BOOSTED TREES ON A DIET: COMPACT MODELS FOR RESOURCE-CONSTRAINED DEVICES

Anonymous authors
Paper under double-blind review

ABSTRACT

Deploying machine learning models on compute-constrained devices has become a key building block of modern IoT applications. In this work, we present a compression scheme for boosted decision trees, addressing the growing need for lightweight machine learning models. Specifically, we provide techniques for training compact boosted decision tree ensembles that exhibit a reduced memory footprint by rewarding, among other things, the reuse of features and thresholds during training. Our experimental evaluation shows that models achieved the same performance with a compression ratio of 4–16x compared to LightGBM models using an adapted training process and an alternative memory layout. Once deployed, the corresponding IoT devices can operate independently of constant communication or external energy supply, and, thus, autonomously, requiring only minimal computing power and energy. This capability opens the door to a wide range of IoT applications, including remote monitoring, edge analytics, and real-time decision making in isolated or power-limited environments.

1 INTRODUCTION

Modern Internet of Things (IoT) techniques have paved the way for new applications in domains such as home automation, healthcare, agriculture, or industry (Khanna & Kaur, 2020). For instance, in the context of home automation, sensors can be used to automate (smart) lighting and (smart) heating. Another example is using sensor data for predictive maintenance to prevent machine failures and enhance productivity. Microcontrollers are a key building block of these IoT applications, often equipped with sensors to measure parameters such as temperature, humidity, pressure, or vibrations. A key characteristic of such devices is that they generally have very limited computing and memory resources (Ojo et al., 2018). For instance, the broadly used open-source *Arduino Uno R4 Minima* board is equipped with a 32-bit *Renesas RA4M1* microcontroller and an *Arm Cortex-M4* processor running at 48 MHz, 32 KB main memory (RAM), 256 KB flash storage, and 1 KB electrically erasable programmable read-only memory (EEPROM). There are also microcontrollers with even less memory and computing resources, such as the *Arduino Nano* board. Furthermore, the available resources must be shared among the operating system, sensing data, data-processing programs, and machine learning models. Another characteristic is that IoT microcontrollers are often designed for energy efficiency. This makes them well-suited for being deployed in remote locations, where a continuous power supply is not available. Under ideal conditions, such devices can run for several months or even years via batteries. Such applications are also supported by specialized communication protocols such as LoRa.[1]

To minimize energy-intensive data transfers and to enable real-time processing, the on-site analysis of data on the IoT device is preferable if possible. Embedding machine learning methods directly into IoT nodes addresses this need. The key challenge in embedded machine learning is to minimize both compute and memory requirements to enable execution on resource-constrained devices while, at the same time, preserving model quality. This has given rise to the concept of *Tiny Machine Learning (TinyML)* models (Gural & Murmann, 2019; Reddi et al., 2021; Kumar et al., 2017; Warden & Situnayake, 2019). By sufficiently reducing resource demands, the "tiny" models can be run directly on the microcontrollers, enabling compact "smart" devices. An example of a corresponding IoT application is sketched in Figure 1.

[1] LoRa enables small volumes of data to be transferred over several kilometres (Mayer et al., 2019).

Contribution: We propose a framework that allows for compressing boosted decision tree ensembles, one of the most widely used machine learning models for structured data. Our approach, referred to as Trees on a Diet (`ToaD`), relies on (1) regularizers that encourage the reuse of features and thresholds during training and on (2) a specialized memory layout to store the resulting trees. More precisely, we utilize global lookup tables for both features and thresholds, and we store the trees without pointers using an adapted bit-wise encoding. Overall, these modifications lead to tree ensembles with reduced memory requirements, without sacrificing model quality. We showcase the effectiveness of our approach by assessing the quality-memory trade-off in our experimental evaluation. Our results indicate that the adapted training process yields models of comparable performance while achieving compression ratios of 4-16x compared to other baselines.

2 Background

We begin with the background on boosted decision trees and tiny machine learning models.

2.1 Boosted Decision Trees

In contrast to single decision trees, tree ensembles aggregate the outputs of multiple trees. For instance, a random forest (Breiman, 2001) is obtained by constructing the trees independently from each other, introducing randomness to the construction process so that a set of different trees is obtained. In contrast, boosted decision trees are built in an incremental manner, one tree at a time. Two prominent frameworks in this context are XGBoost (Chen & Guestrin, 2016) and LightGBM (Ke et al., 2017).

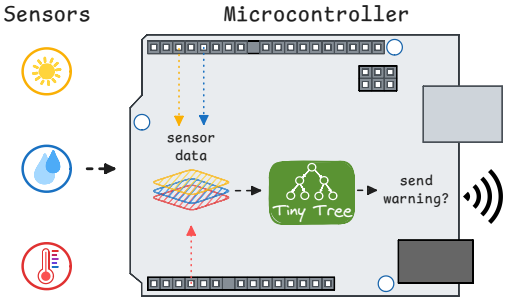

Figure 1: A machine learning model (decision tree) on a microcontroller processes multi-sensor data locally and transmits only relevant events, reducing energy costs; the decision tree must have a minimal compute and memory footprint.

Let $\mathcal{T} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \mathcal{Y}$ be a training (multi)set of data points $\mathbf{x}_i \in \mathbb{R}^d$ with associated labels $y_i \in \mathcal{Y}$. Here, $\mathcal{Y}$ is the set of labels. For regression, we have $\mathcal{Y} = \mathbb{R}$, whereas one is given a finite set $\mathcal{Y} = \{p_1, \ldots, p_c\}$ of classes for classification scenarios. Both XGBoost and LightGBM model ensembles are built in an additive way, resulting in a tree ensemble model T, whose prediction $T(\mathbf{x})$ for a new data point $\mathbf{x} \in \mathbb{R}^d$ is based on the sum of predictions made by K individual decision trees t_k, i.e., $T(\mathbf{x}) = \sum_{k=1}^{K} t_k(\mathbf{x})$, where $t_k(\mathbf{x})$ denotes the prediction made by the tree t_k for an instance $\mathbf{x} \in \mathbb{R}^d$. Such ensembles are built in a way to minimize

$$\sum_{i=1}^{n} \mathcal{L}\left(y_i, T(\mathbf{x}_i)\right) + \sum_{k=1}^{K} \Omega\left(t_k\right), \tag{1}$$

where $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ is a suitable loss function and where $\Omega\left(t_k\right)$ specifies the complexity of the tree t_k. Constructing an optimal tree ensemble T w.r.t. Equation (1) is generally not feasible. Instead, one typically resorts to K boosting rounds, and in each round, one new tree is added to the ensemble built so far so that the Equation (1) is minimized. More precisely, in boosting round $m \geq 1$, one considers the ensemble $T_{m-1} \coloneqq \sum_{k=1}^{m-1} t_k$ of the trees built so far, starting with $T_0 \coloneqq 0$, and aims to find the next best tree t_m that minimizes $\sum_{i=1}^{n} \mathcal{L}\left(y_i, T_{m-1}(\mathbf{x}_i) + t_m(\mathbf{x}_i)\right) + \sum_{k=1}^{m} \Omega(t_k)$. The decision tree t_m itself is also constructed in a greedy manner, starting with the root of the tree being recursively split. To penalize "complex" trees, one usually resorts to $\Omega(t_m) = \gamma L + \frac{1}{2}\lambda \sum_{j=1}^{L} v_j^2$ as regularizer, where L is the number of leaves and $v_1, \ldots, v_L \in \mathbb{R}$ the associated leaf values of t_m.

2.2 Related Work

Decision trees and decision tree ensembles, such as Gradient Boosted Decision Trees (GBDT), have remained very popular, despite the rise of deep learning models (Grinsztajn et al., 2022), especially for structured (tabular-like) data. They are also generally easy to interpret and need few computational resources. Recent advances in tree-based models have also introduced several approaches to enhance their efficiency. For instance, several works focused on improving the training or inference speed of GBDT by means of array structures (Lucchese et al., 2017; Ye et al., 2018) or

via the quantization of gradient statistics (Shi et al., 2022; Jiang et al., 2018; Devos et al., 2020). Quantization of tree parameters was employed by Koschel et al. (2023), who adapted variants of QuickScorer (Lucchese et al., 2017) to enable deployment on IoT devices. Moreover, quantization of GBDT for resource-constrained devices was investigated by Alsharari et al. (2025) and Wang et al. (2023), both targeting FPGAs. Reduction of model size and latency is the motivation for different post-training pruning techniques (Liu & Mazumder, 2023; Guo et al., 2018), some including the refinement of leave values (Devos et al., 2025; Emine et al., 2025; Buschjäger & Morik, 2023). Increased efficiency of tree models by design or within the training process is addressed by works from Kumar et al. (2017) for single decision tree sizes, Ponomareva et al. (2017) for multiclass classification, and Peter et al. (2017) for efficient evaluation of deep trees considering feature acquisition and tree evaluation costs. Further works mostly focusing on optimization of random forests include Ren et al. (2015), Alkhoury et al. (2025), and Nan et al. (2016). Due to a lack of space, we refer to Appendix C for a more detailed discussion of related approaches.

In this work, we focus on gradient-boosted trees. For subsequent deployment we aim to minimize the memory footprint of the ensemble during the training process, while simultaneously ensuring a good model fit and compact model size. We extend the work of Peter et al. (2017) by defining suitable feature costs that aim at reducing the bits required to encode feature indices. We also introduce a corresponding cost regularizer for split thresholds and leaf values, and provide a specialized encoding for the induced binary trees. Moreover, our memory layout includes global threshold arrays shared by all learners.

3 Approach

Pruning or quantization techniques are typically applied before or after the training to reduce the size of trees (see Section 4). However, such methods generally cannot exploit task-specific compression potential. For instance, they are not designed to incorporate the potential in memory saving of feature sharing or the reuse of (leaf/split) thresholds within a single tree or across all trees in the ensemble. Our framework exploits this potential by penalizing unused features and thresholds when growing new trees. In combination with a corresponding memory layout, this yields a substantially smaller memory footprint with reused features and thresholds being stored more compactly.[2]

3.1 Training Compressed Trees

As sketched above, boosted tree ensembles are built in an incremental manner, and in each boosting round m, a new tree t_m is added to the ensemble. The tree t_m is, in turn, also built in a greedy manner by iteratively splitting leaves (starting with the root) if this leads to a better objective (if not, the construction process stops). To assess the quality of such a leaf split, an associated *gain* Δ is computed (Chen & Guestrin, 2016). More specifically, for a leaf associated with a set I of training indices, a split along feature dimension $i \in \{1, \dots, d\}$ with respect to threshold μ induces a potential gain $\Delta(I, i, \mu) \in \mathbb{R}$ (which may also be negative). Leaves are split as long as some split yields a positive gain, always choosing the leaf and split with the highest gain.

The standard gain does not promote the reuse of features or thresholds. Following Peter et al. (2017), we introduce an additional regularizer based on the set of features $F_U \subseteq \{1, \dots, d\}$ and thresholds $\mathcal{T}^f \subset \mathbb{R}$ with $f \in F_U$ that have already been used by the trees $t_1, \dots, t_m$ built so far (including the current tree t_m). The memory layout detailed below allows for storing those features and thresholds in a much more compact manner. In particular, features and thresholds that have already been used in previous trees contribute only marginally to the overall space consumption and are therefore essentially "free of charge". A simple linear regularizer that favors such a reuse of features and thresholds is given by

$$\Omega_l(t_m) = \Omega(t_m) + \iota \cdot |F_U| + \xi \cdot \sum_{f \in F_U} |\mathcal{T}^f|, \tag{2}$$

[2] Our work targets resource-constrained devices where memotry, not latency or energy, is the main bottleneck. In many applications, the model size determines whether a deployment is feasible or not. It is worth pointing out that local on-device inference is generally far more energy-efficient than transmitting data to a remote server and also incurs only minimal latency. Technically, our compression scheme adds only minimal overhead (a few bit-wise operations), so we expect the impact on latency and energy per prediction to be negligible.

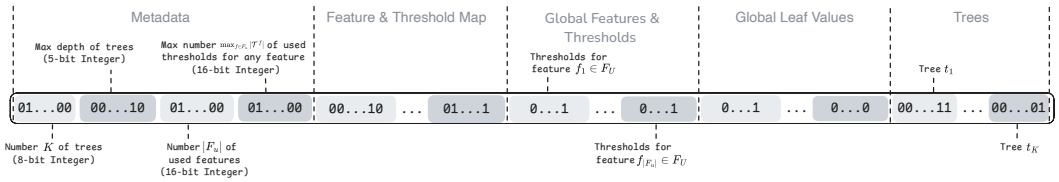

Figure 2: High-level sketch of the memory layout used to store an ensemble of boosted decision trees. The first part stores some metadata, such as the number K of boosted trees or the maximum depth of all trees. The following three parts encode the used features, thresholds, and leaf values. Finally, the references to the features and thresholds for the individual trees are stored.

where $\iota, \xi \in \mathbb{R}^+$ are user-defined hyperparameters. Thus, using a new feature that has not been used so far leads to an increase of $|F_U|$ by 1, and, hence, to an increase of the objective (1) by ι. Accordingly, if a new threshold is used for a feature $f \in F_U$, the objective is increased by ξ.[3] Using this modified regularizer leads to the following modified gain

$$\Delta_l(I, i, \mu) = \Delta(I, i, \mu) - s_f \iota - s_t \xi, \tag{3}$$

where $s_f = 1$ in case a new feature (index) is used and $s_f = 0$ otherwise, and $s_t = 1$ in case a new threshold is used and $s_t = 0$ otherwise, see Appendix A for the derivations.

3.2 Memory Layout

Our memory layout leverages the reuse of feature indices and thresholds enforced by the modified regularizer. A high-level sketch of the overall memory layout is shown in Figure 2. In a nutshell, it reduces the memory footprint of boosted tree ensembles through two mechanisms:

1. *Bit-wise encoding:* Encoding the information in a bit-wise manner allows to store a minimal representation of information compared to the use of higher level data types that may use non-minimal representations (e.g. a `bool` occupies eight bits in memory in `C`).
2. *Shared thresholds and leaf values:* Global arrays are used to store threshold and leaf values, which are referenced within internal tree nodes and leaves. Sharing values across trees can substantially reduce the bits needed for storing thresholds and leaf weights.

The memory layout comprises five components. The first stores metadata, including the number K of trees, the maximum tree depth, the number $|F_U|$ of used features, and the maximum number of thresholds $\max_{f \in F_U} |\mathcal{T}^f|$ associated with any feature. In addition, three global arrays and the individual decision trees $t_1, \ldots, t_K$ are stored. Figure 3 illustrates the bit-wise encoding of a simple model with two exemplary decision trees. We now detail the individual components.

3.2.1 Bit-Wise Encoding

Boosted ensembles typically employ shallow, nearly balanced trees with a small depth (e.g., a depth of up to 5). Such trees can be stored efficiently using pointer-less schemes. More precisely, the root is stored at index $i = 0$, and for a node at index i, the left child is stored at index $2 \cdot i + 1$ and the right child at index $2 \cdot i + 2$. For example, in Figure 3, two such array-based representations are given. Here, the root n_1 of tree t_1 is stored at index 0, and its two children n_2 and n_3 are stored at indices 1 and 2, respectively. We distinguish between internal nodes and leaf nodes:

- *Internal nodes:* For each internal node n_i, two pieces of information are stored, namely a reference for the feature $f_i \in F_U$ that is used for splitting along with an index for the associated threshold value μ_j^i (j-th threshold associated with feature f_i). For instance, for tree t_2 and node n_1, we store a reference `10` for feature f_3 and an index `0` for the associated threshold μ_1^3. The reference `10` can be used to loop up the relevant information

[3] For example, if a feature corresponds to temperature values, it may be sufficient to restrict the set of admissible thresholds to, e.g., 0 and 20 degrees Celsius. Alternative regularization schemes are also conceivable. Another option is, for instance, $\Omega_e(t_m) = \Omega(t_m) + \iota \cdot \sum_{j=1}^{|F_U|} j + \xi \cdot \sum_{j=1}^{p} j$ with $p = \sum_{f \in F_U} |\mathcal{T}^f|$, which imposes an exponentially increasing penalty on the number of distinct features and thresholds. In practice, however, the linear regularizer has been shown to be highly effective and is, hence, used in this work.

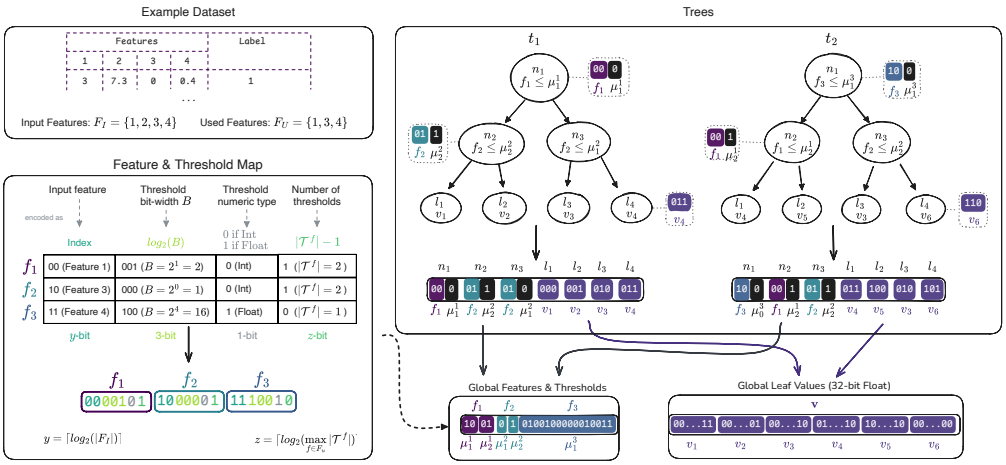

Figure 3: Illustration of a bit-wise encoding of a boosted tree ensemble with two trees. Each tree is stored in a bit-wise manner, with each internal node storing a reference to a feature index and a reference to a threshold index. For instance, for the left child n_2 of the root of the tree t_1, a reference to feature f_2 is stored along with a reference to the associated threshold μ_2^2. For feature f_2, there are two thresholds used by the entire ensemble, namely μ_1^2 and μ_2^2, which can be used by any node of any of the trees (e.g. t_2 in node n_3). Accordingly, the leaf values stored in the leaves of the trees are shared (e.g. v_4 is used in leaf l_4 of tree t_1 and leaf l_1 of t_2) and stored in one array. Since the bit-size of the thresholds varies, additional metadata is stored in the `Feature & Threshold Mapping` table/array. For instance, there are two thresholds for feature f_1 of bit-size 2 (i.e., four different values), whereas there are two 1-bit thresholds for feature f_2.

in the `Feature & Threshold Map`. In this case, f_3 corresponds to the fourth feature, 2^4 bits are used to represent the thresholds (floating point), and one threshold was used overall for that feature. Using the index `0` and along with the information stored for the other features, one can then loop up the threshold value μ_1^3.

- *Leaf nodes:* For each leaf node, a reference to its leaf value v is stored. For instance, for the leaf l_1 of tree t_1, a reference `000` to the leaf value v_1 is stored. All these leaf values are shared across all the trees and are stored in the array `Global Leaf Values`.

The details for each feature are stored in a bit-wise manner in the array `Feature & Thresholds Map` that is referred to when decoding the nodes. To process such a reference we need to know the bit widths of following information:

(a) `Input feature index`: For a given dataset with features $F_I = \{1, \dots, d\}$, the number d of features is known and can be encoded using $\lceil \log_2(|F_I|) \rceil$ bits.

(b) `Threshold bit-width`: Threshold values are assumed to be representable as 1-bit (binary feature), 2|4-bits (small integers), or 8|16|32-bits (floating point with different precision or integers). The threshold bit-width per feature can be stored as a power of two, requiring only three bits to represent the aforementioned values (2^0 to 2^5).

(c) `Threshold numeric type`: The representation can be either a floating point number (float) or a fixed point number (integer), and can be stored using a single bit, allowing for big integers and floating point values to be used.

(d) `Number of thresholds`: The maximum number $\max_{f \in F_U} |\mathcal{T}^f|$ of thresholds among all features can be determined at training time and can be encoded using $\lceil \log_2(\max_{f \in F_U} |\mathcal{T}^f|) \rceil$ bits. Since features with 0 thresholds are not included, we map the value 0 to the threshold count 1 (i.e., the bit value $+1$ is the actual count).

Hence, the array-based representation `Trees` of the trees $t_1, \dots, t_K$ along with the `Feature & Threshold Map` are used to store the trees and additional information to retrieve the actual split feature indices and thresholds and leaf values in the two global arrays, which are described next.

3.2.2 SHARING THRESHOLDS AND LEAF VALUES

The threshold values are stored on a per-feature basis in a single array, see `Global Features & Thresholds` in Figure 3, which is referenced by nodes throughout the entire tree ensemble. For example, consider a tree node n_1 referencing the first feature, f_1, along with its corresponding first threshold, μ_1^1, as depicted in tree t_1 in Figure 3. The `Feature & Threshold Map` allows for calculating the offset for each feature by determining the memory consumption of all previous features. Therefore, the associated threshold value (i.e. `10` for μ_1^1) can be extracted and decoded to its original representation (i.e. $10 \rightarrow (int)2$ for μ_1^1). This permits the variation of both the bit size and precision between different features within a single array.

The leaf values are stored (globally) in the array `Global Leaf Values` using a fixed 32-bit floating point representation. This allows for a high precision in leaves and a reuse across the different trees in the ensemble without feature reference. For example, consider the fourth global leaf value v_4 in Figure 3, which is referenced by both the leaf l_4 in tree t_1 and leaf l_1 of t_2.

4 EXPERIMENTS

To assess the quality of our compression approach and the impact of the additional parameters on the results, we ran three kinds of experiments: (1) A performance comparison of `ToaD` with other GBDT and tree ensemble optimization methods, (2) an univariate sensitivity analysis evaluating the threshold and feature penalties independently, and (3) a multivariate analysis combining both penalties. For the evaluation, the `ToaD` models were trained with varying hyperparameters (grid-search). The maximum number of iterations ranges from 2^0 to 2^{10}, maximum depth per tree from 2^0 to 2^3, and ι and ξ from 2^{-10} to 2^{15}. Moreover, ι and ξ were set to 0 in every possible combination. This results in 32,076 models trained per dataset.

4.1 IMPLEMENTATION DETAILS

Our implementation builds upon the LightGBM framework.[4] The penalties were added as optional hyperparaneters to the training process of GBDT. The parameter ι is introduced as variable `tinygbdt_penalty_feature` and ξ as variable `tinygbdt_penalty_threshold`. Moreover, logging the used features and thresholds enables tracking of the memory consumed by the selected memory layout. Therefore, the optional variable `tinygbdt_forestsize` allows training models for a specific memory limitation (such as 32KB on an Arduino Uno Rev 3). Experiments were conducted on a collection of eight widely used publicly available datasets (see Appendix B for specifics). We split all datasets into training and test sets using an 80/20 ratio, respectively. Model fitting was conducted on the training set, and the test was used to measure the final induced quality of the models. As metrics for quality measurement of the resulting models, accuracy is used for classification datasets and the R^2 score for regression datasets (Lewis-Beck & Lewis-Beck, 2015). Note that, for both metrics, higher values indicate better model performance.

4.2 MODEL COMPARISON TO BASELINES

We compared the performance of our approach to that of other efficient GBDT (compression) methods. LightGBM is considered an established framework for training boosted trees (Ke et al., 2017). It was used for comparison in both the standard and quantized version. For quantization, the threshold and leaf values were reduced to 16-bit floating point precision. In addition LightGBM was evaluated in an array-based structure, i.e. it was stored without pointers but assuming all trees are complete as described in Section 3.2.1. This allows the comparison between `ToaD` and LightGBM under a unified pointer-less layout. Moreover, different pruning methods were evaluated with cost-efficient gradient boosting (CEGB) (Peter et al., 2017) and minimal cost-complexity pruning (CCP) (Breiman et al., 1984). All models were trained with the same hyperparameters as the `ToaD` models, i.e. all combinations of 2^0 to 2^{10} maximum trees and maximum tree depth between 2^0 and 2^3. Alongside related work (e.g., Buschjäger & Morik (2023)), we calculated the memory usage of a model

[4] All the source code and the experimental setup will be made publicly available upon acceptance. An anonymous code repository is already available at `https://anonymous.4open.science/r/ToaD/experiments/README.md`.

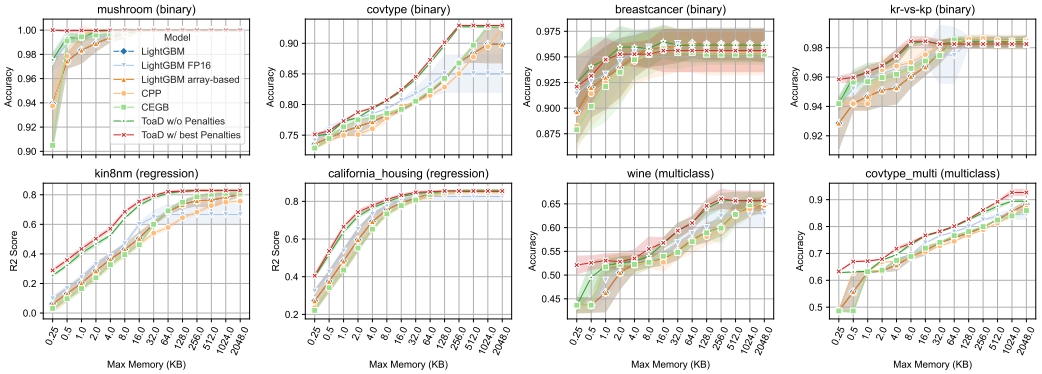

Figure 4: Accuracy vs. memory (KB) for ToaD and baselines. All models were trained on multiple train-test splits and the best model performance at a given memory limit from the hyperparameter analysis is depicted. The points show the mean across the splits, the errorbars show the respective standard deviation.

with 128 bits per node, assuming all values are stored in single precision (float32) and 64 bits for quantized half-precision models.[5] In constrast to Buschjäger & Morik (2023), we assume the information about a node being a leaf can be encoded by a specific feature and child node identifier, thus no additional boolean values are required. Moreover, boosted trees do not need to store the class information within a leaf but create one ensemble per class.

For the calculation of the memory footprint of the ToaD models, the proposed memory layout is used. For the ToaD models, it is distinguished between the memory layout without applying penalization during training, i.e., $\iota = 0$ and $\xi = 0$, and the best-performing models with penalization. For the comparison visualized in Figure 4, the best-performing models with a memory consumption less than or equal to the respective upper limit were chosen from the grid search results. Thus, the number or depth of trees of models at the same memory limit for the same dataset may differ for different model types. We expect ToaD to outperform the baseline implementations as it does not require pointers to its children, it encodes boolean values with only 2 bits, and a suitable penalty configuration encourages the model to reuse features and thresholds. Thus ToaD especially benefits from the pointer-less tree layout which is strong for complete trees, whereas a standard child-pointer layout also allows deeper non-complete trees without wasting too many resources.

4.2.1 Results Baseline Comparison

Since the primary use case for ToaD is microcontrollers, typical memory limits are considered for the respective performance comparison. The results are depicted in Figure 4. For almost all investigated configurations, ToaD outperforms the baseline approaches, even more when considering the models from the penalized training. On all tested multiclass datasets, the ToaD approaches are superior to the other models across all memory limits. The same holds for the regression datasets until the performance saturates, starting with different memory limits, with some of the competitors catching up to the same score with increasing memory available. In the interesting memory range up to 128 KB, before some model performances saturate, the competing methods need 4 to 16 times the memory to achieve the same performance. For example, on the covertype multiclass dataset, the best ToaD model at 2 KB achieves an accuracy of 69 %, which quantized LightGBM as the best competitor only matches with 8 KB, while float32 LightGBM even needs 16 KB. A further finding includes that ToaD outperforms array-based LightGBM, showcasing the effectiveness of our approach beyond the pure pointer-less layout.

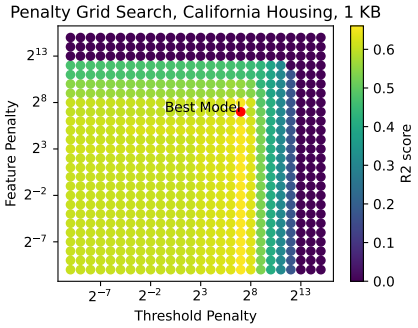

Figure 5: Model performance on California Housing with a 1 KB memory limit under varying penalties.

[5]Each node stores four values: one feature identifier, one threshold, and two child pointers.

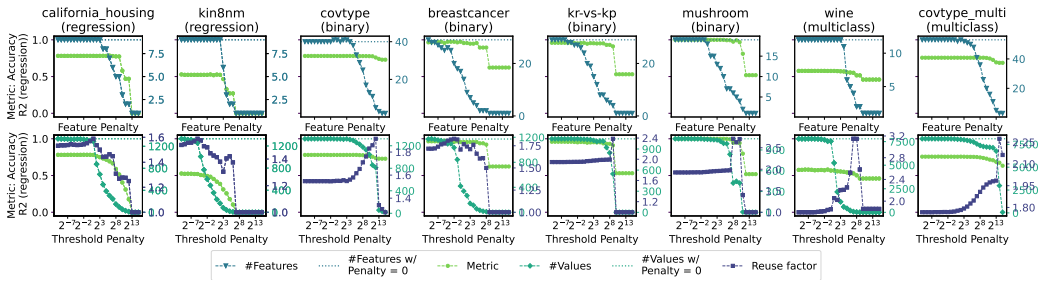

Figure 6: Influence of penalties ι and ξ on the number of thresholds and the number of features, displayed on the right y-axis, alongside the respective performance scores shown on the left y-axis. The threshold penalty figure additionally depicts the reuse factor. The maximum number of iterations per model is set to 256 during training with a maximum tree depth of 2.

Figure 5 shows an exemplary grid for model performances at a given memory limit for the California Housing dataset. The maximum memory size is fixed, allowing for an unlimited number of trees and nodes. As the `forestsize` parameter determining the memory limitation can be set by the user within our implementation, this graph can easily be generated for any memory size to determine the best penalty configuration. This approach helps identify the best-performing model for a given dataset on memory-limited hardware.[6]

4.3 Univariate Sensitivity Analysis

We conducted a univariate sensitivity analysis of the two newly introduced penalties to assess their individual effects on the model. During training, we varied the feature penalty ι and the threshold penalty ξ independently over the range 2^{-10} to 2^{15}, setting the other parameter to zero. We then tracked the number of nodes and leaves, the number of global values (thresholds and leaf values), and the test set performance. Furthermore, to evaluate how efficiently threshold values were reused, we calculated the reusing factor RF as the ratio between the sum of the nodes and leaves and the global number of values. In a naive implementation, the number of leaves and nodes equals the count of values, resulting in $RF = 1$. With the ratio RF, we can determine how many of these values are reused by the `ToaD` approach. For instance, a value of $RF = 1.5$ can be interpreted as a model reusing 50% of its threshold and leaf values, while $RF = 2.0$ indicates that, on average, each value is used twice. Results of this analysis are depicted in Figure 6 with a maximum number of iterations of 256 and a maximum depth per tree of 2, as `ToaD` is meant to be especially useful for shallow trees. A selection of results for further hyperparameter settings can be found in Appendix E.2.

4.3.1 Results Feature Penalty

Figure 6 (top) shows the univariate sensitivity analysis for the feature penalty ι. For $\iota < 1$, the number of features is largely unchanged, except for a notable drop in the Breast Cancer dataset. The value of the feature penalty at which it takes effect varies between different datasets. For datasets with few features (California Housing and kin8nm), the accuracy drops shortly after the number of features decreases. We assume the few features in this dataset are essential for accurate predictions. In contrast, datasets with more features show a slower and later accuracy decline, as the penalty first removes less relevant features. For example, the Covertype model loses only $\approx 2\%$ accuracy when $\iota = 2^{12}$, while the feature count drops from 35 to 5.

4.3.2 Results Threshold Penalty

In the bottom row of Figure 6, the performance metric for the model, the number of global values, and the reuse factor (RF) are depicted against a varying threshold penalty ξ. For all datasets, an increase in the threshold penalty decreases the number of global values used by the model. For the maximum penalty of $\xi = 2^{15}$, the number of global values approaches 1 for all models, i.e., the model only consists of one tree with the root node. The trend of the performance metrics differs between the datasets. The accuracy drops abruptly after a certain penalty for most of the binary datasets, whereas the performance of the other datasets declines more gradually. The models trained

[6] A script to reproduce the figures is available in the repository.

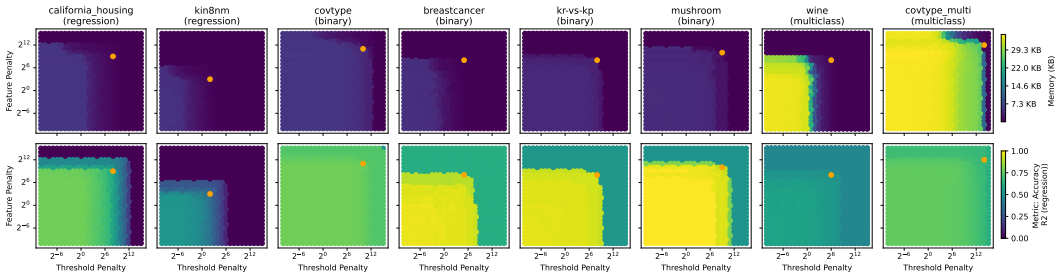

Figure 7: Influence of ι and ξ on the needed memory (KB) and the representative metric (accuracy or R^2 score). Orange dots mark penalty configuration combinations that are a good trade-off between model accuracy and memory. The maximum number of iterations per model is set to 256 during training with a maximum tree depth of 2.

on the Covertype dataset record the slightest decrease of about 6% from the smallest to the largest penalty value. At the same time, the number of values used for the model drops from 1323 to only 18. These differences probably arise from the size of the datasets, as the more stable models have more data points to choose from.

The reuse factor follows a similar pattern across all datasets. At first, it increases for higher penalties, but with very high penalties, it starts to decrease abruptly. At its peak, every model achieves a RF of at least 1.5 with the wine dataset, reusing each value more than three times for a penalty around 2^8. In contrast, for the highest investigated threshold penalty of $\xi = 2^{15}$, all models–except for the Covertype dataset–result in $RF = 1$, meaning each value is only used once per feature. This decrease in RF for very high ξ values can be explained by fewer global values being available for reuse, thus reducing reuse opportunities across training splits. Interestingly, the RF drops later than the number of values; actually, first it increases, and with the sharp decrease of the RF, also the accuracy plunges. Of special interest for the training process are the penalties with RF peaks, as here the model performance is still satisfactory, but the number of global thresholds is low.

4.4 Multivariate Sensitivity Analysis

To assess the combined effects of our penalties, we performed a multivariate sensitivity analysis for both penalty parameter values. As in the univariate sensitivity analysis, we used the parameter space 2^{-10} to 2^{15} for both parameters, resulting in $26 \times 26 = 676$ trained models per dataset and tree setting. Figure 7 shows the memory consumption next to the performance metric in a grid of the combined penalties with a maximum of 256 iterations with a maximum tree depth of 2. A selection of results for further hyperparamter settings can be found in Appendix E.3.

The memory requirements for all datasets decreased significantly as penalties increased, with each dataset having a specific threshold at which memory usage drops rapidly. The different looks of the multiclass dataset memories are reasoned by them using one tree ensemble per class; thus, more trees and memory are needed. For the larger datasets (Covertype, California Housing), the difference in memory consumption is stronger, starting at around 5 KB for small penalties and dropping to around 80 Byte for larger penalties. On average, predictive performance is better for smaller penalties as more features and thresholds are used. Independent of the dataset and objective, after a certain threshold in the feature penalty, model predictions are not better than guessing. This is expected as the penalties reduced tree complexity, thus it loses its predictive capabilities with the omitted features and thresholds.

By combining both penalty parameters, we can select a model that maintains high predictive performance while having a significantly smaller memory footprint. This creates solutions that are equally viable, known as nondominated solutions (Deb, 2011), where no solution is better in both predictive performance and memory usage simultaneously. If we know the minimal predictive performance we need for a task, we can select the corresponding microcontroller accordingly. Exemplary points are marked in orange where the accuracy is good, but the memory usage dropped.

To summarize, there is no globally optimal penalty setting as it depends on the dataset being used and the number and depth of trees allowed during training. For all datasets, we found that higher

memory usage does not necessitate an incline in predictive performance. Instead, the metrics remained similar until there was a considerable decrease in memory usage.

5 Conclusion & Future Work

We propose two hyperparameters and a new memory layout to optimize the memory footprint of boosted decision tree ensembles. First, custom penalties within boosted decision tree training were implemented that encourage a boosted tree to reuse features and thresholds and thus create smaller models. An univariate analysis has shown an effective decrease in the number of utilized values but almost unchanged performance for specific penalty values. Then, the decision tree memory layout `ToaD` was introduced. Building upon index-based trees that enable a pointer-less node sequence and global value lookup, it allows the reuse of threshold values multiple times per feature and allows storing thresholds with fewer bits, e.g. only 1 bit for boolean values. Our experiments show that we can store models with a significantly smaller memory footprint than baseline methods while maintaining the same accuracy, supporting the application of powerful boosted decision trees on resource-constrained devices.

Although the linear penalizer already performed well, a deeper analysis of more sophisticated penalizers may reveal even better performance and thus would prove to be an interesting extension to the present work. Adapting our method to reuse leaf values more effectively could also prove useful as well as the transfer to other variants of decision tree ensembles. Lastly, to further assess the effectiveness of the proposed method, its deployment to different microcontroller units would be valuable.

References

Fouad Alkhoury, Sebastian Buschjäger, and Pascal Welke. Splitting stump forests: tree ensemble compression for edge devices (extended version). *Machine Learning*, 114(10), August 2025. ISSN 1573-0565. doi: 10.1007/s10994-025-06866-2.

Majed Alsharari, Son T. Mai, Roger Woods, and Carlos Reaño. Efficient integer-only-inference of gradient boosting decision trees on low-power devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 72(1):241–253, January 2025. ISSN 1558-0806. doi: 10.1109/tcsi.2024.3446582.

Jock Blackard. Covertype. UCI Machine Learning Repository, 1998. URL `https://doi.org/10.24432/C50K5N`.

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

Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification And Regression Trees*. Chapman and Hall/CRC, 1 edition, 1984. ISBN 9781315139470. doi: 10.1201/9781315139470. URL `https://doi.org/10.1201/9781315139470`.

Sebastian Buschjäger and Katharina Morik. Joint leaf-refinement and ensemble pruning through l_1 regularization. *Data Mining and Knowledge Discovery*, 37(3):1230–1261, March 2023. ISSN 1573-756X. doi: 10.1007/s10618-023-00921-z.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 785–794, San Francisco California USA, August 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL `https://dl.acm.org/doi/10.1145/2939672.2939785`.

Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, November 2009. ISSN 0167-9236. doi: 10.1016/j.dss.2009.05.016. URL `https://doi.org/10.1016/j.dss.2009.05.016`.

Kalyanmoy Deb. *Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction*, pp. 3–34. Springer London, London, 2011. ISBN 978-0-85729-652-8. doi: 10.1007/978-0-85729-652-8_1. URL `https://doi.org/10.1007/978-0-85729-652-8_1`.

Laurens Devos, Wannes Meert, and Jesse Davis. Fast gradient boosting decision trees with bit-level data structures. In Ulf Brefeld, Elisa Fromont, Andreas Hotho, Arno Knobbe, Marloes Maathuis, and Céline Robardet (eds.), *Machine Learning and Knowledge Discovery in Databases*, pp. 590–606, Cham, 2020. Springer International Publishing. ISBN 978-3-030-46150-8. doi: 10.1007/978-3-030-46150-8_35. ECML PKDD 2019.

Laurens Devos, Timo Martens, Deniz Can, Wannes Meert, Hendrik Blockeel, and Jesse Davis. Compressing tree ensembles through level-wise optimization and pruning. In *Forty-second International Conference on Machine Learning*, 2025.

Youssouf Emine, Alexandre Forel, Idriss Malek, and Thibaut Vidal. Free lunch in the forest: Functionally-identical pruning of boosted tree ensembles. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(16):16488–16495, April 2025. ISSN 2159-5399. doi: 10.1609/aaai.v39i16.33811. URL https://doi.org/10.1609/aaai.v39i16.33811.

Zoubin Ghahramani. kin8nm. URL https://www.openml.org/search?type=data&id=189.

Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems*, volume 35, pp. 507–520. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/0378c7692da36807bdec87ab043cdadc-Paper-Datasets_and_Benchmarks.pdf.

Huaping Guo, Hongbing Liu, Ran Li, Changan Wu, Yibo Guo, and Mingliang Xu. Margin & diversity based ordering ensemble pruning. *Neurocomputing*, 275:237–246, 2018. ISSN 0925-2312. doi: 10.1016/j.neucom.2017.06.052.

Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. Protonn: Compressed and accurate knn for resource-scarce devices. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1331–1340. PMLR, 2017. URL https://proceedings.mlr.press/v70/gupta17a.html.

Albert Gural and Boris Murmann. Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications. In *Internatinal Conference on Machine Learning (ICML)*, pp. 2515–2524, 2019.

Jiawei Jiang, Bin Cui, Ce Zhang, and Fangcheng Fu. Dimboost: Boosting gradient boosting decision tree to higher dimensions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD/PODS '18, pp. 1363–1376. ACM, May 2018. doi: 10.1145/3183713.3196892. URL https://doi.org/10.1145/3183713.3196892.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems (NeurIPS)*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

R. Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997. ISSN 0167-7152. URL https://doi.org/10.1016/S0167-7152(96)00140-X.

Abhishek Khanna and Sanmeet Kaur. Internet of Things (IoT), Applications and Challenges: A Comprehensive Review. *Wireless Personal Communications*, 114(2):1687–1762, September 2020. ISSN 1572-834X. doi: 10.1007/s11277-020-07446-4. URL https://doi.org/10.1007/s11277-020-07446-4.

Simon Koschel, Sebastian Buschjäger, Claudio Lucchese, and Katharina Morik. Fast inference of tree ensembles on arm devices. 2023. doi: 10.48550/ARXIV.2305.08579. URL https://arxiv.org/abs/2305.08579.

Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 kb ram for the internet of things. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1935–1944. PMLR, August 2017. URL https://proceedings.mlr.press/v70/kumar17a.html.

Colin Lewis-Beck and Michael Lewis-Beck. *Applied regression: An introduction*, volume 22. Sage publications, 2015.

Brian Liu and Rahul Mazumder. Forestprune: Compact depth-pruned tree ensembles. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent (eds.), *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pp. 9417–9428. PMLR, 25–27 Apr 2023. URL https://proceedings.mlr.press/v206/liu23h.html.

Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Quickscorer: Efficient traversal of large ensembles of decision trees. In *Machine Learning and Knowledge Discovery in Databases*, pp. 383–387, Cham, 2017. Springer International Publishing. ISBN 978-3-319-71273-4. doi: 10.1007/978-3-319-71273-4_36. URL https://doi.org/10.1007/978-3-319-71273-4_36.

Philipp Mayer, Michele Magno, Thomas Brunner, and Luca Benini. Lora vs. lora: In-field evaluation and comparison for long-lifetime sensor nodes. In *International Workshop on Advances in Sensors and Interfaces (IWASI)*, pp. 307–311. IEEE, 2019.

Mushroom. UCI Machine Learning Repository, 1981. URL https://doi.org/10.24432/C5959T.

Feng Nan, Joseph Wang, and Venkatesh Saligrama. Pruning random forests for prediction on a budget. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/3948ead63a9f2944218de038d8934305-Paper.pdf.

Mike O. Ojo, Stefano Giordano, Gregorio Procissi, and Ilias N. Seitanidis. A Review of Low-End, Middle-End, and High-End Iot Devices. *IEEE Access*, 6:70528–70554, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2879615. URL https://doi.org/10.1109/ACCESS.2018.2879615.

Sven Peter, Ferran Diego, Fred A Hamprecht, and Boaz Nadler. Cost efficient gradient boosting. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/4fac9ba115140ac4f1c22da82aa0bc7f-Paper.pdf.

Natalia Ponomareva, Thomas Colthurst, Gilbert Hendry, Salem Haykal, and Soroush Radpour. Compact multi-class boosted trees. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 47–56, 2017. doi: 10.1109/BigData.2017.8257910. URL https://doi.org/10.1109/BigData.2017.8257910.

Vijay Janapa Reddi, Brian Plancher, Susan Kennedy, Laurence Moroney, Pete Warden, Anant Agarwal, Colby R. Banbury, Massimo Banzi, Matthew Bennett, Benjamin Brown, Sharad Chitlangia, Radhika Ghosal, Sarah Grafman, Rupert Jaeger, Srivatsan Krishnan, Maximilian Lam, Daniel Leiker, Cara Mann, Mark Mazumder, Dominic Pajak, Dhilan Ramaprasad, J. Evan Smith, Matthew Stewart, and Dustin Tingley. Widening access to applied machine learning with tinyml. *CoRR*, abs/2106.04008, 2021. URL https://arxiv.org/abs/2106.04008.

Shaoqing Ren, Xudong Cao, Yichen Wei, and Jian Sun. Global refinement of random forest. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

Alen Shapiro. Chess (King-Rook vs. King-Pawn). UCI Machine Learning Repository, 1983. URL https://doi.org/10.24432/C5DK5C.

Yu Shi, Guolin Ke, Zhuoming Chen, Shuxin Zheng, and Tie-Yan Liu. Quantized training of gradient boosting decision trees. In *Advances in neural information processing systems (NeurIPS)*, volume 35, pp. 18822–18833. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/77911ed9e6e864ca1a3d165b2c3cb258-Paper-Conference.pdf.

Hongfei Wang, Zhanfei Wu, Xiangwei Wang, Longyun Bian, and Hai Jin. Hardgbm: A framework for accurate and hardware-efficient gradient boosting machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(7):2122–2135, 2023. doi: 10.1109/TCAD.2022.3218509.

Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with Tensorflow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Inc., 2019.

William Wolberg, Olvi Mangasarian, and Nick Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1993. URL https://doi.org/10.24432/C5DW2B.

Ting Ye, Hucheng Zhou, Will Y. Zou, Bin Gao, and Ruofei Zhang. Rapidscorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 941–950, London United Kingdom, July 2018. ACM. ISBN 978-1-4503-5552-0. doi: 10.1145/3219819.3219857. URL https://dl.acm.org/doi/10.1145/3219819.3219857.

A TRAINING COMPRESSED ENSEMBLES

Constructing an optimal tree ensemble T w.r.t. Equation (1) is computationally infeasible. Therefore, one resorts to K boosting rounds. In each boosting round m, one considers $T_{m-1} \coloneqq \sum_{k=1}^{m-1} t_k$ together with a new tree t_m that is built in this round (starting with $T_0 \coloneqq 0$) (Chen & Guestrin, 2016). The new tree is chosen such that

$$\sum_{i=1}^{n} \mathcal{L}\left(y_i, T_{m-1}(\mathbf{x}_i) + t_m(\mathbf{x}_i)\right) + \sum_{k=1}^{m} \Omega(t_k) \tag{4}$$

is minimized. A common choice for the regularizer Ω is

$$\Omega(t_m) = \gamma L + \frac{1}{2}\lambda \sum_{j=1}^{L} v_j^2,$$

where L is the number of leaves and $v_1, \ldots, v_L \in \mathbb{R}$ are the associated leaf values of t_m. This penalizes trees with many leaves and large absolute leaf values. The modified regularizer

$$\Omega_l(t_m) = \Omega(t_m) + \iota \cdot |F_U| + \xi \cdot \sum_{f \in F_U} |\mathcal{T}^f|, \tag{5}$$

proposed in Section 3, with user-defined hyperparameters $\iota, \xi \in \mathbb{R}^+$, penalizes the use of new features and thresholds in a linear manner. Similar to Peter et al. (2017), incorporating Ω_l into the objective leads to a modified gain compared to standard boosted decision trees (Chen & Guestrin, 2016). More precisely, using gradient statistics of the form

$$\begin{aligned} g_i &\coloneqq \frac{\partial}{\partial z}\mathcal{L}(y_i, z)\Big|_{z=T_{m-1}(\mathbf{x}_i)} \\ h_i &\coloneqq \frac{\partial^2}{\partial^2 z}\mathcal{L}(y_i, z)\Big|_{z=T_{m-1}(\mathbf{x}_i)} \end{aligned}$$

and omitting constant terms, one obtains the simplified objective

$$\sum_{j=1}^{L} \left[G_{I_j} v_j + \frac{1}{2}\left(H_{I_j} + \lambda\right) v_j^2 \right] + \gamma L + \iota |F_U| + \xi \cdot \sum_{f \in F_U} |\mathcal{T}^f| \tag{6}$$

with $G_S \coloneqq \sum_{i \in S} g_i$ and $H_S \coloneqq \sum_{i \in S} h_i$. Here, I_j denotes the set of training indices assigned to the j-th leaf of the current tree t_m, i.e., $I_j = \{i \mid q(\mathbf{x}_i) = j\}$, where $q : \mathbb{R}^d \to \{1, \ldots, L\}$ maps an input instance to the corresponding leaf of t_m (i.e., $t_m(\mathbf{x}) = v_{q(\mathbf{x})}$).

The decision tree t_m is constructed greedily by iteratively splitting leaves (starting at the root) if this improves Objective (6); otherwise, the construction stops. More specifically, given a leaf associated with a set I of training indices, a split on feature $i \in \{1, \ldots, d\}$ with threshold μ divides I into two subsets, I_L (left leaf) and I_R (right leaf). The corresponding gain is then given by

$$\begin{aligned} \Delta_l(I, i, \mu) &\coloneqq \frac{1}{2}\left(\frac{G_{I_L}^2}{H_{I_L} + \lambda} + \frac{G_{I_R}^2}{H_{I_R} + \lambda} - \frac{(G_I)^2}{H_I + \lambda} \right) - \gamma - s_f \iota - s_t \xi, & (7) \\ &= \Delta(I, i, \mu) - s_f \iota - s_t \xi, & (8) \end{aligned}$$

where $s_f = 1$ if a new feature is used (and $s_f = 0$ otherwise), and $s_t = 1$ if a new threshold is used (and $s_t = 0$ otherwise). Thus, the modified regularizer (5) introduces the additional terms $-s_f \iota$ and $-s_t \xi$, corresponding to the cost of using a new feature or threshold, respectively.

B DATASETS

An overview of the eight datasets used for experiments in Section 4 along with their statistics are presented in Table 1. The binary Covertype (`covtype`) dataset (Blackard, 1998), the California Housing regression dataset (Kelley Pace & Barry, 1997), the KRKPA7 (`kr-vs-kp`) dataset (Shapiro, 1983) and the Breast Cancer dataset (Wolberg et al., 1993) are commonly solved using boosted decision trees. The kin8nm dataset (Ghahramani), which describes robotics decision making, and the Mushroom dataset (Mushroom, 1981), which is for mushroom edibility classification, cover data that might be of interest for analysis on constrained edge devices.

Table 1: Datasets

Dataset	***Instances***	***Features***	***Task***
Covertype (Blackard, 1998)	581,012	54	Binary & multiclass classification
California Housing (Kelley Pace & Barry, 1997)	20,640	8	Regression
kin8nm (Ghahramani) [7]	8,192	8	Regression
Mushroom (Mushroom, 1981)	8,124	22	Binary classification
Wine Quality (Cortez et al., 2009)	6,497	11	Multiclass classification
KRKPA7 (Shapiro, 1983)	3,196	36	Binary classification
Breast Cancer Wisconsin (diagnostic) (Wolberg et al., 1993)	569	30	Binary classification

C EXTENDED RELATED WORK

Decision trees and decision tree ensembles, such as GBDT, have remained very popular, despite the rise of deep learning models (Grinsztajn et al., 2022). These tree-based methods are highly valued for their interpretability, particularly when dealing with small datasets, mixed data types, and constrained computational resources. Recent advances in tree-based models have also introduced several approaches to enhance their efficiency. For instance, cost-efficient gradient boosting introduces mechanisms to account for feature acquisition and tree evaluation costs (Peter et al., 2017), which involves penalizing the use of new features based on their cost and minimizing the number of split nodes an input traverses during inference. Other work (Ponomareva et al., 2017) presents a method to train compact boosted tree ensembles for multi-class classification using vector-valued trees and layer-by-layer boosting. Bonsai (Kumar et al., 2017), instead, yields a single decision tree model and reduces the model size by projecting the input data into a low-dimensional subspace. Two other approaches are QuickScorer (Lucchese et al., 2017) and its enhanced version, RapidScorer (Ye et al., 2018), decision tree ensembles both developed for search engines, where array structures are used to reduce model size. These methods prioritize enhancing processing speed over minimizing memory footprint by summarizing features and split values for joint calculation. Recently, Koschel et al. (2023) further adapted these approaches to IoT contexts, focusing on ARM CPUs prevalent in such devices. Their work involves adapting QuickScorer variants for these CPUs and applying fixed-point quantization to split nodes and leaf values, emphasizing computational enhancements over memory optimization. DimBoost (Jiang et al., 2018) made use of lower precision values during the calculation of gradient histograms in training as one of their contributions to improve the performance of GBDT for high-dimensional data. Additionally, Shi et al. (2022) demonstrated that using just two or three bits suffices to represent gradients in GBDT training. Devos et al. (2020) represent input data as well as gradients in bit-level data structures to improve the runtime of GBDT training.

The related works sketched above aim mostly at enhancing the training process, leading to improvements in memory and energy consumption as ancillary benefits. The notable exception are the works by Kumar et al. (2017) and Ponomareva et al. (2017), which explicitly address a memory footprint reduction in the context of resource-constrained devices, achieving compact and effective tree-based models. However, the approach of Kumar et al. (2017) only considers single decision trees and conceptually modifies the tree building process and the underlying models. Ponomareva et al. (2017)

[7] `https://www.cs.toronto.edu/~delve/data/kin/desc.html`

focus on multi-class classification problems. However, the vector-values used within trees may increase the memory footprint of tree models for binary classification or for low-number multi-class problems. Additionally, an evaluation of memory consumption is not conducted directly, but rather the number of trees is used as an estimate. Specific memory consumption of different Machine Learning (ML) models and compression techniques is considered by Buschjäger & Morik (2023) and (Devos et al., 2025). Their works combine ensemble pruning and update of leaves values as post-training compression methods to reduce the size of random forests and GBDTs respectively. Lossless post-training pruning of boosted ensembles is proposed by Emine et al. (2025). Liu & Mazumder (2023) on the other hand focuses on pruning deep layers of trees to tackle growing memory demands because of exponential increase in the number of nodes for deeper layers. There are many other works optimizing non-decision tree based methods for deployment on resource-constrained devices, for example based on k-nearest neighbors Gupta et al. (2017). In this work, we focus on gradient boosted trees, and at minimizing the memory footprint of the overall ensemble in the course of the training process. We extend the work of Peter et al. (2017) by defining feature costs that aim at reducing the bits required to encode feature indices. We also introduce a special cost for adding new thresholds to each feature and provide a specialized encoding for the induced binary trees, including global threshold arrays that are shared across all individual learners.

D Comparison to Random Forest

`ToaD` as a compression method developed for GBDT is compared to a baseline random forest (RF) method for classification tasks as well as version pruned along Guo et al. (2018) as shown in Figure 8. Only classification results are included as most pruning methods are not implemented for regression tasks. The results show an advantage of RF in multiclass classification. This is reasoned by boosted trees being usually trained with one ensemble per class whereas RFs store the class information in the nodes. An extension of our work in the future could include optimizing `ToaD` for multiclass classification, for example based on the approach of Ponomareva et al. (2017).

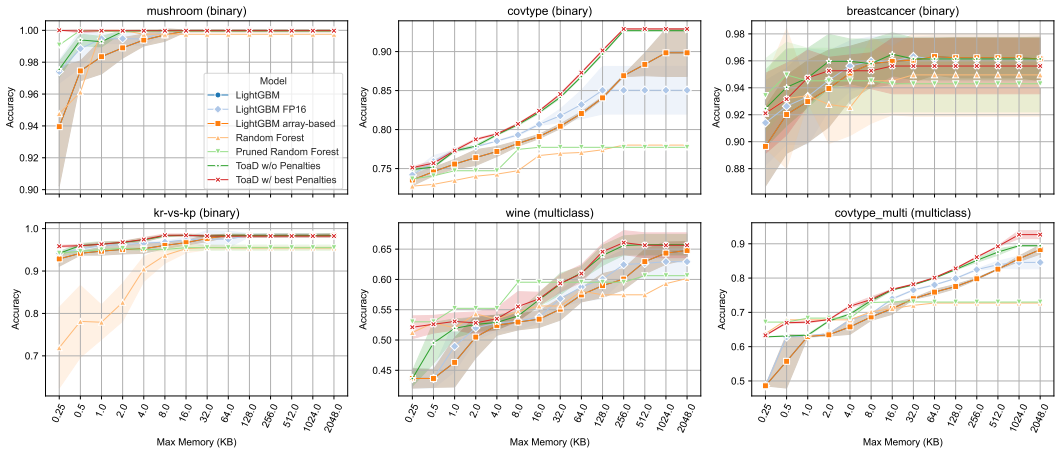

Figure 8: Comparison of baseline LightGBM method and `ToaD` to baseline random forests and random forests pruned by the method presented by Guo et al. (2018). Best models at each memory threshold are depicted. Models were trained on multiple train-test data splits.

E EXPERIMENTS

E.1 RUNTIME EXPERIMENTS

For an estimate for the expected runtime, we deployed a `ToaD`-model processing the covtype binary dataset at a memory limit of 0.5 KB. The model consists of four complete trees with a depth of four. We measured 20 runs, with each run having 500 predictions. We decided to stop at 20 runs as the variance between the runs was minimal. Experiments were conducted on the Seeed Xiao ESP32-S3 and the Arduino Nano 33 BLE Rev2. To ensure that programs are not further optimized, we used random numbers as input. The prototypes are available at the anonymous repository[8]. Note that currently the `ToaD`-program is only a first prototype and there are many options for optimization. A single prediction took 0.51 milliseconds for the `ToaD`-model on the Arduino Nano. In contrast to the LightGBM program, this is a slowdown by a factor of ~ 5. For the ESP32-S3, a single prediction took 0.14 milliseconds, with a respective slowdown in contrast to LightGBM of a factor of ~ 8. Those findings should be read as an outlook for future work to incorporate optimization techniques to close the runtime gap. As already stressed in the main paper, the observed latency degradation is not a significant factor in real-world deployments. At below millisecond inference time for `ToaD`, the overall latency and energy consumption of real-world use cases is dominated by the device measuring the input data and potentially transmitting the results.

Hardware	Average Prediction Runtime (µs)	
	`ToaD`	LightGBM
XIAO ESP32S3	137.08	17.63
Arduino Nano 33 BLE	512.89	102.16

Table 2: Runtime for one inference run for the covtype binary dataset with four trees, each with a depth of four.

E.2 UNIVARIATE SENSITIVITY ANALYSIS

Influence of feature penalty ι and threshold penalty ξ on the number of thresholds and the number of features, for different hyperparamter settings are displayed in the following figures. The respective top row shows the influence of ι on the number of features and the performance when $\xi = 0$. The respective top row shows the influence of ξ on the number of thresholds and the performance when $\iota = 0$. Additionally the reuse factor RF is displayed as described in Section 4.3.

We can observe similar patterns in the evolution of used features and thresholds and the corresponding performance as described in Section 4.3.

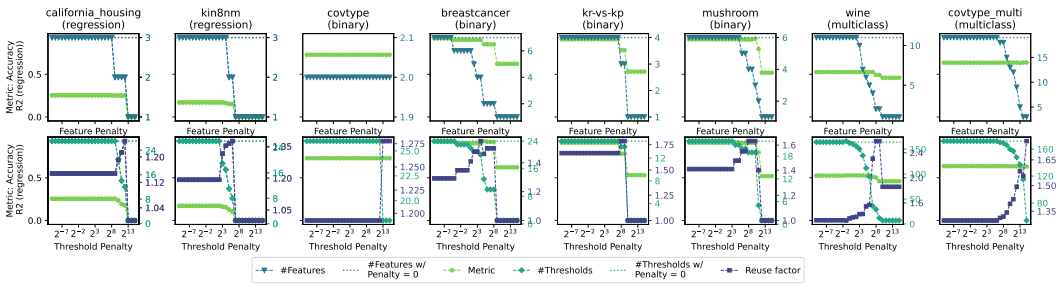

Figure 9: $max\_iterations = 4$, $max\_depth = 2$

[8] `https://anonymous.4open.science/r/ToaD/experiments/latency/README.md`

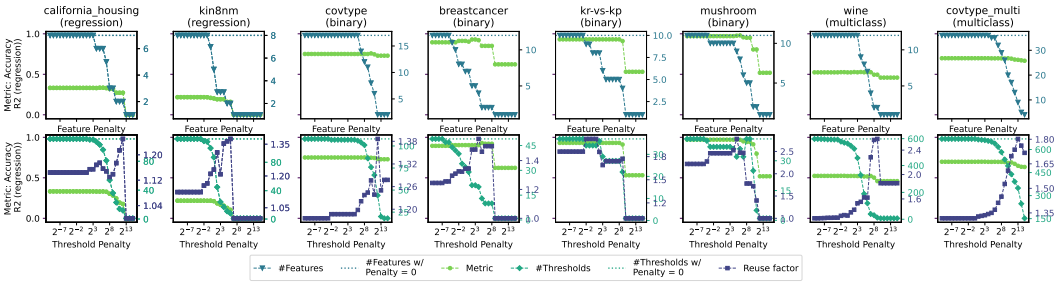

Figure 10: $max\_iterations = 4$, $max\_depth = 4$

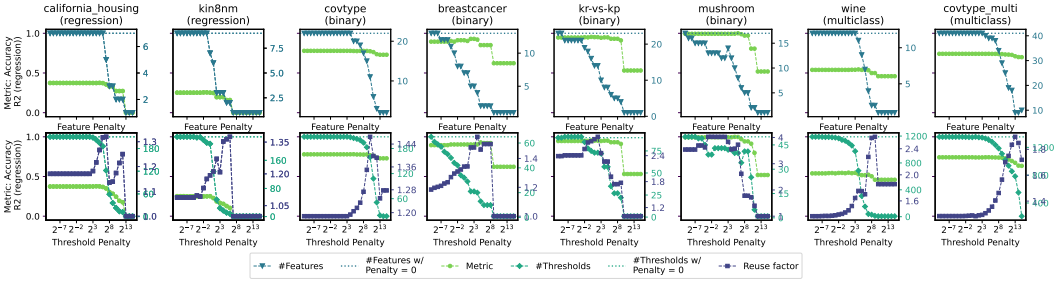

Figure 11: $max\_iterations = 4$, $max\_depth = 8$

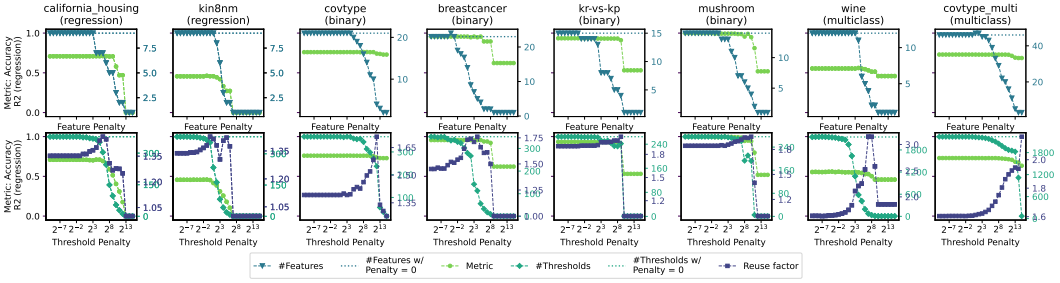

Figure 12: $max\_iterations = 64$, $max\_depth = 2$

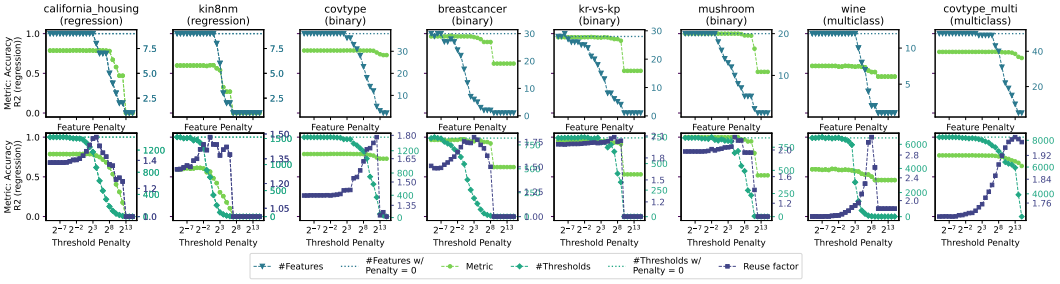

Figure 13: $max\_iterations = 64$, $max\_depth = 4$

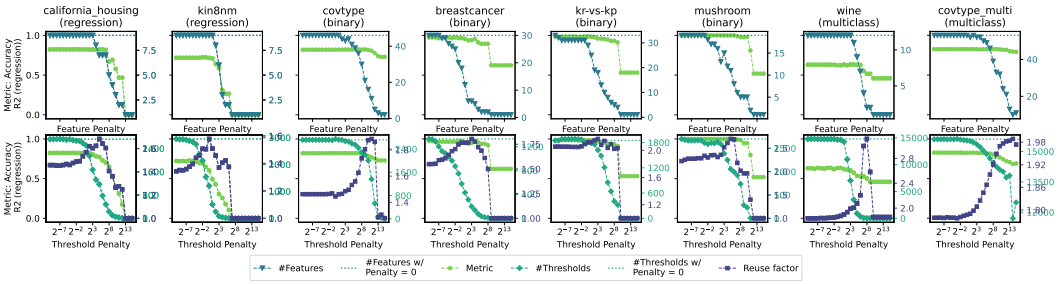

Figure 14: $max\_iterations = 64$, $max\_depth = 8$

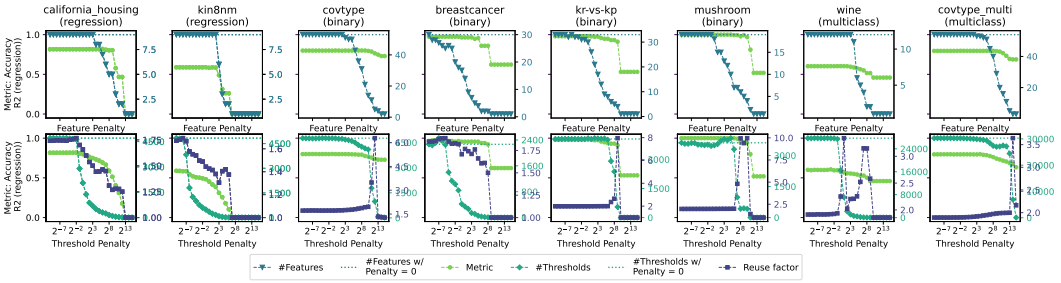

Figure 15: $max\_iterations = 1024$, $max\_depth = 2$

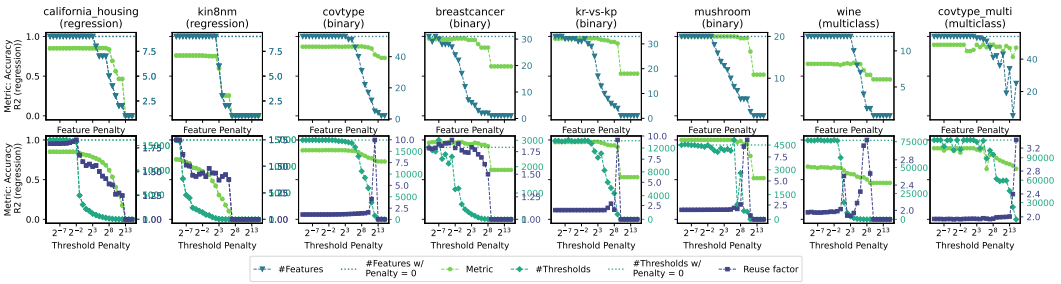

Figure 16: $max\_iterations = 1024$, $max\_depth = 4$

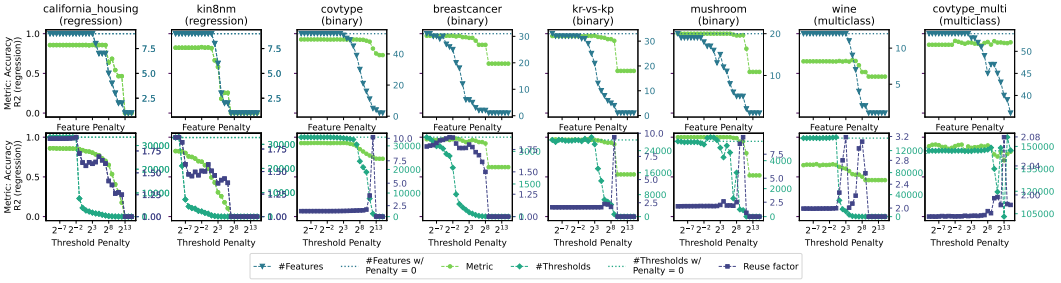

Figure 17: $max\_iterations = 1024$, $max\_depth = 8$

E.3 Multivariate Sensitivity Analysis

In the following figures the influence of ι and ξ on the needed memory (KB) on the top row and the representative metric (accuracy or R^2 score) on the bottom row are depicted for different hyperparameter settings.

Similarly as depicted in Figure 7 and described in Section 4.4 we can find useful penalty combinations that provide a useful trade-off between a small decrease in performance but a significant decrease in memory consumption for most of the given examples.

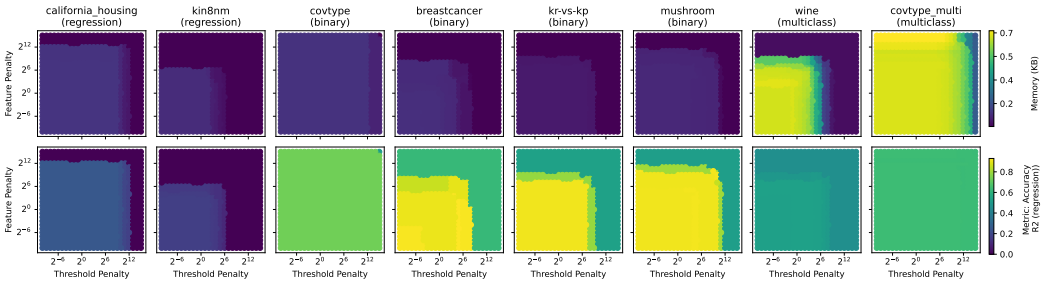

Figure 18: $max\_iterations = 4$, $max\_depth = 2$

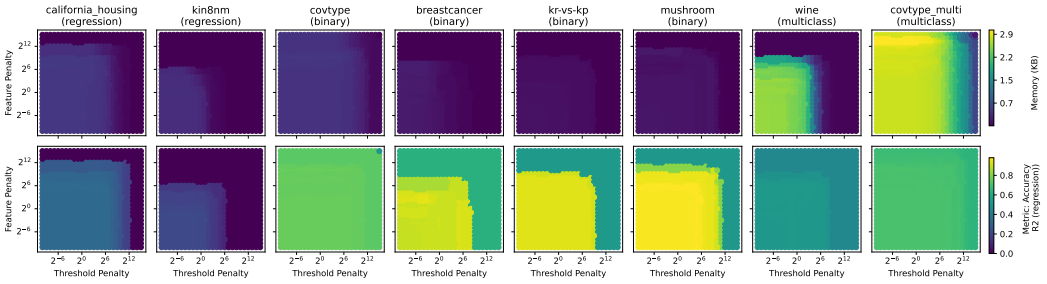

Figure 19: $max\_iterations = 4$, $max\_depth = 4$

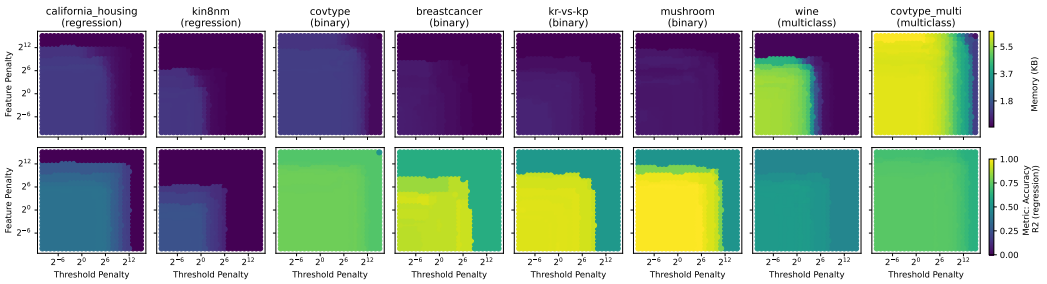

Figure 20: $max\_iterations = 4$, $max\_depth = 8$

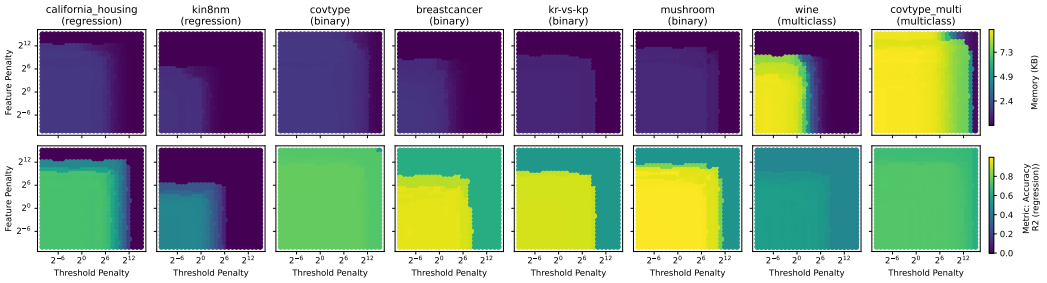

Figure 21: $max\_iterations = 64$, $max\_depth = 2$

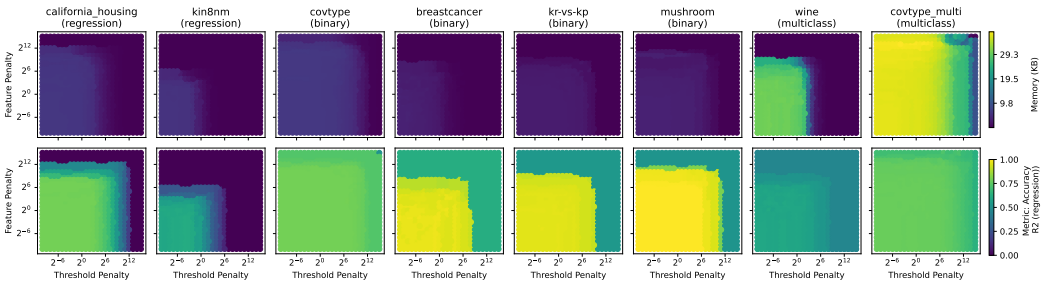

Figure 22: $max\_iterations = 64$, $max\_depth = 4$

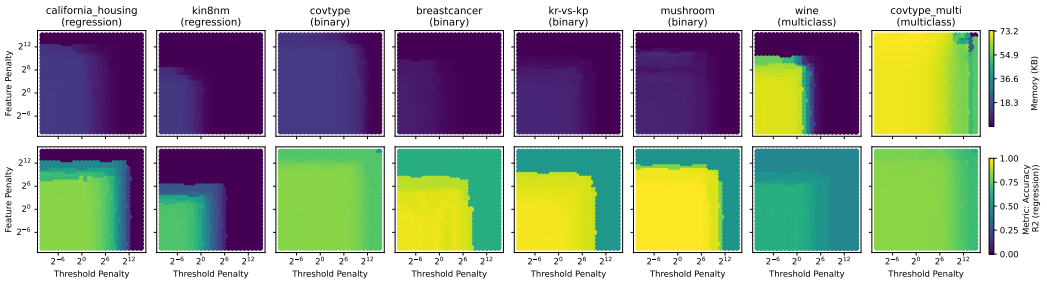

Figure 23: $max\_iterations = 64$, $max\_depth = 8$

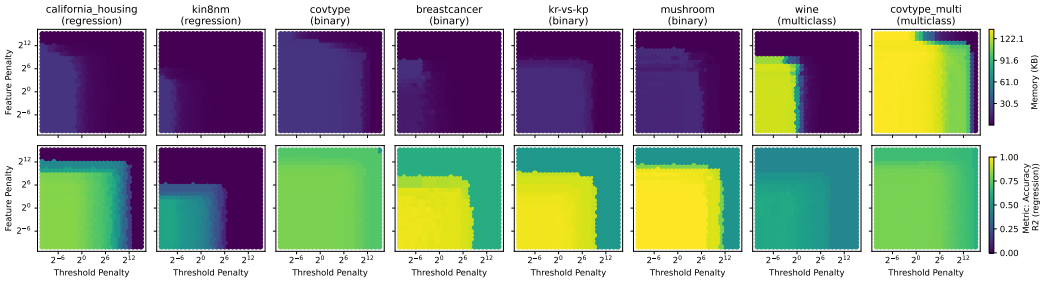

Figure 24: $max\_iterations = 1024$, $max\_depth = 2$

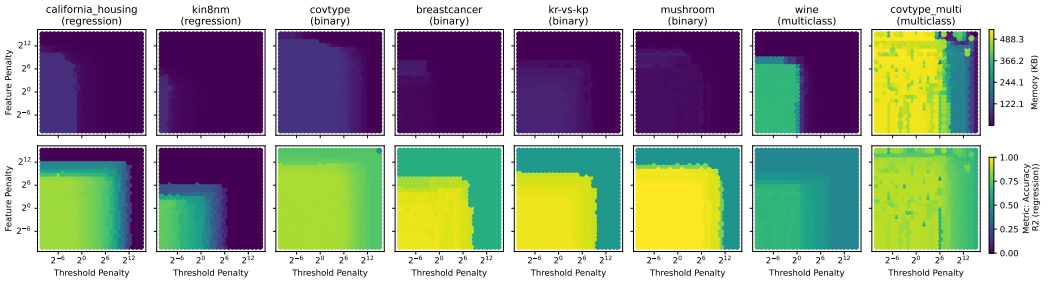

Figure 25: $max\_iterations = 1024$, $max\_depth = 4$

F Large Language Model Usage

This manuscript has undergone sentence-level improvements using Large Language Models (LLMs) to enhance clarity and readability. However, all scientific ideas, methods, results and conclusions are the exclusive work of the authors.