Black-Box Models for Non-Functional Properties of AI Software Systems

Birte Friesel
birte.friesel@uos.de
Universität Osnabrück
Osnabrück, Germany

Olaf Spinczyk
olaf@uos.de
Universität Osnabrück
Osnabrück, Germany

ABSTRACT

Non-functional properties (NFPs) such as latency, memory requirements, or hardware cost are an important characteristic of AI software systems, especially in the domain of resource-constrained embedded devices. Embedded AI products require sufficient resources for satisfactory latency and accuracy, but should also be cost-efficient and therefore not use more powerful hardware than strictly necessary. Traditionally, modeling and optimization efforts focus on the AI architecture, utilizing methods such as neural architecture search (NAS). However, before developers can start optimizing, they need to know which architectures are suitable candidates for their use case. To this end, architectures must be viewed in context: model post-processing (e.g. quantization), hardware platform, and run-time configuration such as batching all have significant effects on NFPs and therefore on AI architecture performance. Moreover, scalar parameters such as batch size cannot be benchmarked exhaustively. We argue that it is worthwhile to address this issue by means of black-box models before deciding on AI architectures for optimization and hardware/software platforms for inference. To support our claim, we present an AI product line with variable hardware and software components, perform benchmarks, and present notable results. Additionally, we evaluate both compactness and generalization capabilities of regression tree-based modeling approaches from the machine learning and product line engineering communities. We find that linear model trees perform best: they can capture NFPs of known AI configurations with a mean error of up to 13 %, and can predict unseen configurations with a mean error of 10 to 26 %. We find linear model trees to be more compact and interpretable than other tree-based approaches.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; **Machine learning**; • **Software and its engineering** → *Extra-functional properties.*

KEYWORDS

AI, Performance Prediction, Regression Trees, Product Lines

CAIN'22, May 16–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9275-4/22/05…$15.00
https://doi.org/10.1145/3522664.3528602

ACM Reference Format:
Birte Friesel and Olaf Spinczyk. 2022. Black-Box Models for Non-Functional Properties of AI Software Systems. In *1st Conference on AI Engineering - Software Engineering for AI (CAIN'22), May 16–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3522664.3528602

1 INTRODUCTION

Models for non-functional system properties (NFP models for short) have proven to be useful for system design in a wide area of use cases. Given a system (or product line) configuration, they provide estimates for attributes such as ROM/RAM requirements, AI inference latency and throughput, or energy consumption. Having NFP models at hand allows system designers to estimate a system's properties before building or using it, compare different solutions, and even automatically configure system components to optimize relevant NFPs [15, 20].

For example, a generic AI software system for agricultural machinery must be able to handle a variety of tasks. One use case might be pixel-accurate segmentation and classification of harvested and non-harvested areas, and fine-tuning of machinery for optimal yield in an AI-supported harvest scenario. Other use cases may include bounding-box segmentation and classification of obstacles in the way of a machine, or classification of individual fruit. For each task, developers can choose from a wide range of AI architectures, corresponding AI platforms, platform-specific optimizations, and pre-trained models for transfer learning.

Although minimizing hardware cost is a general theme in the industry, acceptable trade-offs and constraints for inference on low-cost hardware depend on the use case and may be fine-tuned further by developers and customers. For instance, obstacle detection should react at least as fast as a human driver would, but doubling the hardware cost just to reduce the latency from 140 to 120 milliseconds may not be worthwhile. On the other hand, AI-driven optimization of a combine harvester has ample processing time, as changing the properties of large mechanical equipment is an inherently slow process. Depending on additional factors such as optional 5G cloud or edge offloading, cooling constraints, or over-the-air updates, even a single use case may result in multiple products with different optimization goals.

Traditional product line engineering workflows require a domain expert to judge how configuration options affect the desired non-functional properties and suggest suitable system configurations [16]. Their input is required whenever a new product (i.e., a new set of configuration constraints and optimization goals) is developed.

NFP models offer an alternative solution that does not rely on manual input. Given n individual configuration options and m non-functional properties, the idea is to perform state space exploration and benchmarks once, and automatically transform the result into a model $F : M \rightarrow \mathbb{R}^m$. The model describes how configuration options ($\vec{x} \in M$ with an n-dimensional state space M) affect individual non-functional properties ($y \in \mathbb{R}$). This way, whenever a new use case or product is developed, the already-available NFP model can be used to find suitable configurations [16]. When new hardware/software platforms or AI architectures become available, the NFP model can be updated by only running benchmarks of not-yet-known configurations.

In an AI context, NFP models can be used in two ways.

The white-box variant is similar to performance models used for Neural Architecture Search [1, 11]. It builds NFP models based on AI architecture attributes such as number of floating point operations or number of layers and hardware attributes such as number of cores or floating point performance. This allows it to predict the performance of previously unseen AI architectures, at the cost of requiring a large amount of training samples.

The black-box variant, on the other hand, distinguishes between different architectures and hardware platforms by identifier only. It should not be used to predict the performance of unseen architectures, but is well suited for exploring a set of common architectures and determining which one performs best in a specific setting.

As this paper focuses on tailoring a set of known architectures and platforms towards specific use cases, we only consider the black-box variant. We compare NFP modeling approaches from the product line engineering and machine learning communities: regression trees, model trees, and tree boosting [3, 7, 17].

We show that

- the variability within a real-world AI software system can be formally expressed as a feature model (section 2),
- both AI architecture and software/hardware configuration affect the non-functional properties of AI applications (section 4), and that
- while both regression trees and model trees are suitable for modeling the performance of AI software system configurations, non-binary model trees achieve the best interpretability and have a low generalization error (section 5).

We examine abstraction (model accuracy when querying NFPs of known configurations) and generalization (model accuracy when predicting NFPs of unseen configurations). Additionally, we examine both binary and non-binary tree structures, and how they affect model size and accuracy.

In the next section, we define the product line and feature model we use throughout this paper. We then introduce the NFP modeling methods under consideration in section 3. After presenting benchmark results (that is, how feature selection affects NFP values) in section 4, we compare complexity and accuracy of NFP models for these results in section 5. We examine related work in section 6, and conclude in section 7.

2 AI PRODUCT LINE

A *product line* is a set of components that can be assembled and configured for specific use cases (*products*) in a common domain. One of its key components is a *feature model*, or *variability model*, that formally describes how its components can be configured and how they depend on each other.

Our product line has four main sources of variability: the task (detection or segmentation) and associated AI architectures, the AI platform and associated optimization or quantization settings, runtime settings (batch size), and the hardware platform used for inference. Note that we do not consider hardware platform variability such as different memory, clock speed, or CPU/GPU core configurations; we always use all available resources for inference.

We consider 26 KERAS architectures for classification, 18 architectures downloaded from TensorFlow Hub for bounding box segmentation, and twelve architectures for pixel-accurate segmentation. Hardware platforms are Raspberry Pi 4 (4 GB version), nVidia Jetson Nano, nVidia Jetson Xavier NX, Google Coral EdgeTPU, and an i.MX 8M development kit. All of them use an operating system and TensorFlow (Lite) installation with arm64 support. The nVidia Jetson platforms use CUDA GPU acceleration for TensorFlow models; i.MX 8M uses NNAPI GPU acceleration for TensorFlow Lite models. The Coral EdgeTPU supports hardware acceleration for suitably compiled TensorFlow Lite models.

On the platform side, we use the `tf.lite.Optimize.DEFAULT` flag to convert TensorFlow models to TensorFlow Lite using *Default*, *Float16*, or *Int8* optimization. For Float16, we additionally set the preferred data type to 16-bit float. For Int8, we configure all operators and data types (including input/output) to use 8-bit integers. On the Coral EdgeTPU, we additionally use the edgetpu compiler to compile Int8 models for the EgdeTPU accelerator.

We perform Default and Float16 optimizations both with and without a representative dataset (that is meant to improve quantization by looking at representative input data). For Int8 optimization, the representative dataset is mandatory. We note that TensorFlow Lite (especially with 8-bit quantization) only supports a limited set of operators; some models cannot be converted.

Fig. 1 shows an excerpt of the feature model, excluding some hardware platforms, optimizations, and AI architectures for brevity. Filled circles indicate mandatory features that must be selected if the parent feature is selected. Child nodes with circle segments indicate alternative features: if the parent feature is selected, exactly one child feature must be selected. For example, in our use case, the AI platform must be either TensorFlow or TensorFlow Lite. In both cases, if the parent feature is disabled, all child features must be disabled as well.

Note that we do not visualize additional dependencies between different features. These include constraints such as EdgeTPU optimized TensorFlow Lite models only being available on the Coral EdgeTPU platform, AI architectures without batching support, and hardware platforms where a TensorFlow installation is not sensible due to scarce resources and lack of precompiled software packages.

With a feature model at hand, an NFP model is not far away. A naïve approach would be to simply annotate each feature node with its effect on the resulting NFP values, such as inference time or accuracy [15]. For instance, Int8 quantization might reduce inference latency by 10 %, and a Jetson Xavier might be thrice as fast as a Raspberry Pi 4.

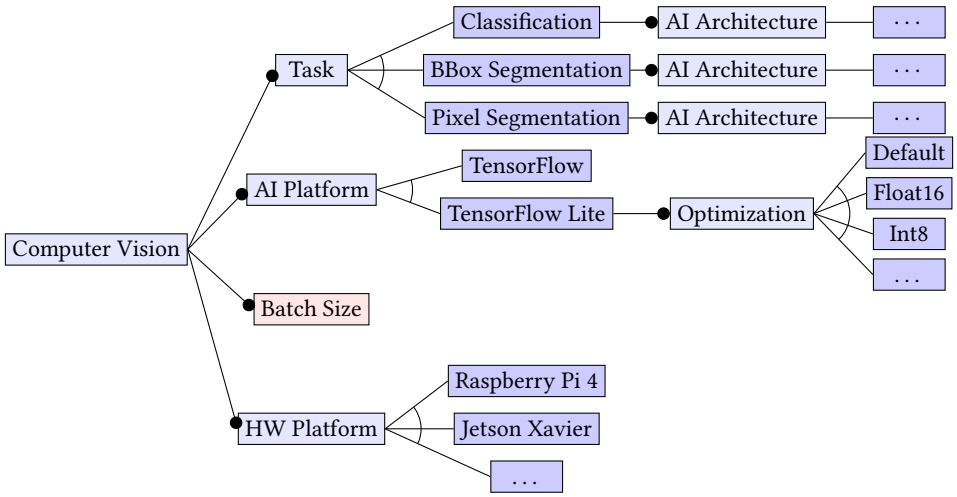

Figure 1: Feature Model excerpt for the computer vision AI product line used in this paper. Lines with filled circles indicate mandatory child features. Lines with half-circles indicate alternatives: exactly one child feature must be selected.

In most practical applications, it is not that simple [19]. There is no single, well-defined latency or throughput effect for "Raspberry Pi 4" or "Int8". Instead, latency and throughput depend on complex interactions between hardware platform, AI platform, AI architecture, and runtime settings such as batch size. Accuracy, on the other hand, depends not just on the AI architecture, but also the AI platform and optimizations such as quantization, which can decrease accuracy. The same applies to other non-functional properties, such as model size or energy use.

NFP models handle this by considering the whole product line configuration as a *feature vector* and mapping feature vectors to NFP values. For example, considering just hardware platform and batch size, the configuration space could be defined as

$$M = \{\text{Raspberry Pi 4}, \text{Jetson Xavier}, \dots\} \times \mathbb{N}_{>0}$$

and a valid configuration vector $\vec{x} \in M$ would be (Jetson Xavier, 16). For each non-functional property we are interested in, the NFP model must provide a function $f : M \to \mathbb{R}$.

In the next section, we present several modeling methods borrowed from the software product line and machine learning communities that can be applied to this task. All of them are based on regression trees.

3 MODELING METHODS

All modeling methods use observations $S = \{y_1, y_2, \dots\}$ of corresponding feature vectors $\vec{x}_1, \vec{x}_2, \dots$ to learn how a system configuration affects a single non-functional property y. Each system configuration is a vector of variable values. We use x_i to refer to the i-th configuration variable, regardless of the concrete configuration.

The configuration variables are closely related to the feature model (Fig. 1). Typically, NFP models use boolean and scalar variables. Each feature toggle, such as the Raspberry Pi 4 hardware platform, the Pixel Segmentation task, or the Float16 optimization, is associated with a single boolean variable that is true if and only if the corresponding feature is selected. Each scalar feature, such as batch size, is associated with a single scalar variable. We note that $\sum_{i \in I} x_i \leq 1$ holds for all groups I of alternative features (e.g. all AI architectures for classification).

Alternatively, we can treat each group of alternative features as a single categorial variable: $x \in \{\text{Raspberry Pi 4}, \text{Jetson Xavier}, \dots\}$ for the hardware platform, $x' \in \{\text{TensorFlow}, \text{TensorFlow Lite}\}$ for the AI platform, and so on. This is closer to the semantics of the underlying feature model, but rarely used in literature. To understand whether this is deliberate or not, we examine NFP models both with and without support for categorial variables.

We first present three regression tree variants and Linear Model Trees. These are prevalent in the software product line NFP modeling community [7]. We then examine the Extreme Gradient Boosting algorithm that is popular in machine learning applications [3].

3.1 Regression Trees

Regression Trees (also known as Classification and Regression Trees, or CART) have been introduced in 1984 and continue to be relevant to this day [2]. They can be used to identify sets of similar product-line configurations and assign a constant NFP value for each set.

Mathematically, they express a piecewise constant function $f : M \to \mathbb{R}$ by means of a binary regression tree. The input domain M is an arbitrary combination of boolean, scalar, and categorial variables. Each non-leaf node holds a binary decision such as $x_1 < 3$ or $x_2 \in \{A, B, C\}$, and each leaf holds a value defining the function output for the partial system configuration described by the path from the root to the leaf.

Fig. 2 shows an example of a regression tree for a boolean-only variability model. Leaf values $\mu(\cdots)$ are the arithmetic mean of the corresponding observations. For example, $f(x_1\overline{x_2}\overline{x_3}) = \mu(x_1\overline{x_3})$.

In practice, most CART generation algorithms are tailored towards specific input domains. Here, we consider three variants: *Scalar CART* ($\mathbb{R}^n \to \mathbb{R}$) as implemented in the Python3 scikit-learn package, *Boolean CART* ($\{0,1\}^n \to \mathbb{R}$) as used by DECART [7], and a custom *Non-Binary CART* method with support for categorial variables ($M \to \mathbb{R}$).

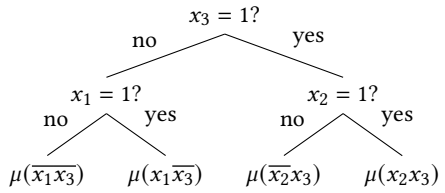

Figure 2: A regression tree. Each leaf holds a constant model μ for the corresponding partial system configuration.

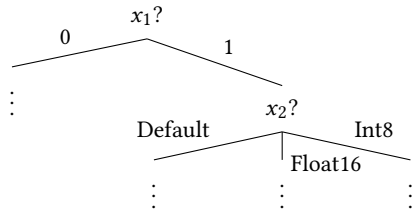

Figure 3: A non-binary regression tree with boolean variable x_1 and categorial variable x_2.

3.1.1 Scalar CART. The key idea of CART generation algorithms is to greedily select binary splits of the observations that minimize the *loss* (i.e., model error). This way, the regression tree is refined recursively until a stop criterion is satisfied. Common stop criteria are number of samples ($|S| < T$) or standard deviation ($\sigma(S) < T$) with a user-provided threshold T, or tree size and depth limits. Properly chosen, these minimize the risk of overfitting.

We briefly outline the scalar CART generation algorithm used in this paper and refer to Breiman et al. for an in-depth discussion [2].

(1) For each variable x_i, let $T_i = \{t_{i,1}, t_{i,2}, \dots\}$ be the ordered set of its unique values.
(2) If a stop criterion is satisfied: return a leaf node using the mean of observed data $\mu(S)$ as model value.
(3) For each pair $(x_i, t_{i,j})$, split S into partitions $S_{i,j,\text{left}}$ (containing only samples with $x_i \leq t_{i,j}$) and $S_{i,j,\text{right}}$ ($x_i > t_{i,j}$).
(4) Select the pair $(x_i, t_{i,j})$ with the lowest associated loss and transform it into a regression tree node.
(5) Repeat recursively with $S := S_{i,j,\text{left}}$ (left child node) and $S := S_{i,j,\text{right}}$ (right child node).

A typical loss function is the sum of squared residuals:

$$\sum_{y \in S_{i,j,\text{left}}} (y - \mu(S_{i,j,\text{left}}))^2 + \sum_{y \in S_{i,j,\text{right}}} (y - \mu(S_{i,j,\text{right}}))^2$$

We note that Scalar CART generation supports both scalar and boolean configuration variables, as $\{0, 1\} \subset \mathbb{R}$. We also note that functions expressed by CART are well-defined. For any system configuration, there is a path from the root node to a leaf node with a matching partial system configuration.

3.1.2 Boolean CART. In software product line engineering, feature models often exclusively use boolean configuration variables: features can be enabled/disabled, but do not have scalar configuration attributes. In this case, $T_i \subseteq \{0, 1\}$ for all variables, so there is just one split candidate for each variable. This allows for efficient data acquisition methods, such as DECART [7].

Instead of $S_{i,j,\text{left}}$ and $S_{i,j,\text{right}}$, we may now write $S_{i,\text{left}}$ (for $x_i = 0$) and $S_{i,\text{right}}$ ($x_i = 1$) without ambiguity. Apart from that, model generation algorithm and stop criteria remain the same.

3.1.3 Non-Binary CART. In principle, extending Scalar CART to also support categorial variables is trivial.

Let x_i be a categorial variable, and let T_i be the set of its unique values (e.g. hardware or AI architecture identifiers). Given the power set $\mathcal{P}(T_i)$, let $P_{i,j}$ refer to the j-th element of $\mathcal{P}(T_i)$ using an arbitrary (but deterministic) sorting order. Change step (3) as follows: for each pair $(x_i, P_{i,j})$, split S into partitions $S_{i,j,\text{left}}$ (containing only samples with $x_i \in P_{i,j}$) and $S_{i,j,\text{right}}$ ($x_i \notin P_{i,j}$). The rest of the algorithm remains the same.

However, we expect that this method does not offer significant advantages over boolean configuration variables. While it allows similar configuration variable values to be grouped (e.g. resulting in a common sub-tree for Default and Float16 optimizations), comparable tree structures can be achieved with boolean-only variables and binary splits. Additionally, for each categorial variable, $O(2^{|T_i|})$ different splits must be performed in order to find the one with the lowest loss.

Instead, we examine a more radically different variant: *Non-Binary CART* [9]. When encountering a categorial configuration variable with n different values, these use a single tree node with n children – one for each value. We expect the resulting trees to be more compact. They should also be closely related to the structure of the feature model, whose groups of alternative features also have one child for each alternative. See Fig. 3 for an example.

We perform non-binary CART generation as follows.

(1) For each variable x_i, let $T_i = \{t_{i,1}, t_{i,2}, \dots\}$ be the set of its unique values in an arbitrary (but deterministic) order.
(2) If a stop criterion is satisfied: return a leaf node using the mean of observed data $\mu(S)$ as model value.
(3) For each variable x_i, split S into partitions $S_{i,1}, \dots, S_{i,|T_i|}$ with $S_{i,k} = \{y_j \in S | (\vec{x}_j)_i = t_{i,k}\}$.
(4) Select the variable x_i with the lowest associated loss and transform it into a regression tree node.
(5) Repeat recursively with $S := S_{i,k} \forall k$, resulting in a total of $|T_i|$ child nodes.

Stop criteria and loss function remain virtually unchanged:

$$\sum_{k=1}^{|T_i|} \sum_{y \in S_{i,k}} (y - \mu(S_{i,k}))^2$$

When querying a non-binary CART, categorial variables may take values that were not part of the training set and are therefore not part of the tree structure. Whenever encountering a decision node where this is the case during traversal, we return the mean model value of all of its sub-trees. This ensures that the functions expressed by non-binary CART are well-defined.

For example, when queried with $x_1 = 1, x_2 = \text{Int8EdgeTPU}$, the tree shown in Fig. 3 returns the mean of the sub-tree values for Default, Float16, and Int8.

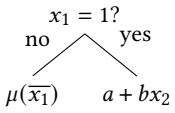

Figure 4: A linear model tree. Each leaf holds a constant model μ or a linear model $a + bx_i + cx_j + \cdots$.

3.2 Linear Model Trees

Linear Model Trees (LMT) are an extension of CART. They also rely on regression trees, but use both static values and linear functions in leaves [17]. This allows them to express piecewise linear functions $\mathbb{R}^n \to \mathbb{R}$ and better capture configuration parameters with linear influence on the model value.

For instance, the model tree fragment in Fig. 4 describes memory requirements during AI inference. In this example, we consider the batch size (scalar) and two AI architectures (modeled with boolean variables), one of which does not support batching. If the AI architecture with batch support is used ($x_1 = 1$), memory usage depends on the batch size: $f(\vec{x}) = a + bx_2$. Otherwise, it is constant ($\mu(\overline{x_1})$).

Many model tree generation algorithms work by first building a scalar CART, and then using a bottom-up *pruning algorithm* combined with linear regression analysis to transform sub-trees into linear functions. Details vary between publications, the M5' algorithm appears to be a prominent example [22]. Here, we use a top-down approach instead. We first generate a boolean CART that ignores scalar variables, and then transform leaf nodes that are influenced by scalar variables into linear functions by means of linear regression [6]. We consider both binary LMT (with boolean and scalar variables) and non-binary LMT (with boolean, categorial, and scalar variables).

3.3 Tree Boosting

In contrast to previous models, *tree boosting* (including *Extreme Gradient Boosting*) is an ensemble learning approach. This means that the model consists of a group (also referred to as *forest* or *ensemble*) $\mathcal{F} = \{f_1, f_2, \ldots f_K\}$ of regression trees instead of just a single one. Each tree may have a different structure. The model output is the sum of the individual tree models: $\mathcal{F}(\vec{x}) = \sum_{i=1}^{K} f_i(\vec{x})$.

Learning accurate forests from training data efficiently and without overfitting is considerably more challenging than for the previously presented single-tree models. We sketch the most important ideas here, and refer to Chen and Guestrin for details [3].

Extreme Gradient Boosting (XGBoost) employs a regularized loss function to penalize complex forests and thus reduce the risk of overfitting. Given a non-regularized loss function l, such as the sum of squared residuals, an ensemble $\mathcal{F} = \{f_1, \ldots, f_K\}$, and observations $y_1, \ldots, y_n$ of system configuration $\vec{x}_1, \ldots, \vec{x}_n$, the XGBoost loss function $\mathcal{L}$ is defined as follows.

$$\mathcal{L}(\mathcal{F}) = \sum_{i=1}^{n} l(\mathcal{F}(\vec{x}_i), y_i) + \sum_{k=1}^{K} \Omega(f_k)$$

The regularization term $\Omega(f)$ calculates the complexity of each tree f based on the number of leaves and leaf values.

Forest generation is an iterative process. In each iteration, a new tree is constructed in such a way that it minimizes the (regularized) loss $\mathcal{L}$, and added to the forest. The number of iterations (and, therefore, trees) is configured by the user. Tree generation, in turn, employs the CART generation algorithms shown previously. However, it uses a custom loss function that greedily selects splits to reduce $\mathcal{L}$. Thus, XGBoost tree generation takes the accuracy of the entire forest and not just the current tree into account.

Additionally, leaf values in the newly added tree are scaled by the *shrinkage* hyper-parameter $\eta \in [0, 1]$. Essentially, this controls the learning rate: A low η value reduces the influence of individual trees on the model outcome, thus allowing additional trees to fine-tune the ensemble.

4 BENCHMARKS

We now present the benchmarks we used to acquire the model input data (i.e., the set of observations S), and noteworthy observations.

4.1 Data Acquisition

We performed a thorough state space exploration by benchmarking all valid combinations of hardware platform, AI platform, optimization settings, and AI architecture. For TensorFlow, we used batch sizes of 1, 2, 4, 8, 16, 32, and 64, if sufficient memory was available. For TensorFlow Lite, we used batch sizes of 1, 4, 8, and 16.

Although it is desirable to minimize the amount of benchmarks required for model generation, we decided against it in this evaluation. Data-efficient model generation is an active research area in itself and the recommended data acquisition method may depend on the type of NFP model [14, 26]. In order to avoid misconceptions due to unsuitable input data sets, we err on the side of caution. Running the complete set of benchmarks for our product line takes about three days and is fully automated.

All benchmarks were performed using Python3 and TensorFlow 2.4 (nVidia Jetson) or TensorFlow 2.5 (other hardware). We captured the following non-functional properties.

- *Latency* (s): Wall-clock time spent in `model.predict` or `interpreter.invoke`, excluding the first inference (warm-up) and excluding the time spent for image preprocessing.
- *Throughput* (FPS): Batch size divided by median latency of the entire benchmark run.
- *Accuracy* (%): F1-Score of pre-trained KERAS models on the ILSVRC 2012 classification validation set. We did not benchmark segmentation accuracy.
- *Memory Usage* (MB): Memory allocated during benchmark execution. Includes memory overhead for image preprocessing and a small amount of benchmark data storage.
- *Model Size* (MB): Size of serialized model in the TensorFlow SavedModel or TensorFlow Lite Buffer format.

Additionally, we consider *Hardware Cost* (€) based on the "HW Platform" selection and end-user development kit prices.

We stored raw, unaveraged samples in our observation set, and ran each latency, throughput, and memory usage benchmark twice. This way, we have at least two samples for each distinct system configuration $\vec{x}$. We note that these may have different NFP values, e.g. due to runtime fluctuations caused by cache and background load effects.

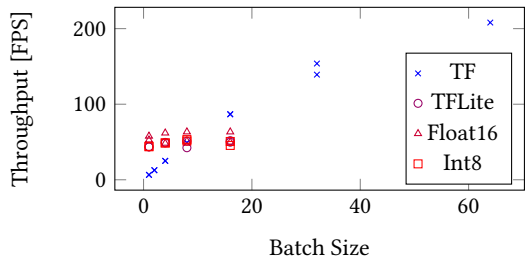

Figure 5: MobileNet v3 throughput on a Jetson Xavier board.

We did not repeat accuracy and model size measurements. Model size is always the same, and a configuration's accuracy only depends on the validation set, which was identical for each KERAS model.

4.2 Benchmark Results

There is already a plethora of benchmarks covering how AI architectures affect accuracy and latency in the literature [11, 21, 24]. To complement this, we examine how configuration parameters such as batch size, platform, or TensorFlow Lite optimization setting affect latency and other non-functional properties of the AI architectures present in our benchmark dataset. Raw data is available at https://ess.cs.uos.de/git/artifacts/cain22-black-box-models.

4.2.1 Batch Size. From a latency and throughput perspective, we find that most TensorFlow models benefit from batching, whereas it has almost no positive effect on TensorFlow Lite models. With TensorFlow, while higher batch sizes do lead to higher latency, they also significantly increase throughput. With TensorFlow Lite, adjusting the batch size linearly increases the latency and has no significant effect on throughput in almost all cases. The best we observe is a 50 % increase in throughput when increasing the batch size from one to 16. For low batch sizes, TensorFlow Lite is faster than TensorFlow; the break-even point at which switching to TensorFlow leads to faster inference varies.

As an example, Fig. 5 shows MobileNet v3 classification throughput with both AI platforms on a Jetson Xavier. In this case, TensorFlow is faster when using at least eight images per batch. We see these effects on CPU-bound and on GPU-/TPU-accelerated platforms, and also with other tasks and AI architectures.

We find that memory usage and batch size have a linear relationship on most platforms utilizing CPU-only inference. With AI accelerators, on the other hand, we find that changing the batch size may have no influence at all on memory usage, or only marginally increase it. This is the case on Jetson Nano and Jetson Xavier (GPU-accelerated TensorFlow), and Coral EdgeTPU (TPU-accelerated TensorFlow Lite for compatible AI architectures). As far as we know, this is due to eager memory allocation strategies used on these accelerator platforms: on startup, they allocate a predefined chunk of memory, and increase the allocated space if it is insufficient.

4.2.2 AI Platform and Quantization. We observe significantly higher memory usage for TensorFlow execution than for TensorFlow Lite. This is also the case when looking at CPU-bound TensorFlow on Raspberry Pi 4. This confirms that TensorFlow Lite implementations are generally optimized towards low memory footprint.

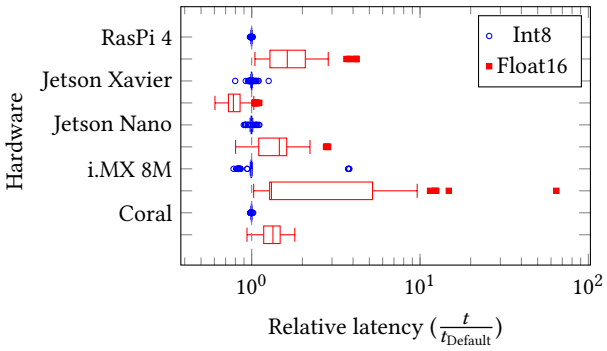

Figure 6: Inference latency of TensorFlow Lite Int8 and Float16 quantization relative to Default optimization settings.

When comparing TensorFlow Lite quantization settings, we see that 8-bit integer quantization more than halves the memory usage compared to 16-bit float quantization. The same applies to model size: 8-bit integer quantized models take up about half as much space as 16-bit float models; these are in turn about half as large as (unquantized, 32-bit) TensorFlow models. As an 8-bit integer takes up exactly half as much space as a 16-bit float, and integer-only executable code is typically smaller, this is an expected result.

Fig. 6 shows the relative latency of 8-bit integer quantization and 16-bit float quantization compared to default TensorFlow Lite optimization settings for pairs of identical AI architecture and batch size configurations. We see that TensorFlow Lite inference latency is lowest when using 8-bit integer quantization on almost all Coral, i.MX 8M and Raspberry Pi 4 benchmarks. On Jetson Xavier, by contrast, 16-bit float quantization is often the fastest option for TensorFlow Lite. We observe negligible latency differences between quantization with and without a representative dataset, and between default and 8-bit integer configuration.

On the accuracy side, we find that the F1 score for TensorFlow Lite with Float16 optimization is up to ten percentage points higher than the F1 score of the Default/Int8 variants for some KERAS models. This can be explained by the significant changes required for integer quantization, and their much lower dynamic range compared to floating-point numbers. Whether using integer instead of floating-point quantization is worth the trade-off depends on optimization goals, use-case constraints, and the AI architecture.

4.2.3 Hardware Choice. Inference latency on a specific hardware platforms depends on the platform's overall performance, CPU/GPU features and whether they are used by the TensorFlow (Lite) implementation, available accelerators, and compatibility between accelerators and AI architecture components (e.g. specific layer types or convolution kernels). Fig. 7 shows how switching from a Raspberry Pi 4 to another hardware platform affects inference latency for pairs of identical AI architecture, batch size, and quantization settings. For the Coral EdgeTPU platform, it shows both CPU and TPU-accelerated latency; the latter only considers integer-quantized models.

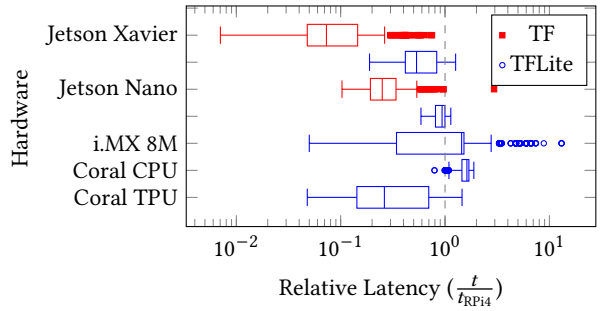

Figure 7: Inference latency by hardware architecture relative to Raspberry Pi 4.

Unsurprisingly, we see that GPU-accelerated TensorFlow on Jetson Xavier and Jetson Nano is almost always faster than CPU-bound TensorFlow on Raspberry Pi 4, and that the Jetson boards also provide advantages for TensorFlow Lite inference. The Coral EdgeTPU platform, on the other hand, is slightly slower when it comes to CPU-only inference, and often considerably faster if the EdgeTPU accelerator can be used.

We also see that performance on i.MX 8M varies wildly: it can be both much better and much worse than Raspberry Pi 4. Our i.MX 8M development kit uses NNAPI for GPU-accelerated inference with TensorFlow Lite models, providing a significant latency reduction when it is working as intended. However, the NNAPI version appears to be incompatible with several modern TensorFlow operators and is therefore unable to properly offload their execution, resulting in sub-par performance for some models.

These observations are also interesting from a cost optimization point of view. Although Raspberry Pi 4 is by far the most affordable option in our collection, it is not the least powerful one. The most powerful option for TensorFlow inference, however, is also the most expensive (Jetson Xavier).

4.2.4 Summary. Overall, we have seen that there are numerous interactions between configurable features and non-functional properties of AI software systems. Both boolean/categorial features such as hardware selection and quantization, and the scalar batch size configuration can have significant impact on the non-functional properties of an AI product.

Therefore, we conclude that it is worthwhile to perform thorough benchmarks of AI software system components before deciding on a specific configuration for optimization and product development. NFP models help abstract and generalize the benchmark results to a certain degree, and can even structure the performance implications of product line features in a human-readable manner. In the next section, we show how well the different modeling approaches abstract from and generalize on the benchmark results.

5 MODEL ACCURACY

One might argue that a simple look-up table that maps a feature vector $\vec{x}$ to the NFP values observed in benchmarks is sufficient for NFP-aware design and optimization of AI products, but such an approach has two notable flaws.

First, as we have seen, batch size influences performance trade-offs, but running benchmarks for any batch size between 1 and 64 (or even 128) is impractical. Instead, we need a model that is capable of predicting system performance for batch sizes that were not part of the training data.

Second, tree-based models are generally considered to be human-readable and can be efficiently queried by product line optimization software [12]. Regression tree generation algorithms naturally sort features from most important (close to the root) to least important (close to a leaf or not present at all), which benefits both human and algorithmic users of NFP models.

Therefore, we now examine how well the modeling methods introduced in section 3 perform when faced with our observations. We consider the following questions. **Q1.** How compact are the resulting models, and how accurately do they describe the observations? **Q2.** How well do they generalize when faced with previously unseen system configurations? **Q3.** How do non-binary tree structures affect model size, accuracy, and interpretability?

To answer these, we create scalar regression tree (CART), boolean regression tree (DECART), linear model tree (LMT), and Extreme Gradient Boosting (XGB) models using the algorithms introduced in section 3. For DECART and LMT, we consider both binary tree structures (using boolean and scalar feature vectors) and non-binary tree structures (scalar and categorial); we indicate the non-binary version with an *nb* suffix. For CART and XGB, we use external implementations which only support binary tree structures.

The product line variant for binary trees has 73 boolean feature variables. The non-binary one has five categorial variables, one of which (the AI Platform) behaves like a boolean as it only has two values. Both have a single scalar variable: the batch size. In total, we use 11,601 benchmark results for model training and evaluation.

Raw data, source code, and evaluation scripts are available at https://ess.cs.uos.de/git/artifacts/cain22-black-box-models.

5.1 Compactness and Abstraction

To assess **Q1**, we examine the complexity (tree depth and number of nodes) and symmetric mean absolute percentage error (SMAPE) of each model. Given predictions $P = \{p_1, \ldots, p_n\}$ and ground truth $Y = \{y_1, \ldots, y_n\}$, it is defined as follows.

$$\text{SMAPE}(P, Y) = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|p_i - y_i|}{\frac{|p_i|+|y_i|}{2}}$$

As pointed out in the introduction, we are interested in compact reproduction of benchmark data, and intend to use our NFP models within the range of NFP values present during benchmark generation. Therefore, we assess model performance on training data in this section. While we cannot detect over-fitting this way, seeing how the accuracy changes between different modeling approaches and model sizes still gives worthwhile insights into model performance. In the next section, we address generalization capabilities (Q2) with cross validation.

We use two additional models to put the results into perspective. The *static* model serves as upper bound for prediction error: it does not care about feature vectors and simply uses $\mu(S)$ to predict NFP values. The *LUT* (look-up table) model partitions observations by feature vector into sets $S_1, S_2, \ldots$ so that $\forall k : \forall y_i, y_j \in S_k : \vec{x}_i = \vec{x}_j$.

Table 1: Model size (# Nodes) and Symmetric Mean Absolute Percentage Error (SMAPE) of binary and non-binary (*nb*) regression tree models on training data. Static and look-up-table (LUT) models serve as upper and lower accuracy bound, respectively.

	SMAPE [%]		# Nodes and SMAPE [%]											
Application	Static	LUT	DECART		CART		LMT		XGB		$\text{DECART}_{\text{nb}}$		LMT_{nb}	
Latency	103.5	5.0	2,775	77.3	10,381	12.1	2,775	12.9	158,870	12.1	6,881	12.1	1,847	12.9
Throughput	87.9	3.6	2,775	8.4	10,381	3.6	2,775	6.7	50,968	4.9	6,857	3.6	1,843	6.7
Accuracy	14.6	0.0	223	0.0	263	0.0	223	0.0	356	10.4	189	0.0	189	0.0
Memory Usage	79.9	1.1	2,775	37.3	10,381	1.1	2,775	4.1	89,624	1.6	6,464	1.1	1,708	4.1
Model Size	81.9	0.0	963	0.0	1,889	0.0	963	0.0	14,442	0.1	1,389	0.0	1,053	0.0
Hardware Cost	56.1	0.0	11	0.0	11	0.0	11	0.0	220	0.1	7	0.0	7	0.0

When the model value for a feature vector $\vec{x}$ is requested, the LUT model selects the set S_k with matching feature vectors, and returns $\mu(S_k)$. This makes it incapable of generalizing, but an excellent lower bound for the model error.

We expect that accuracy, model size, and hardware cost exhibit the lowest model error. Accuracy only depends on AI architecture and quantization (and is only modeled for KERAS architectures); model size only depends on AI architecture and quantization as well; and hardware cost only depends on the selected AI hardware. Latency, throughput, and memory usage, on the other hand, are affected by all configuration variables.

Table 1 lists model size (number of regression tree nodes) and model error on training data for all combinations of NFP and model type. We immediately see that CART and $\text{DECART}_{\text{nb}}$ perform best in all settings, and reach the lower bound for model error in five of six cases. On average, linear model trees and XGBoost are only slightly less accurate, though XGB has a high error in the simple accuracy model. Binary DECART error is up to 77 %.

Model size, on the other hand, is lowest for LMT_{nb}, followed by DECART and LMT. CART are up to five times larger. The XGB forest uses up to 15 times more nodes than CART and is still less accurate. Tree depth (not shown) also differs significantly: all non-binary trees have a maximum depth of five, whereas binary tree depth ranges from 30 to 60.

CART and $\text{DECART}_{\text{nb}}$ performance is unsurprising: they can add a leaf node for each individual batch size, or groups of batch sizes with similar NFP values, and therefore mirror the observations almost perfectly. This comes at the cost of a large model.

Our implementation of linear model trees, on the other hand, uses linear functions to express the influence of scalar variables. This makes the model much smaller. However, as the influence of batching is not perfectly linear, they have a slightly higher error when it comes to latency, throughput, and memory usage. For NFPs that are independent of batch size, both perform equally well.

We would have expected XGBoost to perform at least as well as CART. However, tree boosting algorithms require carefully selected hyper-parameters (e.g. learning rate and number of trees) to achieve optimal results. Here, we set the number of trees to 20, and limited the maximum depth to 40. While we did perform a hyper-parameter exploration, we may not have found the optimal combination. This, in turn, is also a noteworthy data point: XGBoost is very powerful, but also very complex, and tends to generate large forests. It may be better suited for large-scale, high-dimensional modeling problems.

As DECART does not support scalar parameters, its high model error for batch size-dependent NFPs is to be expected. This is a good indicator of just how important scalar parameter support in AI NFP models is. The identical model size of DECART and LMT underlines that our version of LMT are really just DECART plus linear regression.

5.2 Generalization

We now examine how the modeling approaches perform when faced with system configurations that were not present during training. To address this (and, thus, **Q2**), we perform parameter-aware 10-fold cross validation and calculate symmetric mean absolute percentage error (SMAPE) for each model.

Parameter-aware cross validation partitions observations into training and validation sets not based on their position in the set of observations S, but on their feature vector $\vec{x}$. This ensures that $\vec{x}_t \neq \vec{x}_v$ for any pair of training set entry (with configuration $\vec{x}_t$) and validation set entry ($\vec{x}_v$). So, the validation set only contains system configurations that were not part of the training set.

In this case, we evaluate the static model (upper error bound) using cross validation as well. We do not use cross validation for the LUT model (lower error bound), as it is unable to generalize. We perform training and validation with ten different pairs of training and validation set, and report the mean model error.

Table 2 shows the cross validation results. Here, we see that linear model trees are well-capable of generalization thanks to the linear functions used in leaves. They achieve the lowest model error in all cases. LMT_{nb} provide competitive results, with 1 to 3 % higher error than binary LMT.

The advantage of linear model trees is most notable when predicting latency and memory usage, which are affected by the configured batch size in the majority of benchmark observations. All other modeling methods fare significantly worse in these cases. The less complex throughput and accuracy models, on the other hand, can be expressed by CART, DECART, and XGB with only slightly higher model error.

Overall, we see that linear model trees (and, to some extent, CART and XGB) are capable of generalization and can predict configurations that were not present during training with acceptable accuracy. However, we do not recommend varying too many variables at once: attempting to predict performance of an unknown AI architecture on an unknown hardware platform will generally not lead to usable data.

Table 2: Symmetric Mean Absolute Percentage Error (SMAPE) of binary and non-binary (*nb*) regression tree models with 10-fold cross validation. Static and look-up-table (LUT) models serve as upper and lower accuracy bound, respectively; LUT error is calculated without cross validation.

	SMAPE [%]							
Application	Static	LUT	DECART	CART	LMT	XGB	$\text{DECART}_{\text{nb}}$	LMT_{nb}
Latency	103.5	5.0	94.7	46.9	25.9	45.0	43.3	26.5
Throughput	87.9	3.6	10.7	12.7	10.5	15.6	19.5	11.2
Accuracy	14.8	0.0	12.3	12.4	12.3	13.9	15.7	15.7
Memory Usage	79.9	1.1	49.7	32.5	13.7	28.0	33.7	14.5
Model Size	82.0	0.0	0.1	0.9	0.1	1.1	0.8	0.6
Hardware Cost	56.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0

5.3 Tree Structure

When comparing LMT and LMT_{nb} as well as CART and $\text{DECART}_{\text{nb}}$, almost all pairs of binary and non-binary model variant have the same number of leaf nodes. However, the internal tree structure differs drastically: non-binary trees are shallow, with a maximum depth of five in our case, whereas binary tree depth ranges from 30 to 60. This is directly related our AI product line's variability model: it has no more than five categorial configuration options, but 73 distinct boolean ones. It also results in the model size differences reported earlier: with non-binary splits, the same number of leaves can be reached from a smaller amount of inner tree nodes.

For interpretability, non-binary trees are good news. Even with large models, as each non-leaf node corresponds to a common *group* of features (e.g. AI architectures) rather than a single feature (e.g. a single AI architecture), humans can immediately ascertain the relative importance and effects of configuration decisions.

For example, the root node of the LMT_{nb} model for throughput is the neural network architecture, followed by hardware platform and AI platform. Therefore, neural network selection has the highest impact on throughput, and hardware and platform changes may be less helpful than choosing a different neural network when certain throughput goals must be met. Also, leaf nodes belonging to the TensorFlow platform are frequently modeled as a function of batch size, whereas TensorFlow Lite throughput is often modeled as a constant value, reflecting our observations in section 4.2.1.

Regarding model error on training data, we note that the number of leaf nodes dictates how many different system configuration groups a model can distinguish. When using comparable model generation algorithms and obtaining an identical number of leaf nodes, as is the case for LMT and LMT_{nb} as well as CART and $\text{DECART}_{\text{nb}}$, we expect identical model error. Indeed, Table 1 shows that this is the case.

Generalization error is not as straightforward. When queried for a categorial feature value that was not part of the training set, binary trees end up in a leaf in which all relevant features from the same feature group (e.g. AI architectures for bounding-box segmentation) are disabled, and the least important (irrelevant) features are "don't care". Non-binary trees, on the other hand, compute the mean model value for all known features from this feature group.

Table 2 shows that LMT_{nb} generalization error is slightly higher than LMT error, whereas $\text{DECART}_{\text{nb}}$ and DECART error do not show a clear trend. We conclude that binary tree structures appear to be better suited for generalization. Nevertheless, non-binary tree structures – when applied to a suitable model, such as LMT – only have a small negative impact on generalization error.

So, to answer **Q3**: non-binary tree structures (specifically, LMT_{nb}) are useful for interpretation of observed AI software system behaviour. Model error without cross validation is identical between binary and non-binary trees; generalization error of non-binary LMT is only slightly higher than binary LMT error.

6 RELATED WORK

Operating systems, video codecs, and other applications offer a plethora of tunables. NFP models help find appropriate configurations for these. Therefore, both efficient state space exploration and accurate NFP modeling methods have a long history in the software product line engineering community [16]. We now give a brief overview of related NFP modeling methods for software product lines and AI applications.

When it comes to software product lines, both efficient data acquisition and generation of accurate models are active research areas. For instance, the authors of *Data-Efficient CART* (DECART) show that, for most software systems, a complete state space exploration is not required when considering only boolean features. Given n variables, they build regression trees with a model error of less than 10 % with less than $10n$ observations for several product lines [7]. However, they do not support scalar features, and do not consider the interaction between software and hardware features.

L2S addresses hardware-software interactions by detecting correlations between configurable features and hardware (or workload) changes, and generating benchmark configurations accordingly. This way, Jamshidi et al. are able to achieve an error of 7 to 20 % when modeling the latency of both conventional and AI software systems with a minimal number of benchmark measurements and with hardware changes [8]. Their approach uses CART for NFP modeling, but has not been tested with scalar feature variables. We expect that it also works well with linear model trees.

FLASH uses CART with boolean and scalar variables internally, and combines data acquisition with multi-objective optimization [14]. Optimization goals must be known beforehand, making it unsuitable for our application.

While CART appear to be the most common NFP modeling method in the software product line community [16], *Linear Model Trees* are known to be a suitable modeling method for boolean and

scalar inputs as well [13]. A recent software fault prediction study reports 5 to 50 % model error [18].

Non-tree approaches have shown to be useful for explicit handling of model uncertainty. *Fourier Learning* is capable of providing guaranteed accuracy bounds, at the cost of requiring a large amount of samples [26]. *P4* uses probabilistic programming to obtain interpretable model functions [4]. Both focus on boolean-only configuration spaces.

Considering the step towards white-box NFP models, Yu et al. examine the effect of GPU hardware on the training time of neural networks. However, their *Habitat* model does not map neural network architecture or configuration to training performance. Instead, model input is the training latency on a benchmark GPU and a performance model of benchmark as well as target GPU, and model output is the expected latency on the target GPU [24].

In a similar vein, *MNasNet* performs neural architecture search to optimize accuracy and latency of mobile neural networks. Instead of NFP models, it relies on actual smartphones for latency measurements [21]. While this guarantees highest accuracy, it also means that each measurement takes a significant amount of time.

Li et al. provide a white-box model for GPU latency of CNN models with support for embedded Jetson TX boards, and find that convolution parameters may have non-linear effects on inference latency [10]. Their model does not consider hardware variations.

Oboe, on the other hand, uses a combination of bilinear and polynomial models on meta-features extracted from ML models and datasets to predict dataset-dependent accuracy and latency [23].

When it comes to variable hardware, *NN-Meter* is able to predict TensorFlow Lite DNN latency by decomposing networks into hardware-specific execution units and predicting their latency by means of a non-linear model [25]. The authors report a root mean square error of less than 6 % for an Adreno 640 GPU.

In the area of TinyML (i.e., small, 8-bit integer-quantized neural networks suitable for execution on microcontrollers), Banbury et al. show that the operation count is a suitable predictor for model latency [1]. Their *MicroNets* approach utilizes simple linear functions to predict model latency based on operation count. They also find that image processing networks have a different operation count to latency mapping than audio processing networks, so NFP models must provide one function for each AI application domain.

Liberis et al. confirm this: While the number of floating point operations alone is an unsuitable performance indicator for GPU inference, predicting microcontroller inference latency from the number of multiply-accumulate operations works well [11].

Finally, non-binary trees have been used for process monitoring [5]. We are not aware of prior applications in NFP models.

7 CONCLUSION AND FUTURE WORK

We have presented an AI software system product line for image classification and segmentation tasks consisting of variable AI architecture, AI platform, quantization settings, batch size, and hardware. In a thorough configuration space evaluation, we have found complex interactions between product line configuration (e.g. hardware and AI platform selection and configuration) and non-functional properties such as memory requirements and throughput.

Combined with significant cost differences between hardware platforms, this underlines the usefulness of NFP models for AI software systems. The AI architecture is but a small component in the overall performance and price tag of an AI product. Hardware and AI platform selection and configuration also play a significant role, and should be considered during product design and AI architecture selection. When done properly, this helps in finding the most efficient combinations of AI architecture, platform, and hardware for AI products, and helps build them in a cost-efficient manner.

We have found that linear model trees are both the most compact and (for generalization error) most accurate of all evaluated tree-based modeling methods. While regression trees (CART) have a lower abstraction error on training data, they are also significantly larger. Non-binary linear model trees offer far better interpretability than binary tree structures, at only slightly increased model error. This makes them well suited for visualization and manual analysis of NFP models.

In summary, we conclude that tree-based black-box models for non-functional properties of AI software systems, even if seemingly mundane, provide important insights and reasonable accuracy. Linear model trees provide the best combination of model accuracy and complexity (i.e., model size). Users that are most interested in high accuracy may stick to conventional (binary) linear model trees, whereas users interested in analysis and visualization should consider non-binary tree structures.

We are currently working on an NFP-aware configuration software for AI products. In the future, we plan to use these NFP models to assist developers in making informed choices about cost-efficient hardware selection and configuration, while respecting optimization goals and performance constraints. After manually configuring relevant system components (e.g. the AI task, latency requirements, and optimization goals), the goal is for the configuration software to automatically suggest optimal configuration candidates (e.g. groups of hardware platform, AI architecture, and configuration).

Finally, it is also interesting to transform the black-box models into white-box models that are capable of predicting performance of arbitrary AI architectures (based on their layout) on arbitrary hardware platforms (based on performance models). This way, non-functional properties of both new AI architectures and new hardware platforms could be queried without mandatory NFP model re-training. As related work has shown that this is possible at least in certain domains, e.g. for GPUs (Habitat [24]) or MCUs (MicroNets [1]), we consider this to be an interesting avenue of further research.

ACKNOWLEDGMENTS

This paper was partly supported by funds of the Federal Ministry of Food and Agriculture (BMEL) based on a decision of the Parliament of the Federal Republic of Germany via the Federal Office for Agriculture and Food (BLE) under the innovation support programme.

REFERENCES

[1] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 517–532.

[2] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 1984. *Classification and regression trees*. Routledge. https://doi.org/10.1201/9781315139470

[3] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[4] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 684–696. https://doi.org/10.1145/3324884.3416620

[5] B. Ferreira, R. G. Silva, and V. Pereira. 2017. Feature selection using non-binary decision trees applied to condition monitoring. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–7. https://doi.org/10.1109/ETFA.2017.8247642

[6] Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. 2018. Parameter-Aware Energy Models for Embedded-System Peripherals. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 1–4. https://doi.org/10.1109/SIES.2018.8442096

[7] 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 Softw. Engg.* 23, 3 (June 2018), 1826–1867. https://doi.org/10.1007/s10664-017-9573-6

[8] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3236024.3236074

[9] Stanislav Keprta. 1996. Non-binary classification trees. *Statistics and Computing* 6, 3 (1996), 231–243.

[10] Jinyang Li, Runyu Ma, Vikram Sharma Mailthody, Colin Samplawski, Benjamin Marlin, Songqing Chen, Shuochao Yao, and Tarek Abdelzaher. 2021. Towards an Accurate Latency Model for Convolutional Neural Network Layers on GPUs. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 904–909. https://doi.org/10.1109/MILCOM52596.2021.9652907

[11] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. 2021. μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3437984.3458836

[12] Wei-Yin Loh. 2014. Fifty years of classification and regression trees. *International Statistical Review* 82, 3 (2014), 329–348.

[13] D. Malerba, F. Esposito, M. Ceci, and A. Appice. 2004. Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 5 (2004), 612–625. https://doi.org/10.1109/TPAMI.2004.1273937

[14] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[15] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. 2012. Modelling and Multi-Objective Optimization of Quality Attributes in Variability-Rich Software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages* (Innsbruck, Austria) *(NFPinDSML '12)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/2420942.2420944

[16] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044. https://doi.org/10.1016/j.jss.2021.111044

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

[18] Santosh Singh Rathore and Sandeep Kumar. 2016. A decision tree regression based approach for the number of software faults prediction. *ACM SIGSOFT Software Engineering Notes* 41, 1 (2016), 1–6.

[19] 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, 3-4 (2012), 487–517. https://doi.org/10.1007/s11219-011-9152-9

[20] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2010. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *2010 Asia Pacific Software Engineering Conference*. 147–155. https://doi.org/10.1109/APSEC.2010.26

[21] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[22] Yong Wang and Ian H Witten. 1996. Induction of model trees for predicting continuous classes. (1996).

[23] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. 2019. OBOE: Collaborative Filtering for AutoML Model Selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1173–1183. https://doi.org/10.1145/3292500.3330909

[24] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 503–521. https://www.usenix.org/conference/atc21/presentation/yu

[25] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3458864.3467882

[26] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 365–373. https://doi.org/10.1109/ASE.2015.15