MABSplit: Faster Forest Training Using Multi-Armed Bandits

Anonymous Author(s)
Affiliation
Address
email

Abstract

Ensemble learning methods such as random forest are some of the most widely used machine learning models, especially in domains that necessitate interpretability. We present an algorithm that accelerates the training of random forest and other popular tree-based learning methods. At the core of our algorithm is a novel and fast node-splitting subroutine, dubbed MABSplit, used to efficiently find split points when constructing decision trees. Our algorithm borrows techniques from the multi-armed bandit literature to judiciously determine how to allocate computational power across potential split points. We provide theoretical guarantees that MABSplit improves the computational complexity from linear to logarithmic in the number of data points. Even on small datasets such as MNIST, our algorithm leads to 7x faster training (an 85% reduction in training time) without any decrease in test accuracy. We demonstrate similar speedups when the MABSplit subroutine is used across a variety of forest-based variants, such as Extremely Random Forests and Random Patches. We also show our algorithm can be used in both classification and regression tasks. Finally, MABSplit outperforms existing methods in test performance and feature importance calculations under a fixed computational budget.

1 Introduction

The Random Forest (RF) algorithm is a supervised learning technique that is widely used for classification and regression tasks [21, 11]. In RF, an ensemble of decision trees (DTs) is trained for the same prediction task. Typically, *bootstrapping* [10] is used to generate multiple sample sets from a dataset, each of which is used to train a separate DT. Moreover, in the process of training each DT, a random subset of the features is considered at each candidate split. The prediction of the different DTs is then aggregated to provide a single prediction. Common aggregation schemes include majority voting in classification tasks and averaging in regression tasks. By aggregating the prediction of the different DTs in the ensemble, RFs tend to be more robust against noise and overfitting [41] and are capable of capturing multiple patterns in the data, as opposed to a single DT [44].

RF has gained tremendous popularity due to its flexibility, usefulness in multi-class classification and regression tasks, high performance across a broad range of data types, and relatively low computational complexity [46, 42, 8, 27]. Furthermore, RF inherits the interpretability of decision trees, as the prediction of each individual DT can be explained through a sequence of binary decisions. RFs have been successfully applied in contexts as varied as the prediction of legal court decisions [24], solar radiation analysis [8], and the Higgs Boson classification problem [2]. As we move into the era of big data, a simple and flexible machine learning technique such as RF is expected to play a key role in processing large datasets and providing accurate and interpretable predictions.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

The need for training prediction models on massive datasets and doing so on compute-constrained hardware (such as smartphones and Internet-of-Things devices) requires the development of new algorithms that can deliver faster performance without sacrificing results [46]. For this reason, recent work has proposed ways to accelerate the training of RFs, both at the algorithmic level and at the hardware level. At the algorithmic level, the focus is on the fast construction of each individual decision tree. A decision tree is built by identifying a feature f and a threshold t that best split the data points according to the prediction targets. The data points are split based on f and t, and the process is repeated for each resulting subset. The key computational bottleneck is the subroutine required for identifying the pair (f, t) that provides the best split for a set of N data points, which typically requires $O(N)$ computations per split. Recent proposals include computing (or estimating) f and t from a subsample of the data points and a subset of the features, or quantizing the feature values. The latter technique creates a histogram of values of each feature across the datapoints and restricts t to be at the edges of histogram bins. In Subsection 1.1, we discuss these approaches in more detail.

While existing approaches provide significant speedups in the training of RFs, they are often based on a pre-defined strategy in terms of the number of data points and number of features to consider at each split. As such, they are not able to adapt to the data distribution. Moreover, when comparing different candidate features f for a split, all features are treated equally and the quality of their split is computed based on the same number of data points. Intuitively, this is wasteful as features that are not informative for the prediction task can be identified based on a very small number of data points. Alternatively, an adaptive scheme could better allocate computational resources towards a promising set of candidate features and achieve a better tradeoff between computational cost and accuracy.

In this work, we propose MABSplit, a fast subroutine for the node splitting problem, which adaptively refines the estimate of the "quality" of each feature-threshold pair (f, t) as a candidate split. In this way, bad splitting candidates can be discarded early, which can lead to significant computational savings. The main idea behind our algorithm is to reduce the node-splitting task to a multi-armed bandit problem [26, 1, 4, 43], where each pair (f, t) is a distinct arm. The unknown parameter of each arm, μ_{ft}, corresponds to the quality of the split of the data points $\mathcal{X} = \{x_1, \ldots, x_N\}$ based on feature f and threshold t. The split quality is measured in terms of how much the split would reduce label impurity, for a chosen label impurity measure such as Gini impurity. An arm (f, t) can be "pulled" by computing the reduction of label impurity on a small subset of data points from $\mathcal{X}$. This allows us to compute an estimate $\hat{\mu}_{ft}$ and a corresponding confidence interval, which can be used in a batched variant of the Upper Confidence Bound (UCB) and successive elimination algorithms [26, 48] to identify the best arm (f, t).

We demonstrate the benefits of MABSplit on a variety of datasets, for both classification and regression tasks. Even on small datasets such as MNIST, our algorithm leads to 7x faster training (an 85% reduction in training time), without any decrease in test accuracy, over an exact implementation of RF that searches for the optimal (f, t) pair via brute-force computation. Additionally, we demonstrate similar speedups when using MABSplit across a variety of forest-based variants, such as Extremely Random Forests and Random Patches.

1.1 Related work

Random Forests were originally proposed by Ho [21]. In its basic formulation, a RF involves constructing n_{tree} DTs, each based on a bootstrap sample of N data points and a random subset of the features (known as random subspacing [22]). More recently, the need for training RFs on large-scale datasets has prompted the development of several techniques to accelerate the training process via both software and hardware techniques.

Software acceleration of RF: Most of the software/algorithmic acceleration techniques are based on accelerating the training of each individual DT. FastForest [46] accelerates the node-splitting task using three ideas: subsampling a pre-specified number of datapoints (subbagging) without replacement, subsampling a pre-specified number of features dependent on the current number of data points (Dynamic Restricted Subspacing), and dividing values of a given feature into T bins where T depends on the number of data points at the node (Logarithmic Split-Point Sampling, inspired by the single-tree SPAARC algorithm [47]). MABSplit is inspired by the ideas in FastForest, but does not require the number of datapoints or features to be pre-specified as these are determined adaptively.

Recent work has discussed using adaptivity to identify the best split. For example, Very Fast Decision Tree (VFDT) [14] and Extremely Fast Decision Tree (EFDT) [31] are incremental decision tree learning algorithms in which trees can be updated in streaming settings. Acceleration at the node-splitting task is achieved by adaptively selecting a subset of data points sufficient to distinguish the best and second best features. This is similar to our approach, but the sampled data points are evaluated for all features, unlike in MABSplit, which adaptively discards unpromising feature splits. The F-forest [17] algorithm also applies adaptivity and utilizes an upper bound on the impurity reduction of a split in order to discard candidate splits early and save on computation. This is similar in spirit to the goal of MABSplit, but is based on a deterministic upper bound on the impurity reduction (as opposed to our statistical estimate) and is still linear in N per split computation, even when considering a fixed number of splitting thresholds per feature.

Other variations of RF have been proposed to improve training time. Random Patches [30] builds trees based on a subset of data points and a subset of features (which is fixed for each entire DT, unlike in RF). Extra-Trees (ETs) [18] draws a random subset of K feature at each node and, for each one, chooses a number R of random splits, and then selects the split that yields the largest impurity reduction from among these KR candidate splits. Other recent work attempts to accelerate RF training by identifying the optimal number of decision trees needed in the forest [34], a form of hyperparameter tuning. The MABSplit subroutine can also be incorporated into these methods.

Hardware acceleration of RF: The training of RFs can also be significantly accelerated through the use of specialized hardware. For instance, the implementation of RF available in Weka [15] allows trees to be trained on different cores and reduces forest build time. A GPU-based parallel implementation of RF has also been proposed in [19]. These solutions require specialized hardware (e.g., GPU-based PC video cards) and are inappropriate for everyday users locally executing data-mining tasks on standard PCs or smartphones. As such, it is also desirable to develop techniques to improve prediction performance and processing speed on standard platforms.

2 Preliminaries

We consider N data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ where each $\mathbf{x}_i$ is an M-dimensional feature vector and y_i is its target. Following standard RF literature, we consider flexible feature types such as numerical or categorical; RFs also deal naturally with missing values [23]. We also consider both categorical targets for classification and numerical targets for regression. With a slight abuse of notation, we use $\mathcal{X}$ to mean either the set of indices $\{i\}$ or the values $\{(\mathbf{x}_i, y_i)\}$, with the meaning clear from context.

A random forest contains n_{tree} decision trees, each constructed using a set of bootstrapped N samples (sampled with replacement) and a random set of features (typically $\sqrt{M}$ [9]) at each node for a given tree; it outputs the majority vote across trees for the classification task and average prediction across all trees for the regression task [21, 11]. We focus on the top-down approach of constructing DTs by choosing a variable at each step that best splits the set of labels at a given node [41]. Specifically, consider a node with n data points $\mathcal{X}$ and m features $\mathcal{F}$. Let $\mathcal{X}_{\text{L},ft}$ and $\mathcal{X}_{\text{R},ft}$ be the left and right child subsets of $\mathcal{X}$ when $\mathcal{X}$ is split according to feature f at threshold t. The approach finds the split that best reduces the label impurity; i.e., it finds

$$f^*, t^* = \underset{f \in \mathcal{F}, t \in \mathcal{T}_f}{\arg\min} \frac{|\mathcal{X}_{\text{L},ft}|}{n} I(\mathcal{X}_{\text{L},ft}) + \frac{|\mathcal{X}_{\text{R},ft}|}{n} I(\mathcal{X}_{\text{R},ft}) - I(\mathcal{X}), \tag{1}$$

where $I(\mathcal{X})$ measures the impurity of targets $\{y_i\}_{i \in \mathcal{X}}$ and $\mathcal{T}_f$ is the set of feasible thresholds for feature f. Common impurity measures include Gini impurity or entropy for classification, and mean-squared-error (MSE) for regression [12]:

$$\text{Gini} : 1 - \sum_{k=1}^{K} p_k^2, \quad \text{Entropy} : -\sum_{k=1}^{K} p_k \log_2 p_k, \quad \text{MSE} : \frac{1}{n} \sum_{i \in \mathcal{X}} (y_i - \bar{y})^2, \tag{2}$$

where K is the total number of classes, $p_k = \frac{1}{n} \sum_{i \in \mathcal{X}} \mathbb{I}(y_i = k)$ is the proportion of class k in $\mathcal{X}$ in classification and $\bar{y} = \frac{1}{n} \sum_{i \in \mathcal{X}} y_i$ is the average target value in regression. We note that our proposed algorithm, MABSplit, does not assume a particular structure of $I(\cdot)$.

While the conventional RF considers all $(n-1)$ possible splits at a feature f, in this work we focus on the histogram-based variation which chooses the threshold from a set of predefined values $\mathcal{T}_f$,

e.g., $|\mathcal{T}_f|$ equally-spaced histogram bin edges; this variation is substantially more computationally efficient, offers comparable accuracy, and has been used in most state-of-the-art implementations [40, 13, 25, 47].

3 MABSplit

Without loss of generality, consider a classification task with K classes. RF constructs n_{tree} DTs, each of which requires recursively partitioning data points at a given node across a set of $\sqrt{M}$ features. Consider a node with n data points $\mathcal{X}$, m features $\mathcal{F}$, and T possible thresholds for each feature $\mathcal{T}_f$. We note that n and m at the given node may or may not be the same as N and M of the entire dataset. A naïve algorithm finds the best feature-threshold pairs in Eq. (1) by evaluating the label impurity reduction for each feature-threshold over all n data points, which incurs computational cost linear in n.

We present MABSplit, which can substantially reduce this complexity to sublinear in n. Note that Eq. (1) is equivalent to

$$f^*, t^* = \underset{f \in \mathcal{F}, t \in \mathcal{T}_f}{\arg\min} \frac{|\mathcal{X}_{\text{L},ft}|}{n} I(\mathcal{X}_{\text{L},ft}) + \frac{|\mathcal{X}_{\text{R},ft}|}{n} I(\mathcal{X}_{\text{R},ft}) \tag{3}$$

and so we focus on solving Eq. (3). Let $\mu_{ft} = \frac{|\mathcal{X}_{\text{L},ft}|}{n} I(\mathcal{X}_{\text{L},ft}) + \frac{|\mathcal{X}_{\text{R},ft}|}{n} I(\mathcal{X}_{\text{R},ft})$ be the optimization objective for feature-threshold pair (f, t). Computing μ_{ft} exactly is at least $O(n)$[1], but we can instead estimate μ_{ft} with fewer computations by drawing $n' < n$ independent samples uniformly with replacement from $\mathcal{X}$. As shown in Subsec. 3.1, it is possible to construct a point estimate $\hat{\mu}_{ft}(n')$ and a $(1-\delta)$ confidence interval (CI) $C_{ft}(n', \delta)$ for the parameter μ_{ft}, where n' and δ determine estimation accuracy. The width of CIs generally scales with $\sqrt{\frac{\log 1/\delta}{n'}}$. To estimate the solution to Problem (3) with high confidence, we can then choose to sample data points for different features to different degrees of accuracy. Intuitively, promising features with small values of μ_{ft} should be estimated with high accuracy, while less promising ones can be discarded without being evaluated on too many data points.

The specific adaptive estimation procedure is described in Algorithm 1. It can be viewed as a batched version of the conventional UCB algorithm [26, 48] combined with successive elimination [16], and is straightforward to implement. Algorithm 1 uses the set $\mathcal{S}_{\text{solution}}$ to track all potential solutions to Problem (3); $\mathcal{S}_{\text{solution}}$ is initialized as the set of all feature-threshold pairs $\{(f, t)\}$. Then, for each potential solution $(f, t) \in \mathcal{S}_{\text{solution}}$, Algorithm 1 maintains its mean objective estimate $\hat{\mu}_{ft}$ and $(1-\delta)$ CI C_{ft}.

In each iteration, a new batch of data points $\mathcal{X}_{\text{batch}}$ is evaluated for all potential solutions in $\mathcal{S}_{\text{solution}}$ and allows us to update the histogram for each feature f and make each estimate $\hat{\mu}_{ft}$ more accurate. Based on the current estimate, if a candidate's lower confidence bound $\hat{\mu}_{ft} - C_{ft}$ is greater than the upper confidence bound of the most promising candidate $\min_{f,t}(\hat{\mu}_{ft} + C_{ft})$, we remove it from $\mathcal{S}_{\text{solution}}$. This process continues until there is only one candidate in $\mathcal{S}_{\text{solution}}$ or until we have sampled more than n data points. In the latter case, we know that the difference between the remaining candidates in $\mathcal{S}_{\text{solution}}$ is so subtle that an exact computation is warranted. We then compute those candidates' objectives exactly and return the best candidate in the set.

3.1 Point estimate and confidence interval for impurity metrics

We consider two widely used impurity metrics, Gini impurity and entropy, although mean estimates and confidence intervals for other metrics can be derived similarly (more details are provided in Appendix 3). We consider point estimates and confidence intervals of μ_{ft} based on a set of n' points, $\{(\mathbf{X}_i, Y_i)\}_{i=1}^{n'}$, sampled independently and with replacement.

Let $p_{\text{L},k} := \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(x_{if} < t, y_i = k)$ and $p_{\text{R},k} := \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(x_{if} \geq t, y_i = k)$ denote the proportion of the full n data points in class k and each of the two subsets ("left" and "right",

[1] More precisely $O(nK)$, but we focus on the dependency on n here.

Algorithm 1 MABSplit ($\mathcal{X}, \mathcal{F}, \mathcal{T}_f, I(\cdot), B, \delta$)

$\mathcal{S}_{\text{solution}} \leftarrow \{(f,t),\ \forall\, f \in \mathcal{F},\ \forall t \in \mathcal{T}_f\}$ ▷ Set of potential solutions to Problem (3)
$n_{\text{used}} \leftarrow 0$ ▷ Number of data points sampled
For all $(f,t) \in \mathcal{S}_{\text{solution}}$, set $\hat{\mu}_{ft} \leftarrow \infty$, $C_{ft} \leftarrow \infty$ ▷ Initialize mean and CI for each arm
for all $f \in \mathcal{F}$ **do**
    Create empty histogram, h_f, with $|\mathcal{T}_f| = T$ equally spaced bins
while $n_{\text{used}} < n$ and $|\mathcal{S}_{\text{solution}}| > 1$ **do**
    Draw a batch sample $\mathcal{X}_{\text{batch}}$ of size B with replacement from $\mathcal{X}$
    for all unique f in $\mathcal{S}_{\text{solution}}$ **do**
        for all x in $\mathcal{X}_{\text{batch}}$ **do**
            Insert x_f into histogram h_f ▷ Each insertion is $O(1)$
    for all $(f,t) \in \mathcal{S}_{\text{solution}}$ **do**
        Update $\hat{\mu}_{ft}$ and C_{ft} based on histogram h_f ▷ Update running mean and CI in $O(T)$
    $\mathcal{S}_{\text{solution}} \leftarrow \{(f,t) : \hat{\mu}_{ft} - C_{ft} \leq \min_{f,t}(\hat{\mu}_{ft} + C_{ft})\}$ ▷ Retain only promising splits
    $n_{\text{used}} \leftarrow n_{\text{used}} + B$
if $|\mathcal{S}_{\text{solution}}| = 1$ **then**
    return $(f^*, t^*) \in \mathcal{S}_{\text{solution}}$
else
    Compute μ_{ft} exactly for all $(f,t) \in \mathcal{S}_{\text{solution}}$
    return $(f^*, t^*) = \arg\min_{(f,t)\in\mathcal{S}_{\text{solution}}} \mu_{ft}$

respectively). Note that

$$\sum_k p_{\text{L},k} = \frac{|\mathcal{X}_{\text{L},ft}|}{n} \quad \text{and} \quad \sum_k p_{\text{R},k} = \frac{|\mathcal{X}_{\text{R},ft}|}{n}. \tag{4}$$

Furthermore, let $\hat{p}_{\text{L},k} := \frac{1}{n'}\sum_{i=1}^{n'} \mathbb{I}(X_{if} < t, Y_i = k)$ and $\hat{p}_{\text{R},k} := \frac{1}{n'}\sum_{i=1}^{n'} \mathbb{I}(X_{if} \geq t, Y_i = k)$ denote the empirical estimates of $p_{\text{L},k}$ and $p_{\text{R},k}$ based on the n' subsamples drawn thus far. Then $\{\hat{p}_{\text{L},k}, \hat{p}_{\text{R},k}\}_{k=1}^K$ jointly follow a multinomial distribution (since for each random data point $(\mathbf{X}_i, Y_i)$, exactly one of these variables is incremented) satisfying

$$\mathbb{E}[\hat{p}_{\text{L},k}] = p_{\text{L},k}, \quad \text{Var}[\hat{p}_{\text{L},k}] = \frac{1}{n'} p_{\text{L},k}(1 - p_{\text{L},k}),$$
$$\mathbb{E}[\hat{p}_{\text{R},k}] = p_{\text{R},k}, \quad \text{Var}[\hat{p}_{\text{R},k}] = \frac{1}{n'} p_{\text{R},k}(1 - p_{\text{R},k}).$$

Using Eqs. (2) and (4), and the definition of μ_{ft} after Eq. (3), we can write

$$\text{Gini impurity}: \ \mu_{ft} = 1 - \frac{\sum_k p_{\text{L},k}^2}{\sum_k p_{\text{L},k}} - \frac{\sum_k p_{\text{R},k}^2}{\sum_k p_{\text{R},k}}, \tag{5}$$

$$\text{Entropy}: \ \mu_{ft} = -\sum_k p_{\text{L},k} \log_2 \frac{p_{\text{L},k}}{\sum_{k'} p_{\text{L},k'}} - \sum_k p_{\text{R},k} \log_2 \frac{p_{\text{R},k}}{\sum_{k'} p_{\text{R},k'}}, \tag{6}$$

where we can use the empirical parameters $\{\hat{p}_{\text{L},k}, \hat{p}_{\text{R},k}\}_{k=1}^K$ as plug-in estimators for the true parameters $\{p_{\text{L},k}, p_{\text{R},k}\}_{k=1}^K$ in the equations above to produce the point estimate $\hat{\mu}_{ft}$. In Algorithm 1, after each batch of B data points is obtained, we update $\hat{p}_{\text{L},k}$ and $\hat{p}_{\text{R},k}$, which can then be used to update our point estimate $\hat{\mu}_{ft}$ and the corresponding CIs. The CIs of $\hat{\mu}_{ft}$ are based on standard error derived using the delta method [45]. As in standard applications of the delta method, these estimates $\hat{\mu}_{ft}$ are asymptotically unbiased and the confidence intervals are asymptotically valid. Appendix 3 provides further details including derivation of CIs and discussion of convergence properties.

3.2 Algorithmic and implementation details

We considered sampling with replacement Algorithm 1 primarily for the ease of theoretical analysis. In practice, we found sampling without replacement was more computationally efficient and did not significantly change the results, and was used in the actual implementation.

For DTs, in classification tasks, we allow individual DTs to provide soft votes and average their predicted class probabilities to determine the forest's predicted class label, following existing approaches. This is in contrast with hard votes, in which each DT is only permitted to produce its best label and the forest's prediction is determined by majority voting. For the fixed-budget experiments described in Section 5, we terminate tree construction and do not split the next node if doing so would violate our budget constraints.

Finally, we note that a head-to-head comparison with common implementations of forest-based algorithms, such as Weka's [15] or `scikit-learn`'s [37], would be unfair due to their extensive hardware- and language-level optimizations in those implementations. As such, we reimplemented these baselines in Python and focused on algorithmic improvements. We verified the quality of our own implementations agreed with that of `scikit-learn` in Appendix **??**. The only difference between our model and the baselines is the call to the node splitting subroutine (MABSplit in our model, and the exact, brute-force solver for the baseline models); thus, any improvements in runtime are due to improvements in the node-splitting algorithm. This is verified by profiling the implementations and measuring the relative time spent in the node-splitting subroutine versus total runtime (see Appendix 5). This allows us to focus on algorithmic improvements as opposed implementation-specific optimizations. Our implementation of these baselines may also be of independent interest.

4 Analysis of the Algorithm

In this section, we prove that MABSplit returns the optimal feature-threshold pair for a node split with high probability. Furthermore, we provide bounds on computational complexity of MABSplit, leading to the logarithmic dependency on the number of data points n.

Consider a node with n data points $\mathcal{X}$, m features $\mathcal{F}$, and T possible thresholds for each feature ($|\mathcal{T}_f| = T$ for all f). Suppose $(f^*, t^*) = \arg\min_{f\in\mathcal{F}, t\in\mathcal{T}_f} \mu_{ft}$ is the optimal feature-threshold pair at which to split the node. For any other feature-threshold pair (f, t), define $\Delta_{ft} \coloneqq \mu_{ft} - \mu_{f^*t^*}$. To state the following results, we will assume that, for a fixed feature-threshold pair (f, t) and n' randomly sampled datapoints, the $(1-\delta)$ confidence interval scales as $C_{ft}(n', \delta) = O(\sqrt{\frac{\log 1/\delta}{n'}})$. This is the case for the confidence intervals of Gini impurity and entropy in Subsec. 3.1 [45]. With this assumption, we state the following theorem:

Theorem 1. *Assume* $\exists c_0 > 0$ *s.t.* $\forall \delta > 0, n' > 0, C_{ft}(n', \delta) < c_0\sqrt{\frac{\log 1/\delta}{n'}}$. *For* $\delta = \frac{1}{n^2mT}$, *with probability at least* $1 - \frac{1}{n}$, *Algorithm 1 returns the correct solution to* (3). *Furthermore, Algorithm 1 uses a total of* M *computations, where*

$$\mathbb{E}[M] \leq \sum_{f\in\mathcal{F}, t\in\mathcal{T}_f} \min\left[\frac{4c_0^2}{\Delta_{ft}^2}\log(n^2mT) + B, 2n\right] + 2mT. \tag{7}$$

Theorem 1 is proven in Appendix 1. Intuitively, Theorem 1 states that with high probability, MABSplit returns the optimal feature-threshold pair at which to split the node. The instance-wise bound Eq. (7) suggests the computational cost of a feature-threshold pair (f, t), i.e., $\min\left[\frac{4c_0^2}{\Delta_{ft}^2}\log(n^2mT) + B, 2n\right]$, depends on Δ_{ft}, which measures how close its optimization parameter μ_{ft} is to $\mu_{f^*t^*}$. Most reasonably different features $f \neq f^*$ will have a large Δ_{ft} and incur a $O(\log(n^2mT))$ computation that is sublinear in n.

In general, MABSplit takes only $O(\log(n^2mT))$ computations per feature-threshold pair if there is reasonable heterogeneity among them. As proven in Appendix 2 in [4], this is the case under a wide range of distributional assumptions on the μ_{ft}'s, e.g., when the μ_{ft}'s follow a sub-Gaussian distribution across the pairs (f, t). These assumptions ensure that MABSplit has an overall complexity of $O(mT\log(n^2mT))$, which is sublinear in the number of data points n. However, we note that in the worst case, MABSplit may take $O(n)$ computations per feature-threshold pair when most feature-threshold pairs are equally good, for example in highly symmetric datasets where all splits reduce the impurity equally. Other recent work provides further discussion on the conversion between an instance-wise bound like Equation (7) and an instance-independent bound such as $O(mT\log n)$ [4, 48, 6, 43, 3, 5].

Finally, we note that δ is a hyperparameter governing the error rate. It is possible to prove results analogous to Theorem 1 for arbitrary δ.

5 Experimental Results

We demonstrate the advantages of MABSplit in two settings. In the first setting, the baseline models with and without MABSplit are trained to completion, and wall-clock training time and test performances are measured. In the second setting, we consider training each forest with a fixed computational budget and study effect of MABSplit on generalization performance.

Datasets: We run experiments on public datasets to validate the behavior of MABSplit, in both classification and regression. For classification, we use the MNIST dataset [29] consisting of 60,000 training and 10,000 test images of handwritten digits, where each black-and-white image is represented as a 784-dimensional vector. For regression, we generate 200,000 datapoints with 50 features, 6 of which are correlated with the targets and 44 of which are pure noise, using `scikit-learn`'s `make_regression` function [36]; 160,000 datapoints are used for training and the remaining 40,000 for test. We call this dataset the Random Linear Model dataset. The two datasets are available under MIT and BSD licenses, respectively.

Baselines: We compare the histogrammed versions of three baselines with and without MABSplit: Random Forest (RF), Extremely Randomized Forests (ExtraTrees) [18], and Random Patches (RP) [30]. In Random Forests, each tree fits a bootstrap sample of N datapoints and only considers random subset of $\sqrt{M}$ features at every node split. Extremely Randomized Forests are identical to Random Forests except for two differences. First, in regression problems, all features are considered at every node split (in classification problems, we still use only $\sqrt{M}$ features). Second, the histogram edges of a feature are randomly chosen from a uniform distribution over that feature's minimum and maximum value. In classification problems, each histogram has $\sqrt{M}$ bins and in regression problems, each histogram has M bins. These conventions follow standard implementations [36]. Note that in Extremely Randomized Forests, the bins in a feature's histogram need not be equally spaced. In Random Patches, the dataset is reduced to α_n of its original datapoints and α_f of its original features, where α_n and α_f are constants. The reduced dataset is fixed for the training of the entire forest. Full settings for all experiments are given in Appendix 6.

5.1 Wall clock time comparisons

In the first setting, we compare baseline models with and without MABSplit in terms of wall-clock training time. Tables 1 and 2 show that MABSplit provides similar test accuracy but leads to faster training than the usual brute-force algorithm for node-splitting for almost all baselines in both classification and regression tasks. Across various tasks, MABSplit leads to 2x-7x faster training, a reduction of training time of 50-85%. These benefits are wholly attributable to MABSplit as the only difference between successive rows is the node-splitting subroutine (see Appendix 5).

5.2 Fixed budget comparisons

In the second setting, we consider training models under a fixed computational budget. As before, insertion into a histogram is taken to be an $O(1)$ operation. This is justified if the histogram's thresholds are evenly spaced, in which the correct bin in which to insert a value can be indexed directly. If the bins are unevenly spaced, we may perform binary searches to locate the correct bin, which is $O(\log T)$ and does not depend on n, or cache the results of these binary searches for an evenly-spaced grid across the range of the given feature's value.

Intuitively, the MABSplit algorithm allows for splitting a given node with less data point queries and histogram insertions than the brute-force solution. As such, when the computational budget is fixed, forests trained with MABSplit should be able to split more nodes and therefore train more trees than forests trained with the brute-force solver. Prior work suggests that increasing the number of trees in a forest improves generalization performance by reducing variance at the cost of slightly increased bias [20].

Tables 3 and 4 demonstrates the generalization performance of different models as the computational budget is held constant for classification and regression, respectively. When using MABSplit, the

Classification: MNIST			
Model	Wall-Clock Time (s)	Number of insertions	Test Accuracy
RF	379.2 ± 4.8	$3.5e7 \pm 2.1e5$	$77.0 \pm 0.5\%$
RF + MABSplit	$\mathbf{57.0 \pm 1.2}$	$\mathbf{4.6e6 \pm 6.5e4}$	$\mathbf{77.0 \pm 0.3\%}$
ExtraTrees	440.1 ± 5.1	$4.2e7 \pm 0.0$	$76.7 \pm 0.4\%$
ExtraTrees + MABSplit	$\mathbf{65.5 \pm 3.1}$	$\mathbf{5.1e6 \pm 6.0e4}$	$\mathbf{77.1 \pm 0.7}$
RP	1144.0 ± 53.1	$1.1e8 \pm 1.6e6$	$61.7 \pm 1.2\%$
RP + MABSplit	$\mathbf{168.5 \pm 4.1}$	$\mathbf{1.6e7 \pm 3.6e5}$	$\mathbf{71.6 \pm 1.0\%}$

Table 1: Wall-clock training time, number of histogram insertions, and test accuracies for various models with and without MABSplit. MABSplit accelerates these models by approximately 7x (an 85% reduction in training time) while achieving comparable (and sometimes better) accuracy. The number of histogram insertions correlates strongly with wall-clock training time, which justifies our focus on accelerating the node-splitting algorithm by focusing on histogram insertions.

Regression: Random Linear Model		
Model	Wall-Clock Time (s)	Test MSE
RF	32.1 ± 0.6	807.5 ± 13
RF + MABSplit	$\mathbf{13.9 \pm 0.1}$	$\mathbf{750.2 \pm 12.7}$
ExtraTrees	23.0 ± 0.1	1781.4 ± 34.9
ExtraTrees + MABSplit	$\mathbf{10.3 \pm 0.2}$	$\mathbf{1769.9 \pm 1.0}$
RP	6.2 ± 0.4	9024.6 ± 406.4
RP + MABSplit	$\mathbf{3.3 \pm 0.3}$	$\mathbf{9013.7 \pm 409.2}$

Table 2: Wall-clock training time and test MSEs for various models with and without MABSplit. MABSplit accelerates these models by approximately 2x (an 50% reduction in training time) while achieving comparable results. We omit the number of histogram insertions in favor of wall-clock time for simplicity; unlike in classification, the different baseline regression models have widely varying histogram bin counts. Since the histogram insertion complexity scales as $O(\log T)$, the comparison across models is not fair.

trained forests consist of more trees and demonstrate better generalization performance across all models.

We also apply MABSplit to compute feature importances under a fixed budget. We follow the common approach of computing the feature importances of multiple forests and then measuring the stability of feature selection across forests [32, 38]; see Table 5.

Classification: MNIST (10K subsamples training data points), budget = 10M		
Model	Number of trees	Test Accuracy
RF	5.8 ± 0.2	$53.1 \pm 1.0\%$
RF + MABSplit	$\mathbf{100 \pm 0}$	$\mathbf{64.6 \pm 0.4\%}$
ExtraTrees	5.0 ± 0.0	$49.0 \pm 0.7\%$
ExtraTrees + MABSplit	$\mathbf{100 \pm 0}$	$\mathbf{63.8 \pm 0.5\%}$
RP	21.8 ± 0.5	$32.1 \pm 0.7\%$
RP + MABSplit	$\mathbf{37.4 \pm 0.8}$	$\mathbf{40.2 \pm 2.5\%}$

Table 3: Classification performance under a fixed computational budget (number of histogram insertions). MABSplit pares down the computation required to train a tree for each type of model, resulting in more trained trees and better generalization performance.

Regression: Random Linear Model, Budget = 2,400,000$*Q$		
Model	# of trees	Test MSE
RF ($Q = 10$)	2.0 ± 0	4356.2 ± 51.2
RF + MABSplit ($Q = 10$)	$\mathbf{100 \pm 0}$	$\mathbf{3095.6 \pm 9.5}$
ExtraTrees ($Q = \frac{1}{2}$)	$\mathbf{3 \pm 0}$	$\mathbf{10037 \pm 4.1}$
ExtraTrees + MABSplit ($Q = \frac{1}{2}$)	100 ± 0	10068.1 ± 1.1
RP ($Q = 1$)	2.0 ± 0	8184.8 ± 565.6
RP + MABSplit ($Q = 1$)	$\mathbf{51.6 \pm 2.2}$	$\mathbf{8103.4 \pm 616.4}$

Table 4: Regression performance under a fixed computational budget (number of histogram insertions). MABSplit pares down the computation required to train a tree for each type of model. This results in better performance for the Random Forest baseline, which resamples features at each node split, but not in the Extreme Random Forest or Random Patches model, which analyze a fixed set of features. Those two models underfit the data as the signal is very sparse - only 3 out of 50 available features are informative.

Regression: Random Linear Model, Budget = 1,000,000$*Q$		
Importance Model	Dataset	Stability
RF + MID($Q = 3.5$)	random classification	0.397 ± 0.031
RF + MABSplit +MID ($Q = 3.5$)	**random classification**	$\mathbf{0.732 \pm 0.014}$
RF + MID ($Q = 5$)	random regression	0.664 ± 0.091
RF + MABSplit + MID ($Q = 5$)	**random regression**	$\mathbf{0.796 \pm 0.014}$
RF + Permutation ($Q = 5$)	digit	0.555 ± 0.022
RF + MABSplit + Permutation ($Q = 5$)	**digit**	$\mathbf{0.69 \pm 0.02}$
RF + Permutation ($Q = 5$)	random regression	0.305 ± 0.039
RF + MABSplit + Permutation ($Q = 5$)	**random regression**	$\mathbf{0.694 \pm 0.019}$

Table 5: Stability scores under a fixed computational budget (number of histogram insertions). MABSplit pares down the computation required to train a tree for each type of model. This results in better performance for the Random Forest stability baseline. `scikit-learn` datasets for both classification and regression were used.

6 Discussions and Conclusions

In this work, we presented a novel algorithm, MABSplit, for determining the optimal feature and corresponding feature threshold at which to split a node in tree-based learning models. Unlike prior models such as Random Patches, in which the hyperparameters α_n and α_f must be chosen manually, MABSplit requires no tuning and queries only as much data as is needed. Indeed, robustness to choice of hyperparameters is one of primary appeals of algorithms like RF [39].

MABSplit can avoid the expensive $O(n\log n)$ sort used in many existing baselines and the $O(n)$ computational complexity of their corresponding histogrammed versions. MABSplit can be used in conjunction with existing software- and hardware-specific optimizations and with other methods such as Logarithmic Split-Point Sampling and boosting [46]. In boosting, MABSplit has the potential advantage of only needing to update target variables on-the-fly as needed, as opposed to current boosting approaches which update target variables for the entire dataset at each iteration. Additionally, MABSplit may permit easier parallelization due to lower memory requirements than existing algorithms, which may enable greater use in edge computing, and may be adaptable to streaming settings.

References

[1] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best arm identification in multi-armed bandits. In *International Conference on Learning Theory*, pages 41–53, 2010.

[2] Mourad Azhari, Abdallah Abarda, Badia Ettaki, Jamal Zerouaoui, and Mohamed Dakkon. Higgs boson discovery using machine learning methods with pyspark. *Procedia Computer Science*, 170:1141–1146, 2020.

[3] Vivek Bagaria, Tavor Z Baharav, Govinda M Kamath, and N Tse David. Bandit-based monte carlo optimization for nearest neighbors. *IEEE Journal on Selected Areas in Information Theory*, 2(2):599–610, 2021.

[4] Vivek Bagaria, Govinda M. Kamath, Vasilis Ntranos, Martin J. Zhang, and David Tse. Medoids in almost-linear time via multi-armed bandits. In *International Conference on Artificial Intelligence and Statistics*, pages 500–509, 2018.

[5] Tavor Z Baharav, Gary Cheng, Mert Pilanci, and David Tse. Approximate function evaluation via multi-armed bandits. In *International Conference on Artificial Intelligence and Statistics*, pages 108–135. PMLR, 2022.

[6] Tavor Z. Baharav and David Tse. Ultra fast medoid identification via correlated sequential halving. In *Advances in Neural Information Processing Systems*, pages 3650–3659, 2019.

[7] G. P. Basharin. On a statistical estimate for the entropy of a sequence of independent random variables. *Theory of Probability & Its Applications*, 4(3):333–336, 1959.

[8] Lamara Benali, Gilles Notton, A Fouilloy, Cyril Voyant, and Rabah Dizene. Solar radiation forecasting using artificial neural network and random forest methods: Application to normal beam, horizontal diffuse and global components. *Renewable energy*, 132:871–884, 2019.

[9] Simon Bernard, Laurent Heutte, and Sébastien Adam. Influence of hyperparameters on random forest accuracy. In *International workshop on multiple classifier systems*, pages 171–180. Springer, 2009.

[10] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[11] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[12] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[13] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[14] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2000.

[15] Frank Eibe, Mark A Hall, and Ian H Witten. The weka workbench. online appendix for data mining: practical machine learning tools and techniques. In *Morgan Kaufmann*. Elsevier Amsterdam, The Netherlands, 2016.

[16] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. In *Journal of Machine Learning Research*, volume 7, pages 1079–1105, 2006.

[17] Yasuhiro Fujiwara, Yasutoshi Ida, Sekitoshi Kanai, Atsutoshi Kumagai, Junya Arai, and Naonori Ueda. Fast random forest algorithm via incremental upper bound. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2205–2208, 2019.

[18] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[19] Hrakan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat. Cudarf: a cuda-based implementation of random forests. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 95–101. IEEE, 2011.

[20] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *Elements of Statistical Learning*. Springer, 2009.

[21] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[22] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[24] Daniel Martin Katz, Michael J Bommarito, and Josh Blackman. A general approach for predicting the behavior of the supreme court of the united states. *PloS one*, 12(4):e0174698, 2017.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[26] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[27] SK Lakshmanaprabu, K Shankar, M Ilayaraja, Abdul Wahid Nasir, V Vijayakumar, and Naveen Chilamkurti. Random forest for big data classification in the internet of things using optimal features. *International journal of machine learning and cybernetics*, 10(10):2609–2618, 2019.

[28] Gabriele Lanaro, Quan Nguyen, and Sakis Kasampalis. *Advanced Python Programming: Build high performance, concurrent, and multi-threaded apps with Python using proven design patterns*. Packt Publishing.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Gilles Louppe and Pierre Geurts. Ensembles on random patches. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 346–361. Springer, 2012.

[31] Chaitanya Manapragada, Geoffrey I Webb, and Mahsa Salehi. Extremely fast decision tree. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1953–1962, 2018.

[32] Kristin K. Nicodemus. Letter to the editor: On the stability and ranking of predictors from random forest variable importance measures. *Briefings in Bioinformatics*, 12(4):369–373, 04 2011.

[33] Sarah Nogueira, Konstantinos Sechidis, and Gavin Brown. On the stability of feature selection algorithms. *J. Mach. Learn. Res.*, 18(1):6345–6398, 2017.

[34] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. How many trees in a random forest? In *International workshop on machine learning and data mining in pattern recognition*, pages 154–168. Springer, 2012.

[35] Liam Paninski. Estimation of entropy and mutual information. *Neural Comput.*, 15(6):1191–1253, jun 2003.

[36] 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*, 12:2825–2830, 2011.

[37] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[38] Miriam Piles, Rob Bergsma, Daniel Gianola, Hélène Gilbert, and Llibertat Tusell. Feature selection stability and accuracy of prediction models for genomic prediction of residual feed intake in pigs using machine learning. *Frontiers in Genetics*, 12, 2021.

[39] Philipp Probst, Marvin N Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: data mining and knowledge discovery*, 9(3):e1301, 2019.

[40] Sanjay Ranka and Vineet Singh. Clouds: A decision tree classifier for large datasets. In *Proceedings of the 4th knowledge discovery and data mining conference*, volume 2, 1998.

[41] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers-a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005.

[42] Eduarda MO Silveira, Sergio Henrique G Silva, Fausto W Acerbi-Junior, Monica C Carvalho, Luis Marcelo T Carvalho, Jose Roberto S Scolforo, and Michael A Wulder. Object-based random forest modelling of aboveground forest biomass outperforms a pixel-based approach in a heterogeneous and mountain tropical environment. *International Journal of Applied Earth Observation and Geoinformation*, 78:175–188, 2019.

[43] Mo Tiwari, Martin J Zhang, James Mayclin, Sebastian Thrun, Chris Piech, and Ilan Shomorony. Banditpam: Almost linear time k-medoids clustering via multi-armed bandits. *Advances in Neural Information Processing Systems*, 33:10211–10222, 2020.

[44] Kagan Tumer and Joydeep Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection science*, 8(3-4):385–404, 1996.

[45] Aad W Van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[46] Darren Yates and Md Zahidul Islam. Fastforest: Increasing random forest processing speed while maintaining accuracy. *Information Sciences*, 557:130–152, 2021.

[47] Darren Yates, Md Zahidul Islam, and Junbin Gao. Spaarc: a fast decision tree algorithm. In *Australasian Conference on Data Mining*, pages 43–55. Springer, 2018.

[48] Martin Zhang, James Zou, and David Tse. Adaptive monte carlo multiple testing via multi-armed bandits. In *International Conference on Machine Learning*, pages 7512–7522, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We have all reviewed the claims in the abstract and introduction and confirmed that they reflect the paper's contributions and scope.
   (b) Did you describe the limitations of your work? [Yes] We discuss the assumptions necessary for MABSplit to work in Section 4 and also discuss failure modes when these assumptions are violated. We also describe further room to improve our work in Section 6
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We discuss potential negative societal impacts in the Future Work section
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We have all reviewed the ethics review guidelines and ensured that our paper conforms to them.
2. If you are including theoretical results...

(a) Did you state the full set of assumptions of all theoretical results? [Yes] The statement of Theorem 1 contains a full description of assumptions, which is further expounded upon before and after the theorem statement.

(b) Did you include complete proofs of all theoretical results? [Yes] The proof of Theorem 1 is provided in Appendix 1

3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Yes, full specifications of all experiments are provided in Section 5 and Appendix 6, and full, documented source code is provided to reproduce our results.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Yes, full specifications of all experiments are provided in Section 5 and Appendix 6, and full, documented source code is provided to reproduce our results.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Every numerical measurement in Tables 1, 2, 3, and 4 contains confidence intervals

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Full specifications of computing resources are provided in 6

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] Yes, the only existing assets we used were publicly available datasets, which we cited in Section 5

(b) Did you mention the license of the assets? [Yes] We explicitly mention the licenses in Section 5.

(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] Yes, the only new assets created are in the form of code, which is entirely submitted in the supplemental material

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] Yes, both datasets we use are publicly available and we mention the licenses in Section 5.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The two datasets are the MNIST dataset and a mathematical dataset; neither have any personally identifiable information or offensive content.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We did not use crowdsourcing or conduct research with human subjects.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We did not use crowdsourcing or conduct research with human subjects.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We did not use crowdsourcing or conduct research with human subjects.

Appendix

1 Proofs

In this section, we present the proof of Theorem 1.

Proof. Following the multi-armed bandit literature, we refer to each feature-threshold pair (f, t) as an arm and refer to its optimization object μ_{ft} as the arm parameter. Pulling an arm corresponds to evaluating the change in impurity induced by one data point i at one feature-threshold pair (f, t) (i.e., arm) and incurs an $O(1)$ computation. This allows us to focus on the number of arm pulls, which translates directly to sample complexity.

First, we show that, with probability at least $1 - \frac{1}{n}$, all confidence intervals computed throughout the algorithm are valid, in the sense that they contain the true parameter μ_{ft}. For a fixed (f, t) and a given iteration of the algorithm, the $(1 - \delta)$ confidence interval satisfies

$$\Pr\left(|\mu_{ft} - \hat{\mu}_{ft}| > C_{ft}\right) \leq \delta.$$

Note that there are at most $\frac{n}{B}$ rounds and hence at most $\frac{nmT}{B} \leq nmT$ confidence intervals computed across all arms and all steps of the algorithm, where B is the batch size. With $\delta = \frac{1}{n^2 mT}$, we see that $\mu_{ft} \in [\hat{\mu}_{ft} - C_{ft}, \hat{\mu}_{ft} + C_{ft}]$ for every arm (f, t) and for every step of the algorithm with probability at least $1 - \frac{1}{n}$, by a union bound over at most nmT confidence intervals.

Next, we prove the correctness of Algorithm 1. Let $(f^*, t^*) = \arg\min_{f \in \mathcal{F}, t \in \mathcal{T}_f} \mu_{ft}$ be the desired output of the algorithm. First, observe that the main `while` loop in the algorithm can only run $\frac{n}{B}$ times, so the algorithm must terminate. Furthermore, if all confidence intervals throughout the algorithm are correct, it is impossible for (f^*, t^*) to be removed from the set of candidate arms. Hence, (f^*, t^*) (or some (f, t) with $\mu_{ft} = \mu_{f^*t^*}$) must be returned upon termination with probability at least $1 - \frac{1}{n}$. This proves the correctness of Algorithm 1.

Finally, we consider the complexity of Algorithm 1. Let n_{used} be the total number of arm pulls computed for each of the arms remaining in the set of candidate arms at a given point in the algorithm. Notice that, for any suboptimal arm $(f, t) \neq (f^*, t^*)$ that has not left the set of candidate arms, we must have $C_{ft} \leq c_0 \sqrt{\frac{\log 1/\delta}{n_{\text{used}}}}$ by assumption. With $\delta = \frac{1}{n^2 mT}$ as above and $\Delta_{ft} = \mu_{ft} - \mu_{f^*t^*}$, if $n_{\text{used}} > \frac{4c_0^2}{\Delta_{ft}^2} \log(n^2 mT)$ then

$$2(C_{ft} + C_{f^*t^*}) \leq 2c_0 \sqrt{\log(n^2 mT)/n_{\text{used}}} < \Delta_{ft} = \mu_{ft} - \mu_{f^*t^*},$$

and

$$\begin{aligned}
\hat{\mu}_{ft} - C_{ft} &> \mu_{ft} - 2C_{ft} \\
&= \mu_{f^*t^*} + \Delta_{ft} - 2C_{ft} \\
&\geq \mu_{f^*t^*} + 2C_{f^*t^*} \\
&> \hat{\mu}_{f^*t^*} + C_{f^*t^*}
\end{aligned}$$

which means that (f, t) must be removed from the set of candidate arms at the end of that iteration. Hence, the number of data point computations M_{ft} required for any arm $(f, t) \neq (f^*, t^*)$ is at most

$$M_{ft} \leq \min\left[\frac{4c_0^2}{\Delta_{ft}^2} \log(n^2 mT) + B, 2n\right].$$

Notice that this holds simultaneously for all arms (f, t) with probability at least $1 - \frac{1}{n}$. We conclude that the total number of arm pulls M satisfies

$$\begin{aligned}
\mathbb{E}[M] &\leq \mathbb{E}[M \mid \text{all confidence intervals are correct}] + \frac{1}{n}(2nMT) \\
&\leq \sum_{f \in \mathcal{F}, t \in \mathcal{T}_f} \min\left[\frac{4c_0^2}{\Delta_{ft}^2} \log(n^2 mT) + B, 2n\right] + 2mT,
\end{aligned}$$

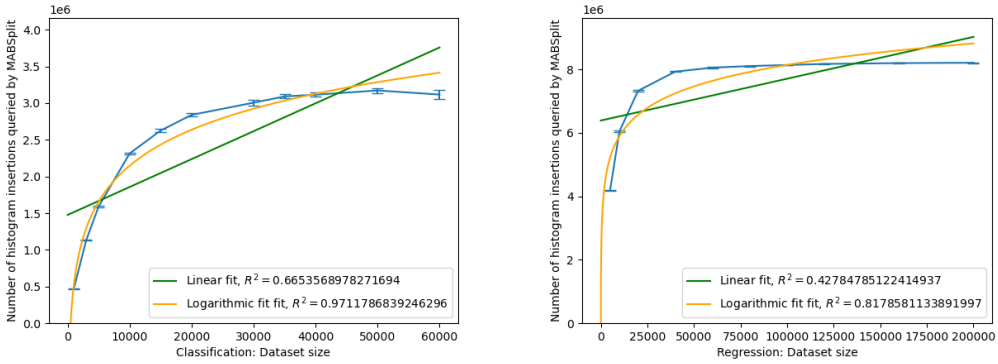
where we used the fact that the maximum number of computations for any arm is $2n$. As argued before, since each arm pull involves an $O(1)$ computation, M also corresponds the total number of computations. □

2 $O(\log n)$ scaling of MABSplit

In Theorem 1, we demonstrated that MABSplit scales logarithmically in data set size. In this section, we empirically validate this claim.

Figure 1 (a) demonstrates the number of data points queried by MABSplit for a single node split, i.e., a single call to MABSplit, as the dataset size increases, for various subset sizes of MNIST. For each sample size, a sample is drawn with replacement from the original MNIST dataset. The model is trained using Gini impurity in the typical classification setting. Figure 1 (b) demonstrates the same plot for various subset sizes of the Random Linear Model dataset.

Figure 1 also shows the best linear and logarithmic fits to each dataset. The relatively high R^2 values of the logarithmic fits ($R^2 = 0.97$ and $R^2 = 0.82$) compared to that of the linear fits ($R^2 = 0.66$ and $R^2 = 0.43$) suggests that the scaling of MABSplit is logarithmic (and therefore sublinear) with dataset size.



(a) (b)

Appendix Figure 1: Scaling of MABSplit with different data subset sizes for (a) the MNSIT digit classification task and (b) the Random Linear Model regession task. In both tasks, MABSplit appears to scale logarithmically, not linearly, with dataset size.

3 Mean Estimation and Confidence Interval Constructions

In this section, we discuss the estimation of the means μ_{ft} and construction of their confidence intervals via plug-in estimators and the delta method.

Let $p_{\mathrm{L},k}$, $p_{\mathrm{R},k}$, $\hat{p}_{\mathrm{L},k}$, and $\hat{p}_{\mathrm{R},k}$ be the same as defined in Subsection 3.1. Furthermore, let $\mathbf{p} = [p_{\mathrm{L},1}, \cdots, p_{\mathrm{L},K}, p_{\mathrm{R},1}, \cdots, p_{\mathrm{R},K}]^T$ and $\hat{\mathbf{p}} = [\hat{p}_{\mathrm{L},1}, \cdots, \hat{p}_{\mathrm{L},K}, \hat{p}_{\mathrm{R},1}, \cdots, \hat{p}_{\mathrm{R},K}]^T$. Then, $n'\hat{\mathbf{p}}$ follows a multinomial distribution with parameters $(n', 2K, \mathbf{p})$.

Let $\boldsymbol{\theta} = [p_{\mathrm{L},1}, \cdots, p_{\mathrm{L},K}, p_{\mathrm{R},1}, \cdots, p_{\mathrm{R},K-1}]^T$ and $\hat{\boldsymbol{\theta}} = [\hat{p}_{\mathrm{L},1}, \cdots, \hat{p}_{\mathrm{L},K-1}, \hat{p}_{\mathrm{R},1}, \cdots, \hat{p}_{\mathrm{R},K-1}]^T$. Then, by the Central Limit Theorem,

$$\sqrt{n'}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}) \overset{D}{\sim} \mathcal{N}(0, \Sigma), \tag{8}$$

where $\Sigma_{ii} = \theta_i(1-\theta_i)$ and $\Sigma_{ij} = -\theta_i\theta_j$.

Next, write μ_{ft} in terms of $\boldsymbol{\theta}$ for the impurity metrics as

$$\text{Gini impurity}: \ \mu_{ft}(\boldsymbol{\theta}) = 1 - \frac{\sum_{k=1}^{K}\theta_k^2}{\sum_{k=1}^{K}\theta_k} - \frac{\sum_{k=K+1}^{2K-1}\theta_k^2 + (1-\sum_{k=1}^{2K-1}\theta_k)^2}{1-\sum_{k=1}^{K}\theta_k}, \tag{9}$$

$$\text{Entropy}: \ \mu_{ft}(\boldsymbol{\theta}) = -\sum_{k=1}^{K}\theta_k \log_2 \frac{\theta_k}{\sum_{k'=1}^{K}\theta'_k} - \sum_{k=K+1}^{2K-1}\theta_k \log_2 \frac{\theta_k}{1-\sum_{k'=1}^{K}\theta'_k} - (1-\sum_{k=1}^{2K-1}\theta_k)\log_2 \frac{(1-\sum_{k=1}^{2K-1}\theta_k)}{1-\sum_{k=1}^{K}\theta_k}. \tag{10}$$

For a given impurity metric, let $\nabla\mu_{ft}(\boldsymbol{\theta})$ be the derivative of μ_{ft} with respect to $\boldsymbol{\theta}$. From the delta method,

$$\sqrt{n'}(\hat{\mu}_{ft}(\boldsymbol{\theta}) - \mu_{ft}(\boldsymbol{\theta})) \overset{D}{\sim} \mathcal{N}(0, \nabla\mu_{ft}(\boldsymbol{\theta})^T\Sigma\nabla\mu_{ft}(\boldsymbol{\theta})), \tag{11}$$

where the CIs can be constructed accordingly. These CIs are asymptotically valid as $n', n \to \infty$. For other impurity metrics such as MSE, the CIs can be similarly derived by writing the corresponding μ_{ft} in terms of $\boldsymbol{\theta}$.

Model	Task and Dataset	Performance Metric	Test Performance
RF (ours)	Classification: 20 Newsgroups	Accuracy	$74.1 \pm 2.8\%$
RF (`scikit-learn`)	Classification: 20 Newsgroups	Accuracy	$76.2 \pm 1.7\%$
ExtraTrees (ours)	Classification: 20 Newsgroups	Accuracy	$66.5 \pm 5.1\%$
ExtraTrees (`scikit-learn`)	Classification: 20 Newsgroups	Accuracy	$62.6 \pm 2.8\%$
RF (ours)	Regression: California Housing	MSE	0.679 ± 0.022
RF (`scikit-learn`)	Regression: California Housing	MSE	0.672 ± 0.028
ExtraTrees (ours)	Regression: California Housing	MSE	0.696 ± 0.055
ExtraTrees (`scikit-learn`)	Regression: California Housing	MSE	0.695 ± 0.082

Table 6: Comparison of our re-implementation of baselines with the the implementations available in `scikit-learn`. No statistically significant differences are apparent, which suggests that our re-implementations are accurate.

4 Comparison of baseline implementations and scikit-learn

In this section, we compare our re-implementation of common baselines to those in popular packages to verify the accuracy of our re-implementation. Specifically, we compare our implementations of Random Forest Classifiers, Random Forest Regressors, Extremely Random Forest Classifiers, and Extremely Random Forest Regressors to those of `scikit-learn`.

We omit comparisons of the Random Patches models as their correctness is implied by that of the Random Forest model, as the Random Patches model consists of applying the Random Forest model to subsampled data and features.

For classification, we compare our implementations on a the 20 newsgroups dataset filtered to two newsgroups, `alt.atheism` and `sci.space`. The dataset is embedded via TF-IDF and projected onto their top 100 principal components, following standard practice [36]. The train-test split is the standard one provided by `scikit-learn`.

For all classification problems, we average the predicted probabilities of each tree in the forest ("soft voting") as opposed to only allowing each tree to vote for a single class ("hard voting"), following the implementation in `scikit-learn` [36].

For regression, we compare our implementations on the California Housing dataset, subsampled to 1,000 points as performing the regression on the full dataset of approximately 20,000 points is computationally prohibitive. The train-test split is the standard one provided by `scikit-learn` [36].

Table 6 presents our results. In all cases, our re-implemented baselines do not present a statistically significant difference in performance than the models present in `scikit-learn`, which suggests that our re-implementations are correct. Performance is measured over 20 random seeds to compute confidence intervals.

5 Profiles

In this work, we focused on the reducing the runtime at the *algorithmic* level, i.e., reducing the complexity of computing the best feature-threshold split. In this section, we justify this choice by demonstrating that most of the time spent in our re-implementation of the baseline algorithms is spent in computing the best feature-threshold split.

Figure 2 demonstrates the wall-clock time spent inside various functions when fitting a Random Forest classifier on two subsets of the MNIST dataset of sizes 5,000 and 10,000. Most of the time is spent inside the computation of the best feature-threshold split, which scales approximately as dataset size and motivates our work's focus on improving the performance of the split-identification subroutine. When using MABSplit, the time spent to identify the best feature-threshold split is reduced significantly (Figure 3).

Figure 4 also contains an example callgraph demonstrating callers and callees for the fitting procedure of a Random Forest, for easier interpretation of Figures 2 and 3.

Incl.	Self	Called	Function	Location
28 530 355 227	63 786 876	(0)	fit	forest_base.py
28 463 927 621	3 352 727	5	fit	tree_base.py
28 080 739 176	51 124 826	1 174	calculate_best_split	node.py
28 029 614 350	10 035 680	154	solve_exactly	solvers.py
27 757 770 220	49 001 781	154	sample_targets	solvers.py
25 297 154 221	8 489 709 632	2 790	add	histogram.py
16 709 128 009	13 411 464	2 790	__call__	function_base.py
16 695 716 545	16 563 117 384	2 790	_vectorize_call	function_base.py
2 390 857 717	244 556 769	2 790	get_impurity_reductions	criteria.py
1 694 515 913	1 080 737 273	64 170	get_gini	criteria.py

(a)

Incl.	Self	Called	Function	Location
62 475 103 619	130 137 743	(0)	fit	forest_base.py
62 341 106 050	4 011 875	5	fit	tree_base.py
61 569 426 508	112 705 528	1 320	calculate_best_split	node.py
61 456 720 980	12 413 551	155	solve_exactly	solvers.py
61 137 274 586	78 318 817	155	sample_targets	solvers.py
58 108 915 088	19 636 444 229	3 037	add	histogram.py
38 288 693 888	27 780 631	3 037	__call__	function_base.py
38 260 913 257	38 008 248 809	3 037	_vectorize_call	function_base.py
2 922 821 783	300 147 575	3 037	get_impurity_reductions	criteria.py
2 069 337 645	1 329 814 899	69 851	get_gini	criteria.py

(b)

Appendix Figure 2: Profiles for the node-splitting algorithm using the exact solver/brute force computation, the canonical algorithm for computing the best feature-threshold split, for 5,000 (top) and 10,000 (bottom) datapoint subsets of MNIST. The "Function" column is the name of the called function, The "Incl." column is the time spent in the function and any called subroutines, and the "Self" column is the time (in nanoseconds) spent in only the function and *not* in any called sub-routines. All times are in nanoseconds. When increasing the dataset size, the overhead spent outside of the `solve_exactly` function grows negligibly from about 0.5 seconds to about 1 second. However, the time spent in the `solve_exactly` function and any called subroutines grows from about 28 seconds to about 61 seconds and constitutes approximately 98% of the increase in wall-clock time. This observation motivates our focus on improving the subroutine used to identify the best feature-threshold split. This profile was generated with `cProfile` and visualized with `pyprof2calltree` [28].

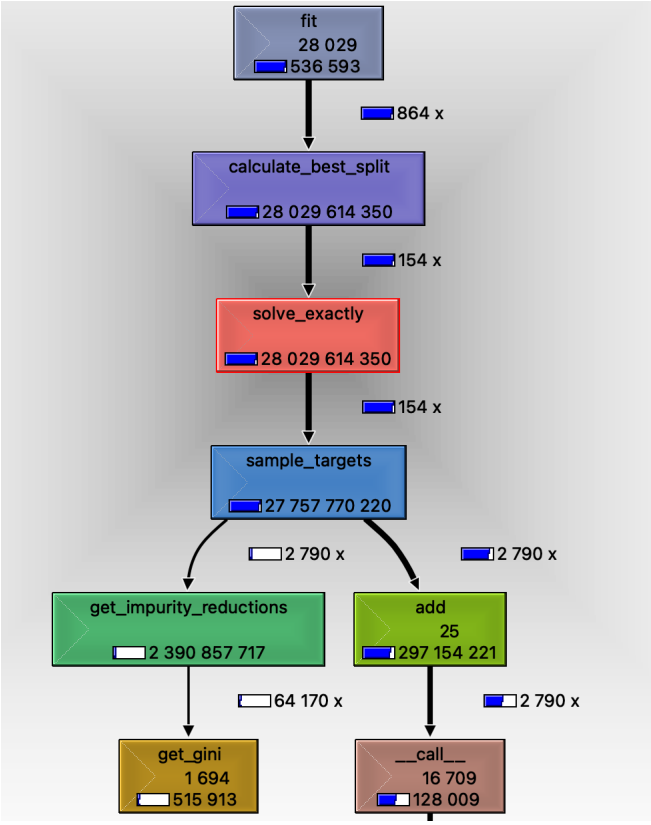
Incl.	Self	Called	Function	Location
19 661 963 096	41 969 883	(0)	fit	forest_base.py
19 617 643 699	3 092 763	5	fit	tree_base.py
19 230 991 254	50 655 813	1 235	calculate_best_split	node.py
19 180 335 441	23 821 486	153	solve_mab	solvers.py
18 876 412 727	54 401 292	171	sample_targets	solvers.py
16 264 631 640	5 516 196 393	2 874	add	histogram.py
10 655 097 887	14 682 830	2 874	__call__	function_base.py
10 640 415 057	10 522 707 847	2 874	_vectorize_call	function_base.py
2 525 591 375	263 120 240	2 874	get_impurity_reductions	criteria.py
1 780 190 442	1 136 355 365	64 900	get_gini	criteria.py

(a)

Incl.	Self	Called	Function	Location
35 191 513 750	146 189 424	(0)	fit	forest_base.py
35 037 137 603	4 275 199	5	fit	tree_base.py
34 213 584 573	148 596 702	1 302	calculate_best_split	node.py
34 064 987 871	33 510 699	155	solve_mab	solvers.py
33 666 053 648	99 185 719	201	sample_targets	solvers.py
30 175 621 314	10 467 512 322	3 286	add	histogram.py
19 550 499 483	28 674 755	3 286	__call__	function_base.py
19 521 824 728	19 318 153 735	3 286	_vectorize_call	function_base.py
3 336 452 440	354 850 749	3 286	get_impurity_reductions	criteria.py
2 340 068 097	1 498 557 797	72 086	get_gini	criteria.py

(b)

Appendix Figure 3: Profiles for the node-splitting algorithm using MABSplit, for 5,000 (top) and 10,000 (bottom) datapoint subsets of MNIST. The "Function" column is the name of the called function, The "Incl." column is the time spent in the function and any called subroutines, and the "Self" column is the time (in nanoseconds) spent in only the function and *not* in any called sub-routines. All times are in nanoseconds. When increasing the dataset size the time spent in the `solve_mab` function and any called subroutines only grows from approximately 20 seconds to approximately 35 seconds to identify the best feature-threshold split. This profile was generated with `cProfile` and visualized with `pyprof2calltree` [28].



Appendix Figure 4: Example call graph of the `fit` subroutine for the forest-based models in our re-implementation when the forest includes a single tree to be split only once. The `fit` method of the forest calls the `fit` method of its only tree, which calls `calculate_best_split` method of the root note, which calls the respective solver (`solve_exactly` for the canonical algorithm or `solve_exactly` for MABSplit), where the majority of wall-clock time is spent.

6 Experiment Details

Here we provide full details for the experiments in Section 5. All experiments were run on 2019 MacBook Pro running MacOS 12.3.1 (Monterey) with a 2.4 GHz 8-Core Intel Core i9 processor, 64 GB 2667 MHz DDR4 RAM, and an Intel UHD Graphics 630 1536 MB Graphics Card. In all experiments, trees of depth between 3 and 5 were trained.

6.1 Runtime Experiments

For the classification runtime experiments presented in Table 1, all models used the full MNIST train dataset of 60,000 images and test performance was measured on the 10,000 standard test images. Performance was averaged over 5 random seeds.

For the regression runtime experiments presented in Table 2, all models used data generated by `scikit-learn.datasets.make_regression`, with `n_samples=200000, n_features=50`, and `n_informative=3`. The first 160,000 datapoints were allocated to the training set and the remaining 40,000 to the test set; no validation set was constructed as no hyperparameter tuning was performed. The performance of the Random Forest model was averaged over 5 random seeds and the performances of the ExtraTrees and RandomPatches models were averaged over 20 random seeds.

6.2 Budget Experiments

For the classification budget experiments presented in Table 3, all models used the MNIST dataset subsampled to the first 10,000 points for training data, whereas performance was still measured on the full test dataset of 10,000 points. The budget (number of histogram insertions permitted) for each model was $784 * 10000 * 1.3 = 10192000$. The performance of each model was averaged over 5 random seeds.

For the regression budget experiments presented in Table 4, all models used data generated by `scikit-learn.datasets.make_regression`, with `n_samples=200000, n_features=50`, and `n_informative=3`. The first 160,000 datapoints were allocated to the training set and the remaining 40,000 to the test set; no validation set was constructed as no hyperparameter tuning was performed. The budgets of Random Forest, ExtraTrees, and Random Patches models were set to $24,000,000$, $1,200,000$, and $24,000,000$ respectively. The performance of the Random Forest model was averaged over 5 random seeds and the performances of the ExtraTrees and RandomPatches models were averaged over 20 random seeds.

6.3 Stability Experiments

Two measures for calculating feature importance are used in Table 5. The first is the out-of-bag Permutation Importance (OOB PI). For a feature f, the OOB PI is calculated by measuring the difference between the trained model's out-of-bag error on the original data with its out-of-bag error of all the data with all out-of-bag datapoints' f values shuffled. The second measure of feature importance is the Mean Decrease in Impurity (MDI). The MDI for a feature f is the average decrease in impurity of all nodes where f is selected as the splitting criterion.

Once feature importances have been calculated, the top k most important features for the model are selected and the stability of these k features is measured via standard stability formula [33].

The stability comparison experiments are conducted on various models with the settings described in Table 5. The Random Classification dataset is generated via `scikit-learn`'s `datasets.make_classification` function with `n_samples=10000, n_features=60`, and `n_informative=5`. The Random Regression dataset is generated by `scikit-learn`'s `datasets.make_regression` with `n_samples=10000, n_features=100`, and `n_informative=5`.

7 Limitations

7.1 Theoretical Limitations

Crucial to the success of MABSplit are the assumptions described before and after Theorem 1. In particular, we assume that their is reasonable heterogeneity amongst the true impurities of different feature-value splits. Such assumptions are common in the literature and have been validated on many real-world datasets [4, 48, 6, 43, 3, 5].

We also note that the assumptions that each CI scales as $\sqrt{\frac{\log 1/\delta}{n'}}$ may be violated when using certain impurity metrics. For example, the derivative of the entropy impurity criterion with respect to some p_k approaches ∞ when $p_k \to 0$. In this case, we cannot apply the delta method from Appendix 3 to compute finite CI intervals that scale in the way we require. In such settings, it may be necessary to compute the confidence intervals in other ways, e.g., following [35] or [7].

We note that in the worst case, even when all assumptions are violated, MABSplit is never worse than the brute-force algorithm in terms of sample complexity. In the worst case, it is a batched version of the brute-force algorithm.

7.2 Practical Limitations

We note that MABSplit may perform worse than brute-force node-splitting on very small datasets, where the overhead of sampling the data in batches outweighs any potential benefits in sample complexity.

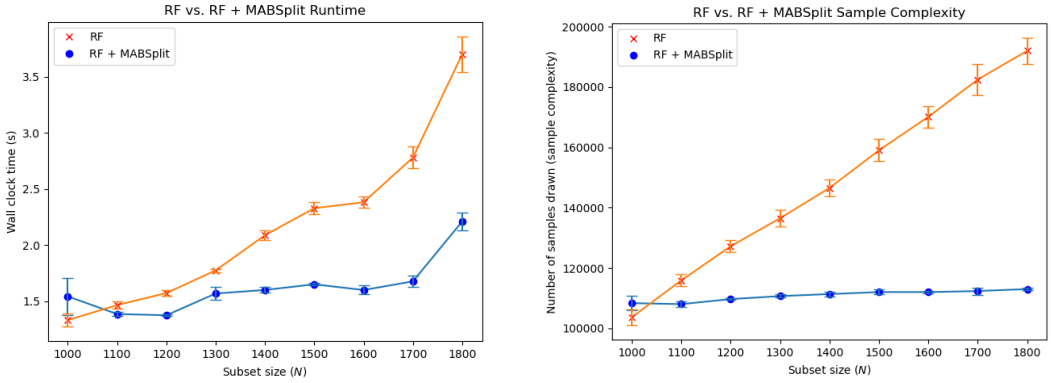
In this work, we avoided a direct runtime comparison with `scikit-learn` because `scikit-learn` utilizes a number of low-level implementation optimizations that would make the comparison unfair. In order for practitioners to take full advantage of MABSplit, however, it may be necessary to implement MABSplit within the `scikit-learn` library.

In doing so, it may be possible that MABSplit makes it difficult or impossible to use existing optimizations in the `scikit-learn` library. An example of this is vectorization: because the brute-force node-splitting algorithm queries the data in a predictable way, each datapoint can be queried more quickly than in MABSplit. Despite MABSplit's advantages in sample complexity, the disadvantages of being unable to use implementation optimizations like vectorization may outweight MABSplit's benefits.

Many of these risks may be ameliorated by int. We anticipate that many optimizations will still apply: for example pre-fetching data to have it in caches close to the CPU, manual loop unrolling, etc. We leave an optimization implementation of MABSplit inside the `scikit-learn` library to future work.

8 Comparison on Small Datasets

In this section, we investigate the performance of MABSplit on small datasets. Figure 5 demonstrates the performance of MABSplit, both in wall-clock training time and sample complexity, for various subset sizes of MNIST. Our results that RF+MABSplit outperforms the standard RF algorithm, in both sample complexity and wall-clock time, when the dataset size exceeds approximately 1100 datapoints. However, we also note that the main use case for MABSplit is when the data size is large and it is computationally challenging to run standard forest-based algorithms.



(a) (b)

Appendix Figure 5: (a) Wall-clock training times and (b) sample complexities of a random forest model with and without MABSPlit, for various subset sizes of MNIST. For dataset sizes below approximately 1000, the exact random forest model performs better in terms of sample complexity and wall-clock time. Above 1100 datapoints, the MABSplit version demonstrates better sample complexity and wall-clock time. Error bars were computed over 3 random seeds. Test performances were not different at a statistically significant level.