Bolt: Fast Inference for Random Forests

Eduardo Romero-Gainza
romerogainza.1@osu.edu
The Ohio State University
USA

Christopher Stewart
cstewart@cse.ohio-state.edu
The Ohio State University
USA

Angela Li
li.10011@osu.edu
The Ohio State University
USA

Kyle Hale
khale1@iit.edu
Illinois Institute of Technology
USA

Nathaniel Morris
nathaniel.morris@amd.com
AMD Research
USA

ABSTRACT

Random forests use ensembles of decision trees to boost accuracy for machine learning tasks. However, large ensembles slow down inference on platforms that process each tree in an ensemble individually. We present Bolt, a platform that restructures whole random forests, not just individual trees, to speed up inference. Conceptually, Bolt maps every path in each tree to a lookup table which, if cache were large enough, would allow inference with just one memory access. When the size of the lookup table exceeds cache capacity, Bolt employs a novel combination of lossless compression, parameter selection, and bloom filters to shrink the table while preserving fast inference. We compared inference speed in Bolt to three state-of-the-art platforms: Python Scikit-Learn, Ranger, and Forest Packing. We evaluated these platforms using datasets with vision, natural language processing and categorical applications. We observed that on ensembles of shallow decision trees Bolt can run 2–14X faster than competing platforms and that Bolt's speedups persist as the number of decision trees in an ensemble increases.

CCS CONCEPTS

• **Software and its engineering** → **Allocation / deallocation strategies**; *Middleware*; *Main memory*; *Client-server architectures*; • **Computer systems organization** → Client-server architectures; • **Computing methodologies** → Parallel algorithms; *Artificial intelligence*.

KEYWORDS

Decision tree, Ensemble model, Cache, Branch misprediction, Random Forest, Interpretability

ACM Reference Format:
Eduardo Romero-Gainza, Christopher Stewart, Angela Li, Kyle Hale, and Nathaniel Morris. 2022. Bolt: Fast Inference for Random Forests. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3528535.3531519

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3531519

1 INTRODUCTION

Random forests, deep forests [30], and gradient-boosted forests [8] combine predictions from multiple decision trees, i.e., an ensemble. Compared to individual trees, ensembles can boost accuracy for classification and regression tasks by reducing overfitting errors caused by bias in the modeling algorithm. Extant platforms for random forests process each tree in the ensemble independently. While correct and easy to understand, this approach causes inference speed to slow down as ensemble size grows. Even though trees are created independently, multiple trees within an ensemble use the same sequence of features for prediction, leading to wholly or partially redundant paths. By processing each tree independently, platforms waste resources during inference on redundant paths.

Decision trees can be transformed into lookup tables and subsequently stored in processor cache to speed up inference. Each path maps to one or more entries in the lookup table and each entry stores a leaf-node result. During inference, input data is encoded as a lookup table entry to retrieve the correct leaf node. To ensure that input data maps to exactly one lookup table entry, the encoding scheme must use every feature in the decision tree. That is, the encoding scheme produces the address in the lookup table that stores the correct leaf-node result, and during inference, this path is retrieved using this address. Henceforth, we use the terms lookup table entry and lookup address interchangeably.

Lookup tables require only one memory access for inference on a given decision tree. In comparison, breadth-first processing of decision trees requires one memory access for each node along the path from root to leaf. Lookup tables also avoid conditional control flow, alleviating pressure on the branch predictor. While transforming decision trees to lookup tables has benefits, it also has severe drawbacks for random forests:

1. **Lookup tables have large storage demands:** Lookup tables require distinct addresses for each possible input. The address space is 2^f where f is the number of features in the tree. The address space can easily exceed the size of processor cache, such that inference requires slow accesses to main memory.
2. **Random forests use ensembles, multiplying storage demands:** Ensemble models, such as random and deep forests [30] use many decision trees to achieve high accuracy and, as a result, further increase storage demands.
3. **Redundant paths between trees inflate storage demands:** Lookup tables encode paths in one or more addresses. If trees in

a random forest ensemble have redundant paths, each path may produce multiple entries in lookup tables.

Prior work has used lookup tables sparingly to manage storage demands. Forest Packing [7] restructures trees so that hot paths can be processed in one access to processor cache. Other paths are processed using standard, breadth-first tree traversal. Ranger [26] batches multiple inputs together for inference, accessing cache once to process multiple input data. However, inference workloads increasingly demand low response times and cannot wait to batch queries. Python Scikit-Learn, the most widely used platform for random forests, exploits the Intel MKL library for efficient layout optimizations for tree structures.

We present Bolt [20], a platform for fast inference in ensemble models. Bolt transforms fully trained random forests from an ensemble of decision trees to an ensemble of lookup tables. Unlike prior approaches, Bolt enumerates paths for every tree in a random forest and clusters similar paths. Redundant paths can be stored in the same lookup table entry. Bolt also partitions lookup tables to fit in processor cache by limiting the number of features and using parallel cores for concurrent access. We implemented Bolt in C++, transforming random forests trained using Python Scikit-Learn. The majority of our evaluation focuses on random forests, although we also tested deep forest workloads. Further, since Bolt does not affect training, techniques used by Bolt on random forests are generalizable to all decision tree ensembles. Using widely used machine learning datasets, we found that Bolt executes inference 2X faster than Forest Packing, the current state-of-the-art platform, and at least 14X faster than Python Scikit-Learn on ensembles consisting of shallow - i.e. limited height - decision trees.

Specifically, we make the following contributions:

- We present a method to manage storage and processing demands to transform random forests into ensembles of lookup tables using clustering and bloom filters.
- We present a method for exploring parameters, searching for inference latency given a forest and available cores.
- We present an efficient C++ implementation and evaluate these methods in terms of inference speed on multiple types of random forests and datasets.
- Our evaluation shows the strengths and weaknesses of our approach. Bolt's speedup over Forest Packing persists as ensembles grow to include more trees. However, Forest Packing provides faster inference as the height of trees within the ensemble increases.

The remainder of this paper is structured as follows: Section 2 describes related work. Section 3 provides a motivation for the alternative approach Bolt takes. Section 4 provides background on random forests and presents the design of Bolt. Section 5 presents our implementation of Bolt and a networked classification service. Section 6 presents empirical evaluations of Bolt, and Section 7 draws conclusions.

2 RELATED WORK

Explainable machine learning (ML) models allow humans to understand the factors that influence the models' output given input data [4, 12, 14]. Explain-ability is increasingly important when a models' output directly affects the cyber-physical world with little or no human intervention, e.g., automated edge and cloud workloads [3, 10, 16, 19], Internet of Things [29], and autonomous vehicles [6, 22]. Explainable models help humans audit and trust software-driven systems. In particular, decision trees, a ML model structured as a tree wherein each node represents a *test* on an feature of the input data [25], support innately explainable characterizations, e.g., salience maps [15, 17]. Random forests, ensembles of decision trees, are also explainable [4]. The ability to explain automated resource management decisions was a primary reason to employ tree-based models in recent research. For example, Maji et al. [16] use decision trees to detect the presence of cloud interference. Grohmann et al. [10] showed that random forests can provide accurate models of key performance indicators that can enable monitor-less performance management.

Random forests build each tree within an ensemble independently, normally by sub-sampling the training data and set of features available as internal nodes. Machine learning experts structure the trees, choosing parameters like the maximum height and thresholds for splitting nodes and/or declaring leaf nodes. The structure of trees within a random forest matters for explain-ability and classification accuracy. Explainable trees should respect the limits of the human brain [11]: fewer variables (i.e., shallow trees) are easier to explain. Gradient-boosted forests [8] and deep-learning forests [30] achieve state-of-the-art results for classification and regression accuracy. Theoretical studies suggest that these approaches benefit from forests composed from shallow trees [2]. Thus, establishing the importance of improving inference time for short trees.

2.1 Approaches to speed up inference

Forest Packing is the closest related work. Browne et al. also seek to speed up response time for AI inference services [7]. Their approach implicitly creates a partial lookup table by storing trees in depth-first order. Nodes in the same path are loaded into the same cache line and checked against input data. Paths are organized by how frequently they are accessed in testing data, prioritizing cache lines for hot paths. However, testing data may not reflect the statistical path distribution observed when a forest runs inference as a service. Test data is meant to broadly cover a wide range input data, whereas inference services target narrow use cases. For complex data used on a wide range of services, hot paths will likely differ. By explicitly mapping all paths into lookup tables, Bolt forests can cache whichever paths are used most frequently by a service. Another key difference is that Bolt does not follow pointers from node to node. By using a dictionary, Bolt reduces branch mispredictions and separates compute and cache capacity concerns. Further, Bolt is less dependent on system scheduling and instruction-level parallelism. While Browne et al. note that random forests support explainable inference, they do not evaluate local explanation workloads. Bolt uses associative arrays to track salient features. Bolt can do such tracking with one memory access per tree inference, meaning that Bolt can produce a list of salient features as inference is produced, but it seems a comparable implementation in Forest Packing would require an additional memory access for node. Besides this difference, Forest Packing also compresses nodes and uses cache efficiently during inference.

Furthermore, their approach seems more effective for deeper decision trees in which even partial lookup tables are hindered by the number of features.

Similarly, Ranger [26] also focuses on creating an efficient version of Random Forests. However, Ranger processes trees in a breadth-first order, and does not differ in principle from traditional tree execution, instead, it optimizes storage by "avoiding copies of the original data, saving node information in simple data structures and freeing memory early" [26]. Other optimizations of Ranger, apply to training time, rather than inference as a service. Consequently, Ranger performs inference at similar or only slightly better times than other libraries, such as Scikit-learn, that also perform tree inference through branching on each node. When using Ranger as a service, the absence of lookup tables hurts the performance, however, when batching queries Ranger can benefit from its optimizations and achieve very low response times.

Prior work has explored deterministic finite automaton (DFA) on custom hardware [23, 24] and for processing XML and JSON files [1]. These efforts share our goals of efficiently using processor resources for DFA workloads. Xie et al. optimize random forests for execution on GPUs, using similar concepts as Bolt, e.g., redundancy elimination [27]. While the two approaches target different types of processors and are not directly comparable, we note that both approaches achieve state of the art speedup with comparable absolute throughput for low-latency inference serving. The choice of GPU versus CPU depends on many contextual factors. However, Bolt and Tahoe [27] clarify some performance tradeoffs. Bolt includes unique implementation optimizations targeted at random forests. Design ideas for Bolt were published previously without substantial evaluation [21]. Joshi et al. [13] characterized the design of machine learning optimizations on the edge as levering model compression and/or conditional computation. Bolt employs both.

3 APPROACH

As described before, ensembles of decision trees are more accurate, while also slower and often redundant. Other solutions, approach the efficiency problems of ensemble models by maintaining frequently visited nodes in cache, thus profiting from redundancies, while paying a cache miss penalty in the remaining nodes. Mapping the tree into a lookup table, on the other hand, considers all possible paths of a tree, thus avoiding branching, but may not be faster than branching if the final lookup table is too large. Our approach profits from redundancy by first, merging identical paths across multiple trees, then clustering similar paths across trees - as opposed to adjacent paths in the same tree as other approaches - and during inference quickly discarding irrelevant groups of paths while maintaining storage requirements low.

For clarity, consider the case of a school bus which must drop off students in their respective homes. An inefficient approach would ask each student where to go, and drop them off before proceeding to the next student, without considering that some students might be neighbors. If dropping off a student is equivalent to loading a node in memory, this approach is equivalent to breadth-first processing, branching at every node. Alternatively, establishing a route that visits every street in the city, even those in which no students live, may route more efficiently and take advantage of the existence of neighbors. However, if there are many more streets than students, this may be slower than the first approach. If driving through a street is equivalent to testing if a path matches input features, this is equivalent to naïvely creating large lookup tables. Bolt mixes the two approaches, by doing the equivalent, in our metaphor to establishing a bus route, but quickly discarding streets in which no students live. Section 4 will formally describe the idea presented by this metaphor.

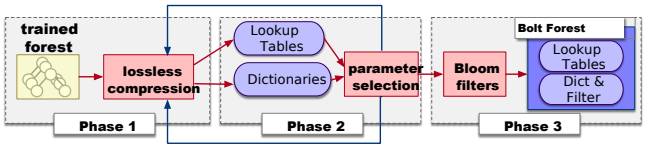


Figure 1: Bolt (1) compresses a given trained forest, (2) explores parameters to reduce storage demand and latency and (3) uses filters to reduce memory lookups.

Not only is Bolt more efficient because of the previously described approach but also, it allows for more flexibility with scaling. With Bolt's clustering of paths, a single sample can be parallelized across multiple nodes in a distributed system, by assigning tables to different processors. Other approaches allow for multiple samples to be executed in parallel or to execute groups of trees in the forest in different nodes. Bolt can still do the previous two parallelization methods while permitting further splitting of a single sample.

4 DESIGN

We focus on binary trees where nodes are features, and edges indicate boolean values associated with a feature and a threshold value (0 or 1). Given an input sample, at every node, the input feature is compared to the threshold-value, and an edge is taken accordingly. A matching path from root to leaf nodes means that each feature-value pair in the path is also present in the input data. Each tree has exactly one matching path for a given input. The terminal node (leaf node) of the path represents the inference result (classification). Results from each tree are aggregated to produce a final classification.

Bolt: Figure 1 presents our approach for processing forests that safely[1] transforms trees into lookup tables, manages storage demand and speeds up inference and explanation workloads. Bolt accepts three inputs: a trained forest, number of available CPU cores, and cache capacity of each core. The output is a collection of structures that we call *dictionaries*[2] and trees mapped into lookup tables ready for inference and corresponding to the original forest.

Bolt consists of three phases. Phase 1 splits the entire forest into several tree paths and clusters similar paths (among all trees) into tables to reduce storage demands. Phase 2 evaluates the outcome of Phase 1 against the expected size of each table (storage) and of each dictionary (latency) and searches for parameters that reduce inference latency. Phase 3 speeds up path matching by quickly filtering out tables that comprise only non-matching paths, thereby avoiding unnecessary memory accesses.

[1] Informally, *safety* means that transformations preserve classification results for all inputs.
[2] These are not traditional dictionaries in the sense of associative maps with $O(1)$ lookup.

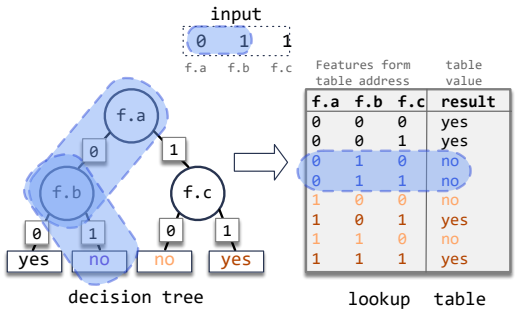


Figure 2: Lookup tables encode each path in a tree as an address that directly maps to a classification result. Features that are irrelevant for particular path in a tree must be considered in the table.

4.1 Phase 1: Clustering and Compression

Recall from Figure 2 that naïve mapping forms a lookup table address from *every* feature present in a tree and the possible values that feature can take on (here only 0 or 1). In that figure, even though the highlighted path formed by $(f_a, 0) \rightarrow (f_b, 1)$ does not include f_c, f_c still must be present in the address formed from the features, since addresses (*lookup table* indexes) are of fixed length. This results in two duplicate entries in the lookup table with the same result, where f_c is treated as a "don't care" in the address. This approach wastes space and inflates storage demand exponentially, since it necessitates 2^n table entries for n binary features, thus making lookup tables untenable for forests comprising complex trees. Note here that n comprises *all* distinct features used in *all* trees in the forest (not all trees use the same features), so it can grow quite quickly. Even for modest $n > 25$, the memory used by the lookup table (2^{25} entries $\simeq 32MB$) will easily outstrip the capacity of a modern machine, e.g. the 20MB L3 cache on an Intel Xeon E6-2620. We propose an alternative memory-mapping approach to manage storage demands. The key insight we leverage for our approach is that several trees within a forest may share paths, presenting an opportunity for compression.

Figure 3 shows how Bolt transforms an input random forest comprising two decision trees into a set of data structures we can use for fast, cache-friendly inference. First, paths (consisting of a series of feature-value pairs) for each tree in the forest are enumerated and sorted lexicographically (1). The sorted paths from the trees are then merged into a single, sorted list of paths *for the entire forest* (2). Clusters are formed by incrementally adding paths from this sorted list. This is done until a tunable threshold for the number of *uncommon* feature-value pairs is reached. In the example, $(b, 1)$ and $(h, 0)$ in the second and third rows in the table at (2) both represent pairs that have not yet been seen, so we add them to our first (green) cluster. We set our threshold to 2 in this example, so we cannot add more paths to this cluster. We begin a new (yellow) cluster and repeat the process. Note that this clustering threshold is a hyperparameter of our model, and it will be tuned during optimization (§4.2).

The important thing to note about the clusters is that, at this stage, each cluster will have its own *compressed* lookup table (as opposed to a single large table for the whole forest). We can see the path clusters superimposed on the original forest (3) in the third column of Figure 3. By design, *each* cluster shares a unique set of feature-value pairs common to all paths in that cluster. In the figure, $(a, 0)$ is common for the green cluster, $(a, 1)$ is common for the yellow cluster, and $(h, 1)$ is common for the blue cluster. The commonality of these features *within the cluster* allows us to extract them out and use them as an identifier that determines membership of inputs in cluster-specific lookup tables (5). This is accomplished using our "dictionary" data structure (4). When an input vector arrives, its features are compared against entries in the dictionary to match it to an appropriate lookup table. For example, an input of `0100` would match the first dictionary entry (storing the common pair $(a, 0)$). The lookup would then be directed to the first (green) lookup table. Note that now we only have ten lookup table entries and three dictionary entries; "don't care" entries are almost entirely eliminated. This is in contrast to the 16-entry table that would be used in the naïve approach, shown on the right side of the figure.

In general, this approach reduces demand significantly when the number of features in the tree grows. Let a full, binary, trained decision tree be given, and let h be its height. Let no node be repeated in different subtrees. A naïve lookup table of this tree would require 2^{2^h-1} entries. Let a dictionary extract the first k features from every path (assume $h > k$). Then, the dictionary would create 2^k lookup tables, each mapped in its corresponding dictionary entry. Each of these mappings would have $2^{2^{h-k}-1}$ entries in the table. Thus, the total entries in the table required by the dictionaries approach is $2^{2^{h-k}-1} \times 2^k = 2^{2^{h-k}+k-1} < 2^{2^h-1}$. Therefore, for any positive integers $h > 1, k$ where $h > k$, the dictionary approach leads to a smaller table. However, increasing the dictionary size also increases lookup latency, as each input must be compared to each entry of the dictionary (again, this is not a traditional dictionary in the sense of constant-time lookup). This size/latency trade-off is explored in the next section. After clustering, and compression, Bolt outputs a dictionary in which every entry maps to a unique lookup table. However, these tables must be recombined into one single table to help identifying false positives (more details in §4.3), and to avoid the use of pointers and the consequent branch misses. That is, Bolt hashes every entry in each of the lookup tables, corresponding to each dictionary entry, into one big recombined lookup table. These paths from the smaller lookup tables are hashed using their feature values and the entry ID corresponding to the dictionary entry that maps to the sub-table. After recombination, this stage has one dictionary and one lookup table for the entire forest. Each dictionary entry has an entry ID and a set of feature-value pairs, and the lookup table contains all paths in the forest.

4.2 Phase 2: Parameter Selection

As described in the previous section, Bolt makes use of dictionaries to reduce the storage demands of lookup tables. However, during inference each inputs must be compared to *every* entry in the dictionary. While this lookup does not require memory accesses and uses fast bit-wise operations in lieu of branching, a large number of dictionary entries can become the bottleneck during inference, particularly if the lookup table already fits in cache (fast memory accesses). Given the inverse proportionality between number of entries in the dictionary and the size of the lookup table, the clustering threshold

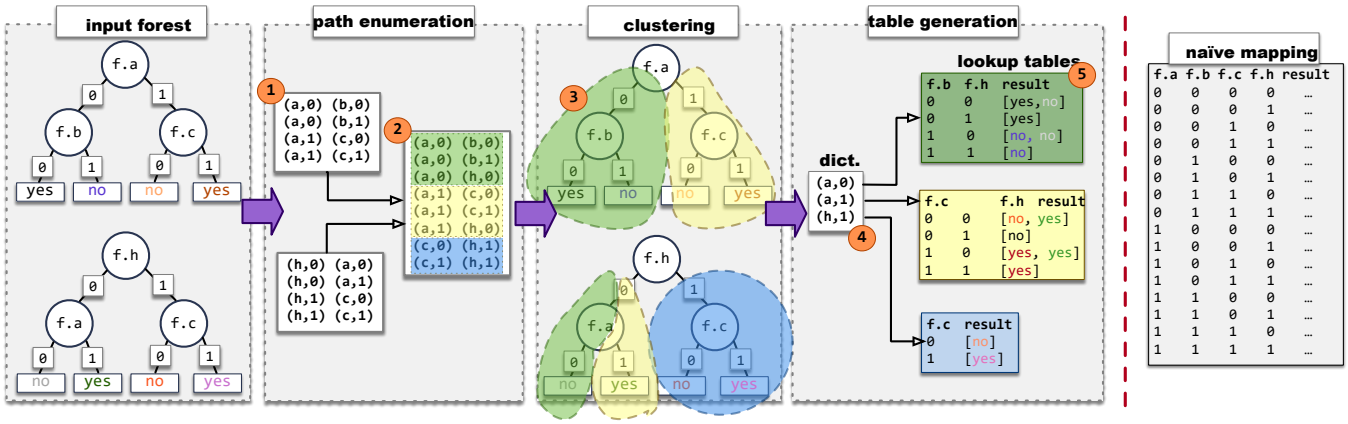


Figure 3: Compression of input forests in Bolt.

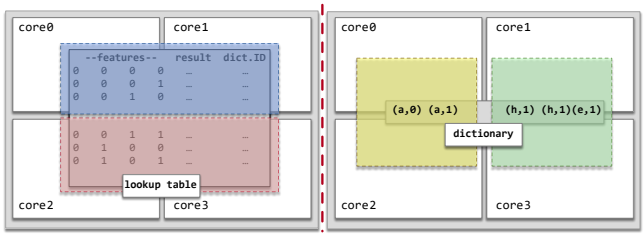


Figure 4: Four core example of parallelization on Bolt. This example shows two partitions of the lookup table in and two partitions of the dictionary.

described in the previous section, which controls dictionary size, must be carefully tuned.

However, tuning this parameter is not trivial. The ideal value depends on many factors, such as the number of paths in the forest with features in common, time consumed by a memory access, cache size, number of dictionary entries that would lead to a lookup in memory, among others. Furthermore, the possibility of scaling to multiple cores adds even more complexity. Figure 4 shows one possible division of a Bolt forest across four cores. In this example, both the lookup table and the dictionary are split into two partitions each. Partitioning lookup tables requires running two copies of the dictionary. For any input, a comparison is made with the key features of the dictionary entries. If there is a match, a memory lookup is attempted (a binary sequence is generated with the mapped features and the address is mapped to an index of the lookup table using the entry ID of the dictionary), if the address searched is within the partition of the lookup table that corresponds to that core, a result is computed. Lookup table partitioning decreases storage demand per individual core but may only indirectly affect latency. Dividing the lookup table only improves latency if cache misses have a big impact on performance of the specific workload. Figure 4 also shows the division of the dictionary. When the dictionary is partitioned, a copy of each lookup table is made. During inference, each core compares the input with the key features of the available dictionary entries, and performs the corresponding memory accesses. The partitioning of the dictionary directly impacts latency, but the overhead of aggregating results must be considered. Different values for inter-core communication latency and different methods for result aggregation can lead to different partition strategies. The combination of lookup table size, dictionary size, the number of splits of these data structures across cores, and the overhead of partitioning complicates modeling the ideal strategy given a workload. Bolt searches the space given by these parameters by running the forest with different parameter settings and selecting those partitioning strategies that lead to best results. Bolt can explore values within a given set of parameters, or given specific parameters, it can test the effect of small deviations from the given settings. In general, if the lookup table already fits in cache, parameter changes that lead to less dictionary entries will yield better results. Bolt explores different parameter strategies and outputs a set of lookup tables and dictionaries that give the best performance (latency) given a forest and the specified hardware.

4.3 Phase 3: Improving Lookup Table Selection

The use of dictionaries to compress a lookup table renders many entries in the dictionary irrelevant for a particular input. For the dictionaries to be effective in reducing latency, the decision to access a lookup table given the features in a dictionary entry must be a fast one. To solve this, Bolt uses bloom filters [5], a probabilistic data structure used to query set membership. Unlike perfect hashing that correctly labels inputs and non-members, bloom filters can report false positives; some non-members can be labeled members but members are never labeled as non-members (i.e. no false negatives). When non-members greatly outnumber members, like in a Bolt forest with many dictionary entries, bloom filters can afford fast, resource-lean membership lookups.

During inference, for every dictionary entry, Bolt uses bit-wise operations to simultaneously decide if the dictionary entry is relevant to the input, and compute the location of the lookup table that would be accessed if the dictionary entry is relevant to the input. As shown in Figure 3, Bolt dictionaries distinguish between common and uncommon features. Common features are those that are present with the same value (same node and same edge) in every path that was clustered into a particular entry in the dictionary, or

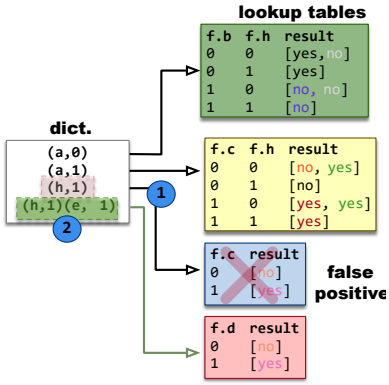


Figure 5: False positives.

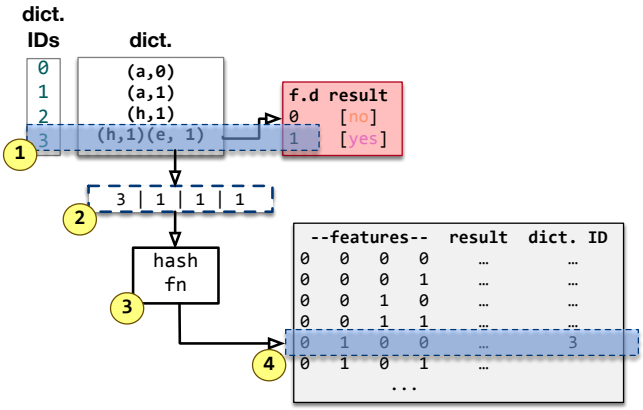


Figure 6: Recombining tables.

more precisely, into the lookup table, pointed at by the dictionary entry. Uncommon features are all other features that are present in any path that was clustered in that particular dictionary entry. Given input features, Bolt uses common and uncommon features to create a binary sequence representing a mapping of the input in the corresponding lookup table. Then, Bolt uses a bit-mask representing the common features to decide membership of the input in the table mapped by that entry. If the input matches the common features, then the generated sequence is hashed into the recombined lookup table described above and the response is retrieved; otherwise, the dictionary entry is ignored.

The approach described above suffices when the dictionary entries map entire subtrees. However, because Bolt allows for grouping of paths from different trees or subtrees, and because of the greedy algorithm for clustering, false positives are possible. As shown in Figure 5, a false positive in entry i of the dictionary implies the common features of entry i are a subset of the common features of some other entry j, and the entry j would lead to a correct memory access. The example on Figure 5 shows an expanded dictionary from Figure 3 in which an extra entry (2) has been added. This entry's common features are a superset of the previous entry's common features: $(f_h, 1)$. Therefore, any input that matches the bit-mask corresponding to (2) will also match a bit-mask for $(f_h, 1)$. To correct for this, Bolt uses the entry ID of the dictionary entry when hashing the binary sequence into the recombined lookup table.

Additionally, every table entry contains the entry ID of the dictionary entry that would map to that portion of the table. When a response is retrieved from the recombined lookup table, the entry ID of the dictionary which produced the lookup is compared to the entry ID in the lookup table. A response is only counted if there is a match.

Figure 6 illustrates the process of transforming the lookup tables of each dictionary entry into one combined table. Using the entry highlighted in (1), the entry ID and the values of all features (1) are used to hash into another table. During inference, an input that matches (1) will be mapped into (4), and upon comparison of the entry ID in (1) and (4), (both equal to 3) the lookup will be labeled a true positive. The same input, would also cause a lookup on the entry 2 of the dictionary, however, upon hashing,the lookup will be labeled as a false positive once it finds a mismatch in dictionary IDs.

Using the entry ID as part of the hashing function and matching such entry ID with the path found in the lookup table correctly identifies false positives. This extra step highlights the need for combining tables. Given the recombined lookup table, when erring on imposters, Bolt bloom filters only pay the penalty for one memory access, but after performing the memory access, Bolt can be certain about the relevance of the lookup made.

4.4 Correctness of Bloom Filters

Section 4.3 describes how bloom filters are used to achieve fast and correct memory accesses. Here, we present an argument for why the described approach is correct. Recall from the previous section that Bolt distinguishes common and uncommon features of dictionary entries. Thus, a true positive is generated by an input sample that matches all features in a path belonging to the dictionary entry, not just those that are common to the dictionary entry. While a false positive matches only the common features but does not fully match any path in the entry. Also, recall that during inference, a binary sequence is generated by comparing input features with features in the dictionary entry. The hashing function takes both this binary sequence and the dictionary entry ID as inputs, and outputs an address in the lookup table. Thus, the approach above is correct if true positives are always hashed to the unique path with which all features are common and false positives are hashed into lookup table entries that are not marked by the entry ID of the dictionary entry that triggered the lookup.

The first condition above follows from (a) the definition of true positive and (b) the path expansion described in Section 4.1. Note that (a) does not suffice since the binary address produced by input evaluation may be longer than the original path in the decision tree. However, since all paths in a dictionary entry are expanded in the lookup table to include all possible values of irrelevant features, one of the lookup table entries must match input. Thus, for a true positive input, the binary address will match that of the desired path and the entry ID will also be identical. Since those are the only two inputs to the hashing function, the correct lookup table entry will be found.

The second condition - that false positives are not hashed into lookup table entries with the same entry ID, depends on the number of conflicts of the hashing function. If many distinct paths are mapped to the same location, then the probability that a false positive will not be identified is increased. However, it is already a requirement that the final lookup table not have any conflicts so that entries can be accessed quickly. Further, by design, the final lookup table is much larger than the tables in dictionary entries. At a minimum the final lookup table must be $2^{\lceil \log_2 p \rceil}$ where p is the total number of paths to be assigned to the table. Typically this would be much larger than any individual dictionary entry. The probability of incorrectly accounting for a false positive is the same as the probability of having two distinct binary sequences mapping into the same location, while using the same entry ID and the same hashing function. Further, the probability of having a false positive is already restricted by the probability of having two dictionary entries where one set of common features is a subset of the other. Even for small ensemble models, we found this to be unlikely.

4.5 AI Inference with Bolt Forests

Figure 7 depicts the workflow for an inference service that uses Bolt forests. Before the inference begins, the processing engine has access to a dictionary in which each entry corresponds to a group generated during compression (one of the small lookup tables). Each entry in the dictionary has a list of feature-value pairs present in the paths of that dictionary entry. Additionally, the processing engine has access to a lookup table (the recombined lookup table) that contains results corresponding to each path (its data) and the dictionary entry ID of the dictionary entry that should map to that result (used to identify false positives during probabilistic hashing). Input data is sent via network to a front-end. The front-end calls the inference processing engine of each core. The inference processing engine of each core iterates over all dictionary entries. The processing engine performs hashing using bloom filters as described above to avoid unneeded memory lookups. Dictionary entries with matching values on common features trigger results lookup. The location of the lookup table given by the bloom filter is only accessed if the corresponding entry is located in the partition of the lookup table that corresponds to the current core. Recall from Figure 4 that cores may only have access to a portion of the dictionary and a portion of the lookup table. Therefore, it is possible that the dictionary on one core leads to a hashed location not contained in the core's lookup table.

If a dictionary entry on a core leads to a portion of the lookup table not in said core, the dictionary entry is ignored by the current core. This does not imply loss of accuracy because, if this happens, another core is guaranteed to produce a match, so the result will be counted in aggregation. The guarantee comes from the duplication of the lookup table upon splitting the dictionary or viceversa. Formally, consider the following: Let the dictionary be split in $d_1, ..., d_n$ partitions and the lookup table in $t_1, ..., t_m$ partitions. Let there be $C = nxm$ available cores. Suppose in some core c_k with partitions d_i and t_j an input matches a dictionary entry located in d_i which points to a location in partition t_p where $j \neq p$. In this case, core c_k can safely discard the lookup because there exists a core c_q with partitions d_i and t_p which will add the appropriate result. If the result

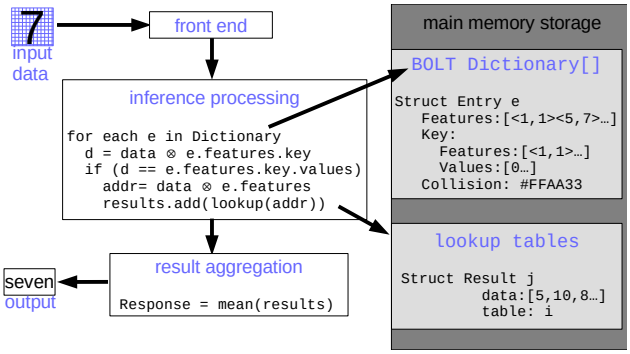


Figure 7: Workflow for AI inference using Bolt forest. This example shows digit recognition where input data is a 28 x 28 image.

is looked up in the correct partition, a check that the dictionary entry ID stored in the table matches the dictionary entry that prompted the lookup is performed. If the two ID numbers do not match, the result is discarded as a false positive. If the ID matches, then, the result of that lookup is saved. After looking up all relevant results (paths) in all cores, the results are aggregated into the final output. Note, Bolt forests provide one implementation. Alternatively, the front-end can connect to other forest implementations. In this paper, we hypothesize that Bolt forests will reduce processing time, measured between front-end receipt and aggregation output.

4.6 Discussion

Bolt uses novel data structures, i.e., dictionary entries with embedded filters and lookup tables, to speed up random forest workloads. Figure 7 depicts workflow for a standard random forest where matching paths contribute equally to the output. In Section 6.3, we will extend this infrastructure to support complex forests. For example, deep forests [30] use multiple layers of forests to improve accuracy. Bolt can be applied to each layer to speed up processing.

Bolt restructures random forest workloads to (1) achieve strong cache locality (via compression, predictable dictionary access patterns, and fewer main memory accesses), (2) avoid branch mispredictions (via bit mask operations and eschewing breadth-first traversal) and (3) make full use of hardware available (via exploring the space of parameters to yield minimal latency). Additionally, for a given forest and hardware, Bolt can diagnose bottlenecks caused by limited LLC cache capacity and too slow architectural processing speed (GHz). The former occurs when storage demand exceeds cache capacity; the latter when clustering yields too many entries in the dictionary. This analysis makes Bolt useful for capacity planning problems, such as, given a forest workload, which processor provides best performance and trend analysis; such as, given future processors, what forests will achieve fast inference and local explanation.

5 IMPLEMENTATION

The MNIST data set trained using Python Scikit-learn with 100 constituent trees can yield over 5M leaf-node results. Verbose data layouts for lookup table and dictionary entries can inflate storage

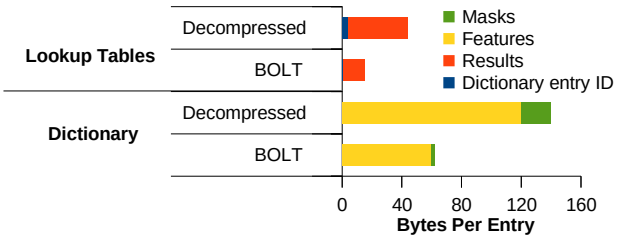


Figure 8: Our implementation compresses memory-mapped data structures to reduce storage demand. Results shown are for the MNIST data set.

demands. We implemented efficient data layouts for Bolt that (1) compress data in memory and (2) use fast, bit-level operations for decompression.

We implemented tools for parameter exploration that lead to parsimonious storage demand and optimize latency given hardware and a trained forest. Our tools iterated over three parameters: Threshold of uncommon features per entry, table partitions and dictionary partitions. The threshold of uncommon features dictates the size of the final lookup table and of the dictionary. Table partitions dictate how many cores will be used to reduce memory demand, and dictionary partitions indicate how many cores will be dedicated to checking the dictionary entries. The final number of cores must be $t \times d$, where t is the number of lookup table partitions and d is the number of dictionary partitions.

In all experiments we used Python Scikit-Learn to train the forests. Given a trained tree, we converted each tree in the forest to DOT files [9], an edge-oriented textual layout. To be sure, Python Scikit-Learn produces binary trees by default. We implemented tools that extracted paths from root to leaf nodes from the DOT files. We then clustered similar paths and expanded them to create a dictionary and lookup tables. At this stage, our tools discovered key properties of the trained forest:

- *Largest value used in binary split:* While features on different datasets can take on any value, information gain is often maximized by splitting on relatively smaller values. Dictionary entries reserved only enough bits for feature values to represent the maximum value used in a split. Recall each dictionary entry must store multiple feature-value pairs to capture paths from root to leaf node. Compared to naïvely using integers to represent features and values, this approach can save 80 bytes per entry on MNIST (see Figure 8), for example. For other datasets, normalization and other small adjustments can be used to achieve the above result. For instance, the LSTW dataset [18] contains location data for traffic incidents. North and South coordinates only have a range of values that span 180 degrees, so, by shifting the scale (from [-90,90] to [0, 180], all of the information can be stored in one byte without losing prediction power.

- *The largest feature set across all dictionary entries:* Our implementation creates bitmaps to capture key features and their corresponding binary value. This property determines the right size of bitmasks. Figure 8 compares against the simple approach of using Boolean arrays (1 byte) to implement masks.

- *99^{th} percentile results value:* Most results can be represented with few bits, but a few results can require many bits. When storing results in the lookup table, we eschewed standard integer data types that, while large enough to represent all results, often wasted precious bits. Our scripts found knee-points; a number of bits that represented a large fraction of the results. The typical result was represented using those knee-points. Atypical results used additional space. This approach compressed table entries by 3X.

- *Dictionary entry ID:* When storing dictionary entry ID in the lookup tables, some optimizations were possible due to the specific features of the bloom filter that we used. Since every tree was binary, we were able to perform all operations using integers and performing bit-wise operations. Thus, as described in the previous section, when comparing an input sample with all the feature-value pairs in a dictionary entry, the result is a binary address (sample address). The generated binary sequence is encoded according to the Boolean value of the original root-leaf path of the tree (lookup address). This address can be interpreted as an integer, thus an index of the lookup table. The properties of this encoding are exploited by reducing the entry ID in each lookup table's entry to one byte per ID. Recall from Section **??** that true positives cannot be missed by our approach and that false negatives have a low probability of being incorrectly considered. Also, recall from Section 4.3 that false negatives only occur if the dictionary entry's common features for the entry that produced the lookup are a subset of some other entry, and the latter is the entry in which a correct lookup would be triggered. Finally, considering our clustering approach described in Section 4.1, entries in which common features are subsets of each other are likely to be adjacent dictionary entries. Thus, a false negative only exist in dictionary entries adjacent to those in which a true positive would occur. So, distinguishing between adjacent dictionary entries in the lookup table is more important than uniquely identifying dictionary entries. Therefore, the entry ID stored by the table in our implementation is just one byte (mod 256 of the original ID). This helps with storage demands and the probability of error is kept low.

Bolt for Complex Forest Structures: Recent research has shown that complex forest structures can improve classification accuracy. Gradient-boosted trees, e.g., XG-Boost [8], apply weights to trees within a forest. Bolt does not affect the training process and thus can support gradient-boosting by simply adding the corresponding tree weight to each path.

Deep forests, e.g., gcForest [30], use multiple layers of random forests to learn "deep" concepts that translate to improved final accuracy. Precisely, the output of each layer is appended as a feature for subsequent layers. We implemented multi-layer deep forests in Bolt. We compress each layer in isolation, creating a lookup table and a dictionary. Since the output of latter layers depends on previous layers, the dictionaries can be loaded sequentially. Features passed from previous layers are appended to input data.

6 EVALUATION

In this section, we compare Bolt to competing forest platforms. Our evaluation uses Python scripts for front-end processing. The front-end communicates to inference processing engines on a UNIX domain socket. Input samples are executed sequentially without

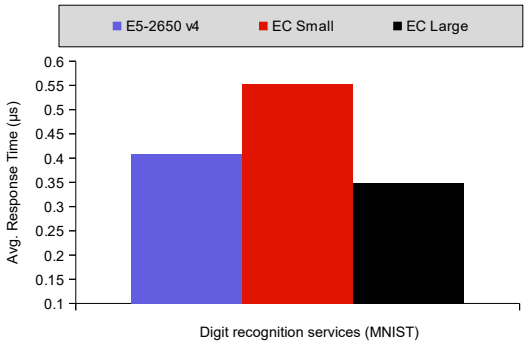


Figure 9: Bolt execution times across different architectures.

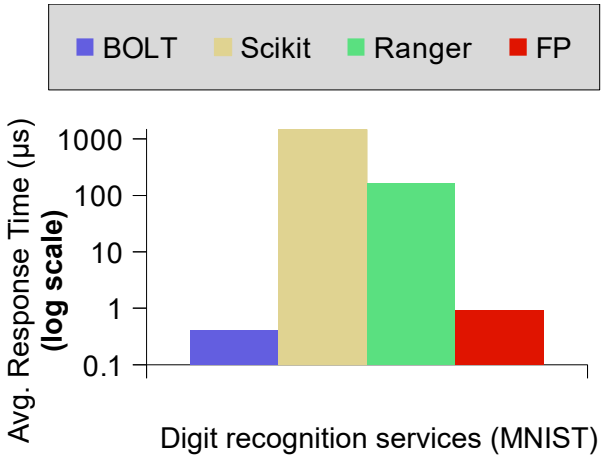


Figure 10: Bolt perform better than competing approaches for small Random Forests

batching and without parallelizing across samples, i.e. all cores execute at least partially, every input sample. Time is measured from the time input samples are received to the moment inference finishes, not including network delays. Bolt is implemented in C/C++, allowing for low level control of data layout and bit operations. All forests in Bolt experiments are trained, and evaluated using Scikit-Learn. Forests are then compressed by Bolt and evaluated independently. The code made available by Forest Packing does not allow for arbitrary forests as input. Instead, Forest Packing accepts pre-set forests already included in their code. For comparison across number of trees and tree depth, we artificially limit the number of trees and nodes in Forest Packing by stopping after reaching the tree setting.

6.1 Datasets

We tested all of our models on MNIST, LSTW [18], and the Yelp Restaurant Review Dataset [28]. MNIST is a common digit recognition workload. Input data to MNIST are 28×28 images of handwritten digits. Each pixel is a feature (784 in total). The output classifies the image as a digit (0–9). MNIST consists of 60000 images used for training and 10000 images used for testing. All inference results for MNIST are reported on the performance of Bolt on the 10000 testing samples. Additionally, the Large-Scale Traffic and Weather Events (LSTW) dataset was used [18]. Input data in LSTW is heterogeneous, including numeric and categorical features related to traffic conditions. LSTW has 11 input features. The output is a categorical assessment of traffic conditions. In general, LSTW requires more data (25M samples) and is harder to predict than MNIST (70K samples). For prediction in LSTW, we used 700000 testing samples, while the rest was used for training. Finally, we used the Yelp Restaurant Review Dataset [28]. This is a Natural Language Dataset with 5200000 user reviews on 174000 businesses from 11 metropolitan areas. For classification, we used the review text as input and the number of stars in the review as output. The text field was processed to remove stop words, take word stems and tokenize into a vector of 1500 features indicating number of appearances of each of the most common 1500 words. Thus, the final input was 1500 numerical features and the prediction target is one numerical feature indicating number of stars of the review.

6.2 Hardware

We tested Bolt on multiple processors with varying cache size and clock frequency. Unless noted otherwise, we used an Intel Xeon(R) E5-2650 v4 @ 2.20GHz with 12 cores, 30 MB of LLC and 132 GB of memory. We also used 2 Google Cloud instances. (1) E2-standard-4 with 4 vCPU's and 16 GB of memory (EC Small), and (2) E2-standard-32 with 32 vCPU's and 128 GB of memory (EC Large), both running Debian GNU/Linux 10. Unless noted otherwise, we report response time with single-threaded Bolt executing on one core. Our default platform runs stock Linux kernel version 3.10.0, Python 3.6.8 and Scikit-Learn 20.

6.3 Results

First, we tested Bolt's performance on one small Random Forest with 10 individual decision trees and a maximum height set to 4. This forest was tested on MNIST's 10000 samples. The reported time is the average response time in microseconds of all samples. On this forest, Figure 9 shows how Bolt achieved a response time in the hundreds of nanoseconds in all three platforms.

Next, we compared the performance of Bolt with alternative approaches on the same small Random Forests. Figure 10 shows the execution time of Bolt, Scikit-Learn, Ranger and Forest Packing on MNIST given a modest forest with 10 trees and a maximum height of 4, when running on one core on our default server. As shown in Figure 10 Bolt can process samples in an average time of $0.4\mu s$ against the $0.9\mu s$ of Forest Packing, while Scikit-Learn achieves $1460\mu s$ and Ranger $160\mu s$. Thus, Bolt outperforms state-of-the-art models for small forests by a factor of at least 2×.

Figure 11 shows a performance comparison of all models on our default servers when the size of the ensemble model varies. Figure 11 (A) shows the effect of increasing maximum depth of each tree, while maintaining the number of trees in the forest constant. The x-axis shows the maximum tree height setting, while the y-axis shows the average response time. All forests have 10 trees. Bolt outperforms Scikit-Learn and Ranger by several orders of magnitude

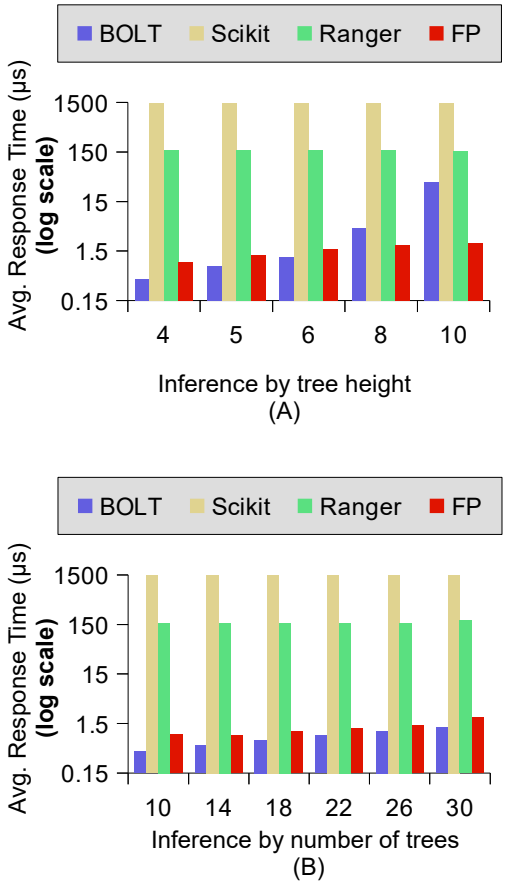


Figure 11: Effects of increasing tree size in all models

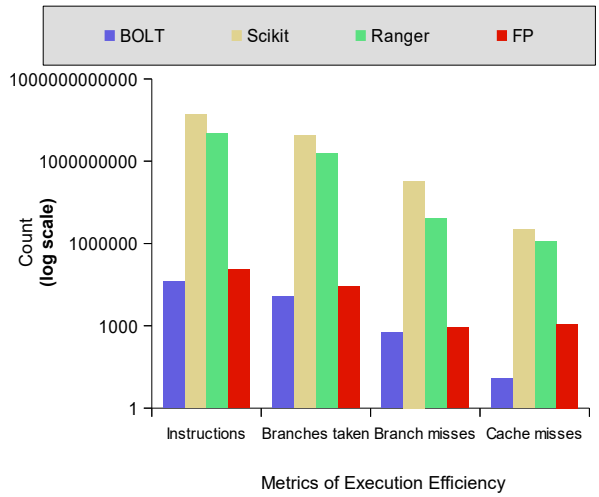


Figure 12: Execution metrics executing MNIST

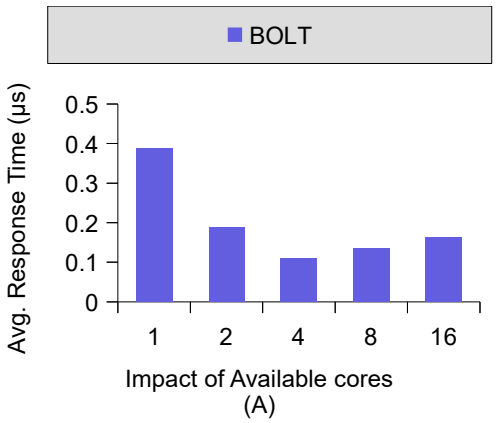
on all settings. However, Bolt outperforms Forest Packing on heights up to 8. For larger forests, the lookup tables and the dictionaries can become too large. Even in these scenarios, Bolt outperforms other models, but Forest Packing still performs better on deeper trees.

On the other hand, Figure 11 (B) shows a similar comparison of different forest sizes, however, in this case, the number of individual trees in the forest grows while maintaining maximum height constant. In Figure 11 (B) Bolt and Forest Packing outperform Scikit-Learn and Ranger by multiple orders of magnitude, while Bolt outperforms Forest Packing in all settings. Bolt achieves a average response times of 0.4, 0.5, 0.7, 0.9, 1 and 1.2 microseconds on all settings in Figure 11 (B) respectively, while Forest Packing achieves average response times of 0.9, 0.9, 1, 1.1, 1.3 and 1.9 microseconds. This result not only shows that Bolt outperforms other state-of-the-art models, but together with Figure 11 (A), it helps clarify the reasons for the slowdown in Figure 11 (A). When the number of trees increases, the total paths also increase, however they increase linearly. In contrast, when the length of each path increases, a full mapping of this path would increase exponentially. While Bolt's data structures mitigate this increase in memory requirement, tree mapping approaches such a Bolt are still more affected by depth of each tree than number of trees.

Continuing the analysis of the causes of Bolt's improved performance, Figure 12 shows performance metrics that illustrate Bolt's effectiveness in managing cache and branching. Figure 12 shows the total instructions, branches taken, branch misses and cache misses on Bolt, Scikit-Learn, Ranger and Forest Packing, when running all 10000 samples of MNIST in a forest with 10 trees and a max depth of 4.

First, Bolt executes MNIST inference in the test set with 40000 instructions versus the 110000 of Forest Packing and the orders of magnitude higher instruction counts of Scikit-Learn and Ranger. Thus, achieving close to a 3× improvement over Forest Packing. Further, Bolt also reduces the total number of branches, thus showing the efficacy of alternative data structures such as Bloom Filters to avoid branching at every node. Bolt achieves a 2× reduction in branches taken with respect to Forest Packing, and 6 orders of magnitude with respect to Scikit-Learn. The reduction of total branches directly affects branch misses, and consequently, Bolt also achieves a lower count of branch misses (556) than all other models. A notable result, is that the percentage of branch misses with respect to branches taken, is highest in Bolt among all other models, despite still maintaining a lower total of branch misses. For instance, Scikit-Learn misses 2.2% of the branches taken, while Bolt has a 4.6% branch miss percentage. This increase could indicate that Bolt might benefit from a better, or more suitable branch predictor, increasing further its advantages. Finally, Figure 12 shows the total cache misses during inference. While both Scikit-Learn and Ranger had cache misses in the order of millions, and Forest Packing was in the order of 1000 cache misses, on this setting, Bolt was able to achieve under 20 cache misses. This result, further highlights the importance of Bolt's data structures to prevent the excessive growth of the path tables and dictionaries, as well as the importance of carefully tuning hyperparameters to achieve the ideal settings.

Such hyperparameter tuning is explored in Figure 13. Here, Figure 13 (A) shows performance times of Bolt when parallelizing



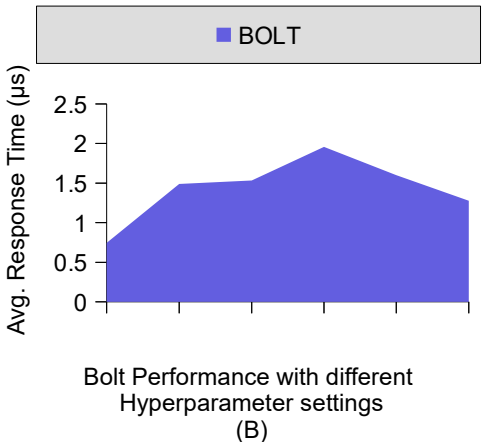


Figure 13: Effect of hyperparameters on Bolt

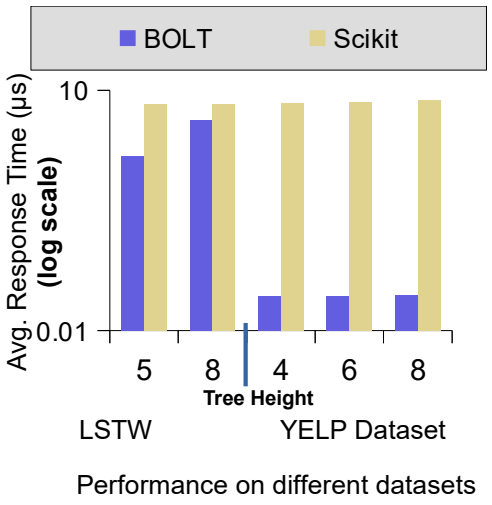


Figure 14: Bolt performance by dataset

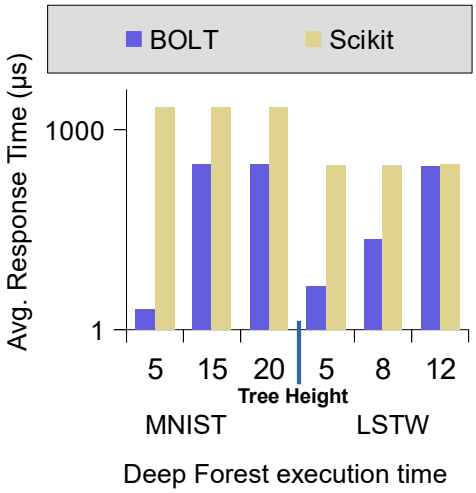


Figure 15: Deep Forest exceution

across different numbers of cores for a modest random forest. As previously discussed, Bolt can parallelize not just by executing multiple trees in different nodes, or different samples in different nodes, but also, a single sample on a single tree can be executed in parallel thanks to the splitting of data structures. Here, parallelization is done by exploring combinations of dictionary and lookup table splits as discussed in Section 4. As demonstrated in Figure 13 (A) parallelization can linearly reduce the execution time previously presented, up to at least 4 cores for a small forest. Bolt can execute the MNIST testing set in 4 cores in an average time of 0.11μ per sample. In this setting, when parallelizing across 8 cores or more, the overhead offsets the advantages of splitting the dictionary and the lookup tables. For small forests, generally, splitting the dictionary tends to be more effective than splitting the lookup tables, since the size of the lookup tables is small enough to fit in cache.

Figure 13 (B) shows some execution times of Bolt when arbitrarily setting both thresholds for dictionary and lookup table sizes discussed in Section 4. As seen in Figure 13 (B), the execution time can vary by as much as 4× based on these parameters. This highlights the importance of Bolt's hyperparameter exploration phase, and the careful tuning of table and dictionary sizes.

Next we show the execution times of Bolt on different datasets representing a variety of workloads. Figure 14 shows the effectiveness of Bolt's approach on heterogenous datasets. We compare Bolt's performance with that of Scikit-Learn on these datasets. On the left, we see the performance of LSTW, while on the right, we see the YELP dataset. On both of these datasets, Bolt achieves a under microsecond average response times for modest forest.

Finally, Figure 15 compares the performance of Bolt when implementing a deep forest [30] compared to Scikit-Learn. The left side of shows the performance of a deep forest on MNIST while the right side shows LSTW. All of this forests are two layer deep forests, in which the output of one forest is passed to the second forest. Both forest also have identical number of trees and maximum depth, differing only in the number of input features. The executions times are higher than in Random Forests as the time to copy over

the results and run two forests, however, we still observe single digit microseconds execution times for modest forests. As before, Bolt is affected by the depth of the trees, however, it still outperforms Scikit-Learn on all deep forests.

7 CONCLUSION

Decision trees can be transformed to lookup tables to speed up inference, but the drawback is an exponential increase in storage demands. Random forests exacerbate storage demands by using ensembles of multiple trees to boost prediction accuracy. However, they also present an opportunity: random forests benefit from shallow trees that use few features. Further, trees in random forests often contain redundant paths that can be combined in lookup tables. Combined with growing processor caches, many core parallelism, and the growing demand for low-latency microsecond inference speeds, we contend that lookup tables should be revisited. In this paper, we presented Bolt, a platform that transforms trained random forests into ensembles of lookup tables. The key insight is to enumerate all paths across the random forest, cluster them, and create lookup tables that consolidate redundant paths. Bolt manages remaining storage demand using a novel combination of bloom filters, optimization, lossless compression, and parallelization. It achieves faster inference speeds than competing state-of-the-art platforms for random forest inference and its speedups persist as ensembles grown in the number of trees. Bolt is not the fastest platform for all forests. In particular, Bolt's latency degrades significantly as tree height increases, because storage demands can not be mitigated.

REFERENCES

[1] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A Scalable in-SRAM Architecture for Pushdown Automata. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-51)*. October 20 - 24, 2018, Fukuoka, Japan. IEEE Press, Piscataway, New Jersey, 921–932. https://doi.org/10.1109/MICRO.2018.00079

[2] Ludovic Arnould, Claire Boyer, and Erwan Scornet. 2020. Analyzing the tree-layer structure of Deep Forests. (2020). arXiv:2010.15690. Retrieved from https://arxiv.org/abs/2010.15690

[3] Yogesh D Barve, Shashank Shekhar, Ajay Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. 2019. FECBench: A holistic interference-aware approach for application performance modeling. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. June 24 - 27, 2019, Prague, Czech Republic. IEEE Press, Piscataway, New Jersey, 211–221. https://doi.org/10.1109/IC2E.2019.00035

[4] Umang Bhatt, Alice Xiang, Shubham Sharma, Adrian Weller, Ankur Taly, Yunhan Jia, Joydeep Ghosh, Ruchir Puri, José M. F. Moura, and Peter Eckersley. 2020. Explainable Machine Learning in Deployment. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency (FAT* '20)*. January 27 – 30, 2020, Barcelona, Spain. Association for Computing Machinery, New York, NY, USA, 648–657. https://doi.org/10.1145/3351095.3375624

[5] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (Jul. 1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Jayson G. Boubin, Naveen T. R. Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. 2019. Managing Edge Resources for Fully Autonomous Aerial Systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing (SEC '19)*. November 7 - 9, 2019, Arlington, Virginia. Association for Computing Machinery, New York, NY, USA, 74–87. https://doi.org/10.1145/3318216.3363306

[7] James Browne, Disa Mhembere, Tyler M Tomita, Joshua T Vogelstein, and Randal Burns. 2019. Forest Packing: Fast Parallel, Decision Forests. In *Proceedings of the 2019 SIAM International Conference on Data Mining (SDM)*. May 2 - 4, 2019, Calgary, Canada. SIAM, Philadelphia, PA, 46–54. https://doi.org/10.1137/1.9781611975673.6

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. August 13 - 17, 2016, San Francisco, California. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[9] Emden Gansner, Eleftherios Koutsofios, and Stephen North. 2006. Drawing graphs with dot. Technical report, AT&T Research, Murray Hill, NJ.

[10] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. December 8–13, 2019, Davis, CA. Association for Computing Machinery, New York, NY, USA, 149–162. https://doi.org/10.1145/3361525.3361543

[11] Graeme S Halford, Rosemary Baker, Julie E McCredden, and John D Bain. 2005. How many variables can humans process? *Psychological science* 16, 1 (2005), 70–76. https://doi.org/10.1111/j.0956-7976.2005.00782.x

[12] Seunghoon Hong, Tackgeun You, Suha Kwak, and Bohyung Han. 2015. Online Tracking by Learning Discriminative Saliency Map with Convolutional Neural Network. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. July 06 - 11, Lille, France. JMLR.org, 597–606. https://doi.org/10.5555/3045118.3045183

[13] Praveen Joshi, Haithem Afli, Mohammed Hasanuzzaman, Chandra Thapa, and Ted Scully. 2022. Enabling Deep Learning for All-in EDGE paradigm. (2022). arXiv:2204.03326. Retrieved from https://arxiv.org/abs/2204.03326

[14] Emre Kıcıman and Matthew Richardson. 2015. Towards Decision Support and Goal Achievement: Identifying Action-Outcome Relationships From Social Media. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. August 10 - 13, 2015, Sydney, Australia. Association for Computing Machinery, New York, NY, USA, 547–556. https://doi.org/10.1145/2783258.2783310

[15] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence* 2, 1 (2020), 2522–5839. https://doi.org/10.1038/s42256-019-0138-9

[16] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating Interference in Cloud Services by Middleware Reconfiguration. In *Proceedings of the 15th International Middleware Conference (Middleware '14)*. December 8 - 12, 2014, Bordeaux, France. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/2663165.2663330

[17] Ioannis Mollas, Grigorios Tsoumakas, and Nick Bassiliades. 2019. LionForests: Local Interpretation of Random Forests through Path Selection. (2019). arXiv:1911.08780 Retrieved from http://arxiv.org/abs/1911.08780

[18] Sobhan Moosavi, Mohammad Hossein Samavatian, Arnab Nandi, Srinivasan Parthasarathy, and Rajiv Ramnath. 2019. Short and Long-Term Pattern Discovery Over Large-Scale Geo-Spatiotemporal Data. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (KDD '19)*. August 4 – 8, 2019, Anchorage, AK. Association for Computing Machinery, New York, NY, USA, 2905–2913. https://doi.org/10.1145/3292500.3330755

[19] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley. 2018. Model-Driven Computational Sprinting. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. April 23 26, 2018, Porto, Portugal. Association for Computing Machinery, New York, NY, USA, Article 38, 13 pages. https://doi.org/10.1145/3190508.3190543

[20] Eduardo Romero. 2022. *Bolt*. Retrieved April 14, 2022 from https://github.com/EduardoRomero83/CADM

[21] Eduardo Romero-Gainza, Christopher Stewart, Angela Li, Kyle Hale, and Nathaniel Morris. 2021. Memory Mapping and Parallelizing Random Forests for Speed and Cache Efficiency. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21)*. August 9 - 12, 2021, Lemont, IL. Association for Computing Machinery, New York, NY, USA, Article 29, 5 pages. https://doi.org/10.1145/3458744.3474052

[22] Christopher Stewart, Deepak Vasisht, and Weisong Shi. 2021. Toward Fully Autonomous and Networked Vehicles. *IEEE Internet Computing* 25, 3 (May 2021), 32–33. https://doi.org/10.1109/MIC.2021.3078346

[23] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards Machine Learning on the Automata Processor. In *High Performance Computing*. Springer International Publishing, New York, NY, USA, 200–218. https://doi.org/10.1007/978-3-319-41321-1_11

[24] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. September 30 - October 2, 2018, Raleigh, North Carolina, USA. IEEE Press, 13–24. https://doi.org/10.1109/IISWC.2018.8573482

[25] Wikipedia. 2021. Wikipedia Decision Tree The Free Encyclopedia. Retrieved May 19, 2022 from https://en.wikipedia.org/wiki/Decision_tree.

[26] Marvin N. Wright and Andreas Ziegler. 2017. ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software* 77, 1 (2017), 1–17. https://doi.org/10.18637/jss.v077.i01

[27] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: Tree Structure-Aware High Performance Inference Engine for Decision Tree Ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. April 26 - 28, 2021, Online Event, United Kingdom. Association for Computing Machinery, New York, NY, USA, 426–440. https://doi.org/10.1145/3447786.3456251

[28] Yelp. 2014. *Yelp Dataset version 6*. Retrieved January 10, 2022 from https://www.kaggle.com/yelp-dataset/yelp-dataset/version/6

[29] Zichen Zhang, Jayson Boubin, Christopher Stewart, and Sami Khanal. 2020. Whole-Field Reinforcement Learning: A Fully Autonomous Aerial Scouting Method for Precision Agriculture. *Sensors* 20, 22 (2020). https://doi.org/10.3390/s20226585

[30] Zhi-Hua Zhou and Ji Feng. 2017. Deep Forest: Towards An Alternative to Deep Neural Networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. August 19 - 25, 2017, Melbourne, Australia. IJCAI, 3553–3559. https://doi.org/10.24963/ijcai.2017/497