Mastering Uncertainty in Performance Estimations of Configurable Software Systems

Johannes Dorn
Leipzig University
johannes.dorn@uni-leipzig.de

Sven Apel
Saarland University
Saarland Informatics Campus
apel@cs.uni-saarland.de

Norbert Siegmund
Leipzig University
norbert.siegmund@uni-leipzig.de

ABSTRACT

Understanding the influence of configuration options on performance is key for finding optimal system configurations, system understanding, and performance debugging. In prior research, a number of performance-influence modeling approaches have been proposed, which model a configuration option's influence and a configuration's performance as a scalar value. However, these point estimates falsely imply a certainty regarding an option's influence that neglects several sources of uncertainty within the assessment process, such as (1) measurement bias, (2) model representation and learning process, and (3) incomplete data. This leads to the situation that different approaches and even different learning runs assign different scalar performance values to options and interactions among them. The true influence is uncertain, though. There is no way to quantify this uncertainty with state-of-the-art performance modeling approaches. We propose a novel approach, P4, based on probabilistic programming that explicitly models uncertainty for option influences and consequently provides a confidence interval for each prediction of a configuration's performance alongside a scalar. This way, we can explain, for the first time, why predictions may cause errors and which option's influences may be unreliable. An evaluation on 12 real-world subject systems shows that P4's accuracy is in line with the state of the art while providing reliable confidence intervals, in addition to scalar predictions.

CCS CONCEPTS

• **Software and its engineering → Software performance**; • **Computing methodologies → Uncertainty quantification**;

KEYWORDS

Probabilistic Programming, Performance-Influence Modeling, Configurable Software Systems, P4

ACM Reference Format:
Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416620

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia*, https://doi.org/10.1145/3324884.3416620.

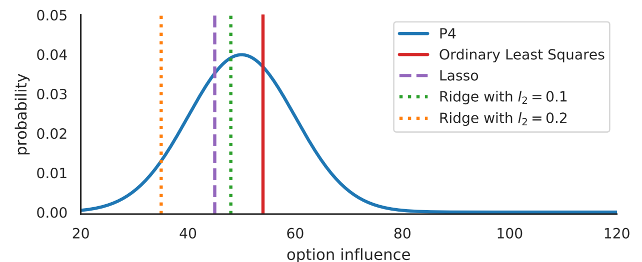

Figure 1: An exemplary option's performance influence modeled by different scalar regression models (bars) contrasted by P4's probability density prediction (blue curve).

1 INTRODUCTION

Modern software systems are often configurable. They offer several configuration options that affect the systems' functional and non-functional properties. Energy consumption, response time, and throughput are examples of non-functional properties, which are commonly subsumed by the term *performance*. Understanding an option's influence on performance and predicting performance for particular configurations is key when it comes to finding optimal system configurations. Finding an optimal configuration is an essential task because (1) many systems are shipped with a sub-optimal default configuration [10, 40], (2) manually exploring configurations does not scale [45], and (3) fine-grained tuning can improve performance up to several orders of magnitude [15, 47]. To determine the influence of individual configuration options and their interactions on performance, a number of machine learning approaches have been proposed, relying on rule-based decision trees [5], symbolic regression [35], and artificial neural networks [7].

To model the influence of options on performance, a set of configurations must be sampled and measured from a system's configuration space. These data are fed into a learning algorithm to fit a model which then allows users to estimate a scalar performance value for a given configuration. However, this scalar value falsely implies a certainty that neglects several sources of uncertainty in the modeling process: (1) measurement bias, (2) model representation and learning process, and (3) incomplete data (e.g., due to sampling bias) [36].

Without a proper uncertainty measure, application engineers may be led to wrong decisions as there is no available information about how certain a learned influence or estimated performance is. Let us assume that we have trained a model using one of the previously mentioned learning approaches. In the ideal case, we

obtain an influence per configuration option and interaction stating the expected contribution to the overall system performance when (de-)selecting an option or combinations of options. Figure 1 illustrates this influence as a scalar number using a vertical bar. Each bar represents a different learning approach to determine the scalar representing the option's influence. As we can see, different learning approaches lead to different scalars, and even a single approach can produce substantially different values arising from different runs and different hyper-parameter settings. Looking at Figure 1, it is unclear which actual effect an option has on the system's performance, and there is no way to quantify this uncertainty with state-of-the-art performance modeling approaches for configurable systems. Likewise, applying an optimization approach to automatically find an optimal configuration using only scalars will yield a single optimal (possibly incorrect) configuration, while there may be other better configurations that the model misjudged due to unconsidered uncertainties.

Our key contribution is the following: We account for uncertainty about an option's and interaction's true influence on performance that may arise from measurement bias, the learning procedure, and incompleteness of data [39]. By making uncertainty explicit across the modeling process using a *Bayesian* rather than a frequentist approach, we foster model understanding for performance engineers, provide clear expectation boundaries for performance estimates of software configurations, and offer a means to quantify when and where a learned model is inaccurate. All these pieces of information are absent in current approaches, which can harm trust in the models and transfer into practice.

To illustrate our approach, we compare the probability distribution describing the possible performance value of an option (in blue) in Figure 1 with the scalars produced by the different learning approaches. Considering the distribution as a whole, we can state how likely the influence of an option or interaction falls into a value range. The spread of the distribution is an important indicator for the certainty of estimation and whether additional data for this option might be necessary. It also gives confidence intervals for predictions and performance optimizations.

Framing the problem of performance modeling in a Bayesian setting can be done via probabilistic programming [31]. It requires the specification of three key components: likelihood, prior, and observations. The *likelihood* expresses a generative model of how the *observations* (i.e., measured configurations) are distributed. The *prior* encodes the belief (or expectation) about each option's and interaction's influence on performance. This is usually stated by a distribution for a specific value range (e.g., uniform distribution between 3 and 5 seconds). Specifying this distribution requires domain knowledge, which is not always available. A key element of our approach is an *automated prior estimation algorithm*, which can be used to learn accurate Bayesian performance-influence models without domain knowledge.

In summary, we propose an approach for performance-influence modeling that incorporates and quantifies the uncertainty of influences of configuration options and interactions on performance. A key ingredient is an automatic prior estimation algorithm that takes the burden of guessing priors from the user. We conduct an evaluation of the reliability of the uncertainty estimates of inferred models and compare the accuracy of our approach to a state-of-the-art point estimate model.

We make the following contributions:

- a probabilistic modeling approach for performance influence modeling of configurable software systems,
- a data preprocessing pipeline to avoid inference failures and to improve model interpretability,
- the tool called P4, which is an open-source implementation of our approach,
- an evaluation of P4's prediction accuracy, and
- an evaluation of the reliability of the uncertainty measures of models inferred with P4.

With our approach, we add to the important trend on explainability and interpretability of machine-learning models. We believe that this is especially important in domains such as software engineering, in which machine-learning models must provide insights and explanations to help improving the field.

2 MODELING UNCERTAINTY

Performance-influence modeling entails different kinds of uncertainty, of which we consider aleatoric and epistemic uncertainty in our work, similar to [19, 20]. *Aleatoric uncertainty* results from errors inherent to the measurements of the training set, *epistemic uncertainty* expresses doubt in the model's parameters. Both can be integrated into a Bayesian performance model, for which we explain the basics in Section 2.3.

2.1 Aleatoric Uncertainty

Performance-influence models describe a system's performance in terms of influences of its configuration options and interactions [33]. A configuration is a set of assignments to all available options from a certain domain (e.g., binary or numeric), that is $c = \{o_1, o_2, \dots, o_n\}$, where n is the number of options and o_i is the value assigned to the i-th option.

We measure the performance of a configuration by configuring a software system, and executing a workload. Formally, we denote a configuration's performance Π as a function that maps a configuration c from the set of valid configurations $\mathbb{C}$ to its corresponding scalar performance value: $\Pi : C \mapsto \mathbb{R}$. For a DBMS, we could choose energy consumption as performance, run a benchmark, and query an external power meter to determine the energy needed. However, there are two notable sources of error arising from measurement, which introduce uncertainty: measurement error and representation error.

2.1.1 Measurement Error. Typically, the measurement process has an inherent error ϵ, which is typically either absolute or relative [37]. Absolute errors ϵ_{abs} affect all measurements equally:

$$\hat{\Pi}(c) \in \left[\Pi(c) - \epsilon_{abs}, \Pi(c) + \epsilon_{abs}\right] \quad (1)$$

By contrast, relative errors ϵ_{rel} are given in percent and affect higher values more severely:

$$\hat{\Pi}(c) \in \left[\Pi(c) \cdot \frac{100 - \epsilon_{rel}}{100}, \Pi(c) \cdot \frac{100 + \epsilon_{rel}}{100}\right] \quad (2)$$

Note that, depending on the context, this interval, called *confidence interval*, can be defined to span all possible measurements for

$\hat{}$ or, altenatively, to contain $\hat{}$ only in a fraction of cases (e.g. 95 %). Unfortunately, this information is rarely available to the user.

The confidence interval of the measurement error constitutes an uncertainty that can be reduced by aggregating repeated measurements, but it is fixed at modeling time (i.e., the time when we fit the model). Moreover, absolute and relative errors are examples for *homoscedastic* and *heteroscedastic* aleatoric uncertainty, respectively. This means that, in the case of relative measurement error, the variance of uncertainty depends on the individual sample (heteroscedastic), whereas it is constant for the absolute measurement error (homoscedastic).

2.1.2 Representation Error. Representation of measurement data requires discretization for storage and processing. We assume a decimal representation for simplicity, as the precision of floating-point representations is more complicated[1]. Discretization can happen on the sensor side before we store the data. For example, an energy meter returning only integer Watt-hour (Wh) values may cause a representation error of +/- 0.5 Wh, while storing the execution time of a benchmark in seconds with two decimals may yield a representation error of +/- 5 ms.

$$\bar{}(c) \in [\hat{}(c) - u, \hat{}(c) + u] \quad (3)$$

That is, in the general case, the performance value at modeling time lies around the measured performance $\hat{}(c)$ within +/- u, the unit length of the discretization. Depending on the use case, the representation error can induce substantial uncertainty.

2.2 Epistemic Uncertainty

Models, in general, and performance-influence models, in particular, never match reality perfectly. While, in our case, aleatoric uncertainty arises from the training data samples, epistemic uncertainty stems from the model chosen and the amount of data provided. Let us assume a linear performance model (c) for a configurable software system with n options:

$$(c) = {}_0 + {}_1 \cdot c(o_1) + \cdots + {}_n \cdot c(o_n) \quad (4)$$

Here, $c(o_i)$ returns the value for the i-th option of configuration c; these values are multiplied with the model parameters , where ${}_0$ is the base performance of the system. However, we can assign different values to to model as a one-point estimate.

A typical use case in practice are linear regression models, which can be fitted to minimize different objective functions. Lasso [38] and Ridge [12] regression are alternatives to Ordinary Least Squares regression, which can be combined into an Elastic Net [48]. Their objectives differ in their way of computing the learning error (L1 and L2 normalization). A tuning parameter changes Elastic Net's error computation function such that there is no single right way to fit a linear model. As Figure 1 shows, we obtain different values for the same coefficient ${}_i$ when applying Lasso, Ridge, and Ordinary Least Squares. Hence, the fitted value for ${}_i$ is *uncertain*, as the blue curve in Figure 1 illustrates.

Another reason why can take different values lies in the training data used. Different samples of configurations — sampled according to different sampling strategies [17] — lead to different values, even with the same error function, as the literature on sampling approaches has demonstrated [9, 18, 35]. Yet, even different hyperparameter settings can result in different coefficients depending on how strong we penalize the learning error. In addition, unless a training set contains all valid samples, we are uncertain whether is a good fit, since increasing the training set size usually improves the prediction accuracy of a regression model by refining and also reduces uncertainty about . Note that although adding samples to the training set reduces epistemic uncertainty, each sample itself is still subject to aleatoric uncertainty.

Instead of specifying the model's weights as a real-valued vector $\in \mathbb{R}^n$, we can formally incorporate uncertainty into by changing it to a probability vector $\hat{}$. This way, each model weight becomes a probability density function that specifies which values for are more probable than others representing the best fit. Thus, for Gaussian-distributed uncertainty, we can specify

$$\hat{} \sim \mathcal{N}(\mu,) \quad (5)$$

as a probability vector, with $\mu, \in \mathbb{R}^n$. We do not know, though, whether uncertainty is Gaussian-distributed for real-valued configurable systems and what are the settings for $\mu,$. To determine this distribution, we need probabilistic programming.

2.3 Probabilistic Programming

Framing the problem of performance modeling in a Bayesian setting can be done via *probabilistic programming* [31]. Users of this paradigm must specify three key components with a *probabilistic programming language* (PPL): likelihood, prior, and observations. With these, the PPL takes care of Bayesian inference according to Bayes' theorem:

$$\overbrace{P(A|B)}^{\text{Posterior}} = \frac{\overbrace{P(B|A)}^{\text{Likelihood}} \overbrace{P(A)}^{\text{Prior}}}{P(B)} \quad (6)$$

We refrain from explaining Bayesian statistics from scratch, but explain in what follows the necessary components for inference. If we assume that A and B are distinct events, then $P(\ )$ maps an event to its probability to occur, $P(\ |\ )$ gives the conditional probability of an event A given that another event B occurs. In the context of probabilistic programming, A is a vector of *random variables* that represents model parameters, whereas B represents observations. A *robability ensity unction* (PDF) is a function of a random variable whose integral over an interval represents the probability of the random variable's value to lie within this interval. Accordingly, $P(\ )$ maps a random variable to its PDF, and $P(\ |\ )$ returns the conditional PDF of a random variable given that another random variable has a certain PDF. With these definitions, we next explain the components of Bayes' theorem that are relevant for probabilistic programming.

Likelihood $P(B|A)$. The likelihood specifies the distribution of observations B assuming that the PDFs for model parameters A are true. With probabilistic programming, the likelihood is typically specified as a generative model that incorporates random variables. Imagine an example in which we repeatedly toss a coin to find out whether and how it is biased. We can represent the probabilities of the possible outcomes, heads and tails, with a Bernoulli distribution

[1] see 754-2019 - IEEE Standard for Floating-Point Arithmetic for precision

$\mathcal{B}(\cdot)$, whose parameter $p \in [0, 1]$ defines the probability of heads. Formally, we first let A be a Bernoulli-distributed random variable and then define the likelihood $P(B|A)$ to be determined by A:

$$A \sim \mathcal{B}(p)$$
$$B|A \sim A$$

While this model has only one random variable, more complex models are possible; however, the inference may not be analytically solvable, requiring approximations such as *Monte Carlo Markov Chain* (MCMC) sampling [26]. Such a generative model can make predictions that are PDFs (i.e., posterior distributions) themselves.

Prior $P(A)$. Priors define our belief about the distribution of our random variables before seeing any training data. Choosing priors naturally requires domain knowledge and is comparable to selecting a optimization starting point. An uninformed prior for the coin-toss example is

$$A \sim \mathcal{B}(0.5),$$

which assumes that both heads and tails are equally probable.

Posterior $P(A|B)$ *from observations* B*:* Given a likelihood, we can finally update our prior beliefs with observations. From a machine-learning stance, observations form the training set. In case of the coin-toss example, running Bayesian inference with 5 observed heads will yield an updated generative model, the posterior, which will give heads a higher probability.

3 BAYESIAN PERFORMANCE MODELING

In this section, we describe our approach of incorporating uncertainty into performance-influence models. Figure 2 provides an overview of all steps involved. In a nutshell, we perform the following tasks: First, we preprocess a given set of measured configurations (i.e., the training set) to ensure that inference (i) does not break and (ii) finishes in a reasonable time. Second, we apply probabilistic programming to build a Bayesian model for a selection of options and interactions thereof. It is key for scalability that this selection comprises the actual set of influencing options and interactions. Third, we estimate the priors for the model's random variables (i.e., options and interactions) and compute a fitted model with Bayesian inference.

3.1 Data Preprocessing

Our approach relies on a training set consisting of a number of sampled configurations that are attributed with their performance. Thus, our approach can be combined with any sampling strategy, such as feature-wise, t-wise [16], or random sampling [4]. However, it is important to process the sample set to avoid inference failures and to promote interpretability, as we explain next.

Similar to Ordinary Least Squares, Bayesian inference is prone to failure if *multicollinearity* exists in the training set, which occurs when the values of independent variables are intercorrelated [3, 11]. Let us consider the following training set for an exemplary software system with options X, Y, Z, and M, illustrating multicollinearity:

B	X	Y	Z	M	$\Pi(\cdot)$
1	1	0	0	1	**10**
1	0	1	0	1	**20**
1	0	0	1	1	**30**

Option B is mandatory. It represents the base functionality of the system, which results from configuration-independent parts of the code. Options X, Y, and Z form an alternative group, that is, the system's constraints enforce that exactly one of them is active in each configuration. An important insight is that an alternative group introduces multicollinearity to a training set because the selection of any single option is determined by the remaining options, for example: Z = 1 − X − Y. Multicollinearity not only hinders inference, but also interpretability. Considering the training set above, we see that the following performance-influence models are accurate with respect to the measurements, but assigning different contributions of individual options:

$$\Pi(C) = 0 \cdot c(\text{B}) + 10 \cdot c(\text{X}) + 20 \cdot c(\text{Y}) + 30 \cdot c(\text{Z})$$
$$\Pi(C) = 5 \cdot c(\text{B}) + 5 \cdot c(\text{X}) + 15 \cdot c(\text{Y}) + 25 \cdot c(\text{Z})$$
$$\Pi(C) = 10 \cdot c(\text{B}) + 0 \cdot c(\text{X}) + 10 \cdot c(\text{Y}) + 20 \cdot c(\text{Z})$$

Because exactly one option of the alternative group is active in each configuration, the base performance of a software system can be attributed to the base functionality B and the options of an alternative with any ratio. For example, option X can have an influence of 10, 5, or none, depending on how we assign the performance to the system's base functionality. Therefore, performance influence models for such systems are difficult to compare and interpret. Here, we do not even know whether an option (e.g., X) is influential at all. This is a problem that related approaches share [5, 34].

Choosing a default configuration provides remedy for multicollinearity inference failures and interpretability problems. That is, we select a default option for each alternative group using domain knowledge or at random. We then remove these options from the training set to achieve the following effects:

- Default options' performance influences are set to 0.
- Multicollinearity arising from alternative groups is reduced, since the selection of a single remaining option of an alternative group cannot be determined without the removed default option (i.e., Z = 1 − X − Y does not hold anymore if any of these options is removed from the training set).

Mandatory options, which must be selected in each configuration, introduce a special case of multicollinearity. Option M is mandatory and therefore present in each configuration and indistinguishable from the base influence. Similar to alternative groups, a model can split the base influence between mandatory options and the base influence with any ratio. Moreover, we can see that such an option does not contribute any information to the model by computing the Shannon information entropy [32]:

$$H(o) = -\sum_{x=0}^{1} P_o(x) \cdot \log_2 P_o(x) \qquad (7)$$

As M is selected in each configuration, its only selection value is 1, with selection probability $P_{\text{M}}(1) = 1$. We see that, therefore,

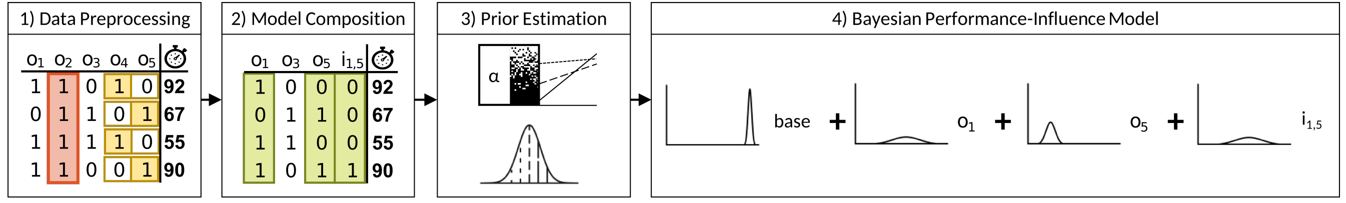

Figure 2: Workflow of P4: 1) preprocess data; 2) compose model from options and interactions; 3) estimate priors for random variables; 4) infer Bayesian performance-influence model.

the information entropy of $\mathbb{M}$ is 0:

$$\begin{aligned} H(\mathbb{M}) &= -P_{\mathbb{M}}(1) \cdot \log_2 P_{\mathbb{M}}(1) - P_{\mathbb{M}}(0) \cdot \log_2 P_{\mathbb{M}}(0) \\ &= -1 \cdot \log_2 1 - 0 \cdot \log_2 0 = 0 \end{aligned} \tag{8}$$

For that reason, we can safely remove mandatory options from the training set. The same applies for dead options, which are never active.

Note that options may only appear to be dead or mandatory as an artifact of the sampling process. That is, it is insufficient to query only the system's variability model for its constraints to detect mandatory or dead options. Hence, we perform constraint mining on the sample set rather than the whole system to overcome this problem. We use the Shannon information entropy in Equation 7 as a means to determine dead options and scan the set of options for combinations that appear to be alternative groups.

3.2 Model Composition

To build a Bayesian model with probabilistic programming, we first need to specify which options and interactions are present in the model. Subsequently, we create random variables from this model structure to account for epistemic and aleatoric uncertainty.

3.2.1 Option and Interaction Filtering. Composing a model from all options and all potential interactions, whose number is exponential in the number of options, is impractical for large software systems, because models with high numbers of parameters are difficult to interpret and, more importantly, inference may become computationally infeasible [14]. Therefore, we apply *model selection* to constrain the number of parameters. In particular, we use a *subset selection approach* [22], because it yields a subset of unaltered options from a parent set, which is not the case for other approaches, such as dimensionality reduction [41]. We build the parent set of available options $\mathbb{S}$ from all options $\mathbb{O}$ of the system in question as well as all pair-wise interactions $\mathbb{I}$ with $\mathbb{S} = \mathbb{O} \cup \mathbb{I}$. We map each pair-wise option i to a virtual option with respect to its constituting options o_r and o_s:

$$c(o_{n+p}) = c(o_r) \cdot c(o_s) \quad \text{with} \quad 1 \leq p \leq |\mathbb{I}| \wedge r \neq s \tag{9}$$

Compared to higher-order interactions, pair-wise interactions have been found to frequently influence performance [34] and to be the most common kind of interaction [21]. However, we acknowledge that considering higher-order interactions may improve the accuracy of our approach [34], at the cost of possibly leading to computationally intractable models.

Subset selection approaches define a filter function $F : \mathbb{S} \mapsto \{0, 1\}$, which yield 1 if an option or interaction of the parent set $\mathbb{S}$ should be considered by the model, and 0, otherwise. The result of subset selection consists of filtered options and interactions:

$$\mathbb{V} = \{ s \mid \forall s \in \mathbb{S} \text{ and } F(s) = 1 \} \tag{10}$$

Similar to previous work [8], we apply Lasso regression [38] on the preprocessed training set. As a result, Lasso assigns zero performance influence to less- and non-influential options and interactions, and it distributes the performance influence among the remaining elements in $\mathbb{V}$. Our Lasso filter selects $\pi \in \mathbb{V}$, whose performance influence $I_{Lasso}(\pi)$ is non-zero according to Lasso regression:

$$F_{Lasso}(\pi) = \begin{cases} 0 & I_{Lasso}(\pi) = 0 \\ 1 & I_{Lasso}(\pi) \neq 0 \end{cases} \tag{11}$$

3.2.2 Applied Probabilistic Programming. We follow related approaches for performance modeling of configurable software systems and chose an *additive model* to make the uncertainty of the options' and interactions' performance influence explicit. We start with a model that takes the form of Equation 4 (which represents the state of the art) with two differences:

(1) Instead of scalar influences $\beta \in \mathbb{R}^n$, we use a probability vector $\hat{\beta}$, whose elements each have a PDF and form the coefficients as explained in Section 2.2.
(2) We use the filtered options and interactions $\mathbb{V}$ from Section 3.2.1 and thus enable our model to capture non-linear performance influence:

$$\begin{aligned} \Pi_{ep}(c) = \; & \hat{\beta}_0 + \hat{\beta}_1 \cdot c(o_1) + \cdots + \hat{\beta}_n \cdot c(o_n) \\ & + \hat{\beta}_{n+1} \cdot c(o_{n+1}) + \cdots + \hat{\beta}_{n+|\mathbb{I}|} \cdot c(o_{n+|\mathbb{I}|}) \end{aligned} \tag{12}$$

To infer the distribution of an option, we need to specify a prior distribution for the probability vector $\hat{\beta}$. This distribution should be continuous (i.e., defined over all $x \in \mathbb{R}$) and have non-zero mass for any $x \in \mathbb{R}$, not to exclude certain values entirely. For performance modeling, we choose the normal distribution $\mathcal{N}(\mu, \sigma)$. It has a mode that, other than the uniform distribution, lets us encode an influence area of high probability. That is, an option's or interaction's influence has a normally distributed probability to fall into an interval to be inferred by probabilistic programming. Note that, even if a normal distribution is not the best fit for all random variables, Bayesian inference can adjust them. We describe how to determine the parameters for chosen prior distributions, such as the mean μ and the standard deviation σ for the normal distribution $\mathcal{N}(\mu, \sigma)$, in Section 3.3.

At this point, we have constructed Π_{ep}, a model that incorporates epistemic uncertainty in $\hat{\Pi}$. To account for aleatoric uncertainty (i.e., the uncertainty in the training set), we use two different models, one for homoscedastic (constant variance) and one for heteroscedastic (variance depending on true performance) aleatoric uncertainty. These models build on Π_{ep}. We adopt the common prior of a normal distribution for both models.

Homoscedastic Model. If we assume that the variance of uncertainty is equal for all training set samples, we can complete our Bayesian model with a normal distribution around $\Pi_{ep}(c)$:

$$\Pi_{ho}(c) = \mathcal{N}(\Pi_{ep}(c), \sigma) \tag{13}$$

This normal distribution is modeled as an additional random variable, whose σ parameter captures the variance of absolute errors in training set samples.

Heteroscedastic Model. To account for errors in the training set that are relative to the training set sample performance, we introduce Π_{rel}, a random variable that captures uncertainty about the error ratio. As an error ratio is in $\mathbb{R}_{>0}$ (i.e., a continuous, positive variable), we choose the Gamma distribution as prior for Π_{rel}. The Gamma distribution with a shape a and a spread parameter b can take a (possibly skewed) bell shape with non-negative values:

$$\Pi_{rel} = \mathrm{G}(a, b) \tag{14}$$

Similar to the homoscedastic model, we define the heteroscedastic model as a normal distribution around $\Pi_{ep}(c)$, but with the product of the epistemic performance prediction and the relative error ratio Π_{rel} as standard deviation:

$$\Pi_{he}(c) = \mathcal{N}(\Pi_{ep}(c), \Pi_{ep}(c) \cdot \Pi_{rel}) \tag{15}$$

3.3 Prior Estimation

Regular Bayesian inference requires the user to estimate prior distributions for the model's random variables from domain knowledge or personal experience. Distributions that are too uninformative (i.e., very wide) can lead to a hold of the inference, whereas distributions that are too informative will also slow down inference if they are imprecise [31]. Our approach automatically chooses which options and interactions are modeled as random variables, such that the user does not need to know which random variables need priors beforehand. For that reason, we employ an automatic prior estimation following the *empirical Bayes approach* [29], which differs from the regular Bayesian approach in that it estimates priors from the training data. As a result, every aspect of Bayesian modeling is automated for the user.

3.3.1 Epistemic Uncertainty Priors. We capture epistemic uncertainty in our Bayesian model in random variables for the base influence and the influences for options and interactions, whose assumed normally distributed priors rely on means μ and standard deviations σ.

We propose a prior estimation algorithm that uses the influence values of other additive models to estimate priors. As models, we use instances of Elastic Net [48] with r evenly distributed ratios of $l_1 \in [0, 1]$. For $l_1 = 1$, Elastic Net behaves like Lasso, for $l_1 = 0$ it behaves like Ridge regression and it interpolates the error functions of both approaches for $0 < l_1 < 1$. We fit 50 Elastic Nets evenly distributed with l_1 on the training set. This way, we obtain a set of 50 models $\mathbb{M}$ with different performance influences $I(\cdot)$ for the previously selected options and interactions. Next, we determine the empirical distribution of influences for each option and interaction:

$$\hat{I}_{\mathbb{M}}(I) = \{ I_m(I) \mid \forall m \in \mathbb{M} \} \tag{16}$$

We could use the mean and standard deviation of $\hat{I}_{\mathbb{M}}$ as prior μ and σ for each option and interaction. However, not all models in $\mathbb{M}$ will fit the training data well. To reduce the influence for unfit models, we weigh each model according to its average error on the training set $\bar{\epsilon}^{-1}(\cdot)$:

$$w = \left\{ \frac{\bar{\epsilon}^{-1}(m_i)}{\sum_{j=1}^{|\mathbb{M}|} \bar{\epsilon}^{-1}(m_j)} \;\middle|\; \forall m_i \in \mathbb{M} \right\} \tag{17}$$

We compute the weighted mean $\mu_w(t)$ and weighted standard deviation $\sigma_w(t)$ for a specific option or interaction t as follows:

$$\mu_w(t) = \frac{1}{\sum_{j=1}^{|\boldsymbol{w}|} w_j} \sum_{\forall i \in \hat{I}_{\mathbb{M}}(t)} w_i \cdot i \tag{18}$$

$$\sigma_w(t) = \sqrt{\frac{1}{\sum_{j=1}^{|\boldsymbol{w}|} w_j} \sum_{\forall i \in \hat{I}_{\mathbb{M}}(t)} w_i \cdot (\mu_w(t) - i)^2} \tag{19}$$

We added the tuning parameter α to enable polynomial weighting. That is, the influence of models with the lowest average error $\bar{\epsilon}$ is increased for $\alpha > 1$. In a pre-study, we empirically evaluated different values for α and found that $\alpha = 3$ yields accurate priors.

3.3.2 Aleatoric Uncertainty Priors. We model aleatoric uncertainty (i.e., uncertainty in each training set sample) as a normal distribution for the homoscedastic model Π_{ho} and as a gamma distribution as the relative uncertainty in the heteroscedastic model Π_{he}. We build the set of all absolute prediction errors for all models $m \in \mathbb{M}$ over the samples in the training set and fit a normal distribution using maximum likelihood estimation to estimate a prior for the aleatoric uncertainty in Π_{ho}. Likewise, we estimate a prior for the gamma distribution in Π_{ho}, but we compute relative prediction errors, instead, to model the error ratio (cf. Equation 15).

3.4 Bayesian Inference and Prediction

As discussed in Section 2.3, Bayesian inference uses prior assumptions on PDFs of random variables that form a generative model, called likelihood, to compute a posterior, that is, an updated belief about the random variable's PDFs. Unfortunately, the posterior to many Bayesian inference problems cannot be computed directly, so recent research in this field has developed algorithms that can estimate the posterior approximately. Two notable classes of inference algorithms are variational inference and Markov chain Monte Carlo [23].

Variational inference algorithms tune the prior distribution's parameters without changing the types of the distributions (i.e., a prior normal distribution stays a normal distribution) [30]. This method is preferred for quick results that do not need to be precise.

Markov chain Monte Carlo (MCMC) algorithms draw samples from the posterior distributions and are able to estimate arbitrary posterior distributions in theory (a prior normal distribution may

by transformed to a skewed distribution). MCMC algorithms are considered more precise, but also slower than variational inference.

We follow a combined approach by first estimating an approximate solution with variational inference and subsequently finetune with the *No-U-Turn Sampler (NUTS)* [13], an MCMC algorithm. We allow 200:000 iterations for variational inference, but abort on convergence. NUTS uses the intermediate result of variational inference and draws 5000 samples from the posterior distributions in total, from which 1000 are reserved for internal tuning. In the ideal case, we obtain 4000 samples of each random variable's posterior distribution, which enables analysis of uncertainty at a high resolution.

Prediction. To predict the performance of a configuration c, we insert c's option selection values into $c(o_1), \ldots, c(o_n)$ and determine active interactions according to Equation 9. We can now draw a number of posterior samples to approximate the distribution for the prediction. Increasing the number of posterior samples makes the approximation more accurate, but also slows down prediction. We draw 1000 posterior samples to yield a good approximation. With this approximation, we can make different kinds of predictions, for which we introduce individual notations. The most informative kind of prediction is the sampled approximation itself (e). Using e, we can compute a confidence interval for a desired confidence $_{ci} \in [0\,\%, 100\,\%]$ ($\bar{}$). This yields the interval around the mode of prediction over which the predicted distribution integrates to $_{ci}$. We use $\bar{}$ to indicate the 95 % confidence interval by default. We can also use mode of the approximation as a single-point estimate prediction ($\hat{}$).

4 EVALUATION

To evaluate our approach, we state three research questions that are in line with related work and are also concerned with the new possibilities of obtaining a confidence interval for performance predictions. Specifically, we answer the following research questions:

> **RQ$_1$:** Can we accurately predict performance as a scalar value with probabilistic programming?

This research question places our approach in relation to a state-of-the-art approach that resorts only to a scalar value. Although this is not the main usage scenario, we evaluate whether our approach has a comparable accuracy.

> **RQ$_2$:** Can we accurately predict performance in terms of a confidence interval with probabilistic programming?

RQ$_2$ refers to the ability that users can specify a confidence interval of predictions. This can substantially effect prediction accuracy and evaluates the strength of our approach.

> **RQ$_3$:** How reliable are predicted confidence intervals?

The third research questions aims at providing a deeper understanding of confidence intervals and incorporated uncertainties in our approach. We evaluate whether the confidence intervals truly capture the uncertainty in the predictions.

Table 1: Overview of the subject systems with domain, number of valid configurations $|C|$, number of options $|O|$, and the kind of performance for prediction.

	Domain	$\lvert C \rvert$	$\lvert O \rvert$	Performance
7z	File archive utility	68 640	44	Compression time
BDB-C	Embedded database	2 560	18	Response time
Sune	Multigrid solver	2 304	32	Solving time
HIPA^{cc}	Image processing	13 485	54	Solving time
HSQLDB	Java-based database	864	21	Energy consumption
JavaGC	Garbage collector	193 536	39	Time
LLVM	Compiler infrastructure	1 024	11	Compilation time
lrzip	File archive utility	432	19	Compression time
Polly	Code optimizer	60 000	40	Runtime
PSQL	Database system	864	14	Energy consumption
VP9	Video encoder	216 000	42	Encoding time
				Energy consumption
x264	Video encoder	1 152	16	Encoding time
				Energy consumption

4.1 Subject Systems

For our experiments, we use 12 real-world configurable software systems that have been used in the literature, as presented in Table 1. We use measured execution time as performance for 10 subject systems from Kaltenecker et al. [18]. For VPXENC and x264, we have additionally measured energy consumption with a different workload. In addition, we consider energy consumption for two further subject systems: PostgreSQL (short PSQL) and HSQLDB [43]. A further description of the systems including the used benchmarks is given at our supplementary Web site[2].

We adopt the procedure of extracting training and test sets from each system's measurement data from Kaltenecker et al. [18]. That is, we apply t-wise sampling with $t \in \{1, 2, 3\}$ to obtain three training test sets, T_1, T_2, T_3, of different sizes. Each system's whole population (i.e., all measurements) form its test set.

4.2 Setup

We implement our approach with the *PyMC3* [31] framework. PyMC3 offers implementations for MCMC, variational inference, as well as confidence interval computation for model parameters and predictions. For maximum likelihood prior estimations, we rely on *SciPy* [42]. The result is a performance prediction tool based on probabilistic programming, *P4* for short.

To answer our research questions, we infer Bayesian models with absolute and relative error with P4 for the chosen subject systems using three training sets T_1, T_2, T_3 on a cluster of machines with Intel Xeon E5-2690v2 CPU and 64GB memory. For the ten subject systems by Kaltenecker et al. [18], we use the training sets provided at their supplementary Web site. For the remaining subject systems, we sample new training sets with SPL Conqueror [35].

For $t = 1$, t-wise sampling is equal to option-wise sampling, which yields $n = |O|$ samples. Since we want to evaluate our approach also for learning interactions among options, creating $n + |I|$ random variables leads to a modeling problem with more variables

[2] https://git.io/JUfjy or an archived version at https://archive.softwareheritage.org/swh:1:dir:5a525f45ec77dbe982081e7f8159e9541391725e/

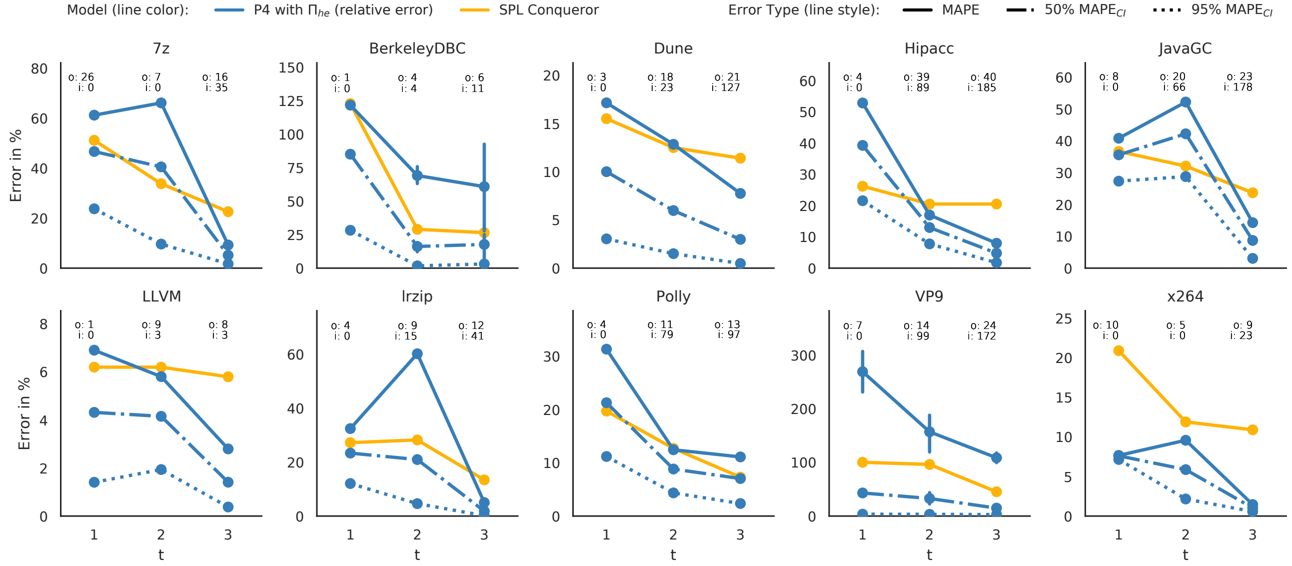

Figure 3: Scalar Mean Absolute Percentage Error (MAPE) of SPL Conqueror compared to the MAPE and interval predictions $MAPE_{CI}$ for confidence levels 50 % and 95 % of P4, with absolute error (Π_{ho} and relative error (Π_{he}) for t-wise sampled training sets. For each subject system's training set, we specify P4's model size in terms of the number of modeled options (o) and interactions (i) below the system's name.

than observations. We avoid this situation by excluding interactions from our model for $\mathcal{T}_1$. This might affect prediction accuracy especially compared to other approaches that do not exclude interactions. We will discuss this in RQ_1.

To account for stochastic elements in MCMC, we run the inference for each system's training set with 5 repetitions. The inference took 418 s, on average, with the times ranging from 98 s for the smallest models and 4 h for the largest. Among all inferences, two failed despite our training set preprocessing in Section 3.1.

4.3 RQ_1: Accuracy of Scalar Predictions

4.3.1 Setup. We chose SPL Conqueror for comparison because it shares the additive model structure with our approach and is used as baseline in the literature [18, 28]. For comparison, we rely on accuracies of SPL Conqueror as reported by Kaltenecker et al. [18]. That is, we consider for RQ_1 the ten subject systems that the original authors have used. Another benefit is that Kaltenecker et al. provided raw measurements of the whole population, so we have a reliable ground truth.

We use the inferred performance-influence models to predict the performance of the whole populations of our subject systems. We adopt the *Mean Absolute Percentage Error* (MAPE) from previous work [18] to quantify prediction accuracy. That is, we first compute the absolute percentage error (APE) for each configuration $c \in C$ with the measured performance $\Pi_{true}(c)$ and predicted scalar performance $\hat{\Pi}(c)$ for our models Π_{ho} and Π_{he}:

$$\mathrm{APE}(c) = \frac{|\Pi_{true}(c) - \hat{\Pi}(c)|}{\Pi_{true}(c)} \tag{20}$$

We then compute the MAPE as the average over all APEs:

$$\mathrm{MAPE}(C) = \frac{\sum_{c \in C} \mathrm{APE}(c)}{|C|} \tag{21}$$

4.3.2 Results. As Figure 3 shows, P4 achieves MAPE scores comparable to SPL Conqueror. Table 2 provides a more fine-grained view. We see that the overall accuracy is higher when using SPL Conqueror, which is to be expected, as only the mode is taken from the performance distribution provided as predictions by our approach. Nevertheless, we observe that, for many systems, especially when using the relative error Π_{he}, we obtain a similar or even better prediction indicated by underscored values. The mean error is, thus, distorted by some larger outliers, such as for HIPA^{cc} and VP9. These systems have many alternative options, so there is a larger uncertainty involved. Since we are not using the provided confidence interval, we deprive our approach of its strength.

Interestingly, compared to $\mathcal{T}_1$, for some subject systems P4 performs worse on $\mathcal{T}_2$. The reason is that the increased number of random variables in P4, due to the additional modeling of interactions, requires more measurements as provided by $\mathcal{T}_2$ to effectively infer performance distributions. Moreover, we see a clear trend that, with an increasing number of measurements, P4 closes the gap in prediction accuracy with SPL Conqueror and even outperforms it for $\mathcal{T}_3$ and Π_{he} for 7 out of 10 systems.

To answer RQ_1, our approach achieves the accuracy of state-of-the-art scalar predictions when a sufficient number of measurements is provided. In the case of fewer measurements, the overhead of learning probability distributions leads to more inaccurate predictions.

Table 2: Scalar Mean Absolute Percentage Error (MAPE) of SPL Conqueror (short SPLC) compared to the MAPE and confidence interval predictions MAPE ($\mathrm{MAPE_{CI}}$) of P4, with absolute error ($_{ho}$) and relative error ($_{he}$) for t-wise sampled training sets. Best scalar MAPE values for each training set are shaded light gray, best overall MAPE values are shaded dark gray.

	SPLC MAPE			$_{ho}$ MAPE			$_{he}$ MAPE			$_{ho}$ -$\mathrm{MAPE_{CI}}$ (95 %)			$_{he}$ $\mathrm{MAPE_{CI}}$ (95 %)		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
7z	51.2	33.8	22.6	70.8	87.5	45.7	61.3	66.2	9.3	7.1	4.8	0	23.8	9.6	1.7
BDB-C	122.9	29	26.5	123.9	58.8	31.3	121.8	69.2	60.9	89.5	2.7	0	28.3	1.7	3.1
Dune	15.5	12.5	11.4	17.2	13.8	9.2	17.1	12.8	7.7	1.8	0.4	0.1	3	1.5	0.5
Hipacc	26.2	20.5	20.5	53	17.8	9.4	52.8	17	8	30.5	5.6	0.3	21.6	7.8	1.7
JavaGC	36.7	32.1	23.7	40.9	65.1	33.4	40.8	52.2	14.3	24.6	16.6	0.7	27.4	28.8	3.1
LLVM	6.2	6.2	5.8	6.9	5.8	2.8	6.9	5.8	2.8	0.2	1.3	0.2	1.4	1.9	0.4
Polly	19.7	12.7	7.3	31	11.5	11.1	31.3	12.4	11.1	6.9	1.3	0.8	11.2	4.3	2.4
VP9	100.3	96.3	45.3	160.3	109.5	88.5	269.4	157.2	108.7	1.9	0.6	0.1	3.5	3.3	2.7
lrzip	27.2	28.2	13.4	45.8	141.7	153.2	32.4	60.1	5	8.3	0	0	12.1	4.6	0.2
x264	20.9	11.9	10.9	9.8	16.5	4.7	7.7	9.6	1.5	0.1	0	0	7.2	2.2	0.6
Mean	42.7	28.3	18.7	56	52.8	38.9	64.2	46.3	22.9	17.1	3.3	0.2	14	6.6	1.6

4.4 RQ$_2$: Accuracy of Confidence Intervals

4.4.1 Setup. Confidence intervals with confidence $_{CI} \in (0\,\%, 100\,\%)$ specify a range in which a given PDF integrates to $_{CI}$. For predictions, a 95 % confidence interval specifies a performance range for which the model is 95 % confident that it contains the true performance value of the corresponding configuration. Consequently, we can expect the true performance to lie outside the 95 % confidence interval in 5 % of predictions. Although we can expect to always capture the true performance with a 100 % confidence interval, such an interval will likely approach $(-\infty, +\infty)$ for PDFs that are defined over $\mathbb{R}$.

Similar to RQ$_1$, we use a relative error metric to answer RQ$_2$. However, for RQ$_2$, we use P4 to predict confidence intervals as prediction, which is the actual strength and novel feature of our approach. Instead of using the APE of a scalar prediction, we compute the confidence interval's APE_{CI} with relation to the closest endpoint of the confidence interval $\bar{}$ for an outlying true performance; we define $\mathrm{APE}_{CI} = 0$ for an confidence interval that includes the measured performance:

$$\mathrm{APE}_{CI}(c) = \frac{\min_{\forall \in \bar{}_{\alpha}(c)} \; true(c)}{true(c)} \tag{22}$$

Hence, the $\mathrm{MAPE_{CI}}$ is the average over all APE_{CI}, similar to Equation 21. For our models $_{ho}$ and $_{he}$, we report the $\mathrm{MAPE_{CI}}$ for predicted confidence intervals with $_{CI} = 95\,\%$ for highly confident predictions and $_{CI} = 50\,\%$ for less confident predictions, for which we expect a narrower interval and, consequently, a higher error.

4.4.2 Results. The dotted lines in Figure 3 illustrate a substantial decrease in prediction error when using a confidence interval rather than a scalar prediction. Note that we report in Figure 3 only $\mathrm{MAPE_{CI}}$'s for $_{he}$; we provide similar results for $_{ho}$ at our supplementary Web site. Table 2 provides further data for $_{ho}$. It reveals that the predicted confidence intervals for 7z, BDB-C, lrzip, and x264 contain all measured performance values when training the absolute model $_{ho}$ on $\mathbb{T}_3$.

Table 3: Five most uncertain features measured by the mean relative confidence interval $^{?}_{j}$ according to Equation 25, of models trained on $\mathbb{T}_1$. Values for the variance inflation factor (VIF) larger 10 are in dark gray (highly problematic) and values between 5 and 10 are in light gray (moderately problematic). Files_30 & BlockSize_1024 were removed from $\mathbb{T}_3$.

		$\mathbb{T}_1$		$\mathbb{T}_3$	
System	Option	?	VIF	?	VIF
VP8 (E)	threads_4	24.64	10.50	0.15	3.24
VP9 (T)	bitRate_1500	16.93	10.58	2.84	2.90
7z (T)	Files_30	11.86	7.13	–	1.67
7z (T)	BlockSize_1024	11.44	7.13	–	1.63
VP9 (T)	variableBitrate	4.91	16.90	0.86	1.99

We illustrate how more training samples allow P4 to decrease uncertainty in internal parameters to achieve better prediction accuracy using the *variance inflation factor* (VIF). The VIF is an indicator for multicollinearity, which can be computed for the activation values of an option o_j in the training set $\mathbb{T}$. It is based on the coefficient of determination R^2. To determine R^2 for an option o_j, we fit a linear regression function f_j to predict whether o_j is active in a configuration $c \cap o_j$ with the remaining options as predictors.

We compute the VIF as follows:

$$\mathrm{VIF}_j = \frac{1}{1 \quad R_j^2} \tag{23}$$

$$R_j^2 = 1 \quad \frac{\sum_{\forall c \in \mathbb{T}} \left(c(o_j) \quad c(o_j) \right)^2}{\sum_{\forall c \in \mathbb{T}} \left(c(o_j) \quad f_j(c \cap o_j) \right)^2} \tag{24}$$

A VIF of 0 indicates an option with no multicollinearity in the training set, while higher values mark increasingly problematic multicollinearity. We adopt the thresholds of 5 and 10 [27, 44] to indicate moderate and highly problematic multicollinearity, respectively.

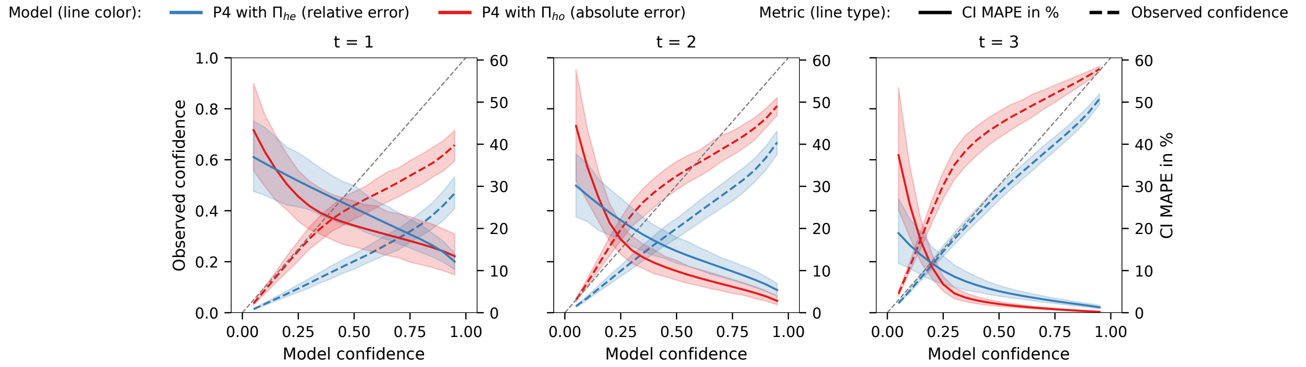

Figure 4: MAPE$_{\text{CI}}$ depending on model confidence (solid) versus uncertainty calibration (dashed) for t-wise training sets, aggregated over all subject systems. Gray dashed line indicates ideal calibration.

Although we could use the VIF as a filter for feature selection (cf. Section 3.3) to remove options with high multicollinearity in the training set, the computational effort required to calculate all R_j^2 makes it infeasible in practice. Hence, we compute the VIF only for the 5 most uncertain options in T_1 to analyze whether multicollinearity is a possible cause for uncertainty of options’ influences. To compute the uncertainty of an option influence ${}^{?}_{j}$, we use its confidence interval $\bar{}_j$ and point estimate $\hat{}_j$. To remove the influence of differing influence scales between software systems, we determine the relative confidence interval width as the ratio of the absolute confidence interval width $|\bar{}_j|$ and the point estimate:

$$ {}^{?}_{j} = \frac{|\bar{}_j|}{\hat{}_j} \tag{25} $$

Looking at Table 3, we see that all five options exhibit either a moderate or even a high VIF for the training set T_1. This points to a situation in which the learning procedure cannot safely assign a performance ratio to the specific option. Investigating this closer, four options are part of an alternative group despite our efforts to avoid multicollinearity by removing one alternative from each alternative group. For option threads_4, we found that it was active in almost every configuration (13 out of 16), reducing the contained information according to Equation 7.

To further confirm our hypothesis that multicollinearity can be a possible cause, we show in Table 3 the uncertainty ${}^{?}_{j}$ and the VIF for these five options using the larger training set T_3. We see a substantial reduction in uncertainty for three options in line with the reduction of the VIF. This strongly indicates that a reduced multicollinearity reduces also the uncertainty of an option’s influence on performance. Options Files_30 and BlockSize_1024 have no uncertainty as they were chosen by P4 to be removed from the alternative group in T_3.

Overall, ${}_{ho}$ yields better results than ${}_{he}$ in most cases, but both approaches always show substantially lower relative errors than scalar predictions. Of course, it would be easy for a model to predict all performance values correctly with a sufficiently large confidence interval. However, our findings for RQ$_3$ demonstrate that P4’s prediction confidence intervals are reliable, as we will discuss in Section 4.5.

> To answer RQ$_2$, using confidence intervals to frame the confidence of predictions substantially reduces the prediction error. That is, our approach is able to model the uncertainty as well as the true performance distributions accurately.

4.5 RQ$_3$: Reliability of Confidence Intervals

4.5.1 Setup. As predictions, our approach can yield confidence intervals with any given confidence level ${}_{CI} \in [0\,\%; 100\,\%]$. We call a model’s predicted confidence intervals reliable if predictions with an ${}_{CI}$ confidence interval contain the measured performance with a similar observed frequency ${}_{obs}$ (i.e., ${}_{obs}({}_{CI}) \approx {}_{CI}$). To compute the observed frequency ${}_{obs}({}_{CI})$ for an ${}_{CI}$ confidence interval, we first define the function within, which returns 1 if the measured performance ${}_{true}(c)$ lies in a predicted confidence interval $\bar{}(c)$, and 0, otherwise:

$$ \text{within}({}_{true}(c), \bar{}(c)) = \begin{cases} 1 & {}_{true}(c) \in \bar{}(c) \\ 0 & \text{else} \end{cases} \tag{26} $$

Second, the observed frequency is computed as the average of within over all configurations of a subject system and their measured performances ${}_{true}(c)$:

$$ {}_{obs}({}_{CI}) = \frac{\sum_{c \in C} \text{within}({}_{true}(c), \bar{}(c))}{|C|} \tag{27} $$

If ${}_{CI} > {}_{obs}({}_{CI})$, the predicted confidence interval is inaccurate more often than we expect and should have been broader; conversely, the predicted confidence interval should be more narrow and thus more informative if ${}_{CI} < {}_{obs}({}_{CI})$. Since using confidence intervals for performance predictions is novel, we have no baseline to which we can compare. Hence, we report the observed frequencies for confidence levels ${}_{CI}$ from 5 % to 95 % in steps of 5 % as well as the average error in percentage to answer RQ$_3$. In addition, we report the MAPE$_{\text{CI}}$ for all confidence intervals.

4.5.2 Results. Figure 4 shows a calibration plot that compares ${}_{CI}$ with ${}_{obs}$ using dashed lines. A model with ${}_{CI} = {}_{obs}$ for all ${}_{CI}$ would yield values along the dashed gray diagonal. Values above the diagonal indicate too broad confidence intervals (i.e., our predictions are more accurate than they should be), values below it signal confidence intervals that are too narrow. The solid lines

in Figure 4 show the mean MAPE_{CI} over all subject systems for both the relative and the absolute model. The shaded area around it constitutes a 95 % confidence interval.

When analyzing the dashed lines, we see that using the absolute error $_{ho}$ yields intervals that are closer to the diagonal than when using the relative error $_{he}$. Moreover, there is a clear trend that, when using more measurements, the intervals become either nearly perfectly aligned or are underestimating the models prediction accuracy. Hence, we see a picture that resembles the picture when using the mode for scalar performance prediction: The approach requires a certain number of measurements to become accurate, but then works robustly.

We can make a further interesting observation when comparing the confidence intervals (dashed lines) with the MAPE_{CI} (solid lines). First and most importantly, we see that using confidence intervals of varying sizes has a clear monotonic relationship with the prediction error. That is, increasing the interval decreases the error. Second, the errors fall rapidly, especially for T_2 and T_3, already when using a narrow interval, such as 25 %. This is good news as this clearly indicates that narrow confidence intervals yield accurate predictions. Third, we observe that (for the solid lines) the uncertainty is higher with fewer measurements, as indicated by the colored area. That is, the model is aware that the measurements are insufficient to actually make trustworthy predictions. This is a feature missing in all scalar prediction approaches. For example, for SPL Conqueror, we have no clue whether the model is confident with a certain prediction. With P4, we have a means to quantify this confidence.

> To answer RQ_3, with enough measurements, our approach yields confidence intervals that contain the true value with a frequency that matches the specified confidence. Even with our smallest training set T_1, confidence intervals with higher specified confidence contain the true value more often.

5 THREATS TO VALIDITY

Threats to internal validity arise from measurement bias. We reuse a measurement set from a recent paper whose authors controlled for this bias by repeating the measurements several times [18]. A threat to construct validity may arise from the model construction process in PyMC3. We selected probability distributions for the random variables based on typical least squares error distributions and best practices for regression modeling in probabilistic programming. External validity refers to the generalizability of our approach. Our data set comprises 12 different subject systems of varying domains and sizes. Moreover, we assessed different properties, such as energy consumption and response time. We made similar observations for all systems such that we are convinced that our approach works on a large and practically relevant class of configurable software systems.

6 RELATED WORK

There is a number of approaches in the field of performance modeling. CART [5] and its improved version DECART [6] use rule-based models to accurately learn performance models with a small number of samples. FLASH [25] is a sequential model-based method that relies on active learning to fit CART [5] models more efficiently. DeepPerf [7] is a deep learning-based approach, which uses sparse neural networks for performance estimation. Zhang et al. propose a framework to model performance influence with Fourier approximation [46], whereas Nair et al. employ spectral learning with dimensionality reduction [24]. SPL Conqueror learns an additive model with step-wise selection of new terms [33]. None of the proposed approaches considers uncertainty in predictions and in the internal representation of influences, producing only scalar estimates.

Notably, the need for incorporating uncertainty in performance modeling has been argued before by Trubiani and Apel [39]. While there are already considerations in other fields for both epistemic and aleatoric uncertainty, such as for computer vision [19], for software engineering, there are only approaches that model some kind of epistemic uncertainty. Antonelli et al. have incorporated uncertainty by allowing two parameters of a performance index for cloud computing systems to be uncertain and thus adapt to changing hardware [1]. Another approach by Arcaini et al. transforms a feature model into two Queueing Networks — one each for the two variants with minimal and maximal performance — and thereby represents uncertainty in performance [2]. To the best of our knowledge, we are the first to follow Trubiani and Apel's call to incorporate both epistemic and aleatoric uncertainty in performance modeling of configurable software systems.

7 SUMMARY

Existing approaches for performance-influence modeling provide only scalar predictions based on modeling influences of options and interactions with scalar values. We argue that these approaches neglect uncertainty arising from the modeling and measurement process. We propose a novel performance-influence modeling approach that incorporates uncertainty explicitly and yields confidence intervals alongside scalar point-estimate predictions. This way, we provide not only a singular number as a performance estimate, but also a posterior distribution and a confidence in which range a performance value lies. Our experiments with 12 real-world software systems show that our implementation, P4, yields scalar prediction accuracies that match the state of the art when provided with a sufficient number of measurements. Further evaluation shows that the confidence intervals provided are reliable and, when used for prediction, achieve competitive accuracies.

The analysis of our trained models indicates that options that are selected in almost every configuration can reduce the amount of information contained in a training set, rendering the option's influence uncertain. This observation calls for a shift in current sampling strategies by taking the information gain more into account, as compared to coverage or uniformness. P4 showed its potential especially with pairwise and triple-wise sampled training sets. Improving P4 for small training sets, hence, remains an open issue. A possible remedy are P4's option influence uncertainties, which may be facilitated in an active learning setup to learn more efficiently.

ACKNOWLEDGMENT

Siegmund's and Apel's work has been funded by the German Research Foundation (SI 2171/3-1, SI 2171/2, and AP 206/11-1).

REFERENCES

[1] Fabio Antonelli, Vittorio Cortellessa, Marco Gribaudo, Riccardo Pinciroli, Kishor S. Trivedi, and Catia Trubiani. 2020. Analytical Modeling of Performance Indices under Epistemic Uncertainty Applied to Cloud Computing Systems. *Future Generation Computer Systems* 102 (2020), 746–761. https://doi.org/10.1016/j.future.2019.09.006

[2] Paolo Arcaini, Omar Inverso, and Catia Trubiani. 2020. Automated Model-Based Performance Analysis of Software Product Lines under Uncertainty. *Journal of Information and Software Technology (IST)* 127 (2020), 106371. https://doi.org/10.1016/j.infsof.2020.106371

[3] Donald E. Farrar and Robert R. Glauber. 1967. Multicollinearity in Regression Analysis: The Problem Revisited. *The Review of Economics and Statistics* 49 (1967), 92–107. https://doi.org/10.2307/1937887

[4] Vibhav Gogate and Rina Dechter. 2006. A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Principles and Practice of Constraint Programming - CP 2006*. Springer, 711–715. https://doi.org/10.1007/11889205_56

[5] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311. https://doi.org/10.1109/ASE.2013.6693089

[6] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering* 23 (2018), 1826–1867.

[7] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106. https://doi.org/10.1109/ICSE.2019.00113

[8] Huong Ha and Hongyu Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 470–480. https://doi.org/10.1109/ICSME.2019.00080

[9] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 517–528. https://doi.org/10.1109/ICSE.2015.69

[10] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-Tuning System for Big Data Analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 261–272.

[11] R. Carter Hill and Lee C. Adkins. 2007. Collinearity. In *A Companion to Theoretical Econometrics*. John Wiley & Sons, Ltd, Chapter 12, 256–278. https://doi.org/10.1002/9780470996249.ch13

[12] Arthur E. Hoerl and Robert W. Kennard. 1970. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics* 12 (1970), 55–67. https://doi.org/10.1080/00401706.1970.10488634

[13] Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *The Journal of Machine Learning Research* 15 (2014), 1593–1623.

[14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Springer. https://doi.org/10.1007/978-1-4614-7138-7

[15] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 39–48. https://doi.org/10.1109/MASCOTS.2016.17

[16] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 46. https://doi.org/10.1145/2362536.2362547

[17] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Software* 37, 4 (2020), 58–66. https://doi.org/10.1109/MS.2020.2987024

[18] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094. https://doi.org/10.1109/ICSE.2019.00112

[19] Alex Kendall and Yarin Gal. 2017. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 5580–5590.

[20] Armen Der Kiureghian and Ove Ditlevsen. 2009. Aleatory or Epistemic? Does It Matter? *Structural Safety* 31 (2009), 105–112. https://doi.org/10.1016/j.strusafe.2008.06.020

[21] Sergiy Kolesnikov, Judith Roth, and Sven Apel. 2014. On the Relation between Internal and External Feature Interactions in Feature-Oriented Product Lines: A Case Study. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 1–8. https://doi.org/10.1145/2660190.2660191

[22] Alan Miller. 2002. *Subset Selection in Regression*. CRC Press. https://doi.org/10.1201/9781420035933

[23] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

[24] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Faster Discovery of Faster System Configurations with Spectral Learning. *Automated Software Engineering* 25 (2017), 247–277. https://doi.org/10.1007/s10515-017-0225-2

[25] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *Transactions on Software Engineering* 46 (2020), 794–811.

[26] Radford M Neal. 1993. *Probabilistic Inference Using Markov Chain Monte Carlo Methods*. Department of Computer Science, University of Toronto.

[27] Robert M. O'Brien. 2007. A Caution Regarding Rules of Thumb for Variance Inflation Factors. *Quality & Quantity* 41 (2007), 673–690. https://doi.org/10.1007/s11135-006-9018-6

[28] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 61–71. https://doi.org/10.1145/3106237.3106273

[29] Herbert E. Robbins. 1956. An Empirical Bayes Approach to Statistics. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. The Regents of the University of California. https://doi.org/10.1007/978-1-4612-0919-5_26

[30] Geoffrey Roeder, Yuhuai Wu, and David K Duvenaud. 2017. Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 6928–6937.

[31] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic Programming in Python Using PyMC3. *PeerJ Computer Science* 2 (2016), e55. https://doi.org/10.7717/peerj-cs.55

[32] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[33] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 284–294. https://doi.org/10.1145/2786805.2786845

[34] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kastner, Sven Apel, Don Batory, Marko Rosenmuller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 167–177. https://doi.org/10.1109/ICSE.2012.6227196

[35] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal* 20 (2012), 487–517. https://doi.org/10.1007/s11219-011-9152-9

[36] Ralph Smith. 2013. *Uncertainty Quantification: Theory, Implementation, and Applications*. Society for Industrial and Applied Mathematics.

[37] John R. Taylor. 1997. *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements* (2nd ed.). University Science Books. https://doi.org/10.1063/1.882103

[38] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58 (1996), 267–288.

[39] Catia Trubiani and Sven Apel. 2019. PLUS: Performance Learning for Uncertainty of Software. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE, 77–80. https://doi.org/10.1109/ICSE-NIER.2019.00028

[40] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[41] Laurens Van Der Maaten, Eric Postma, and Jaap Van den Herik. 2009. Dimensionality Reduction: A Comparative Review. *Journal of Machine Learning Research* 10 (2009).

[42] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa,

Paul van Mulbregt, and SciPy 1. 0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272.

[43] Niklas Werner. 2019. *Energy and Performance Evolution of Configurable Systems: Case Studies and Experiments*. Master Thesis. University of Passau.

[44] Jeffrey Wooldridge. 2012. *Introductory Econometrics: A Modern Approach* (5 ed.). South-Western College Pub.

[45] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 307–319. https://doi.org/10.1145/2786805.2786852

[46] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 365–373. https://doi.org/10.1109/ASE.2015.15

[47] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. ACM, 338–350. https://doi.org/10.1145/3127479.3128605

[48] Hui Zou and Trevor Hastie. 2017. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67 (2017), 301–320. https://doi.org/10.1111/j.1467-9868.2005.00503.x