Realization of Random Forest for Real-Time Evaluation through Tree Framing

Sebastian Buschjäger*‡, Kuan-Hsun Chen†‡, Jian-Jia Chen† and Katharina Morik*
*Artificial Intelligence Unit, TU Dortmund University
†Design Automation For Embedded Systems Group, TU Dortmund University
{sebastian.buschjaeger, kuan-hsun.chen, jian-jia.chen, katharina.morik}@tu-dortmund.de
‡ Both authors contributed equally

***Abstract*—The optimization of learning has always been of particular concern for big data analytics. However, the ongoing integration of machine learning models into everyday life also demand the *evaluation* to be extremely fast and in real-time. Moreover, in the Internet of Things, the computing facilities that run the learned model are restricted. Hence, the implementation of the model application must take the characteristics of the executing platform into account Although there exist some heuristics that optimize the code, principled approaches for fast execution of learned models are rare. In this paper, we introduce a method that optimizes the execution of Decision Trees (DT). Decision Trees form the basis of many ensemble methods, such as Random Forests (RF) or Extremely Randomized Trees (ET). For these methods to work best, trees should be as large as possible. This challenges the data and the instruction cache of modern CPUs and thus demand a more careful memory layout. Based on a probabilistic view of decision tree execution, we optimize the two most common implementation schemes of decision trees. We discuss the advantages and disadvantages of both implementations and present a theoretically well-founded memory layout which maximizes locality during execution in both cases. The method is applied to three computer architectures, namely ARM (RISC), PPC (Extended RISC) and Intel (CISC) and is automatically adopted to the specific architecture by a code generator. We perform over 1800 experiments on several real-world data sets and report an average speed-up of 2 to 4 across all three architectures by using the proposed memory layout. Moreover, we find that our implementation outperforms `sklearn`, which was used to train the models by a factor of 1500.**

***Index Terms*—random forest, decision trees, caching, computer architecture**

I. Introduction

Data collection has become ubiquitous in the Internet of Things, where sensors and restricted computing facilities are embedded into various physical objects [1]. A plentitude of such devices gathers data. Since the devices are most often restricted in their energy, communication and computation resources, data is transmitted to a larger computer or a computing cloud for analysis. The learned model is then sent back to the local device which applies it in order to deliver a certain service such as, e.g., monitoring. In general, for detecting a particular event, the stream of sensor measurements is classified. This requires the learned model to be efficiently executed at a resource-restricted device in real-time. Where efficient learning has always been in the focus of research, the demand to investigate the *efficient application of learned models* has emerged only recently.

In general, tree ensembles are often referred to as one of the best black-box methods available, because they offer high accuracy with only a few parameters to tune [2], [3]. They are among the most efficient methods available for model training, but are not considered to be fast application-wise. Recently, applications of DTs for information retrieval in real time have pushed the issue of efficient traversal of ensembles of DTs forward [4]. We take that approach further by explicitly taking hardware architecture into account.

Where machine learning research has focused for a long time on purely algorithmic properties, the borderline between implementational details and algorithmic contributions has become blurred. In an extensive study of unsupervised methods, the impact of particular implementations, frameworks, programming languages and libraries on the runtime performance has been shown [5]. Particularly for runtime considerations, it has been stated that caching behaviour determines the performance of implemented algorithms even more than algorithmic differences [6]. Taking into account the hardware architecture is discussed in blogs[1], but formal analysis and empirical investigations are rare. A principled account of *memory layout* for the real-time evaluation of DTs is still missing.

Real-time application of tree ensembles such as Random Forests (RF) has become important in many domains, e.g., real-time classification of celestial objects in astrophysics [7], real-time pedestrian detection [8], real-time 3D face analysis [9], real-time classification of noise signals [10], nanopartical sensors [11], etc. Hence, optimization for efficient execution of the learned models becomes important.

Our Contributions: Instead of applying ad-hoc optimizations and application specific enhancements to the execution of tree ensembles, we aim for offering a principled approach. A code generator should automatically adapt to particular parameters of the computer's memory and produce optimized code segments that take the instruction and/or data cache into consideration. Thus,we make the following contributions:

- **Architecture and model awareness:** We introduce a probabilistic view of DT model execution in CPUs.

[1]see, e.g., https://code.facebook.com/posts/975025089299409/evaluating-boosted-decision-trees-for-billions-of-users/

- **Model-aware memory layout:** We introduce a theoretically well-founded memory layout for DTs maximizing locality during execution based on the probabilistic view.
- **Code generator:** We present a code-generator, which exploits the theoretical insights for generating fast implementations of a given tree ensemble.
- **Empirical evaluation:** We perform a total of 1800 experiments across 3 different computer architectures and show that our implementation offers a speed-up factor from $2-4$ on average without changing the prediction accuracy of the model.

The rest of the paper is organized as the following. We recapture tree ensembles and present a probabilistic view of DT execution in Section II. Then, we introduce the locality concept of memory hierarchies (Section III). Section IV discusses related work. We propose an implementation that makes good use of cache memories in Section V. Section VI presents experiments on 12 data-sets across three different computer architectures for different tree sizes. Section VII concludes the paper.

II. Decision Trees and Random Forests

We consider supervised learning problems, in which we infer a model $\widehat{f} : \mathbb{R}^d \rightarrow \mathcal{Y}$ from labelled training data $\{(\vec{x}_i, y_i) | i = 1, \dots, N\}$ to predict the value $\widehat{f}(\vec{x})$ of new, unseen observations. For $\mathcal{Y} = \mathbb{R}$ we have a regression problem, for $\mathcal{Y} = \{0, 1, \dots\}$ we have a classification problem. Tree ensembles train a set of individual trees and combine their predictions to establish a joint model. In the classical Random Forest (RF) approach by Breiman [12], a set of K DTs are trained using different samples of input features. In the literature, other RFs variations have been explored, such as those that train trees on samples of data (bagging) [13] or those that randomly generate trees without training at all [14].

It is common to all these methods, that they use tree-structured predictors as base learners and that they inject some form of randomness into the training. In the theoretical analysis of these methods, we often encounter the fact, that base learners should be as large as possible:

Breiman has shown that bagging in general, and Random Forest specifically, reduce the variance of a biased learner [15]. Thus, for optimal performance, individual trees should minimize the bias error, which implies that they should not be restricted in size. In [16], Breiman extended his formal argument by empirical support. More recent theoretical analysis of RFs such as [17]–[20] consistently support Breimans original claim, that trees should be as large as possible. In short, recommendations for the optimal tree height range between $\mathcal{O}(\log N)$ and $\mathcal{O}(N)$, both, from a theoretical and empirical perspective. This makes RF fundamentally different from other ensemble learners such as Boosting [21], where the size of individual base learners are restricted to reduce over-fitting. When we wish to apply RFs, we need to keep in mind, that individual trees are usually large.

A. A probabilistic view of DT execution

Our goal is to analyze the probability to perform a certain comparison while traversing a DT. Based on this analysis, we can decide for each tree, which implementation and which data layout is the best. Our notation is the following: Each node receives a unique identifier (e.g. in breath-first order) i. We denote the left child of i with $l(i)$ and the right child with $r(i)$. Let M denote the number of leaves in a DT. Since each leaf has a unique path from the root of the DT, there are M different paths from the root node to the leaves. Every node in the tree stores information about the feature, as well as a split-value against which the feature is compared to (split-point). Additionally, every leaf node stores its associated prediction (i.e., 0 or 1).

To classify a sample $\vec{x}$, we begin to traverse the tree starting by its root node and follow the children according to the comparisons at each node until we hit a leaf node. Then, we return the associated prediction value of the leaf node. Every observation takes exactly one path $\pi(\vec{x})$ from the root node to one leaf. To lighten the notation, we drop the argument $\vec{x}$, if we are not interested in the path of a specific observation.

Following the probabilistic view of DT execution in [22], we model each comparison at node i as a Bernoulli experiment in which we will take the path towards the left child with probability $p(i \rightarrow l(i))$ and respectively for the right child with $p(i \rightarrow r(i))$. It holds that $p(i \rightarrow l(i)) = 1 - p(i \rightarrow r(i))$. An example can be found in Figure 1. Note, that the probabilities $p(i \rightarrow l(i))$ and $p(i \rightarrow r(i))$ can be estimated during training by counting the number of samples at each node i taking the left and right path. Assume a path of length L with $\pi = (i_1, i_2, \dots, i_L)$, where i_{j+1} is either the left or the right child of the j^{th} node on the path. Then, following this path consists of a series of Bernoulli experiments each with probability $p(i_j \rightarrow i_{j+1})$. Let $\mathcal{P}$ denote the set of all paths in the tree. Then the probability to take path $\pi \in \mathcal{P}$ is given by

$$p(\pi) = p(i_0 \rightarrow i_1) \cdot \ldots \cdot p(i_{L-1} \rightarrow i_L) = \prod_{j=0}^{L} p(i_j \rightarrow i_{j+1})$$

Again, let i be a node, then there is exactly one path $\pi = (0, \dots, i)$ ending in node i. We call the probability of the path leading to node i the probability of that node,that is $p(i) = p((0, \dots, i))$. Let $\mathcal{T}$ be the set of all nodes in the tree, then we define the probability for every subset of nodes $T \subseteq \mathcal{T}$ as:

$$p(T) = \sum_{i \in T} p(i)$$

B. Problem Definition

In this paper, we consider the performance optimization for executing a given ensemble model, e.g. a Random Forest with the probabilistic information of its DTs. We automatically exploit the parameters of the computer's memory and produce optimized code segments that take the instruction and/or data cache into consideration. We assume the quality of the learned model to be satisfactory and do not change it.

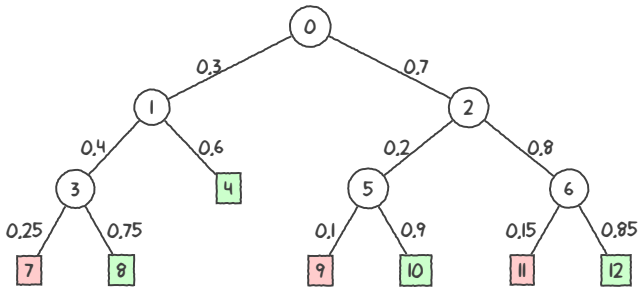

Fig. 1: Binary DT with depth 4. Inner nodes are depicted with circles, leaf nodes are displayed as rectangles. Green nodes indicate a positive and red nodes a negative class. Each node has a unique id and every path is associated with a probability. For example, the probability to go from the root node 0 to its right child 2 is $p(0 \to 2) = 0.7$.

The ensemble is given in an XML or JSON format encoding the trees. Our performance metric is the *runtime* of the resulting implementation of the given forest. Programming a decision tree is a simple task in most programming languages. Take a binary decision tree in Figure 1 as an example. When we execute a node of the DT, we either a) report the associated prediction if the node is a leaf or b) just need to perform a simple comparison and decide whether the next node is the left child or the right child. In any modern programming language, there are (at least) two ways to implement such a decision tree.

- A simple implementation, named the *native* tree, uses a loop to iterate over each node of a tree within a continuous data structure, e.g., arranged by a one-dimensional array. An example code can be found in Listing 1.
- An alternative implementation, named the *if-else* tree, statically generates if-else blocks. Here, the split values of a tree are all hard-coded as constant values into the instructions. An example code can be found in Listing 2.

```
struct Node {
  bool isLeaf;
  unsigned int prediction; // Predicted Label
  unsigned char feature; // Targeted feature
  float split; // Threshold
  unsigned short leftChild;
  unsigned short rightChild;
};
Node tree[] = {{0,0,0,8191,1,2},{0,0,1,2048,3,4},..]}
bool predict(short const x[3]){
  unsigned int i = 0;
  while(!tree[i].isLeaf) {
    if (x[tree[i].f] <= tree[i].split) {
      i = tree[i].left;
    } else {
      i = tree[i].right;
    }
  }
  return tree[i].prediction;
}
```

Listing 1: Native tree structure in C++

```
bool predict(short const x[3]){
  if(x[0] <= 8191){
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  } else {
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  }
}
```

Listing 2: Example for If-else structure in C++.

Based on the given forest model, we will explore how to automatically generate implementations of optimized *native* trees and *if-else* trees. The presentation in this paper and our current implementation aim for generating C++ code. The same method can also be applied for generating code segments in other programming languages, e.g., JAVA, C, etc. The generated code may be further optimized by the compiler, but this is out of the scope of this paper.

III. Memory Locality

Due to the significant performance gap between the main memory (DRAM) and the processor, modern computer architectures have introduced a memory hierarchy. In addition to the main memory, smaller and faster memory subsystems next to the processors, in the forms of *cache* and *scratchpad memory*, are used to hide the *long* memory latencies of DRAM. Drepper provides very insightful discussions about the impact of memory hierarchy on the performance of programs [23].

In this paper, we will consider modern computer systems with instruction and data-caches. The key assumption of the memory hierarchy is the *locality*:

Temporal locality: Recently accessed items will be accessed in the near future, e.g. small program loops

Spatial locality: Items at addresses close to the addresses of recently accessed items will be accessed in the near future, e.g. sequential accesses to elements of an array.

Unfortunately, naive implementations of DTs do not exploit such locality when they classify a set of input data.

The benefit of the *native* tree implementation is the temporal locality of the program, i.e., executing a tree is a simple loop with a few lines of codes. However, the accesses to the nodes of the tree do not have any spatial locality. The execution of a DT follows a *unique path* from the root to a leaf, which are stored in memory addresses that are unfortunately arranged discontinuously, if no attention is made. As a result, the cached data will not be further used, if the distance between each node of the path is greater than the number of nodes that can be loaded into a cache set at once.

As for the *if-else* tree implementation, since the thresholds and the values needed for a split node of a tree are all hard-coded into the instructions, this avoids indirect memory

accesses and has a clear advantage of the reduction of the latency. Therefore, the *if-else* tree implementation does not suffer from missing data locality. However, without awareness of instruction-cache design, the hard-coded instructions may just be loaded into the data-cache once and only used once, so that the advantage of the temporal locality in the instruction-cache is completely abandoned.

There are three types of caches misses [24], namely compulsory, conflict, and capacity cache misses. The *compulsory misses* are due to the first access to a memory block, which by definition is not in cache. The *capacity misses* occur when some memory blocks are discarded from the cache due to the limited cache capacity, i.e., the program working set is much larger than the cache capacity. The *conflict misses* occur in set associative or direct mapped caches when several blocks are mapped to the same cache set.

The implementation of a tree should take the layout of the data (in the *native* tree), the instructions of the branches (in the *if-else* trees), and the size of caches of the particular platform of execution into consideration.

IV. Related Work

Random forests and DTs have been studied in the context of CPUs architecture already. Van Essen et al. present in [25] a comprehensive study of different architectures for implementing RFs on CPUs, FPGAs and GPUs. Based on the CATE algorithm [26], the authors train a RF with DTs constrained by a fixed height. By fixing the tree-size, the authors show an effective pipelining approach for executing DTs on CPUs, FPGAs and GPUs. Note, however, that from a theoretical perspective we know that trees should be as large as possible to offer a small bias error. Thus, this "fixed tree-size" approach may hinder the effectiveness of RF models.

In [27] Asadi et al. introduce different implementation schemes of tree-based models in the context of learning-to-rank tasks. They mainly introduce the two different implementation schemes already discussed in the former section: The first one uses a while-loop to iterate over individual nodes of the tree, whereas the second approach decomposes each tree into its individual if-else structure. For the first implementation, the authors also consider a continuous data-layout (i.e., an array of *structs*) to increase data locality, but do not directly optimize each implementation. Also note, that the authors mainly consider gradient boosted trees. There, the individual trees are usually "weak" in a sense, that they are comparably small, as opposed to larger trees in RFs.

Again in the context of ranking models, Lucchese et al. present the QuickScorer algorithm for gradient boosted trees [28]. In this approach, the authors discard the tree structure, but view each tree traversal as a series of bit operations on a 2^{Δ} dimensional bitvector, where Δ is the height of a tree. Their approach offers significant speed-up, but is limited to trees with fixed height $\Delta \leq 6$. Again, this is a reasonable restriction for gradient boost trees, but not for RF models, where trees tend to be significantly larger.

Kim et al. present in [29] an implementation for binary search trees using vectorization units on Intel CPUs and compare their implementation against a GPU implementation. The authors provide insight in how to tailor the implementation to Intel CPUs by taking into account register sizes, cache sizes, and page sizes. Their work is specialized for Intel CPUs and thus it is not directly applicable for different CPU architectures. Lucchese and colleagues already have noticed, that many nodes are seldomly visited [4]. Buschjaeger and Morik formalize this observation in [22] by estimating the probabilities of specific paths during tree traversal. Based on this probabilistic view of model execution, the authors consider different implementation schemes for tree traversal and theoretically analyze their runtime. Note however, that this model of computation remains at the software level and does not include the memory layout.

V. Optimizations of Decision Tree

So far we have introduced the two standard approaches for implementing DTs. In this section, we are going to discuss optimizations of these two approaches. To do so, we first discuss the downsides of each implementation regarding their caching behaviour. Then, we present optimizations to improve caching.

A. Optimization of If-else Tree

As already mentioned, we can unroll the comparisons of a DT into conditional statements forming an if-else structure (cf. Listing 2). Since the entire tree is transformed into if-else blocks without indirect memory accesses, we can expect this implementation to perform better than the *native* tree structure. However, cache misses may still occur:

a) Reducing compulsory cache misses: When an instruction cache miss takes place, several instructions are sequentially fetched into the instruction cache. When a branch is executed, these prefetched instructions will possibly not be utilized. If we can increase the chance of actually using prefetched instructions, we can reduce the number of the compulsory cache misses. However, DTs are naturally composed of many branches. To reduce the possibility of branch executions for tree $\mathcal{T}$ we can traverse all its paths and swap the children of every node i when $p(i \rightarrow l(i)) \geq p(i \rightarrow r(i)))$. This way, we decrease the possibility to branch out of the current block, which in turn increases the utilization of prefeteched code blocks.

b) Reducing capacity and conflict cache misses: The best case for exploiting the instruction-cache fully is having all the instructions of the *if-else* tree loaded into the instruction-cache. However, if the size of the instructions from the overall tree structure is greater than the size of the instruction-cache, the cached instructions may be evicted out by loading other instructions due to the capacity and conflict cache misses. Considering the usage of DTs, we notice that keeping the instructions of those nodes utilized frequently in the instruction-cache can improve the utilization of the cached instructions, resulting in better performance.

With the above idea, we can define a computation kernel which contains those nodes which are used most of time. For example, note that the root node of a tree is used in every case and thus it should be kept inside the cache all the time. Let $\mathcal{K}$ denote the kernel and let $s(i)$ be a mapping function returning the instruction size of node i. Then, we wish to solve the following optimization problem:

$$\mathcal{K} = \arg\max \left\{ p(T) \middle| T \subseteq \mathcal{T} \text{s.t.} \sum_{i \in T} s(i) \leq \beta \right\} \quad (1)$$

where β is a given budget related to the size of the instruction-cache on the targeted architecture. Given $\mathcal{K}$, we can make sure, that these nodes are likely to remain in the cache, whereas the remaining nodes $\mathcal{L} = \mathcal{P} \setminus \mathcal{K}$ may be evicted more often.

In order to solve Eq. 1 we need to iterate over all possible subsets of $\mathcal{T}$ which might be difficult for large trees. Thus, we propose a greedy approach in which we look at a complete path from root to leaf node: First, we swap the children depending on their probabilities as already explained in the former section. Then, we sort all paths in the tree by their probability. After that, we greedily add a node one by one into $\mathcal{K}$ until the accumulated size of the added nodes b is greater than the given budget $\mathcal{B}$. The rest of nodes are all added into $\mathcal{L}$. Algorithm 1 summarizes the presented approach.

Algorithm 1 Optimized *if-else* tree
Input: Tree $\mathcal{T}$,Paths $\mathcal{P} = \{\pi_1, \ldots, \pi_M\}$
Output: Kernel $\mathcal{K}$, Label $\mathcal{L}$

```
swapChildren(T)
P = sortByProbabilities(P)
b = 0
for π ∈ P do
    for i ∈ π do
        if b + s(i) > B then
            Add i to L
        else
            Add i to K
            b = b + s(i)
        end if
    end for
end for
```

Once the nodes are grouped into $\mathcal{K}$ and $\mathcal{L}$ respectively, we can use `goto` statements to break the sequential generation of if-else blocks: First, we generate if-else blocks for all nodes in $\mathcal{K}$. Once the left/right child of one of those nodes is in $\mathcal{L}$, a goto statement is generated at the same position to replace the original if-else statement. Then, the corresponding if-else statements of this node and its children are all generated into a label block at the end. Listing 3 shows an example based on Listing 2 by applying Algorithm 1.

The question remains, how to estimate the instruction size $s(\cdot)$ of each node. The instruction set size generally differs for the two different types of nodes:

- **Split nodes** require three types of instructions. First, the values of the target feature and the corresponding threshold are loaded into registers. Second, the values inside the registers are compared against constant values and last, a jump out the current block is performed based on the comparison.
- **Leaf nodes** need two types of instructions. First, the return value of the prediction are stored into a register and second, a jump back to the caller of the if-else tree is performed.

Based on the above analyses, we can estimate $s(\cdot)$ by counting the number of generated instructions on the targeted architecture in an isolated example. However, note that in a real application the actual number of instructions may differ based on the compiler, compiler options and actual code used. Thus, this mapping function helps to choose appropriate parameters, but should be viewed as the expected size of a node. Table I summarizes the expected size of instructions for ARM, X86 (Intel) and PPC.[2]

	ARM [Bytes]		PPC [Bytes]		Intel [Bytes]	
Type	Int	Float	Int	Float	Int	Float
Split	20	32	20	48	28	17
Leaf	8	8	8	8	10	10

TABLE I: The expected size of instructions for a split node and a leaf node in a decision tree on ARM (`Raspberry PI 2`), PPC (`NXP T4240 processors`) and Intel (`Intel Core i7-6700`) processors.

```
bool predict(short const x[3]){
  if(x[0] > 8191){
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  } else {
    goto Label0;
  }
Label0:
  {
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  }
}
```

Listing 3: If-else structure in C++ with goto statements

B. Optimization of Native Tree

As shown in Listing 1 we can implement a DT by placing the nodes sequentially in an array and traversing this array by using a simple while loop. We observe that half of the nodes in a tree are leaf nodes, which only store a prediction value. The *naive native* implementation however assumes the same data type for each node, leading to unnecessary overhead. Second, considering the usage of DTs for predicting classes, we notice that the data access pattern in the array is mostly not sequential. The distance between each accessed element

[2]We adopt GNU C++ (g++) compiler version 4.8.3 for ARM, version 4.9.2 for PPC, and version 5.4.0 for Intel with `-O0` option.

becomes bigger when the depth of targeted nodes in the DT becomes greater. This phenomenon violates the spatial locality of the array and abandons the advantage of the cache design, which may result in high cache misses.

a) Reducing compulsory cache misses: Nodes are prefetched into the instruction-cache sequentially. If we can reduce the amount of memory each node needs, we can fit more nodes into the cache and thus reduce compulsory cache misses. For the native implementation we recognize that a leaf node only stores a prediction value, but does not use the pointer to its children, nor does it use the feature index or the split-value. A simple way to reduce the memory consumption is to remove all leaf nodes from this array and move them to a separate array with a specialized data type. However, then we have to layout two arrays in the memory which might be difficult. Thus we propose to abandon the `isLeaf` and `prediction` field of the native solution, but store the prediction of the left (right) child directly in the respective fields `left` (`right`) if it is a leaf node. This method only requires us to layout one array, but offers the same size-reduction as using two arrays.

b) Reducing capacity and conflict cache misses: As mentioned in Section III, if no attention is made, the nodes stored in memory are arranged discontinuously. Thus, when a node is loaded into the cache, the nearby nodes should be on the same path to reduce capacity and conflict cache misses. A sensible way to exploit the data locality is to allocate as many nodes as possible on the same path into the same cache set.

To do so, we propose the following approach, where τ denotes the cache set size: Let `A` be the array in which we place all nodes of $\mathcal{T}$. Further, let $\mathcal{C}$ be the candidate list of nodes in $\mathcal{T}$ which have not been placed in `A` yet and let $\mathcal{S}$ denote the nodes which should be placed in the same cache set. For each node, we greedily choose that child, which has highest probability on the current path and try to place it in $\mathcal{S}$. Once $\mathcal{S}$ contains $\tau - 1$ elements (thus is full), we append all nodes form $\mathcal{S}$ to the array `A`, reset $\mathcal{S}$ and continue with the next cache set. Algorithm 2 summarizes this method. When adding a new node to $\mathcal{S}$, attention has to be paid, because there are to types of nodes (Line 7):

- The current node is a split node. Then we pick the next node based on the children's probabilities and put the more probable child into $\mathcal{S}$ and the other child into the candidate list $\mathcal{C}$.
- The checked node is a leaf node, i.e., it is the end of the path: We pick up a sub-root with the highest probability from the candidate list $\mathcal{C}$ as long as it is not empty. The traverse starts again until $\mathcal{S}$ is full.

If the current $\mathcal{S}$ is full before finishing a traverse of a path (Line 14), two children should be put back to the candidate list $\mathcal{C}$ (Line 16). A sub-root which has the highest probability should be picked up from $\mathcal{C}$ for the next new set $\mathcal{S}$. Once a set is finished, the nodes in it will be allocated into the data array sequentially. To the end, the output of the algorithm is the data array with a path-oriented layout, in which path-oriented sets are sequentially allocated into the array.

Algorithm 2 Optimized *native* tree
Input: Tree-nodes $\mathcal{T}$, maximum nodes per set τ
Output: A data array `A` with the path-oriented layout

```
1: A = [ ]
2: C ← {0}
3: while C ≠ ∅ do
4:     i = arg max_{j∈C}{p(π(j))}
5:     C = C \ {i}
6:     S = {i}
7:     while |S| ≠ τ do
8:         if i is leaf-node and C ≠ ∅ then
9:             i = arg max_{j∈C}{p(π(j))}
10:            C = C \ {i}
11:        else
12:            C = C ∪ arg min{p(i → l(i)), p(i → r(i))}
13:            i = arg max{p(i → l(i)), p(i → r(i))}
14:            if |S| = τ − 1 then
15:                //this is the last node in S
16:                C = C ∪ {l(i), r(i)}
17:            end if
18:        end if
19:        S = S ∪ {i}
20:    end while
21:    A.append(S)
22: end while
23: return A
```

We want to give a quick example to illustrate algorithm 2. Consider the DT in Figure 1 and set the size τ to three. The first path starts from the root node 0, which is a split node. Accordingly, node 2 with a higher probability is chosen, and node 1 is put into $\mathcal{C}$. From the children of node 2, the leaf node 6 is chosen (node 5 is put into $\mathcal{C}$). As now $\mathcal{S}$ is full with three nodes, the algorithm adds the current $\mathcal{S}$ into the output array, prepares a new set $\mathcal{S}$, and picks up a sub-root with the highest probability from $\mathcal{C}$. In the list $\mathcal{C}$, currently there are nodes 1 and 5. Since the probability of node 1 is higher than the probability of node 5, node 1 is the next chosen sub-root. To the end, the delivered sets are: $\{0, 2, 6\}$ and $\{1, 3, 5\}$.

Please, note that the proposed approaches in a) and b) both may be applied while implementing the optimization for *native* trees. To do so, an additional field is required in the node structure that indicates whether the prediction is embedded in the respective fields `left(right)`. In Algorithm 2, the leaf-node case can be skipped technically, whereas the split-node case has to consider this additional field accordingly.

VI. Experiments

In this section we experimentally evaluate the proposed optimizations. We have performed 1800 different experiments by training Decision Trees (DT) [30], Random Forests (RF) [12] and Extremely Randomized Trees (ET) [14] on 12 different data-sets with varying tree-sizes to generate the aforementioned implementations for different architectures, i.e., X86, PPC and ARM CPUs.

Table II shows the data-sets we used during the experiments. All data-sets are available in the UCI Machine Learning Repository [31] with the exception of MNIST [32], IMDB [33] and FACT [34]. In addition to the number of features and the number of examples during test-time, we also report the range of accuracy for the three different models DT, RF and ET. In

all experiments we used the CART algorithm with the Gini-Score criterion for node-splitting and trained models using the `sklearn` package [35]. For RF and ET we used 25 trees. If the respective data-set comes with a pre-computed train/test split we use this. Otherwise we use 75% of the data for training and 25% of the data for testing. Expectantly, DTs often do not achieve high accuracy, whereas RF and ET perform best with large trees. We want to emphasize that we did not perform any hyperparameter optimization with respect to the classification accuracy, but report the accuracy here to validate our tool-chain.

After training, we export the models into a JSON format which is used by our code-generator. During generation, we make sure, that optimized trees retain their accuracy. Note, that `sklearn` uses a probability-based majority vote, whereas we weight all votes equally. Thus, final predictions may differ, but we could not detect any significant change in the final accuracies. Also note, that `sklearn` always produces floating-point split-values. For data-sets with integer features (e.g. letter or MNIST) this was rounded down towards the next integer to circumvent the use of floating-point. This does not change the accuracy neither. Our code is publicly available at https://bitbucket.org/sbuschjaeger/arch-forest.

After the implementations have been generated, we use the `GNU` tool-chain to compile the code with the most aggressive optimizations (`-O3`) enabled. Each implementation is tested individually by using the following protocol: For minimizing unfairness due to caching, we first iterate twice over the test data and perform predictions (burn-in phase). Then, we measure the runtime needed to classify all examples in the test set and repeat this 50 times.

We note that the performance of our implementation compared to `sklearn` might be of interest, since `sklearn` is arguably one of the most-used machine learning library and thus well-known to many practitioners. We found, that our implementation is on average $500 - 1500$ times faster than `sklearn`. However, we admit that this comparison is biased, because large parts of `sklearn` are written in Python and optimized for batch execution.

Therefore, we will focus the remaining evaluation on our implementation. Due to limited space, we will focus on RF models in the remaining parts of this section. We notice that DT and ET result in similar behaviour across all systems and thus do not add much more value to the discussion here. We use the *naive native* implementation as baseline for all experiments and measure the average speed-up for each data-set of each optimization against this implementation.

For the *native* optimization, we choose $\tau = 25$ on X86, $\tau = 8$ on ARM, and $\tau = 8$ for the PPC CPU. For *if-else* optimizations, we use a instruction-cache size $\beta = 128000$ Bytes on X86, $\beta = 32000$ Bytes on ARM, and $\beta = 32000$ Bytes on the PPC CPU. The experiments where performed on a `Intel Core i7-6700` desktop machine with 16 GB RAM for `X86`. For `PPC` we use a `NXP Reference Design Board` with `T4240 processors` and 6 GB RAM. For `ARM` we use a `Raspberry PI 2` with a `ARMv7` CPU and 1 GB RAM.

TABLE II: Summary of data sets for our experiments based on UCI data sets [31], IMDB [33], MNIST [32], FACT [34].

Dataset	# Examples	# Features	Accuracy
adult	8141	64	0.76 - 0.86
bank	10297	59	0.86 - 0.90
covertype	145253	54	0.51 - 0.88
fact	369450	16	0.81 - 0.87
imdb	25000	10000	0.54 - 0.80
letter	5000	16	0.06 - 0.95
magic	4755	10	0.64 - 0.87
mnist	10000	784	0.17 - 0.96
satlog	2000	36	0.40 - 0.90
sensorless	14628	48	0.10 - 0.99
wearable	41409	17	0.57 - 0.99
wine-quality	1625	11	0.49 - 0.68

A. Experiments on the X86 CPU architecture

Figure 2 depicts the average speed-up of the four different optimizations on Intel. First we note, that the *if-else* tree versions are the fastest on Intel and offer a speed-up around 3 across all tree sizes. For smaller tree depth from $1 - 10$, we see that optimizing *if-else* trees only offer marginal speed-up. However, for larger tree depth around 15 and 20 we see, that optimized *if-else* trees can retain their speed-up and outperform un-optimized *if-else* trees with a speed-up factor larger than 3.

Native trees do not perform as well as *if-else* trees on Intel CPUs. Overall, the speed-up compared to *naive native* trees is only marginal for smaller trees below depth 15. Here, both version the *StandardNativeTree* and the *OptimizedNativeTree* offer a speed-up of 1.5 at-most. Interestingly, for larger trees around depth 15 and more, we again notice that our optimizations improve performance.

B. Experiments on the PPC CPU architecture

Figure 3 depicts the average speed-up of the four different optimizations on PPC. We can observe that the results here are similar to Figure 2 in which *if-else* trees always outperform *native* trees with a speed-up in the range from $2-5$. Along with the increment of tree depth, the speed-up from both *if-else* tree versions drop, but especially un-optimized *if-else* trees suffer performance by dropping to almost 2, whereas the optimized version can retain a speed-up around 3.5.

Similar to `X86` CPU, the *native* implementation does not seem to be the best choice here by providing a speed-up under 2 in all cases. However, we also notice, that with increasing tree sizes, the optimizations are more important. It is worth noting, that we can observe some cases where the *native* trees outperform *if-else* trees when tree depth is bigger than 15.

C. Experiments on the ARM CPU architecture

Figure 4 depicts the average speed-up of the four different optimizations on ARM. We observe that the situation on ARM is more fragmented compared to `X86` and `PPC`. In general, we

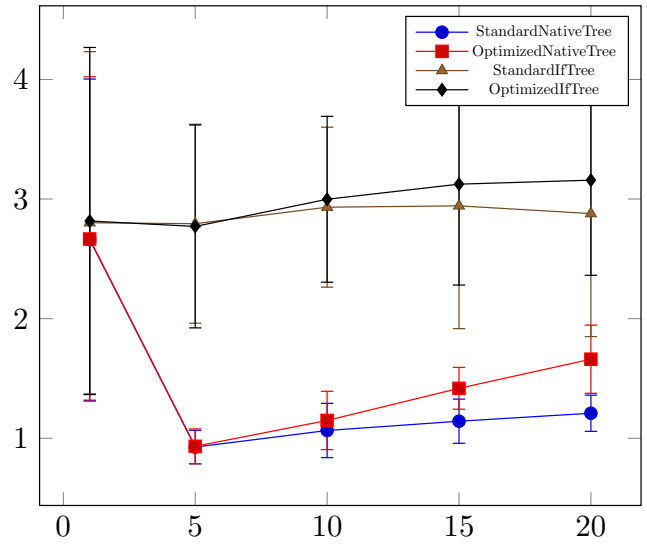

Fig. 2: Average speed-up factor for real-time execution compared to the naive native implementation on Intel for tree sizes from $1-20$.

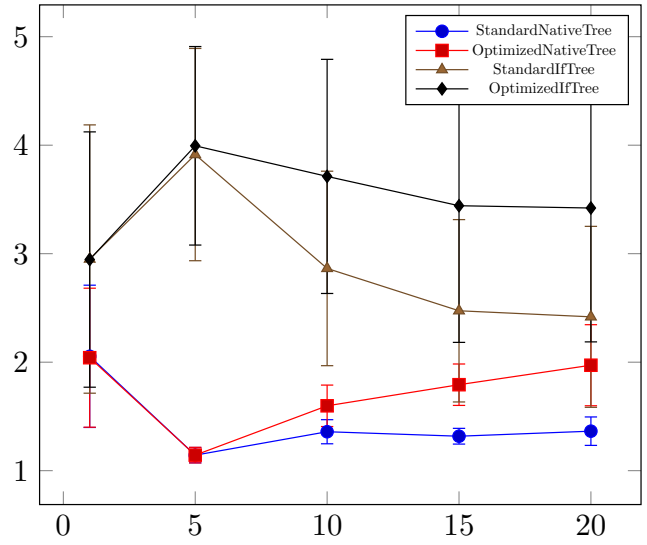

Fig. 3: Average speed-up factor for real-time execution compared to the naive native implementation on PPC for tree sizes from $1-20$.

are able to achieve a speed-up around 4 for small trees, which drops to around $2-3$ for larger trees. Both implementations roughly start with the same speed-up factor for small trees, but then quickly diverge for tree depth around $5-15$. In this range of tree depth we see that *if-else* trees are the fastest choice on ARM. Additionally, we notice that with increasing tree depth cache optimizations become more important and consistently outperform their un-optimized counterpart. Once trees are sufficiently large, we see that the *native* trees match the performance of *if-else* trees again and even outperform

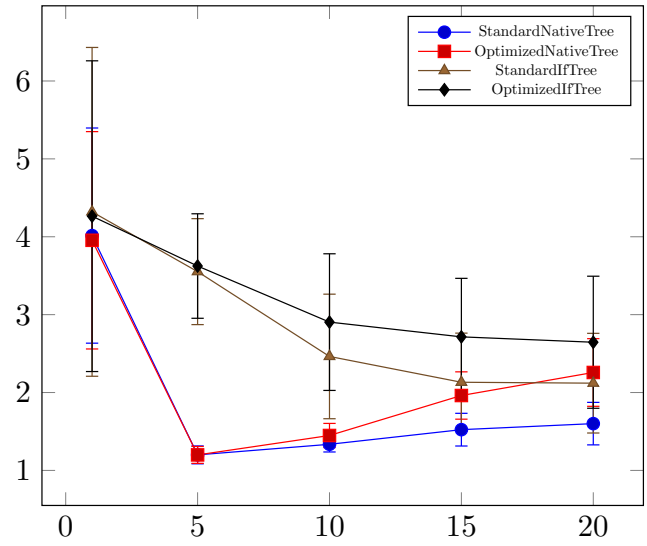

Fig. 4: Average speed-up factor for real-time execution compared to the naive native implementation on ARM for tree sizes from $1-20$.

them for tree depth of 15 and 20 in some cases. In this sense, the results are similar to what we have seen on the `PPC` architecture.

D. Discussion of the experiments

The experiments show overall different behaviour across the three different architectures, but also depict some similarities. Here, we want to discuss these phenomena in terms of the properties of the specific architectures, as well as the concrete CPU models used for experiments. We note, that one of the main architectural differences between `X86`, `ARM` and `PPC` are the available instructions. Since *native* trees only use a small amount of hot-code, the differences between CPU architectures will likely not matter much here. However, looking at *if-else* trees we can expect a larger difference. To further investigate the interplay between CPU architectures and code size, we consider table III in the following. Table III depicts the instruction size of a single tree for varying tree sizes for the FACT data-set (containing floating-point features) and the letter data-set (containing integer features) of the *StandardIfTree* implementation.

Clearly the *if-else* trees are the best choice for Intel CPUs, but why is that so? We find two reasons for that, one of which is related to the architectural specifics of `X86` architecture and one which is related to the specific CPU we used: First, `X86` CPUs are Complex Instruction Set Computers (CISC) offering a very rich set of instructions which include all sorts of specialized operations. Since *if-else* trees unroll the complete tree structure into code, they give the compiler the opportunity to utilize this multitude of instructions to the fullest, by encoding larger parts of the tree in single instructions. And indeed, looking at table III we see that the

Intel CPU almost always requires the fewest instructions per decision tree. Second, the Intel Core i7-6700 CPU used for experiments has a comparably large instruction cache of 256 KiB combined with two larger shared caches of 1 MiB (L2 Cache) and 8 MiB (L3 Cache). Thus, by encoding a single tree in only a few instructions, it is likely to fit it into the larger instruction cache. In contrast, *native* trees do not utilize the CISC architecture and "waste" additional data cache by encoding the tree as data and not as instructions.

Similar to the X86 architecture, we have seen that *if-else* trees perform very well on the PPC architecture, but to a lesser extend. Again, we can try to explain this behaviour in terms of the PPC instruction set architecture and the specific CPU model used for experiments. The PPC CPU architecture is a Reduced Instruction Set Computer (RISC) with performance enhancement for high performance computing. RISC does not offer instructions for specialized operations as CISC does. Thus, the compiler must largely rely on the combination of (comparably) simple instructions to implement *if-else* trees. This in turn results in larger code which is less likely to fit into the instruction cache. Comparing the instruction size of PPC to X86 in table III we see that the PPC architecture indeed requires more instructions compared to X86. Interestingly, this case is less sever for integer features, which can be attributed to the enhancements in this instruction set architecture. Looking at the cache sizes of the T4240 processors we see that it only has 32 KiB instruction cache, but also comes with a 2 MiB shared L2 cache, which is even larger than the Intel Core i7-6700 CPU. For smaller trees around $5 - 10$, the cache sizes are still enough to hold all trees and thus *if-else* trees are still the fastest choice. If trees become large (depth 10 and more), the instruction cache is not enough to hold all trees anymore and we must rely on the larger L2 cache. However, this cache is slower and thus we suffer some performance penalty, which in combination with the larger code size explains the performance drop for larger trees.

Last, we want to discuss the fragmented behaviour of the ARM architecture. Again we try to answer this question in terms of ARM's instruction set architecture and the specific CPU used for experiments. Much like its PPC counterpart, ARM also uses a reduced instruction set architecture (RISC). However, ARM's RISC does not come with specialized instructions for high performance computing and thus the compiler has to rely to a larger extend on the combination of simple instructions for *if-else* implementation. This in turn results in even larger code for integer features, which is less likely to fit into the instruction cache as shown in table III. Interestingly, for floating-point features, we see that the ARM CPU uses fewer instruction than the PPC CPU, which can be attributed to specific CPU model used during experiments. The T4240 processors are optimized towards high-performance computing in a low-power embedded computing setting, such as networking applications and thus are optimized towards integer operations. In contrast, the ARMv7 CPU of the Raspberry PI 2 is a general purpose CPU aimed at the needs of the average user and thus it lays a larger emphasis on floating-point operations compared to the T4240 processors. It has a 32 KiB instruction cache in combination with a significantly smaller 512 KiB L2 shared cache. In comparison to the other CPUs this means, that the ARM CPU has $2 - 16$ times less L2/L3 cache available. For smaller trees around depth $5 - 10$ the cache sizes are still enough to hold all trees and thus *if-else* trees are still the fastest choice. For larger tree sizes however, the instruction cache is not enough any more and *native* structures using the data-cache become faster. However, since the data-cache is also small, we quickly fill both caches to their maximum. Interestingly, if we optimize both *if-else* and *native* trees we end up with roughly the same performance.

VII. Conclusion

In this paper, we advocate for optimizing the evaluation of learned models in the real-time setting. Since tree ensembles in general and specifically Random Forests are among the most powerful black-box methods available, we looked at the application of decision trees from an computer-architectures point of view.

We introduced a probabilistic view of decision tree execution and carefully analysed caching behaviour for data and instructions in this setting. Based on this analysis we were able to derive two optimization for the most common decision tree implementations, namely the *native* and the *if-else* trees. We empirically showed that our optimization are capable of increasing the performance of around $2-5$ for ARM (RISC), $2-4$ for PPC (Enhanced RISC) and around $2-4$ on X86 (CISC) architectures without changing the classification accuracy. We provided an in-depth discussion of these findings in terms of the CPU architectures and specific CPU models used during experiments which may help practitioners to choose the right implementation. Additionally we note, that in all 1800 experiments we neither lost classification accuracy, nor did we lose performance by employing our optimizations, making them always worth applying. Moreover, we found, that our implementation outperforms sklearn by a factor up to 1500 for real-time classification.

Our findings in this paper show that implementing RF can be challenging since the underlying features of computer architectures should be carefully considered. Our algorithms only exploit some features regarding memory hierarchy. Implementations that consider other architectural features like scratchpad, out-of-order execution, hyper-threading, branch prediction, etc. may further improve the performance, which are consider as future work.

Acknowledgment

Part of the work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A1, B2 and C3.

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	224	575	8185	51005	167644
PPC	232	604	7732	51840	170772
ARM	204	604	9040	55012	180628

(a) Kernel of `covertype` with integer features

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	96	415	17023	127330	404722
PPC	96	556	20996	169696	577952
ARM	88	428	18436	154992	542020

(b) Kernel of `fact` with floating point features

TABLE III: The actual size of instructions for executing kernels on different architectures with O3 option.

REFERENCES

[1] M. Stolpe, "The Internet of Things: Opportunities and challenges for distributed data analysis," *SIGKDD Explorations*, vol. 18, no. 1, pp. 15–34, June 2016.

[2] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An empirical evaluation of supervised learning in high dimensions," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 96–103.

[3] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *J. Mach. Learn. Res*, vol. 15, no. 1, pp. 3133–3181, 2014.

[4] C. Lucchese, F. M. Nradini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Quickscorer: Efficient traversal of large ensembles of decision trees," in *Procs. ECML PKDD 2017*. Springer, 2017, pp. 383 – 387.

[5] E. Schubert, H.-P. Kriegel, and A. Zimek, "The (black) art of runtime evaluation: Are we comparing algorithms or implementations?" *Knowledge and Information Systems*, vol. 52, no. 2, pp. 341 – 3778, 2017.

[6] S. Nijssen and J. N. Kok, "Frequent subgraph mines: runtimes don't say everything," in *Procs. 4th Workshop on Mining and Learning with Graphs (MLG)*, 2006, pp. 173 – 180.

[7] J. Buss, C. Bockermann, K. Morik, W. Rhode, and T. Ruhe, "Fact-tools – processing high-volume telescope data," in *26th Astronomical Data Analysis Software and Systems conference (ADASS)*, 10 2016, aDASS.

[8] J. Marín, D. Vázquez, A. M. López, J. Amores, and B. Leibe, "Random forests of local experts for pedestrian detection," in *IEEE International Conference on Computer Vision, ICCV*, 2013, pp. 2592–2599.

[9] G. Fanelli, M. Dantone, J. Gall, A. Fossati, and L. J. V. Gool, "Random forests for real time 3d face analysis," *International Journal of Computer Vision*, vol. 101, no. 3, pp. 437–458, 2013.

[10] F. Saki, A. Sehgal, I. M. S. Panahi, and N. Kehtarnavaz, "Smartphone-based real-time classification of noise signals using subband features and random forest classifier," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*, 2016, pp. 2204–2208.

[11] P. Libuschewski, "Exploration of cyber-physical systems for GPGPU computer vision-based detection of biological viruses," Ph.D. dissertation, Technical University of Dortmund, Germany, 2017. [Online]. Available: http://hdl.handle.net/2003/35929

[12] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] ——, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[14] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[15] L. Breiman, "Bias, variance, and arcing classifiers," 1996.

[16] ——, "Some infinity theory for predictor ensembles," Technical Report 579, Statistics Dept. UCB, Tech. Rep., 2000.

[17] G. Biau, "Analysis of a random forests model," *Journal of Machine Learning Research*, vol. 13, no. Apr, pp. 1063–1095, 2012.

[18] M. Denil, D. Matheson, and N. De Freitas, "Narrowing the gap: Random forests in theory and in practice," in *International conference on machine learning (ICML)*, 2014.

[19] G. Louppe, "Understanding random forests: From theory to practice," *arXiv preprint arXiv:1407.7502*, 2014.

[20] G. Biau and E. Scornet, "A random forest guided tour," *Test*, vol. 25, no. 2, pp. 197–227, 2016.

[21] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997. [Online]. Available: http://www.research.att.com/~schapire/papers/FreundSc95.ps.Z

[22] S. Buschjäger and K. Morik, "Decision tree and random forest implementations for fast filtering of sensor data," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. PP, no. 99, pp. 1–14, 2017.

[23] U. Drepper, "What every programmer should know about memory," 2007.

[24] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[25] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga?" in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 232–239.

[26] R. Prenger, B. Chen, T. Marlatt, and D. Merl, "Fast map search for compact additive tree ensembles (cate)," Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2013.

[27] N. Asadi, J. Lin, and A. P. de Vries, "Runtime optimizations for tree-based machine learning models," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, Sept 2014.

[28] C. Lucchese, F. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2015, pp. 73–82.

[29] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 339–350.

[30] L. Breiman, *Classification and regression trees*. Routledge, 1984.

[31] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[32] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[33] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[34] H. Anderhub, M. Backes, A. Biland, V. Boccone, I. Braun, T. Bretz, J. Buß, F. Cadoux, V. Commichau, L. Djambazov *et al.*, "Design and operation of fact–the first g-apd cherenkov telescope," *Journal of Instrumentation*, vol. 8, no. 06, p. P06008, 2013.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.