Towards Hardware-Aware Tractable Learning of Probabilistic Models

Laura I. Galindez Olascoaga [1] **Wannes Meert** [2] **Marian Verhelst** [1] **Guy Van den Broeck** [3]

Abstract

Smart portable applications increasingly rely on edge computing due to privacy and latency concerns. But guaranteeing always-on functionality comes with two major challenges: heavily resource-constrained hardware; and dynamic application conditions. Probabilistic models are an ideal solution to the unstructured nature of these applications: they are robust to missing data, allow for joint predictions and have small data needs. In addition, ongoing efforts in field of tractable learning have resulted in probabilistic models with strict inference efficiency guarantees. However, the current notions of tractability are often limited to model complexity, disregarding the hardware's specifications and constraints. We propose a novel resource-aware cost metric that takes into consideration the hardware's properties in determining whether the inference task can be efficiently deployed. We use this metric to evaluate the performance versus resource trade-off relevant to the application of interest, and we propose a strategy that selects the device-settings that can optimally meet users' requirements. We showcase our framework on a mobile activity recognition scenario, and on a variety of benchmark datasets representative of the field of tractable learning and of the applications of interest.

1. Introduction

Tractable learning aims to balance the trade-off between how well the resulting models fit the available data and how efficiently queries are answered. Most implementations focus on maximizing model performance and only factor in query efficiency by subjecting the learning stage to a fixed tractability constraint (e.g. max treewidth). While recent notions of tractability consider the cost of probabilistic inference as the number of arithmetic operations involved in a query (Lowd and Domingos, 2012; Lowd and Rooshenas, 2013), they still disregard hardware implementation nuances of the target application. This is of special concern for edge computing on embedded applications, where the target algorithm must run in resource constrained hardware, such as a small ARM or RISC-V embedded processor, or a microcontroller. For such architectures running a lightweight operating system, the overall compute cost is mostly determined by the cost of fundamental arithmetic operations, the interaction with sensor interfaces and the exchanges with the device's memory (Kim et al., 2017).

Efforts towards hardware-efficient realizations of probabilistic models are currently scarce (Tschiatschek and Pernkopf, 2015; Piatkowski et al., 2016; Sommer et al., 2018); in stark contrast with the tremendous progress achieved by neural network implementations (Tan et al., 2018; Howard et al., 2017; Moons et al., 2018). We address these limitations of the field of tractable learning by proposing a novel resource-aware cost metric that takes into consideration the target embedded device's properties (e.g. energy consumption); and system-level configuration (e.g. sensors used). We map these hardware characteristics to the cost vs. performance trade-off space, and propose a set of techniques to find the optimal system-level configuration. Specifically, we address the following points: (a) Section 3 discusses the relevant hardware-aware tractability metrics, and Section 4 defines the problem statement; (b) Section 5 discusses how to exploit the model's properties to exchange task-performance for hardware efficiency, and introduces techniques to find the optimal set of system configurations in the cost vs. performance trade-off space; and (c) Section 6 shows practical examples of these optimal solutions. This work constitutes one of the first efforts to introduce the field of probabilistic reasoning to the emerging domain of edge computing. This is motivated by probabilistic models' traits, several of which are ideal for portable applications that require reasoning on the edge: robustness to missing information, small data needs, joint predictions, and expert knowledge integration. Moreover, unlike fixed neural architecture training, tractable learning allows to explicitly vary the level of complexity of the inference task, which allows to model resource tunability.

[1]Electrical Engineering Department, KU Leuven. [2]Computer Science Department, KU Leuven. [3]Computer Science Department, University of California, Los Angeles. Correspondence to: Laura I. Galindez Olascoaga <laura.galindez@esat.kuleuven.be>.

3rd Tractable Probabilistic Modeling Workshop, 36th International Conference on Machine Learning, Long Beach, California, 2019. Copyright 2019 by the author(s).

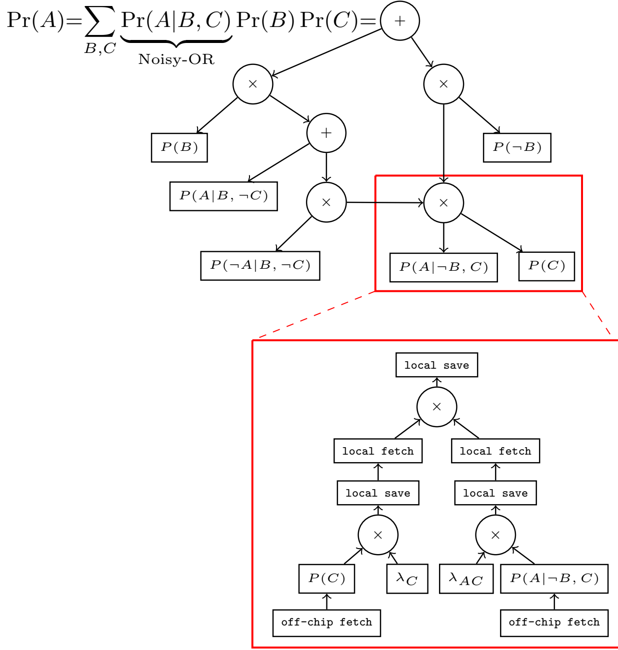


Figure 1. Arithmetic circuit and its mapping to hardware.

2. Background and motivation

We use standard notation: random variables are denoted by upper case letters X and their instantiations by lower case letters x. Sets of variables are denoted in bold upper case $\mathbf{X}$ and their joint instantiations in bold lower case $\mathbf{x}$. Sets of variable sets are denoted with $\mathcal{X}$.

The model representation of choice in this paper is the Arithmetic Circuit (AC), a state-of-the-art, compact representation for a variety of machine learning models such as probabilistic graphical models (Darwiche, 2009) and probabilistic programming (Fierens et al., 2015). Recent developments show how the structure of ACs can also be learned from data (Liang and Van den Broeck, 2019; Liang et al., 2017). Furthermore, ACs can be complemented with deep learning architectures (Xu et al., 2018; Manhaeve et al., 2018) to achieve the best of both worlds. An alternative representation of ACs are Sum-Product Networks (SPNs), which can also encode probability distributions as a computational graph (Poon and Domingos, 2011; Gens and Pedro, 2013). SPNs can be efficiently converted to ACs and vice versa (Rooshenas and Lowd, 2014).

2.1. Probabilistic inference with Arithmetic Circuits

An AC is a directed acyclic graph where inner nodes represent addition or multiplication and leaf nodes are real-valued variables. ACs constitute a standard representation for computing polynomials, but they have proven to be efficient for reasoning over knowledge bases and probabilistic models when a number of additional properties are enforced on them (Darwiche, 2009). Once the circuit is known, the complexity of executing the encoded formula is also known, since operations such as Weighted Model Counting are polynomial in the size of the circuit (Chavira and Darwiche, 2008). This makes ACs a well-suited representation for tractable learning. When used to represent a joint probability distribution over a set of random variables $\mathbf{X}$, the leaf nodes of an AC are either binary indicator variables $\lambda_{X=x}$, where $X \in \mathbf{X}$, or parameters θ. Figure 1 shows an example of an AC that encodes the joint probability distribution of a noisy-OR model (Heckerman, 1993).

This representation allows to perform inference to answer a number of probabilistic queries. For example, given an instantiation $\mathbf{f}$ of $\mathbf{F} \subseteq \mathbf{X}$, the marginal probability $\Pr(\mathbf{f})$ can be computed by setting the indicator variables to 1 if they correspond to instantiations consistent with the observed values, $\lambda_{X=x} \leftarrow 1_{x\sim\mathbf{f}}$, and subsequently performing an upward pass on the AC. In a binary classification task, one can define a class variable C, a feature set $\mathbf{F}$ and a classification threshold T. For a given instance $\mathbf{f}$, the task consists on selecting the class C_T for which the condition $Pr(C|\mathbf{f}) \geq T$ is met. The conditional probability can be calculated by performing two upward passes on the AC[1] that encodes $\Pr(C, \mathbf{F})$, after setting the indicator variables λ in accordance to instance $\mathbf{f}$. ACs' straightforward mapping to embedded hardware and the fact that they readily encode the algorithm necessary for inference, motivates our choice for this probabilistic model representation. Moreover, the process of learning them already entails a trade-off between their predictive performance and their computational efficiency. The following section motivates our proposed hardware-aware tractability metric.

2.2. Motivating example

Consider the mobile classification scenario in Figure 2, where the feature set $\mathbf{F}=\{F_{A1},F_{A2},F_{B1},F_{B2},F_{D1},F_{D2}\}$ is extracted from sensors A, B and D, and where the AC is assumed to be the most compact model that maximizes classification accuracy.

Suppose the feature subset $\mathbf{F_1}=\{F_{A1}, F_{B2}, F_{D1}\}$ provides the maximum accuracy while incurring on the lowest feature cost, a common solution for the problem of feature selection. But when considering also the costs of the sensors, feature set $\mathbf{F_2}=\{F_{B1}, F_{B2}, F_{D1}\}$ turns out to be the better choice, as sensor A is unused and can be turned off. This example shows that trade-off opportunities can be missed when failing to describe realistic hardware-aware system-level costs.

[1] Conditional probability can also be performed by an upward and a downward pass (Darwiche, 2009).

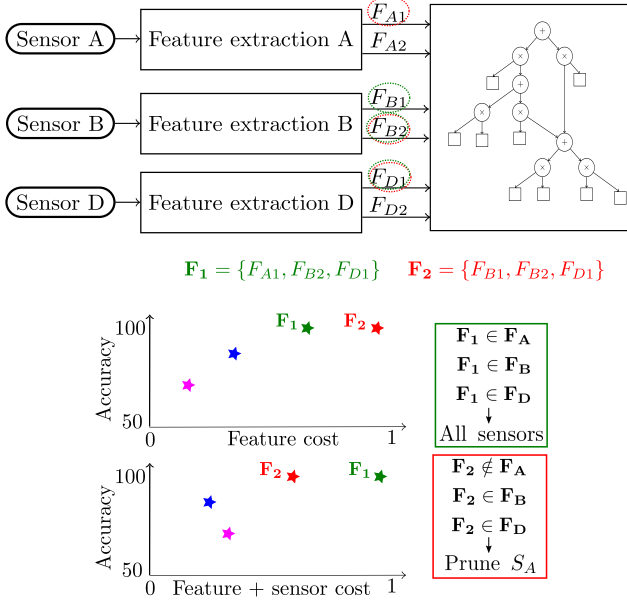


Figure 2. Sensory embedded classification example.

3. Hardware-aware cost

In this section we formalize the notion of hardware-aware cost, the basis of our optimization framework. Let $\alpha = \langle +, \times, \theta \rangle$ be an AC that encodes a joint probability distribution over variables $\mathbf{F}$, extracted from the set of sensor interfaces $\mathbf{S}$. The *hardware-aware cost* ($\mathrm{C_{HA}}$) is defined as:

$$\mathrm{C_{HA}}(\alpha, \mathbf{F}, \mathbf{S}) = \mathrm{C_{AC}}(\alpha) + \sum_{S \in \mathbf{S}} \mathrm{C_{SI}}(S, \mathbf{F}), \qquad (1)$$

where $\mathrm{C_{AC}}$ are the computation costs, pertaining to inference on α, and $\mathrm{C_{SI}}$ are the sensor interfacing and feature extraction costs.

Computation costs. At a high level, a typical embedded hardware architecture entails two components: an off-chip memory, which commonly houses the algorithm's parameter set, and a processing unit where operations are performed and intermediate values are cached in a temporal memory. Performing an upward pass on an AC involves the following actions (see Figure 1): 1) fetching parameters from the off-chip memory, 2) performing arithmetic operations, consisting of additions and multiplications, 3) caching intermediate values in a local memory (e.g. register file or low level cache) and 4) fetching intermediate values from local memory, as needed.[2] Each action has a significantly different hardware resource cost. For example, post synthesis energy models of a simple embedded CPU show that multiplications can require 4 times as much energy as additions, and off-chip memory exchanges 5 times as much energy as multiplications (Horowitz, 2014). When it comes to the design of embedded hardware, energy efficiency is indeed one of the main challenges to address (Kim et al., 2017). Hence, we continue to focus on this resource as a proof of concept without loss of generality; examples of other relevant hardware resource metrics are throughput and latency. It is evident that the total hardware cost of performing a pass on an AC must factor in all the aforementioned exchanges. Let nb be the number of bits used to represents parameters θ and perform arithmetic operations $+$ and $\times$. The *computation cost* ($\mathrm{C_{AC}}$) of AC α is defined as:

$$\begin{aligned}\mathrm{C_{AC}}(\alpha, nb) = \mathrm{C}_+(nb) + \mathrm{C}_\times(nb) \\ + \mathrm{C_{mem}}(nb) + \mathrm{C_{cache}}(nb),\end{aligned} \qquad (2)$$

where the terms in $\mathrm{C_{AC}}$ define the cost incurred by each type of operation. Here, C_+ and $\mathrm{C}_\times$ are the costs of addition and multiplication; $\mathrm{C_{mem}}$ is the cost from fetching parameter leaf nodes from off-chip memory and $\mathrm{C_{cache}}$ is the cost from storing and fetching from local cache (as in Figure 1):

$$\begin{aligned}\mathrm{C}_+(nb) &= \textstyle\sum_a [a =_t +] \cdot \phi_+(nb), \\ \mathrm{C_{mem}}(nb) &= \textstyle\sum_a [a \neq_t + \text{ and } a \neq_t \times] \cdot \phi_{\mathrm{mem}}(nb), \\ \mathrm{C}_\times(nb) &= \textstyle\sum_a [a =_t \times] \cdot \phi_\times(nb), \\ \mathrm{C_{cache}}(nb) &= \textstyle\sum_a [a =_t + \text{ or } a =_t \times] \cdot \phi_{\mathrm{cache}}(nb),\end{aligned}$$

where a denotes a node in α, the equality $=_t$ holds when node a matches the operation type on the right side and $[\beta]$ is equal to 1 when β is true. The function $\phi(.)$ describes the effective cost of the particular operation and can be derived from empirical benchmarks,customized to the target hardware (Horowitz, 2014). When expressing cost in terms of energy consumption, computation costs scale with the precision in number of bits used to represent parameters and perform arithmetic operations (nb), which is typically the same for all nodes in the AC. To conclude, the cost incurred by each node in an AC is determined by its type (whether addition, multiplication, local parameter fetch, or remote memory access) and the resolution of the operation or parameter (in nb).

Sensor interfacing costs The computational block described above is often part of a larger system, which repeatedly performs a task based on external inputs or observations, such as classification. In this scenario, one must factor in the costs incurred by interfacing with the environment or the user. A sensory interface consists of a set of sensors $\mathbf{S}$, which gather, process and digitize environmental information (typically in the analog domain), and a (typically digital) feature extraction unit, which generates the

[2] Depending on the local memory size, one might need to store intermediate values also in off-chip memory but we assume that the local memory size is sufficiently large, but not large enough to store parameters.

feature set $\mathbf{F}$ to be used by the machine learning algorithm. Let $\mathbf{S}$ be the set of available sensors and $\mathbf{F}$ the feature set extracted from them. The sensor interfacing cost ($\mathrm{C_{SI}}$) is:

$$\mathrm{C_{SI}}(S, \mathbf{F}) = \mathrm{C_S}(S) + \sum_{F_S \in \mathbf{F}_S} \mathrm{C_F}(F_S), \quad (3)$$

where $\mathrm{C_S}$ describes the cost incurred by sensor S and $\mathrm{C_F}$ the cost of extracting feature set $\mathbf{F}_S \subseteq \mathbf{F}$. The sensing cost function $\mathrm{C_S}$ can be customized to the target platform and applications through measurements or data sheets. Note that, if no features from a given sensor are used, it can be shut down, and its cost dropped (see Figure 1). In most systems , $\mathrm{C_F}$ can be defined from the type and number of arithmetic and memory operations involved, in a similar fashion to the computation cost estimation $\mathrm{C_{AC}}$, as will be illustrated in the experiments (Section 6.1).

4. Problem statement

We have seen so far that $\mathrm{C_{HA}}$ depends on four system properties:1) the complexity of model α, determined by the number and type of its operations; 2) the size and type of the feature set $\mathbf{F}$; 3) the size and type of the available sensor set $\mathbf{S}$; and 4) the number of bits nb used within α. We refer to an instantiation of these four properties $\sigma = \{\alpha, \mathbf{F}, \mathbf{S}, nb\}$ as a *system configuration*. Clearly, the system configuration also determines the algorithm's performance, defined according to the application of interest. The methods proposed in this work can accommodate any performance metric or miss-classification cost, but we will only consider accuracy, due to its generality. Specifically, we set the classification threshold to $T = 0.5$, and we consider the accuracy of the Bayes-optimal predictions (Acc) over a set of feature instantiations $\{\mathbf{f}_1, ..., \mathbf{f}_l\}$.

Section 2.2 asks to identify the system configuration that incurs the lowest cost for a desired accuracy. Similarly, we might be interested in the configuration that achieves the highest accuracy for a given cost constraint. Thus, the problem we aim to address is how to select the system configurations that map to the Pareto-frontier on the hardware-cost vs. accuracy trade-off space. The inputs to our problem are the class variable C, the available features $\mathbf{F}$ and sensors $\mathbf{S}$ sets, and the set of available precisions $\mathbf{nb}$. The output is the set of Pareto-optimal system configurations $\boldsymbol{\sigma^*} = \{\{\alpha_i^*, \mathbf{F^*}_i, \mathbf{S^*}_i, nb_i^*\}_{i=1:p}\}$.

5. Trade-off space search

We propose to search the cost vs. accuracy trade-off space by scaling four properties (see Section 4):

Model complexity scaling. We learn a set of ACs $\boldsymbol{\alpha}$ of increasing complexity. Each maps to a specific classification accuracy and computation cost $\mathrm{C_{AC}}$ (see Eq. 2). Although discriminative AC learners have shown beyond state-of-the-art classification accuracy (Liang and Van den Broeck, 2019), we have opted for a generative learning strategy, the LearnPSDD algorithm taken from Liang et al. (2017). The motivation for this choice is twofold: this algorithm improves the model incrementally, but each iteration already leads to a *valid* AC, that can be used to populate the set $\boldsymbol{\alpha}$. Moreover, the learner outputs a more robust model, which is demanded by the application range of interest: it must often deal with missing values, either due to malfunction (e.g., a sensor is blocked in an autonomous driving system), or to enforce hardware-cost efficiency (e.g., when energy consumption is excessive, the driving system has the choice to turn off an expensive sensor and the features extracted from it).

Feature and sensor set scaling. We scale the feature set $\mathbf{F}$ by sequentially pruning individual features (see Section 5.1). The effect of feature pruning on classification accuracy has been discussed in numerous works (Friedman et al., 1997; Choi and Broeck, 2018) and the impact on the hardware-aware cost is clear from Eq. 3. Pruning features can also have an impact on the computation costs $\mathrm{C_{AC}}$: recall from Section 2.1 that indicator variables λ or non-observed values are equal to 1 and allow pre-computing the arithmetic operations whose inputs are known (see Algorithm 2). In addition, sensor $S \subseteq \mathbf{S}$ can be pruned when none of the features it originates is used anymore; a strategy that has not been explored by the state-of-the-art, but that is straightforward with our approach, since it considers the full system.

Precision scaling. We consider four different standard IEEE 754 floating point representations , as they can be implemented in almost any embedded hardware platform. Reducing the precision of arithmetic operations and numerical representations entails information loss and results in performance degradation (Hashemi et al., 2017). The effect on computation costs $\mathrm{C_{AC}}$ is clear from Eq. 2.

5.1. Search strategy

Finding the smallest possible AC that computes a given function is Σ_2^p-hard (Buchfuhrer and Umans, 2008), thus computationally even harder than NP. No single optimal solution is known for this problem; it is a central question in the field of knowledge compilation (Darwiche and Marquis, 2002). Optimizing for the lowest-cost/highest-accuracy AC, further increases complexity. We therefore opt for a greedy optimization algorithm. Specifically, we rely on a series of heuristics to search the trade-off space. In each step, we independently scale one of the available configuration properties $\langle \alpha, \mathbf{F}, \mathbf{S}, nb \rangle$, as described in the previous section, and aim to find its locally optimal setting. The search begins by learning the model set $\boldsymbol{\alpha} = \{\alpha_k\}_{k=1:n}$. Starting from each model α_k, we perform a greedy neighborhood search, that

aims to maximize cost savings and minimize accuracy losses by sequentially pruning the sets $\mathbf{F}$ and $\mathbf{S}$, and modifying α_k . At each iteration of Algorithm 1, we evaluate the accuracy and cost of m feature subset candidates, where each considers the impact of removing feature j from the user-defined prunable set $\mathbf{F}_{prunable} \subseteq \mathbf{F}$. We then select the feature and sensor subsets $\mathbf{F}_{select} \subseteq \mathbf{F}, \mathbf{S}_{select} \subseteq \mathbf{S}$ and the pruned model α_{select} (Algorithm 2) that maximizes the cost function CF $= acc/cost_{norm}$, where $cost_{norm}$ is the evaluated hardware-aware cost C_{HA}, normalized according to the maximum achievable cost (from the most complex model available α_n). Note that the feature set selection drives the sensor set selection $\mathbf{S}_{select}$, as described before.

Algorithm 1: Feature selection heuristic

```
Input: α_k , F, F_prunable, S
Output: ⟨𝓕^(k), 𝓢^(k), 𝓐^(k)⟩
F_select ← F_prunable, S_select ← S
α_select ← α_k, m ← |F_select|
⟨𝓕^(k), 𝓢^(k), 𝓐^(k)⟩ ← ⟨F_select, S_select, α_select⟩
while m > 1 do
    for j = 1 to m do
        F_ca,j ← F_select \ F_j
        S_ca,j ← S_select
        for S ∈ S_select do
            if F_ca,j ∉ F_S then S_ca,j ← S_select \ S
            α_ca,j ← ACpruning(α_select, F_ca,j)
            acc_ca,j ← Acc(α_ca,j, F_ca,j)
            cost_ca,j ← C_HA(α_ca,j, F_ca,j, S_ca,j);
        end
        α_select ← arg max_{α_ca} CF(acc_ca, cost_ca)
        F_rm ← arg max_{F∉F_ca} CF(acc_ca, cost_ca)
        F_select ← F_select \ F_rm, m ← |F_select|
    end
    for S ∈ S_select do
        if F_select ∉ F_S then S_select ← S_select \ S
    end
    𝓕^(k) ← 𝓕^(k) ∪ F_select, 𝓢^(k) ← 𝓢^(k) ∪ S_select
    𝓐^(k) ← 𝓐^(k) ∪ α_select
end
return ⟨𝓕^(k), 𝓢^(k), 𝓐^(k)⟩
```

The output of Algorithm 1, $\langle \mathcal{F}^{(k)}, \mathcal{S}^{(k)}, \mathcal{A}^{(k)} \rangle$, is a set of system configurations of the form $\{\{\mathbf{F}_{sel,1}, \mathbf{S}_{sel,1}, \alpha_{sel,1}\}, \ldots, \{\mathbf{F}_{sel,q}, \mathbf{S}_{sel,q}, \alpha_{sel,q}\}\}$, where q=$|\mathbf{F}_{prunable}|$, and the superscript (k) denotes the number of the input model α_k, taken from $\boldsymbol{\alpha}$. For each configuration resulting from Algorithm 1, we can sweep the available precision configurations $\mathbf{nb}$, for a final space described by $\boldsymbol{\sigma}$=$\langle \mathcal{F}, \mathcal{S}, \mathcal{A}, \mathcal{N} \rangle$ of size $|\boldsymbol{\alpha}| \cdot |\mathbf{F}_{prunable}| \cdot |\mathcal{N}|$, where $\mathcal{N}$ contains the selected precision. In the experiments section we show a work-around to reduce search space size and the number of steps required by the Pareto-optimal search. Regarding complexity, the

Algorithm 2: ACpruning

```
Input   : α, F
Output : α_pr
α_pr ← α
foreach node a_{×,+} in α_pr do
    if factor1 is θ and factor2 is λ_f and f ∉ F then
        θ' = evaluate(a_{×,+})
        replace ⟨a_{×,+}, factor1, factor2⟩ ← θ'
    end
end
Return : α_pr
```

feature selection in Algorithm 1 is a greedy search, thus its complexity is linear in the number of features times the number of iterations needed for convergence to the desired accuracy or cost. The AC pruning routine consists of an upward pass on the AC and its complexity is therefore linear in the number of AC nodes.

5.2. Pareto-optimal configuration selection

Algorithm 3 describes how we extract the Pareto-optimal configuration subset, but any convex hull algorithm can be used. The input is $\boldsymbol{\sigma}$=$\langle \mathcal{F}, \mathcal{S}, \mathcal{A}, \mathcal{N} \rangle$ with the accuracy $\mathbf{Acc}$ and cost points $\mathbf{cost}$ evaluated at each configuration setting. The output of this algorithm is the set of Pareto-optimal system configurations $\boldsymbol{\sigma}^*$=$\{\{\alpha_i^*, \mathbf{F}^*{}_i, \mathbf{S}^*{}_i, nb_i^*\}_{i=1:p}\}$, each guaranteed to achieve the largest reachable accuracy for any given cost; or the lowest reachable cost for any given accuracy ($\mathbf{Acc}^*$, $\mathbf{cost}^*$).

Algorithm 3: Pareto-optimal configuration selection.

```
Input   : σ, Acc, cost
Output : σ*, Acc*, cost*
⟨cost*, σ*, Acc*⟩ ← ⟨{}, {}, {}⟩;
⟨cost, σ, Acc⟩ ← sorted(⟨cost, σ, Acc⟩);
Acc_min ← 0 , i ← |σ| + 1;
Acc_max ← ∞;
while i > 0 do
    i ← arg max Acc_{0:i};
    if Acc_max < Acc_i then break;
    σ* ← σ* ∪ σ_i;
    Acc* ← Acc* ∪ Acc_i;
    cost* ← cost* ∪ cost_i;
    Acc_max ← Acc_i
end
Return : σ*, Acc*, cost*
```

The next section illustrates how can our methodology reap the benefits of scalable embedded hardware.

Table 1. Experimental datasets
†: Classification , ⋆: Density est.

Dataset	$\lvert\mathbf{F}\rvert$	$\lvert\mathbf{F}_{prunable}\rvert$	$\lvert\boldsymbol{\alpha}\rvert$
Banknote†	15	15	11
HAR †	28	28	11
Houses †	36	20	6
Jester ⋆	99	20	5
Madelone †	20	20	11
Nltcs ⋆	15	15	11
Wilt †	11	11	11

6. Experimental evaluation

We empirically evaluate the proposed techniques on a relevant embedded sensing use case: the Human Activity Recognition (HAR) benchmark (Anguita et al., 2013). Additionally, we show our method's general applicability on a number of other publicly available datasets (Dua and Graff, 2017; Guyon and Gunn, 2003; Johnson et al., 2013; Lowd and Davis, 2010), two of them commonly used for density estimation benchmarks and the rest commonly used for classification (see Table 1).

Computational costs. The computation costs $\mathrm{C_{AC}}$ are based on the energy benchmarks discussed in Horowitz (2014) and Shah et al. (2019). Table 2 shows the relative costs of each term in $\mathrm{C_{AC}}$ and how they scale with precision nb. The baseline is 64 floating point bits because it is the standard IEEE representation in software environments. For the rest of the experiments, we consider three other standard low precision representations: 32 bits (8 exponent and 24 mantissa), 16 bits (5 exponent and 11 mantissa) and 8 bits (4 exponent and 4 mantissa) (IEEE, 2008).

Dataset pre-processing. For the classification benchmarks, we discretized numerical features using the method in Fayyad and Irani (1993). We then binarized them and subjected them to a 75%-train, 10%-validation and 15%-test random split. For the HAR benchmark, we preserved the timeseries information by using the first 85% samples for training and validation and the last for testing. For the density estimation datasets, we used the splits provided in Lowd and Davis (2010) and we assumed the last feature in the set to be the class variable. On all datasets, we performed wrapper feature selection (evaluating the features' value on a Tree Augmented Naive Bayes classifier) before going through the hardware-aware optimization process to avoid over-fitting on the baseline model and ensure it is a fair reference point. The number of effectively used features $|F|$ is shown in Table 1. In addition, we consider all the features to be in the prunable set $F_{prunable}$ for datasets with less than 30 features. For the rest, we consider the 20 with the highest correlation to the class variable. Within the context of an application, the prunable set can be user-defined. For instance, in a multi-sensor seizure detection application, medical experts might advise against pruning features extracted from an EEG monitor.

Table 2. Experiment computational costs.

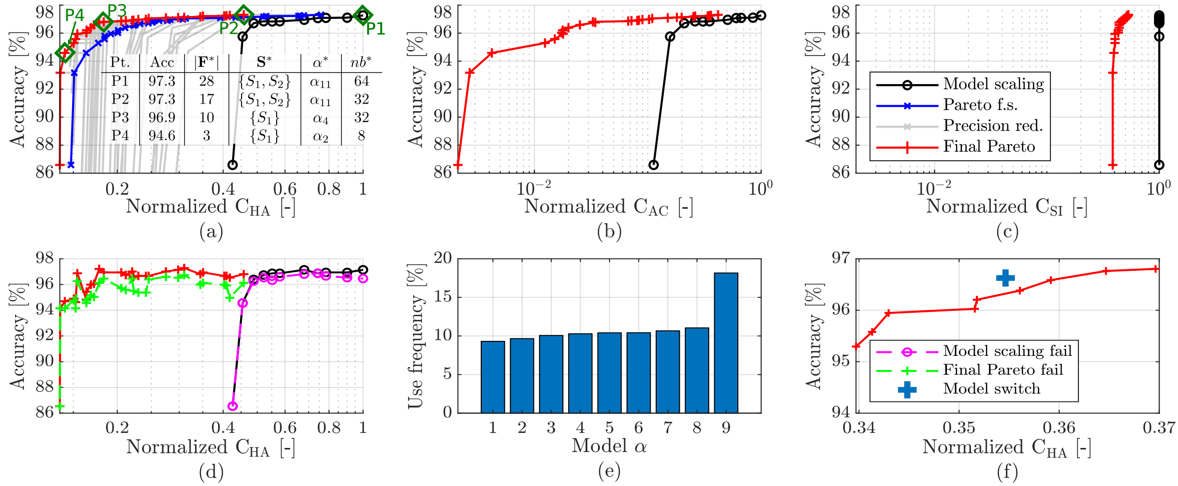
Operation	64 bit	Operation cost
$\mathrm{C_{mem}}$	1	$\phi_{mem} = \gamma_{mem} \cdot nb$
$\mathrm{C_{cache}}$	0.2	$\phi_{cache} = \gamma_{cache} \cdot nb$
$\mathrm{C_{\times}}$	0.6	$\phi_{\times} = \gamma_{\times}^2 \cdot nb^2 \cdot \log(nb)$
$\mathrm{C_{+}}$	0.1	$\phi_{+} = \gamma_{+} \cdot nb$

Model learning We learned the models on the train and validation sets with the LearnPSDD algorithm (Liang et al., 2017), using the same settings reported therein, and following the bottom-up vtree induction strategy. To populate the model set $\boldsymbol{\alpha}$, we retained a model after every $N/10$ iterations, where N is the number of iterations needed for convergence (in the algorithm this is until the log-likelihood on validation data stagnates). Table 1 shows $|\boldsymbol{\alpha}|$ for each dataset.

6.1. Embedded Human Activity Recognition

The HAR dataset aims to recognize the activity a person is performing based on 561 statistical and energy features extracted from smartphone accelerometer and gyroscope data. We perform binary classification by discerning "walking downstairs" from the other activities. For the experiments, we use a total of 28 binary features, 10 of which are extracted from the gyroscope's signal and the rest from the accelerometer, as detailed in appendix D. All computation costs for this dataset are normalized according to the energy consumption trends of an embedded ARM M9 CPU, assuming 0.1nJ per operation (Tarkoma et al., 2014). Sensors are estimated to consume 2mWatt, while the costs of all features is defined as 30 MAC operations (see appendix B for more details).

Pareto optimal configuration. This experiment consisted of three stages, as shown in Figure 3(a): 1) We first mapped each model in $\boldsymbol{\alpha}$ to the trade-off space, as shown in black. 2) Starting from each model, we scaled the feature and the sensor sets $\mathbf{F}$, $\mathbf{S}$, as shown in blue. 3) We then scaled the precision nb of each of these pruned configurations (shown by the grey curves) and we finally extracted the Pareto front shown in red. As shown by the Pareto configurations highlighted in green, our method preserves the highest baseline accuracy by pruning 11 of the available 28 features, which results in a reduced resource consumption of 55%. When willing to tolerate 0.5% less accuracy, our method outputs a configuration that consumes only 20% of the original cost by using a smaller model (α_4), pruning 18 features, turning one sensor off and using a 32 bit representation. Figures 3(a,b) break down the computational cost $\mathrm{C_{AC}}$ and the sen-



Pt.	Acc	$\|\mathbf{F}^*\|$	$\mathbf{S}^*$	α^*	nb^*
P1	97.3	28	$\{S_1, S_2\}$	α_{11}	64
P2	97.3	17	$\{S_1, S_2\}$	α_{11}	32
P3	96.9	10	$\{S_1\}$	α_4	32
P4	94.6	3	$\{S_1\}$	α_2	8

Figure 3. Experiments on the Human Activity Recognition bechmark.

sor costs C_{SI}. When considering only the costs of the AC evaluation (graph (b)), our method results in savings of almost two orders of magnitude with accuracy losses lower than 1%, and up to 3 orders of magnitude when tolerating more losses. Sensor and feature costs, as shown in graph (c) only scale up to 50%, since at least one of the sensors must always be operating. This demonstrates the importance of taking these costs into account: even though computation costs savings are impressive, the system is still limited by the sensing costs.

Robustness and online deployment. The red curve in Figure 3(d) shows that the evaluation of the selected Pareto configurations against testing data stays within a range of $\pm 1\%$ with respect to Figure 3(a). We also assessed the robustness of our method by simulating, per configuration, ten iterations of random failure of varying sizes of feature sets ($|\mathbf{F}|/10$,$|\mathbf{F}|/5$,$|\mathbf{F}|/2$). The green and magenta dotted curves show the median of these experiments for the Pareto configurations and for the original model set. We can see that these trials stay within a range of -2% compared to the fully functional results in red and black, which validates our choice of a generative learner that can naturally cope with missing features at prediction time.

In embedded sensing scenarios, environmental circumstances, power consumption requirements and accuracy constraints commonly vary over time. For example, when battery power is low, a small drop in accuracy to save power might be acceptable. Or when no activity is observed, it might be beneficial to periodically go to a lower power consumption stand-by state. This calls for dynamic operating settings, where the system can switch between different accuracy-cost operating points at run-time. Figure 3(f) shows such a switching scenario with 9 operating-points, which comply with hypothetical user needs: Acc $\geq 95\%$, cost $\leq 40\%$. The implemented policy has two actions: an energy efficient action whenever the classifier has identified no changes in activity during a period of time, and a high reliability action whenever the classifier has identified recent changes in activity (see appendix C for more details). Figure 3(f) contrasts the cost-accuracy performance attained when always using the same model (in red), with the cost-accuracy performance resulting from the implementation of our model-switching policy (blue cross). Even with its simplicity, the proposed policy attains accuracy vs. cost improvements that go beyond the static Pareto front. Figure 3(e) shows how this is achieved by making a balanced use of the 9 available operating points.

6.2. Generality of the method: evaluation on benchmark datasets

We now apply our optimization sequence to the datasets in Table 1. For lack of information on the hardware that originated these datasets, we only consider the computation cost C_{AC}, again evaluated on the cost model of the ARM M9 CPU. Table 3 shows this cost along with the training and testing accuracy ($\mathrm{Acc}_{\mathrm{tr}}$,$\mathrm{Acc}_{\mathrm{te}}$) at four operating points for every dataset. We can see that all the benchmarks strongly benefit from our proposed methodology and that they are all robust when contrasted against the test dataset. Appendix A shows a figure with the Pareto fronts for all the experiments herewith.

7. Conclusions

We proposed a novel hardware-aware cost metric to deal with the limitations of the efficiency vs. performance trade-off considered by the field of tractable learning. Our method obtains the Pareto-optimal system-configuration set in the hardware-aware cost vs. accuracy space. The proposed so-

Table 3. Results for benchmarking datasets [$\mathrm{C_{AC}}$,$\mathrm{Acc_{tr}}$%,$\mathrm{Acc_{te}}$%].

Dataset	Operating pt. 1	Operating pt.2	Operating pt. 3	Operating pt. 4
Banknote	[1, 94.5, 95.6]	[0.17, 94.5, 95.6]	[0.10, 93.1, 94.2]	[0.03, 84.5, 86.7]
Houses	[1, 97.6, 97.4]	[0.09, 97.1, 96.6]	[0.03, 96.1, 95.7]	[0.01, 94.3, 94.0]
Jester	[1, 76.9, 76.7]	[0.40, 77.0, 77.0]	[0.10, 76.2, 76.4]	[0.04, 74.6, 74.7]
Madelone	[1, 68.1, 68.4]	[0.12, 68.6, 69.0]	[0.07, 66.1, 65.7]	[0.03, 65.7, 66.7]
Nltcs	[1, 93.5, 93.9]	[0.12, 93.3, 93.6]	[0.03, 90.8, 91.4]	[0.008, 89.5, 89.5]
Wilt	[1, 97.1, 97.5]	[0.14, 97.1, 97.5]	[0.06 ,97.1 ,97.5]	[0.02, 94.5, 94.6]

lution consists of a sequential hardware-aware search and a Pareto-optimal configuration selection stage. Experiments on a variety of benchmarks demonstrated the effectiveness of the approach and sacrifice little to no accuracy for significant cost savings. This opens up opportunities for the efficient implementation of probabilistic models in resource-constrained edge devices, operating in dynamic environments.

Future work includes learning comprehensive online-model-tuning policies for applications that must perform under dynamic environments, such as autonomous robot navigation and biomedical monitoring applications; and implementing the techniques proposed herewith to trade-off accuracy for time-based hardware resources such as latency and throughput.

Acknowledgements

The authors thank Nimish Shah for helpful discussion and Yitao Liang for the LearnPSDD algorithm and for helpful feedback. This work is partially supported by the EU ERC Project Re-SENSE under Grant ERC-2016-STG-71503; NSF grants #IIS-1657613, #IIS-1633857, #CCF-1837129, DARPA XAI grant #N66001-17-2-4032, NEC Research, and a gift from Intel and Facebook Research.

References

Anguita, D., Ghio, A., Oneto, L., Parra, X., and Reyes-Ortiz, J. L. (2013). A public domain dataset for human activity recognition using smartphones. In *Esann*.

Buchfuhrer, D. and Umans, C. (2008). The complexity of boolean formula minimization. In *International Colloquium on Automata, Languages, and Programming*, pages 24–35. Springer.

Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799.

Choi, Y. and Broeck, G. V. d. (2018). On robust trimming of bayesian network classifiers. *arXiv preprint arXiv:1805.11243*.

Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264.

Dua, D. and Graff, C. (2017). UCI machine learning repository.

Fayyad, U. and Irani, K. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. *IJCAI*.

Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., and De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted CNFs. *Theory and Practice of Logic Programming*, 15.

Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine learning*, 29(2-3):131–163.

Gens, R. and Pedro, D. (2013). Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880.

Guyon, I. and Gunn, S. (2003). Nips feature selection challenge.

Hashemi, S., Anthony, N., Tann, H., Bahar, R. I., and Reda, S. (2017). Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1474–1479. IEEE.

Heckerman, D. (1993). Causal independence for knowledge acquisition and inference. In *Uncertainty in Artificial Intelligence*, pages 122–127. Elsevier.

Horowitz, M. (2014). 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H.

(2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

IEEE (2008). Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70.

Johnson, B. A., Tateishi, R., and Hoan, N. T. (2013). A hybrid pansharpening approach and multiscale object-based image analysis for mapping diseased pine and oak trees. *International journal of remote sensing*, 34(20):6969–6982.

Kim, D., Celio, C., Biancolin, D., Bachrach, J., and Asanovic, K. (2017). Evaluation of risc-v rtl with fpga-accelerated simulation. In *First Workshop on Computer Architecture Research with RISC-V*.

Liang, Y., Bekker, J., and Van den Broeck, G. (2017). Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.

Liang, Y. and Van den Broeck, G. (2019). Learning logistic circuits. *Proceedings of the 33rd Conference on Artificial Intelligence (AAAI)*.

Lowd, D. and Davis, J. (2010). Learning markov network structure with decision trees. In *2010 IEEE International Conference on Data Mining*, pages 334–343. IEEE.

Lowd, D. and Domingos, P. (2012). Learning arithmetic circuits. *arXiv preprint arXiv:1206.3271*.

Lowd, D. and Rooshenas, A. (2013). Learning markov networks with arithmetic circuits. In *Artificial Intelligence and Statistics*, pages 406–414.

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., and De Raedt, L. (2018). Deepproblog: Neural probabilistic logic programming. *NIPS*.

Moons, B., Bankman, D., Yang, L., Murmann, B., and Verhelst, M. (2018). Binareye: An always-on energy-accuracy-scalable binary cnn processor with all memory on chip in 28nm cmos. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE.

Piatkowski, N., Lee, S., and Morik, K. (2016). Integer undirected graphical models for resource-constrained systems. *Neurocomputing*, 173:9–23.

Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE.

Rooshenas, A. and Lowd, D. (2014). Learning sum-product networks with direct and indirect variable interactions. In *International Conference on Machine Learning*, pages 710–718.

Shah, N., Olascoaga, L. I. G., Meert, W., and Verhelst, M. (2019). Problp: A framework for low-precision probabilistic inference. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 190. ACM.

Sommer, L., Oppermann, J., Molina, A., Binnig, C., Kersting, K., and Koch, A. (2018). Automatic mapping of the sum-product network inference problem to fpga-based accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 350–357. IEEE.

Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. (2018). Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*.

Tarkoma, S., Siekkinen, M., Lagerspetz, E., and Xiao, Y. (2014). *Smartphone energy consumption: modeling and optimization*. Cambridge University Press.

Tschiatschek, S. and Pernkopf, F. (2015). On bayesian network classifiers with reduced precision parameters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 37(4):774–785.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Van den Broeck, G. (2018). A semantic loss function for deep learning with symbolic knowledge. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*.

A. Experiments on benchmarking datasets

Figure 4 shows the cost vs accuracy mapping for all datasets. The Pareto curves obtained by the proposed method on the left, and their evaluation on the test dataset on the right.

B. Cost estimation for experiment on HAR

Sensor costs We assume feature extraction and classification take place in an ARM 9 CPU, which consumes, on average 1 Watt to execute 10G operations per second. That means it will consume approximately 0.1nJ per operation (see https://developer.arm.com/products/processors/classic-processors) and will thus require approximately 1 μJ per instance classified (the largest AC requires has 5000 operations and at least two passes are necessary to classify an instance). We assume both the gyroscope and the accelerometer consume at least 2 mW when operating at 10KHz, and that they thus consume roughly 0.2 μJ per operation of the CPU. Note that gyroscopes can consume as much as 10 times more energy than accelerometers, but we can assume that they are both part of a larger system, such as a Inertial Measurement Unit, hence we assign the same cost to both. Thus, we set the cost of the sensors relative to the computation costs, that is, each sensor has a cost 10% the total cost of classifying a single instance in the most complex model available.

Feature costs The features of this dataset are extracted by sampling the sensory signal, applying three low-pass filters and calculating statistical quantities (mean,maximum/minimum,correlation and standard deviation) on the resulting signal. Sampling and extracting the statistical features require a small number of operations in comparison to filtering. For example, calculating the mean of a sample requires a single MAC (Multiply-Accumulate, consisting of a multiplication and an addition) operation, whereas a 3rd order low pass filter will require, at least, 9. Filtering thus takes the bulk of the computations, so we assume a lower bound of 30 MAC operations per extracted feature.

C. Model switching experiment

The Pareto-configuration model set used in this experiment was selected according to hypotetical user's needs: "accuracy must be larger than 95% and cost lower than 40%". The implemented policy is the following:

- When the predictions at $t-5$ to $t-1$ have been "no activity": 1) switch to a simpler model if prediction at t is "no activity", 2) switch to a more complex model when prediction at time t is "activity".
- When the predictions at $t-5$ to $t-1$ have been "ac-

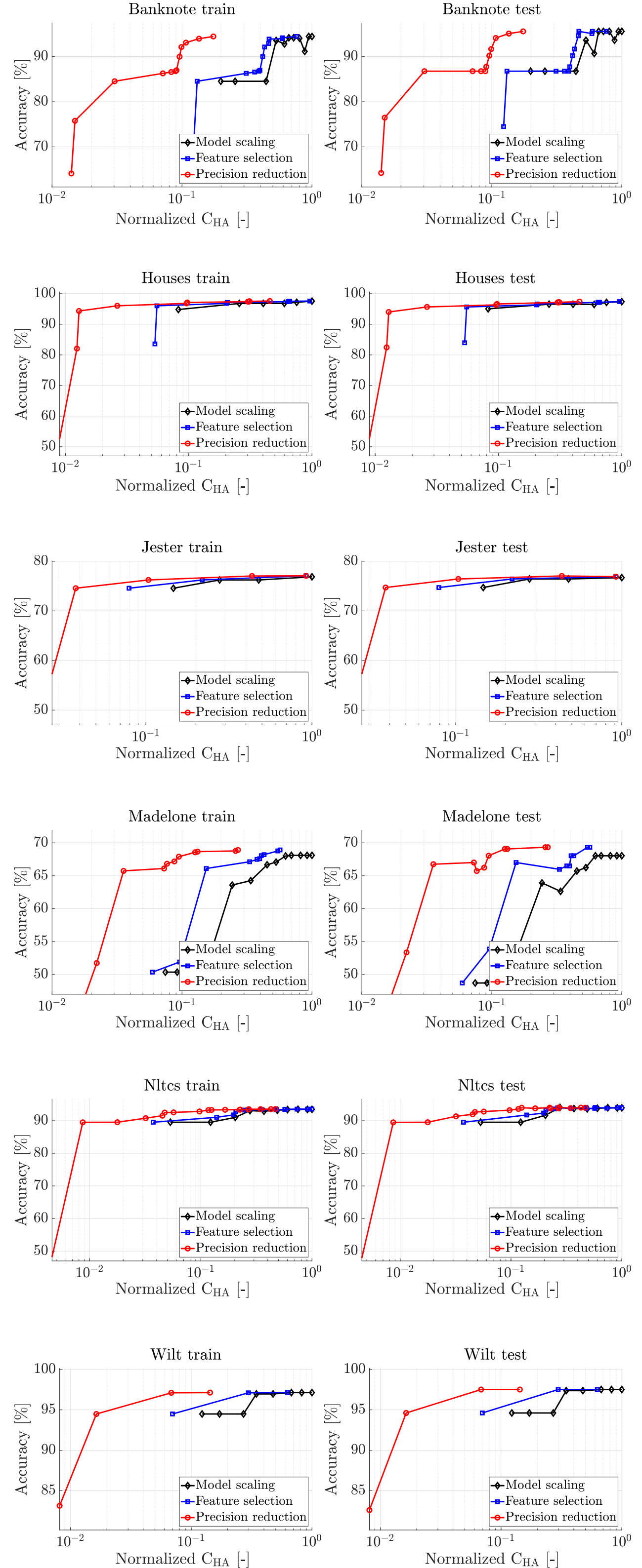


Figure 4. Pareto frontier for benchmarking datasets

tivity": 1) switch to a simpler model if prediction at t is "no activity", 2) switch to a more complex model when prediction at time t is "activity".

D. Features used for experiments on HAR

1 tBodyAcc-mean()-X-value1
2 tBodyAcc-mean()-X-value2
3 tBodyAcc-mean()-Y-value1
4 tBodyAcc-mean()-Y-value2
5 tBodyAcc-max()-Y-value1
6 tBodyAcc-max()-Y-value2
7 tBodyAcc-max()-Y-value3
8 tBodyAccJerk-entropy()-X-value1
9 tBodyAccJerk-entropy()-Y-value2
10 tBodyAccJerk-entropy()-Y-value1
11 tBodyAccJerk-entropy()-Y-value2
12 tBodyAccJerk-entropy()-Z
13 tBodyAccJerk-arCoeff()-X,4
14 tBodyAccJerk-arCoeff()-Y,1
15 tBodyAcc-max()-Z
16 tBodyAccJerk-arCoeff()-Z,1
17 tBodyAccJerk-arCoeff()-Z,2
18 tBodyAccJerk-arCoeff()-Z,3
19 tBodyGyroJerk-corr.()-X,Z-value1
20 tBodyGyroJerk-corr.()-X,Z-value2
21 tBodyGyroJerk-corr.()-X,Z-value3
22 tBodyGyroJerk-corr.()-X,Z-value4
23 tBodyGyro-mean()-X
24 tBodyGyro-std()-Y
25 tBodyGyro-min()-Y
26 tBodyGyro-energy()-X
27 tBodyGyro-arCoeff()-X,1
28 tBodyGyro-correlation()-X,Y