric dependencies based on monitoring data. We create a representative dataset and show that different machine learning approaches perform best, depending on the characteristics of the dependency. Based on these results, we introduce a meta-selector that chooses the most suitable machine learning approach based on the dependency characteristics. In our evaluation, the meta-selector reduces the prediction error compared to the best individual machine learning approach, SVR, by 30%. As a proof of concept, we show that our approach is capable of automatically characterizing a manually modeled dependency from a previous case-study, resulting in a response time prediction accuracy of 92.8%.

CCS CONCEPTS

• **Computing methodologies** → **Supervised learning**; • **Computer systems organization** → **Self-organizing autonomic computing**; • **Software and its engineering** → **Software system models**; *Software performance*;

KEYWORDS

Performance modeling, Model learning, Meta-learning, Parametric dependencies

1 INTRODUCTION

Architectural performance models are a common approach to predict the performance properties of a software system during system design [26] and the impact of reconfigurations at runtime [19]. A major factor in the prediction accuracy of a performance model is its parameterization, i.e. the values for model parameters such as loop frequencies, branching probabilities or resource demands [29]. However, these model parameters often depend on the input parameters of a component, e.g., the size of a list impacts the time required to sort it. Therefore, many architectural performance models allow to explicitly model input parameters and their influence on model parameters as so-called parametric dependencies [3, 9, 16, 20, 28].

Manually modeling parametric dependencies requires expert knowledge, is quite error prone and causes significant manual overhead. For example, in a case stude by Krogman et al. [21] more than 24 hours were required to manually model the parametric dependencies in a small system, which shows that manually modeling parametric dependencies for large systems is infeasible.

Courtois et al. [7] propose to use regression splines combined with automated performance testing to derive functions that describe resource demands based on input parameters. Krogmann et al. [21] use genetic search to find dependencies between a components input parameters and the number of executed bytecode instructions. However, both approaches require dedicated performance tests in order to extract the parametric dependencies.

We propose to derive parametric dependencies solely from monitoring data available at runtime. Identifying between which parameters dependencies exist is a classical feature selection task [6]. Therefore, this paper focuses on the characterization of previously identified dependencies. We create a representative dataset containing parametric dependencies and evaluate how well a range of machine learning approaches can characterize the contained parametric dependencies. We find that no machine learning approach performs well for all parametric dependencies. This is in accordance with the no free lunch theorem [34], which states that machine learning algorithms cannot be universally good. Based on these results we propose a meta-selector, which selects an appropriate machine learning technique for every dependency, based on the characteristics of the available data.

In our evaluation, the meta-selector performs better than any individual machine learning technique. It reduces the parameter prediction error by 30% compared to support vector machines, the best performing individual machine learning technique. As a proof of concept, we show that our approach is capable of automatically characterizing a manually modeled dependency from a previous case-study, resulting in a response time prediction accuracy of 92.8%.

Our approach takes potential dependencies (either labeled by a human or extracted by a different algorithm) as input and automatically characterizes them, i.e., automatically learns how to derive the value of a parameter such as loop frequencies, branching probabilities or resource demands from input parameters. This significantly reduces the required effort compared to manually modeling parametric dependencies and therefore makes the modeling

of dependencies for large systems feasible. Furthermore, the approach works online, requires only run-time monitoring data of the managed application and otherwise treats the application as a black-box. It enables autonomic and online improvement and refinement of already existing performance models. The presented work represents a significant step towards our vision for self-aware performance models [11].

The remainder of this paper is structured as follows: Section 2 describes our approach, including the dataset creation, the dataset characteristics, the evaluation of a multitude of machine learning techniques and finally the construction of our meta-selector. This meta-selector is evaluated in Section 3.1, followed by a proof of concept for the integration of our approach in architectural performance models in Section 3.2. Related work and the limitations of our approach are discussed in Section 4 and Section 5, respectively. Finally, Section 6 concludes the paper and discusses potential future work.

2 APPROACH

We describe how we obtained our datasets in Section 2.1, which machine learning techniques we applied in Section 2.3, followed by why and how we created the meta-selector in Section 2.4.

2.1 Dataset creation

We first create multiple datasets that each contain the measured runtime (a.k.a the response time) of a certain algorithm in dependence of various observable input parameter values. As servers can run various applications and services, all kinds of applications might be relevant for our black-box learning approach. In consequence, we select popular algorithms from different domains (e.g., sorting, image processing, cryptography), and with different characteristics (e.g., number of input parameters, noise intensity or expressiveness of dependency) in order to meet the diversity of real-world application scenarios. Additionally, some datasets were added, where there is no correlation between the input parameters and the runtime (e.g., getRandomInt).

Non-numeric input parameters are transformed into meaningful numeric values (e.g., boolean values to 0 or 1, and enumerations to factors). To obtain the runtime dataset, we run the respective algorithm for 100.000 different input parameter combinations.

For the selection of these measurement points, we propose the following strategy: First, a realistic value range for each input parameter is configured. Next, the actual measurement points are created by dividing the parameter space into equidistant measurement points. The distance between these points depends on the chosen size of the dataset, the number of parameters and the range of each parameter. Next, the chosen tuples are shuffled in a random fashion in order to prevent our datasets from being biased by underlying optimization processes (e.g. Garbage Collection or JIT compiling). Since all tests are done in a single-threaded and sequential fashion, we can safely assume that the measured response times equal the resource demand of the respective resource. In the following, we use the term *runtime*, which can be interchanged with *response time*, *resource demand* or *service demand* in this context.

Table 1 lists the datasets we used in our study. It lists the name of the function or the solved problem and its input parameters.

Table 1: Measured algorithms and input parameters.

Name	Input parameter name	Range
AckermannFunction	n	0-3
	m	0-3
Fibonacci	FibonacciNumber	1-40
	Mode (iter./recOp./rec.)	0-2
FilterArray	ArraySize	0-100000
	FilterKey	0-100000
GaussianFilter	ImageWidth(px)	100-6500
	ImageHeight(px)	100-4010
	Sigma	1.0-10.0
GetRandomInt	MinInt	1-100000
	MaxInt	1-200000
HistogramEqualization	ImageWidth(px)	100-6500
	ImageHeight(px)	100-4010
LoadFile	FileSize(KB)	1-1024
RGBFilter	ImageWidth(px)	100-6500
	ImageHeight(px)	100-4010
RSAEncryption	StringLength	0-30
	KeySizeExponent(2^x)	9-11
RSADecryption	StringLength	0-30
	KeySizeExponent(2^x)	9-11
ScaleImage	ImageWidth(px)	100-6500
	ImageHeight(px)	100-4010
	ScaleFactor	0.1-3.0
SearchArray	ArraySize	0-100000
	Key	0-100000
SHAHashing	StringLength	0-10000
	SHA-Mode (-1/-256/-512)	0-2
SortArray	ArraySize	1-10000
SubsetSum	ArraySize	1-10000
	Sum	1-100000

AckermannFunction calculates the AckermannFunction for the given parameters n and m. As the AckermannFunction is known to drastically increase its runtime for increasing values of n and m, we have to rely on a rather small set of parameter values.

Fibonacci returns the n-th Fibonacci number. The parameter FibonacciNumber defines n, i.e., the index of the requested number. Mode defines if an iterative, an optimized recursive or an unoptimized recursive implementation should be chosen.

FilterArray filters a given array for the given key. Hence, it returns a filtered array only containing elements with a specified key. ArraySize defines the number of elements in the array, FilterKey is the integer representation of the key.

GaussianFilter applies a Gaussian filter function to a given image. ImageWidth (px) and ImageHeight (px) define the width and the height of the processed image, Sigma is the sigma-parameter for the gaussian filter.

GetRandomInt returns a random integer in the range between the given parameters MinInt and MaxInt. Note that

this function should NOT contain any dependencies from the input-parameters to the runtime of the function.

HistogramEqualization enhances the contrast of a given image by agjusting the image intensities. As the procedure is calibrated by the histogram on the image itself, we only have the two parameters ImageWidth (px) and ImageHeight (px), specifying the size of the image.

LoadFile loads a file from disk to the RAM. The only parameter specifying this process is the size of the file in KB as FileSize(KB).

RGBFilter is another image operation, converting the given image to the RGB color channels. The two parameters ImageWidth (px) and ImageHeight (px) specify the size of the image.

RSAEncryption encrypts a message according to the RSA algorithm. The parameter StringLength defines the length of the input message and KeySizeExponent (2^x) defines the length of the key. The range of the KeySizeExponent is between 9-11, which leads to the three possible key sizes 512, 1024 and 2048.

RSADecryption decrypts an encrypted message according to the RSA algorithm. The parameter StringLength defines the length of the resulting non-encrypted message and KeySizeExponent (2^x) defines the length of the key. The range of the KeySizeExponent is between 9-11, which leads to the three possible key sizes 512, 1024 and 2048.

ScaleImage resizes an image and scales its content to be either smaller or bigger than the orignial input. The parameters ImageWidth (px) and ImageHeight (px) define the original image size, the parameter ScaleFactor defines the desired output size. A factor smaller than 1 leads to a reduction of the image size, a factor greater 1 leads to an increase.

SearchArray performs a linear search through the given array and looks for the given key value. The length of the array is defined by ArraySize and the desired item key by Key.

SHAHashing computes the hash of a given string using the SHA algorithm. The length of the string to hash is defined by StringLength. There are 3 modes to operate: SHA-1, SHA-256 and SHA-512. This is configured by the parameter SHA-Mode (-1/-256/-512).

SortArray reorders all (numeric) elements of the given array. The parameter ArraySize defines the number of elements to sort.

SubsetSum calculates if any subset of the given list of (numeric) elements can be found, such that the sum of all elements of the subset equals the given predefined value. The parameter ArraySize defines the number of elements to choose from and the parameter Sum defines the required target sum.

The given parameters are either direct input parameters (e.g., FilterKey) or derivated from the input parameters (e.g. ArraySize). Note that not all parameters do have a (direct) impact on the measured runtime. This is on purpose, as the regressors should be capable of filtering the important parameters.

The framework for executing the runtime tests, the resulting measurements and all other code used in this work is available on GitHub[1]. The results presented in this paper were obtained on a MacBook Air with a Core i5 1.8 GHz and 8 GB RAM, running OS X 10.12 (Sierra).

Table 2: Comparison of runtime distribution in each dataset.

Dataset	Mean (ns)	SD (ns)	Min (ns)	Max (ns)
AckermannF.	611	27226	44	6501140
Fibonacci	9642615	59828774	42	975401791
FilterArray	534018	357394	196	15557496
GaussianFilter	644159655	610063923	2696907	3758818764
GetRandomInt	87	1353	45	270782
HistogramEq.	682101542	648840997	706579	6548593716
LoadFile	398055	394602	11913	12204192
RGBFilter	212744088	185667599	320687	1630478145
RSADecryption	2586784	2755623	308882	26086653
RSAEncryption	143296	97100	41386	6486117
SHAHashing	61019	164588	882	26716009
ScaleImage	326531535	478575701	25683	4081014148
SearchArray	747	1717	44	182087
SortArray	351231	336717	76	65894078
SubsetSum	461134	574735	54	10894929

2.2 Dataset characteristics

We want to ensure that our datasets are diverse and cover various types of parametric dependencies. Hence, we analyze the runtime distributions. As we can see in Table 2, the runtime distribution varies significantly between the different sets, with regards to runtime mean, range and dispersion of the different measurements. Furthermore, the sets differ in algorithm domain, number of input parameters and type of input parameters (numeric vs. nominal).

To further illustrate the diversity of our obtained data sets, we contrast the runtime distributions of three data sets in Figure 1:

SortArray: Figure 1(a) shows all runtime measurements for SortArray in dependence of *ArraySize*. We observe that there is a strong linear relationship between the two variables, and that noise increases as the values of *ArraySize* becomes larger.

Fibonacci: Figure 1(b) depicts our runtime measurements in dependence of the Fibonacci number to be calculated. The colour of the measurement points denotes the second input parameter, *Mode*. We can see that the runtime strongly depends on the computation mode - while it rises exponentially with the value of FibonacciNumber when using the recursive mode (blue), it is more or less constant for the iterative (red) and optimized-recursive mode (green).

SubsetSum: Figure 1(c) shows the measured runtime for SubsetSum with increasing *ArraySize*. Points are coloured based on the value of the input parameter *Sum* for this measurement. It seems like the runtime of SubsetSum depends on neither of the two parameters, but that the measurement points are randomly distributed.

[1] https://github.com/Olifee/automatic_dependency_characterization

This shows that multiple types of parametric dependencies are covered by the datasets, which was our primary goal. Hence, we conclude from Table 2 and the exemplary Figures 1(a), 1(b) and 1(c) that our obtained data sets cover different relevant types of dependencies are adequately diverse.

2.3 Applying Machine Learning Techniques

In order to characterize these dependencies, we use common regression techniques from the area of machine learning. For this, we define the dependent variable as the resource demand or the runtime of the execution and the independent variables as the given parameter values as we expect them to influence this resource demand.

The regressors are chosen from various machine learning approaches, such as decision tree learning, instance-based learning and ensemble learning. We use a single manual configuration for the predictors, as parameter optimization is beyond the scope of this work. In the following, we give an overview of all evaluated predictors and their respective configuration parameter values. We use the implementations from Weka 3.8 [33], a state-of-the-art machine learning library licensed under the GNU General public license 3.0

ZeroR [27]: ZeroR is our baseline algorithm to compare against. It always returns the mean value of the training samples.

k-Nearest Neighbors (kNN) [1]: Here, we dynamically select the number of nearest neighbours between 1 and 5 using hold-one-out evaluation on the training data. Furthermore, we weight neighbours by the inverse of their distance.

Linear regression (LinReg) [8]: The coefficients are updated with batch gradient descent, and squared error loss is used as loss functions.

Stochastic Gradient Descent (SGD) [2]: We use Huber loss as loss function, which is less sensitive to outliers in data than squared error loss. We set the slope δ to 0.001.

Support Vector Regression (SVR) [14]: The threshold parameter ϵ is set to 0.001. We use a polynomial kernel with exponent 1.0.

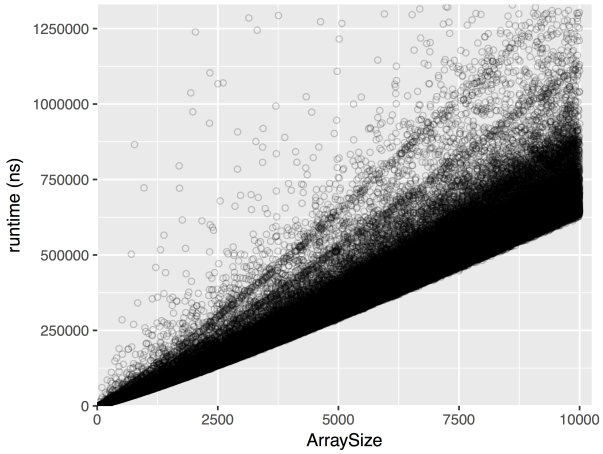
Artificial Neural Net (ANN) [15]: We use a feedforward ANN with the following architecture: The input layer consists of one node per input parameter. It is followed by a single hidden layer with $(\#InputParameter + 1)/2$ sigmoid nodes. After this comes the output layer with unthresholded linear units as output nodes. All layers are fully connected. The learning rate is 0.1, the momentum is 0.2 and we use 2000 epochs for training.

Classification and Regression Trees (CART) [5]: The minimum number of instances per leaf is set to 2. The regression tree is pruned in 3 folds using reduced-error pruning with backfitting.

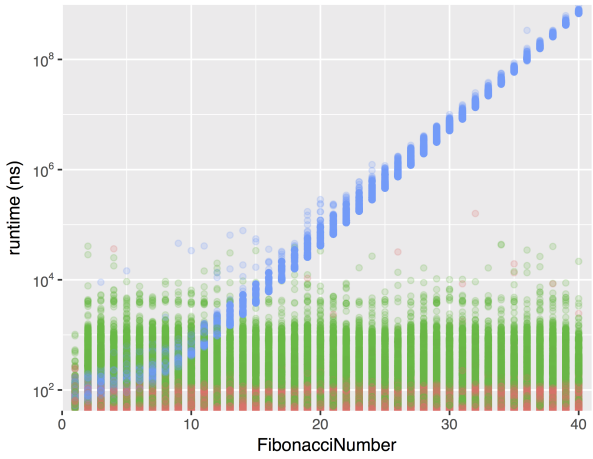
M5 trees (M5) [25]: The minimum number of instances per leaf is set to 5. The leaf nodes use squared error loss for fitting a linear regression model.

Bootstrap aggregating (Bagging) [4]: We use CART as base predictor for our bagging models. The size of the bags equals the size of the training set, and we use a total of 25 bags.

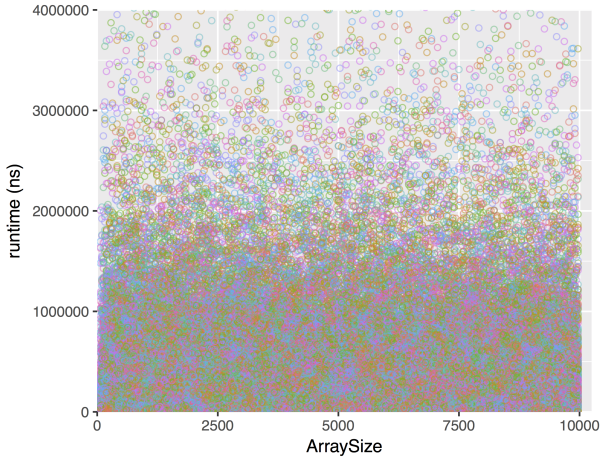
Random forest (RandomF) [18]: The random forest configuration is similar to that of bagging. We use CART as base



(a) SortArray.



(b) Fibonacci.



(c) SubsetSum.

Figure 1: Distribution of runtime values subject to observed input parameters.

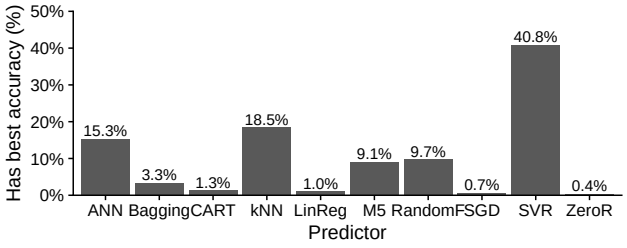


Figure 2: Distribution of the most accurate predictor.

predictor and the bag size is the training set size. Other than in bagging, the minimum number of instances per leaf is set to 1, we use 250 bags. The random variable subset for each decision nodes contains $\log_2(\#InputParameter + 1)$ input parameters.

We systematically measure the prediction performance of different regression techniques on our runtime datasets and thus evaluate their suitability to learn parametric dependencies for performance models. We define the following: A regressor is said to "perform best" if it has the smallest mean absolute error (MAE) among all evaluated regressors for a given training and test set. The mean absolute error (MAE) is defined as follows:

$$MAE = \frac{\sum_{i=1}^{m} |\hat{y}_i - y_i|}{m}, \tag{1}$$

where m is the size of the validation set, $\hat{y}_i$ is the predicted value and y_i the correct measured value of one specific measurement point.

With respect to our online scenario, we use different training set sizes to determine how the prediction performance of each technique scales with the size of the training set. The evaluation process for each runtime dataset is as follows: We randomly shuffle the dataset and take the first 1000 instances as the validation set. Then, we train each predictor on training sets of increasing size and evaluate it on the validation set. The training set sizes, i.e., evaluation steps, are: 10, 100, 500, 1000, 3000, 6000, 9000. Thus, each evaluation run contains 105 single evaluations (15 datasets * 7 training set sizes). We repeat the experiment 10 times for each dataset with different seed values for the shuffling process, resulting in a total of 1050 evaluations per regressor.

Figure 2 shows the distribution of the best regressor on the different evaluation sets. We observe that SVR performs best in most cases, being the best regressor in 40.8% of all cases. However, kNN (18.5%), ANN(15.3%), M5(9.1%) and Random Forest(9.7%) also perform best in some situations. In fact, all applied approaches perform best on at least one evaluation set. Surprisingly, even ZeroR performs best in some cases. This is in accordance with the no free lunch theorem [34], that states that no machine learning algorithm can be the best under all circumstances.

2.4 Meta-Selector Construction

After reviewing the results of Section 2.3, we decided to create a meta-selector. This meta-selector analyzes the characteristics of the dataset in question and then recommends or automatically selects the best suitable approach, based on the results in Section 2.3. We define the following characteristics as features for the training of the meta-selector. These have proven to be (1) easily collectible for any dataset and (2) to have an influence based on our experience with the dataset results:

- Number of training instances (Size)
- Number of parameters (NumParam)
- Range of runtime values (RuntimeRange)
- Coefficient of variance of runtime (RuntimeCV)
- Highest linear correlation between any input parameter and the runtime (HighestCorrelation)
- Lowest linear correlation between any input parameter and the runtime (LowestCorrelation)
- Coefficient of determination (R^2) (R2LinReg).

These features are extracted for all traces collected in Section 2.1. The labels of each trace is the regressor that performed best for the given trace. Therefore, we can train a classifier, to classify and therefore recommend one of the regressor based on the features of any dataset. Using the data sets described in Section 2.1, we get a set of 1050 training samples, each containing the 7 listed features.

As the task of selecting the best approach based on the feature values is a classic supervised machine learning problem, we use a standard classification algorithm to create a decision tree for recommendation. The advantage of using decision trees is that its decisions are traceable and human-readable. In doing so, we hope to gain additional insights on the relations between the dataset characteristics and the algorithm performances. We use the extracted characteristics as features and the best performing regressor as the labeled class. For the construction of the tree, we use Classification and Regression Trees (CART) [5] algorithm, implemented by the WEKA library [33]. We use the same settings as described in Section 2.3, with the difference that we limit the maximum tree depth to four. This should reduce the complexity of the tree and hence improve readability and interpretability.

Figure 3 shows the resulting decision tree. We observe that a low difference between the minimum and the maximum of the measured runtime values (RuntimeRange) together with a low variance (RuntimeCV) leads to the use of SVR or ANN. Other important parameters are the number of parameters (NumParam), the number of training instances (Size) and the lowest correlation between any parameter and the measured runtime (LowestCorrelation).

Table 3: Dataset for meta-selector evaluation.

Name	Input parameter name	Range
ArrayListSerialization	ArrayListSize	1-1000
BinarySearchArray	ArraySize	1-10000
	Key	0-100000
	isSorted	0-1
MatrixMultiplication	numRowsA	1-50
	numColumnsA	1-50
	numColumnsB	1-50
SolveTowersOfHanoi	NumDisks	1-20
TrainMLP	numInstances	1-2000
	numEpochs	1-2000

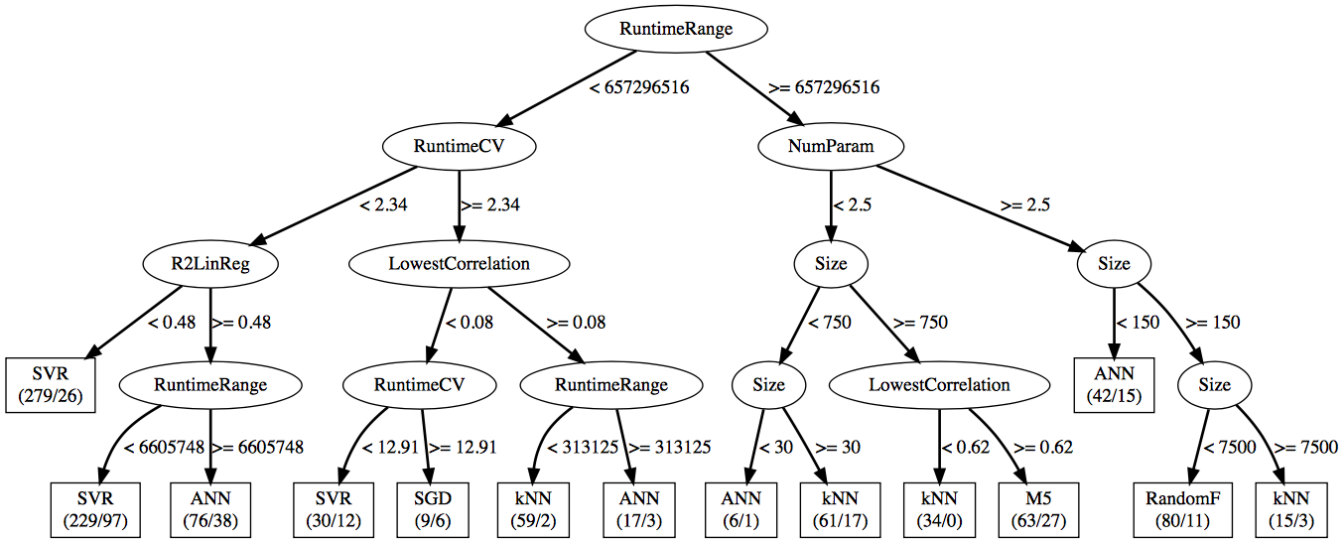


Figure 3: Meta-selector decision tree.

3 EVALUATION

In order to evaluate the proposed approach, we first evaluate the applicability of the meta-selector in Section 3.1 and then present a proof-of-concept using a small end-to-end case study in Section 3.2.

3.1 Meta-selector evaluation

Using the dependent t-test for paired samples, we test if using the meta-selector's recommended technique A significantly reduces the average prediction error μ (i.e., the MAE) compared to using a fixed machine learning approach B that generally performs well (in our case: SVR). The paired t-test is appropriate, as the samples are randomly selected, the data is paired and approximately normal distributed. Thus, we choose the null hypothesis $H_0 = \mu_A \geq \mu_B$ and alternative hypothesis $H_1 = \mu_A < \mu_B$.

The paired samples are obtained on five new runtime datasets, listed in Table 3.

ArrayListSerialization serializes and stores a numerical array to the disk. The parameter ArrayListSize describes the number of elements contained in the array.

BinarySearchArray performs a binary search in a given integer array for a given integer key. ArraySize describes the number of numerical elements contained in the array, Key is the element to search for and isSorted is a boolean value. The value is true, if the given array is sorted and false, if it is not deliberately ordered.

MatrixMultiplication executes a multiplication of two matrices. Their size is specified with numColumnsA (number of columns of matrix A), numRowsA (number of rows of A) and numcolumnsB (number of columns of Matrix B). Note that the number of Rows of B is implicitly defined by the size of A.

SolveTowersOfHanoi is an algorithmic solver of the *Towers of Hanoi* problem. The parameter numDisks describes the number of disks, that need to be moved, i.e., the problem size.

TrainMLP trains a multi-layer perceptron. The two parameters numInstances and numEpochs describe the number of instances used for training and the number of training epochs, respectively.

These datasets were obtained using the same methodology as presented in Section 2.1. Per set, we use 10 different training set sizes ($n = 20, 50, 200, 400, 700, 2000, 4000, 6000, 8000, 10000$), resulting in a total of 50 paired samples. For each sample i, we calculate the difference d_i between the MAE of A and B on the dataset's respective test set:

$$d_i = MAE_i(A) - MAE_i(B).$$

Next, we calculate the mean $\bar{d}$ and the standard deviation σ_d of the differences d. With this, we can calculate the t-value of our dependent samples:

$$t = \sqrt{n}\frac{\bar{d}}{\sigma_d}.$$

The mean MAE difference $\bar{d}$ over all samples is 20884576 ns and the standard deviation σ_d of the differences is 6166633, resulting in the following t-value:

$$t = \sqrt{50} \cdot \frac{20884576}{6166633} = 23.9476.$$

The critical t-value for 49 degrees of freedom and a probability level of 1% is $t(0.99; 49) = 2.404892$. As $t > t(0.99; 49)$, we can reject the null hypothesis, that there is no significant difference between the results using our meta-selector and SVR.

Compared to always choosing SVR for all test sets, the selector improves the overall MAE by 30%. Additionally, it is interesting to note that the best suited approach changes with the number of available measurement points. For two of the data sets, even four different approaches performed best depending on the size of

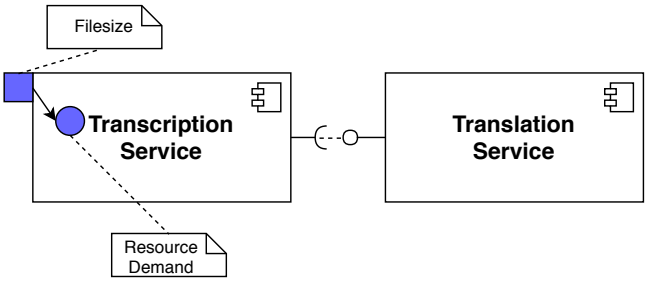


Figure 4: Video transcription case study, adapted from [9].

the training set. This underlines the importance of using a meta-selector.

Our evaluation shows that for the given samples, the average prediction error is significantly lower when using the meta-selector instead of the always applying SVR, the best individual machine learning approach. We infer that our meta-selector is an appropriate approach for prediction technique selection. Overall, the experiment shows that it is feasible and beneficial to adapt prediction techniques to observable dataset characteristics, without applying domain knowledge or manual effort.

3.2 Proof of concept

In a previous paper, Eismann et al. [9] performed case studies for parametric dependencies in the context of a video store application. One of the case studies depicts the automated transcription of subtitles, i.e. the automatic generation of subtitles from the audio track of a video, as shown in Figure 4. The transcription service component receives an audio file and generates subtitles from it. Next, the translation service component translates the resulting subtitles into a variety of languages. In this scenario, the resource demand of the video transcription component depends on the size of the videos that are transcribed. Eismann et al. modeled this parametric dependency manually in this case study.

As a proof of concept, we use the approach presented in this paper to automatically characterize this dependency and compare the performance prediction accuracy of the automatically characterized dependency with the manually modeled dependency. In the previous case study, the distribution of the file sizes was considered to be known. However, our approach is built to work on a stream of monitoring data instead of a distribution. Therefore, we create 100.000 samples from the file size distribution and use them as input for our approach. Aside from this, we are able to reuse the model and solver from the previous case study.

Simulating the manual model predicts a response time of 44565 ms ± 1519 ms (N = 20), while the model with the automatically characterized predicts a response time of 47377 ms ± 1353 ms (N = 20). Compared to the measured response time of 44207 ms, this results in a prediction accuracy of 97.3% and 92.8% respectively. The automatically characterized dependency is slightly worse compared to the manually modeled dependency. Still, the accuracy of the automatically characterized dependency is well within the range considered acceptable for capacity planning [23], without requiring manual effort or expert knowledge. The models used in this case study are available online[2].

4 RELATED WORK

The related work for this approach can be categorized into two research areas: The extraction of dependencies between performance model parameters and software performance curves.

4.1 Extraction of dependencies between performance model parameters

Some approaches have been proposed to automatically extract parametric dependencies, as manual modeling of parametric dependencies is time-intensive and error-prone.

Krogmann et al. [21] run dedicated performance tests in a testbed with bytecode instrumentation and monitoring of input data at interface-level. Next, genetic search is used to learn dependencies between the input values of components and the executed bytecode instructions. Later on, the number of executed bytecode instructions is mapped to resource demands using bytecode benchmarking. The authors state that bytecode instrumentation creates performance overheads of up to 250%, which means the bytecode counting can not be performed continuously in a production environment.

Courtois et al. [7] propose to use regression splines combined with automated performance testing to derive functions that describe resource demands based on input parameters. Compared to linear or polynomial regression, regression splines allow to model non-linear, discontinuous dependencies. The authors introduce so-called pseudo confidence bands as an accuracy measurement for resource functions. Based on these accuracy measurements, their algorithm intelligently selects the number and location of measurement points. This reduces the number of dedicated performance measurements required to construct the regression spline model.

To the best of our knowledge, there is no existing approach to automatically learn parametric dependencies in performance models, that does not require preliminary experiments or access to application source code.

4.2 Software performance curves

Software performance curves aim to model the performance of a system (usually response time) as a function over the configuration and input parameters of the system.

Kwon et al. [22] derive the response time of Android applications from parameters calculated early on in the application execution. During an offline stage, an instrumented version of the application is benchmarked to determine the influence of parameters such as branch counts, loop counts or variable values on the application response time. Using a technique called program slicing, the application is separated into independent code segments. This approach allows to accurately predict the response time of android applications after executing less than two percent of the application code.

Thereska et al. [30] predict the performance of several Microsoft applications based on configuration and input parameters from data collected from several hundred thousand real users. Several

[2] https://github.com/SimonEismann/CharacterizationCasestudyModels

Microsoft applications are instrumented to extract hardware configuration, software configuration and workload characteristics from users "in the wild". The authors apply a classification and regression tree to filter relevant attributes, followed by a similarity search to derive performance predictions.

Westerman et al. [32] analyze the suitability of multivariate adaptive regression splines, classification and regression trees, genetic programming and kringing for the construction of software performance curves. Additionally, three different measurement point selection algorithms are evaluated, which reduce the required number of dedicated performance measurements.

Noorshams et al. [24] investigate the prediction accuracy of linear regression, multivariate adaptive regression splines, classification and regression trees, and cubist (an extension of M5 regression model trees) for virtualized storage systems. The authors propose a general heuristic search algorithm to optimize the parameters of these regression techniques. This algorithm results in a greatly reduced prediction error in their case study.

Faber and Happe [10] derive software performance curves with the use of genetic programming and conduct a thorough parameter optimization. The optimized genetic programming algorithm outperformed MARS in their case study.

Software performance curves are a powerful tool to predict the response time of a system for different workloads. However, unlike architectural performance models, they can not be used to analyze the impact of changes to the system itself, such as scaling, redeployment or system evolution. We use the existing work on software performance curves as guidelines to select the regression approaches we compare in this work.

5 LIMITATIONS AND THREATS TO VALIDITY

In order to explore the limitations of our work, this section reviews the fundamental assumptions of our approach.

One assumption of the proposed approach is that the dependencies to characterize are already given. The identification itself is therefore out of scope for this work. However, there exist other approaches for this problem, based on static code analysis [21] or feature selection [6]. We plan to integrate our work with existing approaches from this area.

Since our approach requires monitoring data, the modeled system needs to be monitored during operation. However, this can be done at low overhead using, for example, the Kieker framework [31]. To further minimize the measuring overhead, resource demands should be estimated instead of directly monitored. There exist a wide variety of approaches in this area [12, 13, 29].

If applied in an online scenario, the accuracy will likely be suboptimal until a certain amount of monitoring data has been gathered. This is due to the fact that the accuracy of machine learning algorithms increases, as the size of the training data increases. However, one intent of the proposed meta-selector is to cope with this exact issue by choosing the most robust approach for the respective scenario.

Although we put in our best efforts to select and create a variety of representative datasets (see Section 2.1), the number of datasets is still limited. However, the number of datasets is sufficient to show that the best performing machine learning approach varies depending the dependency and on how much training data is available. Part of future work will be to create and include more datasets in our study.

Lastly, we acknowledge that our approach only covers dependencies between model parameters on the same call-path. Therefore, the impact of parameters describing the internal state of a component on its resource demand can currently not be explored. An example for this would be a system where the size of a database influences the resource demand of some components. Happe et al. [17] propose an approach to model such dependencies. In order to apply our approach to such dependencies, monitoring data for the parameters describing the system/component state would need to be available.

6 CONCLUSION

We investigate different techniques for black-box learning of parametric dependencies for performance models from monitoring data. The results show that no single approach performs well for all dependencies. Therefore, we construct a meta-selector that classifies a monitoring stream in order to select the best suitable approach for it. Our results show that it is both feasible and beneficial to use the meta-selector, as it reduces the prediction error by 30% compared to using the best individual approach. Furthermore, as a proof of concept, we apply our approach to a previous case study, where our approach enables a prediction accuracy of 92.8% for the service response time without using any expert knowledge. Therefore, our approach is a step towards facilitating automatic model learning and enabling self-aware performance management.

For future work, we propose to repeat the predictor evaluation on a large collection of diverse monitoring datasets from real-world applications to improve the accuracy of the meta-selector. Furthermore, we propose to optimize the configuration parameters of the applied prediction technique, e.g., evaluate different kernel types for SVR or the parameter settings for ANN. Noorshams et al. [24] show that parameter optimization can reduce the prediction error of models by up to 74%.

We assume that the framework's prediction accuracy will significantly improve with parameter optimization. One might also want to investigate if it is beneficial to select model parameters depending on data set characteristics, similar to the meta-classifier. In order to further automatize the process of modelling parametric dependencies, we suggest to investigate the best way to transform non-numeric parameters (e.g., strings, arrays, enums) into numeric values. Possible solutions might be to map objects to their length/size/id, or to use the byte count of an object as numeric values. The resulting approach could be used by the framework to automatically transform non-numeric values, thus enlarge the selection of valid input parameter type.

As of now, the meta-selector solely focuses on prediction accuracy. Future work could additionally consider the time-to-result, as this becomes important in time critical scenarios.

ACKNOWLEDGEMENTS

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1).

REFERENCES

[1] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[2] Dimitri P Bertsekas. 1999. *Nonlinear programming*. Athena scientific Belmont.

[3] E. Bondarev, P. de With, M. Chaudron, and J. Muskens. 2005. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *31st Conference on Software Engineering and Advanced Applications*. 36–43.

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

[5] Leo Breiman. 2017. *Classification and regression trees*. Routledge.

[6] Girish Chandrashekar and Ferat Sahin. 2014. A Survey on Feature Selection Methods. *Computers and Electrical Engineering* 40, 1 (2014), 16–28. https://doi.org/10.1016/j.compeleceng.2013.11.024

[7] Marc Courtois and Murray Woodside. 2000. Using Regression Splines for Software Performance Analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*. 105–114.

[8] Norman R Draper and Harry Smith. 1998. Applied regression analysis 3rd edition.

[9] Simon Eismann, Jürgen Walter, Jóakim von Kistowski, and Samuel Kounev. 2018. Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time. In *IEEE International Conference on Software Architecture*.

[10] Michael Faber and Jens Happe. 2012. Systematic Adoption of Genetic Programming for Deriving Software Performance Curves. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. ACM, New York, NY, USA, 33–44.

[11] Johannes Grohmann, Simon Eismann, and Samuel Kounev. 2018. The Vision of Self-Aware Performance Models. In *Companion of the 2018 IEEE International Conference on Software Architecture*.

[12] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. 2017. Self-Tuning Resource Demand Estimation. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing*. 21–26.

[13] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. 2018. Using Machine Learning for Recommending Service Demand Estimation Approaches. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. 473–480.

[14] Steve R Gunn et al. 1998. Support vector machines for classification and regression. *ISIS technical report* 14, 1 (1998), 5–16.

[15] Kevin Gurney. 2014. *An introduction to neural networks*. CRC press.

[16] Dick Hamlet. 2009. Tools and experiments supporting a testing-based theory of component composition. *ACM Transactions on Software Engineering and Methodology* 18, 3 (2009), 12.

[17] Lucia Happe, Barbora Buhnova, and Ralf Reussner. 2014. Stateful Component-based Performance Models. *Software and Systems Modeling* 13, 4 (2014), 1319–1343.

[18] Tin Kam Ho. 1995. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, Vol. 1. IEEE, 278–282.

[19] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. 2017. Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *IEEE Transactions on Software Engineering* 43, 5 (2017), 432–452.

[20] Heiko Koziolek. 2008. *Parameter dependencies for reusable performance specifications of software components*. Ph.D. Dissertation. Universität Oldenburg.

[21] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. 2010. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Transactions on Software Engineering* 36, 6 (2010), 865–877.

[22] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. 2013. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX Annual Technical Conference*. 297–308.

[23] Daniel Menasce and Almeida Virgilio. 2000. *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*. PTR.

[24] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. 2013. Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 283–294.

[25] John R Quinlan et al. 1992. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, Vol. 92. Singapore, 343–348.

[26] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press.

[27] Suresh Chandra Satapathy, K Srujan Raju, Jyotsna Kumar Mandal, and Vikrant Bhateja. 2015. *Proceedings of the Second International Conference on Computer and Communication Technologies: IC3T 2015*. Vol. 2. Springer.

[28] Murali Sitaraman, Greg Kulczycki, Joan Krone, William Ogden, and Narasimha Reddy. 2001. Performance Specification of Software Components. *SIGSOFT Software Engineering Notes* 26, 3 (2001), 3–10.

[29] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation* 92 (October 2015), 51–71. https://doi.org/10.1016/j.peva.2015.07.005

[30] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. 2010. Practical Performance Models for Complex, Popular Applications. *SIGMETRICS Performance Evaluation Review* 38, 1 (2010), 1–12.

[31] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering*. 247–248.

[32] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. 2012. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 190–199.

[33] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[34] David H Wolpert. 1996. The lack of a priori distinctions between learning algorithms. *Neural computation* 8, 7 (1996), 1341–1390.