

Training Big Random Forests with Little Resources

Fabian Gieseke

Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
fabian.gieseke@di.ku.dk

Christian Igel

Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
igel@di.ku.dk

ABSTRACT

Without access to large compute clusters, building random forests on large datasets is still a challenging problem. This is, in particular, the case if fully-grown trees are desired. We propose a simple yet effective framework that allows to efficiently construct ensembles of huge trees for hundreds of millions or even billions of training instances using a cheap desktop computer with commodity hardware. The basic idea is to consider a multi-level construction scheme, which builds top trees for small random subsets of the available data and which subsequently distributes all training instances to the top trees' leaves for further processing. While being conceptually simple, the overall efficiency crucially depends on the particular implementation of the different phases. The practical merits of our approach are demonstrated using dense datasets with hundreds of millions of training instances.

1 INTRODUCTION

While large amounts of training data offer the opportunity to improve the quality of data mining models, they can also render both the generation and the application of such models very challenging. Ideally, one would like to take all available data into account during the training phase: Using more (i.i.d.) data can—in expectation—not decrease the generalization performance and improves theoretical generalization guarantees. Further, when searching for very rare patterns, ignoring parts of the training data or using subsampling strategies can lead to suboptimal models (simply randomly discarding training instances from the “negative” class can make it difficult to define the decision boundary around the rare “positive” instances). Such learning scenarios often occur in practice, for example in astronomy or remote sensing.

Ensemble methods are among the most successful models in data mining and machine learning [12, 19]. This is especially the case for random forests [3], which often yield very competitive accuracies while being, at the same time, conceptually simple and resilient against small changes of their hyperparameters [8]. Random forests have been extended and modified in various ways, e.g., to fit the requirements of special application domains or to build the involved trees in a parallel or distributed fashion in the context of large-scale learning scenarios. Ideally, one would like to build forests consisting of hundreds or even thousands of trees. However, depending on the data at hand, the construction of such forests can become extremely time- and memory-intensive.

For this reason, there has been a growing interest in developing frameworks and techniques that reduce the practical runtime for both the construction and the application of random forests. A popular line of research focuses on the construction of such tree

ensembles in a parallel or distributed way making use of many individual compute nodes (e.g., by constructing one tree per compute node). While this can significantly reduce the practical runtime, such frameworks naturally require expensive distributed computing environments. Further, the efficient construction of a single tree might cause problems in case the dataset or the tree becomes too large to fit into the main memory of a single system.

In this work, we propose a simple yet effective construction scheme for building random forests with fully-grown trees at large scale. The main idea is to build each of the involved trees in three phases: Starting with a top tree built from a small random subset of the data, one subsequently distributes all training instances to the leaves of that tree. For each leaf subset, one builds one or more associated bottom tree(s). The intermediate leaf subsets can be stored on hard disk and, subsequently, handled individually in parallel. Hence, by using such top trees, one essentially obtains a partition of the data into much smaller and, hence, manageable subsets. Our experimental evaluation shows that our implementation can efficiently handle learning scenarios with hundreds of millions of training instances using systems with both limited computational and memory resources. To the best of the authors' knowledge, no other publicly available implementation exists that can handle datasets of this size without resorting to compute clusters.

2 BACKGROUND

We start by providing the background related to the construction of large-scale random forests. Let $T = \{(x_1, y_1), \dots, (x_n, y_n)\} \subset \mathbb{R}^d \times \mathcal{Y}$ be a set of training patterns with $\mathcal{Y} = \mathbb{R}$ for regression and $\mathcal{Y} = \{c_1, \dots, c_k\}$ for classification. A random forest is an ensemble of M trees whose prediction $f(\mathbf{x})$ for a new pattern $\mathbf{x} \in \mathbb{R}^d$ is based on a combination of the predictions $f_m(\mathbf{x})$ made by the individual trees f_1, \dots, f_M , i.e.,

$$f(\mathbf{x}) = C(f_1(\mathbf{x}), \dots, f_M(\mathbf{x})), \quad (1)$$

where $C : \mathbb{R}^d \rightarrow \mathbb{R}$ depends on the learning scenario. For regression tasks, a common choice is $C(f_1(\mathbf{x}), \dots, f_M(\mathbf{x})) = \frac{1}{M} \sum_{m=1}^M f_m(\mathbf{x})$, whereas $C(f_1(\mathbf{x}), \dots, f_M(\mathbf{x})) = \text{argmax}_{c \in \mathcal{Y}} |\{i \mid f_i(\mathbf{x}) = c\}|$ is the standard choice for classification tasks [3, 12, 19].

A tree is built recursively starting from the root and a subset $T' \subseteq T$ of the training data. Each node splits the available data into two subsets, which are used to build two subtrees becoming the children of the node. A node becomes a leaf when the associated training data subset is *pure* (i.e., only instances with the same label are left) or some other stopping criterion is fulfilled (e.g., a maximum tree depth is reached). A simple example is given in Figure 1. For an internal node corresponding to a subset $S \subseteq T'$ of training instances, one searches for a splitting dimension i and a threshold

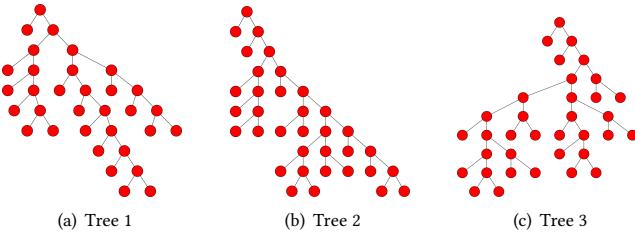


Figure 1: A random forest consisting of three trees.

Algorithm 1 BUILDRANDOMFOREST(T, B, F)

Require: $T = \{(x_1, y_1), \dots, (x_n, y_n)\} \subset \mathbb{R}^d \times \mathcal{Y}$, $B \in \mathbb{N}$, and $F \in \{1, \dots, d\}$.
Ensure: Trees $\mathcal{T}_1, \dots, \mathcal{T}_B$ for T .

- 1: **for** $b = 1, \dots, B$ **do**
- 2: Draw bootstrap sample T' from T
- 3: $\mathcal{T}_b = \text{BUILDTREE}(T', F)$
- 4: **end for**
- 5: **return** $\mathcal{T}_1, \dots, \mathcal{T}_B$

Algorithm 2 BUILDTREE(S, F)

Require: Set $S \subseteq T$ and $F \in \{1, \dots, d\}$.
Ensure: Tree \mathcal{T} built for S

- 1: **if** S is pure (or some other criterion fulfilled) **then**
- 2: **return** leaf node
- 3: **end if**
- 4: $(i^*, \theta^*) = \text{argmax}_{i \in \{i_1, \dots, i_f\} \subseteq \{1, \dots, d\}, \theta} G_{i, \theta}(S)$
- 5: $\mathcal{T}_l = \text{BUILDTREE}(L_{i, \theta})$
- 6: $\mathcal{T}_r = \text{BUILDTREE}(R_{i, \theta})$
- 7: Generate node storing the pair (i^*, θ^*) and pointers to its subtrees \mathcal{T}_l and \mathcal{T}_r . Let \mathcal{T} denote the resulting tree.
- 8: **return** \mathcal{T}

θ that maximize the *information gain*

$$G_{i, \theta}(S) = Q(S) - \frac{|L_{i, \theta}|}{|S|} Q(L_{i, \theta}) - \frac{|R_{i, \theta}|}{|S|} Q(R_{i, \theta}) \quad (2)$$

with $L_{i, \theta} = \{(\mathbf{x}, y) \in S \mid x_i \leq \theta\}$, $R_{i, \theta} = \{(\mathbf{x}, y) \in S \mid x_i > \theta\}$, by minimizing the “impurity” of the subsets assessed by an *impurity measure* Q [3, 12, 19].

The overall construction of a random forest is sketched in Algorithm 1. An ensemble model exploits the diversity of its members. For standard random forests, one usually considers M subsets of the training patterns. These *bootstrap samples* are drawn uniformly at random (with replacement) from T to obtain slightly different training datasets and, hence, trees. Another strategy to introduce randomness is to vary the splitting mechanism at the internal nodes during the construction by, e.g., considering different subsets of features for each node split among which the one with the best splitting quality is selected (or by considering random splitting thresholds, see below). The leaves of a single tree store the label information. For regression problems, one usually computes the mean of all labels associated with the leaf, whereas the most frequent label is stored for classification scenarios (or the distribution over the labels). The prediction $f_m(\mathbf{x})$ for a single tree is obtained

by traversing the tree from top to bottom based on the splitting thresholds stored in the internal nodes until a leaf node is reached. The associated label then determines the prediction of the tree.

2.1 Large-Scale Construction

The construction discussed so far corresponds to standard random forests as proposed by Breiman [3]. Various alternatives have been suggested over the past years. A popular one is the concept of *extremely randomized trees* [10], which is based on “random” thresholds for each feature i in Line 4 of Algorithm 2 (more precisely, a random value between the minimum and the maximum is considered). In practice, resorting to these potentially “suboptimal” splits often yields competitive and sometimes even superior tree ensembles. Further, training such variants might be faster.

Many different random forest implementations have been proposed during the past years. The efficiency of the recursive construction of the individual trees via BUILDTREE heavily depends on the particular random forest variant that is considered and on the heuristics being used to speed up the process. Today’s state-of-the-art software makes use of various other implementation tricks to reduce the practical runtime.¹ A recent trend in data mining is to make use of massively-parallel devices such as graphics processing units (GPUs) to accelerate the generation and application of random forests [7, 11, 13, 23]. However, aiming at full, unpruned trees, such approaches do not seem to improve over a standard multi-core execution. Another line of research considers variants that are tailored towards special cases. For instance, Louppe and Geurts [18] consider small subsets of the data, called *patches*. Each patch is based on a different subset of features and the overall ensemble consists of trees built independently on the patches.

In addition, distributed construction schemes have been proposed that are based on, e.g., *MapReduce* [5]. Several strategies to implement random forests via the MapReduce framework are described by del Río *et al.* [6]. A natural one (which is also implemented by the Apache Mahout™ library) is to consider subsets of the data and to build individual trees/forests for each of these subsets; the overall ensemble is then composed of all individual trees. The PLANAT implementation [21] also resorts to MapReduce. Similarly to our work, it also stops the recursive construction as soon as the subsets become small enough to be handled by a single machine. However, the overall implementation aims at distributed computing environments and the construction of the upper parts of the trees are handled in a conceptually very different way. Further, the trees are built independently from each other.

Finally, efficient implementations exist for the related task of computing ensembles of boosted trees [4]. These ensembles, however, rely on many shallow trees that are built in an iterative fashion. Accordingly, such implementations do not necessarily perform well when building deep, fully-grown trees.

¹For instance, one can keep track of “locally constant” features (i.e., a dimension i does not need to be checked anymore for S and all the descendant nodes in case all patterns in S exhibit the same value w.r.t. dimension i). Further, it is beneficial to represent a bootstrap sample T' via weights instead of duplicating training instances [17]. A popular well-engineered implementation based on highly-tuned C code is provided by the *Scikit-Learn* [22] package.

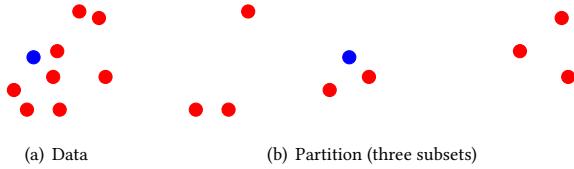


Figure 2: An ensemble built via the three subsets cannot identify the blue object anymore since two of the three models do not contain this instance. Such an effect can also be observed in case $P(y|x)$ is unbalanced for a subregion x .

2.2 Deep Trees

The original random forest implementation proposed by Breiman [3] grows full trees. A simple variant is to build trees up to a certain depth only. While the validation and test accuracies might still be good, the induced forest might lose its capability to deal with rare classes. The so-called m -out-of- n subset strategy partitions the dataset into subsets and builds individual trees/forests for these subsets. While one makes use of all the data, this strategy might yield non-optimal results as well. As discussed by Genuer *et al.* [9] and del Río *et al.* [6], such subset strategies usually cause a shift towards the dominant classes and instances (in a certain region of the feature space). For example, assume that one is given a single “rare” instance (i.e., $P(y = c)$ very small for a class c) and assume that one considers a partition of the data into three equal-sized subsets. The rare object would only be contained in one of the subsets and, hence, would never be predicted via the overall ensemble.

Reweighting strategies applied in a post-processing phase aim at reducing these negative side-effects, see, e.g., del Río *et al.* [6] for several adaptions. However, finding appropriate weights is a challenging task as well and such methods generally focus on shifts in $P(y)$ introduced by subsampling. Such a bias towards a dominant class/label can also occur in a *region of the feature space*, as sketched in Figure 2. Hence, even in case a reasonable amount of labeled instances is given for a rare class, subsampling strategies might lead the model to completely ignore this class, see Genuer *et al.* [9] for a detailed discussion.

3 ALGORITHMIC FRAMEWORK

We propose a wrapper-based approach that can handle massive datasets given single compute node resources.

3.1 Wrapper-Based Construction

We start by outlining the construction of a single tree; the overall implementation simultaneously constructs all trees, which is described in the next section. The basic idea is to build a “top tree” based on a small *random* subset of the training data and to use this tree to obtain a partition of *all* the training instances into (ideally) almost equal-sized subsets.

The overall workflow is shown in Algorithm 3: In the first phase, a top tree \mathcal{T} is built for a small subset S of T drawn uniformly at random (without replacement). Afterwards, all available training instances are distributed to the leaves of the top tree. That is, one

Algorithm 3 BUILDBIGTREE

Require: $T = \{(x_1, y_1), \dots, (x_n, y_n)\} \subset \mathbb{R}^d \times \mathcal{Y}$, $F \in \{1, \dots, d\}$, subset size R , and leaf bucket size M .
Ensure: Tree \mathcal{T} built for T

- 1: Retrieve random subset $S \subset T$ with $|S| = R$
- 2: $\mathcal{T}_{top} = \text{BUILDTopTree}(S, M)$
- 3: $T_1, \dots, T_N = \text{DISTRIBUTE}(\mathcal{T}_{top}, T)$
- 4: $\mathcal{T} = \emptyset$
- 5: **for** $j = 1, \dots, N$ **do**
- 6: $\mathcal{T} = \mathcal{T} \cup \text{BUILDTree}(T_j)$
- 7: **end for**
- 8: **return** \mathcal{T}

Algorithm 4 BUILDTopTree

Require: Set $T' = \{(x_{i_1}, y_{i_1}), \dots, (x_{i_R}, y_{i_R})\} \subset \mathbb{R}^d \times \mathcal{Y}$ leaf desired bucket size M .
Ensure: Root node of a binary tree \mathcal{T}_{top} .

- 1: Create empty stack \mathcal{P} ; create root node N_0
- 2: $i_{node} = -1$
- 3: $\mathcal{P}.push(T', N_0)$
- 4: **while** \mathcal{P} is not empty **do**
- 5: $i_{node} = i_{node} + 1$
- 6: $(\bar{T}, \bar{N}) = \mathcal{P}.pop()$
- 7: **if** $|\bar{T}| < \max(2, M \cdot \frac{R}{n})$ **then**
- 8: $\bar{N}.value = i_{node}$
- 9: **return** \bar{N}
- 10: **end if**
- 11: $(j^*, \theta^*) = \text{argmax}_{(j, \theta)} \bar{G}_{i_{node}}(S)$
- 12: Split \bar{T}' into T'_l and T'_r according to (j^*, θ^*)
- 13: Create left node \bar{N}_l of \bar{N} and $\mathcal{P}.push(T'_l, \bar{N}_l)$
- 14: Create right node \bar{N}_r of \bar{N} and $\mathcal{P}.push(T'_r, \bar{N}_r)$
- 15: **end while**
- 16: **return** N_0

determines for each training instance (x_i, y_i) the index of the leaf within the top tree the pattern x_i would be assigned to. Finally, in the third phase, one computes for each of the induced subsets T_1, \dots, T_M an associated bottom tree, which is then attached to the corresponding leaf of the top tree. This yields the final tree \mathcal{T} . Thus, the overall tree \mathcal{T} basically corresponds to a standard tree built via BUILDTREE with potentially slightly different splits conducted due to the random subset used for the top tree. A direct implementation of this approach, however, does not necessarily yield an efficient implementation due to, e.g., potentially very unbalanced top trees. The modifications needed to render this approach efficient are described next.

3.1.1 Construction of Top Trees. Since the top tree is only built on a small subset, different and potentially “suboptimal” splits might be considered compared to a direct construction via BUILDTREE. Note, however, that the notion of “optimal” is anyway vague in this context. For instance, extremely randomized trees, which resort to simple random splitting thresholds, often even yield competitive or even superior overall ensembles compared to their counterparts that rely on “optimal” splitting thresholds.

In practice, simply resorting a standard BUILDTREE construction scheme might not yield a feasible approach. This is due to the fact that, for some datasets, the top trees might become very unbalanced,

leading to many “small” leaves that do not contain many training instances after the distribution phase. In addition, the standard splitting scheme might yield very big leaves after the distribution phase; while these leaves might be “pure” after the construction phase of the top tree, they might become unpure again after the distribution phase. This actually is a problem since the induced big leaves still have to be processed in the third phase—and this might not be possible given the restricted resources (e.g., such a big leaf might contain hundreds of millions of patterns).

For this reason, we adapt the construction of the top tree, see Algorithm 4: The workflow is essentially the same except for the following two minor yet crucial modifications:

- (1) *Minimal leaf size stopping*: Firstly, the recursive construction only stops as soon as the minimal leaf size is reached. By doing so, we ensure that the resulting leaf buckets are small enough for the further processing (otherwise, almost pure leaves might yield very big leaf buckets). The parameter M specifies the desired maximum size of a leaf bucket *after* the distribution phase. Since the actual number of assigned instances is only known after the distribution of all instances, we resort to an estimate $\bar{M} = \max(2, M \cdot \frac{R}{n})$ for M .
- (2) *Balanced splits*: Secondly, to handle degenerated cases, we consider the following modified information gain criterion $\bar{G}_{j,\theta}(S)$ that favors balanced partitions in the top tree:

$$\bar{G}_{j,\theta}(S) = (1 - \lambda)G_{j,\theta}(S) - \lambda \frac{\|L_{j,\theta}\| - \|R_{j,\theta}\|}{|S|} \quad (3)$$

The second part in the above objective favors “balanced” partitions, which are similar to those done for standard k -d trees [2].² The parameter $\lambda \in [0, 1]$ determines the tradeoff between the standard information gain and favoring balanced partitions.

Finally, no bootstrap samples are drawn for the construction of top trees as well as optimal splits w.r.t. (3) are considered. Note that the adapted information gain is especially important for splits of almost pure nodes. Here, the first part of (3) will yield similar gains for various thresholds. However, the second part will enforce the splits to be balanced.

An example for a very unbalanced tree with a single leaf containing most of the patterns is given in Figure 3 (which is based on the landsat-osm dataset, see Appendix B). The few very big leaves depict a problem since they might become unpure again after the distribution phase and, hence, still have to be considered and processed in the third phase. The adapted construction scheme outlined above with $\lambda = 1$ yields almost equal-sized partitions, all being sufficiently “small” such that the leaves will contain about M patterns *after* the distribution phase.

Note that the adapted splitting scheme actually continues splitting up pure nodes until a leaf size of \bar{M} is reached. While this seems like an unnecessary operation, it is crucial to obtain manageable partitions for the third phase, the construction of the bottom trees. A potential drawback of this approach is that more splits than needed are actually conducted. Since one only knows about the properties of the final leaves after the distribution phase, this cannot be avoided. Further, favoring balanced partitions using, e.g.,

²Note that using the median does not necessarily yield almost equal-sized partitions.

Algorithm 5 DISTRIBUTE

Require: A set $T = \{(x_1, y_1), \dots, (x_n, y_n)\} \subset \mathbb{R}^d \times \mathcal{Y}$ of training patterns and a top tree \mathcal{T}_{top} .

Ensure: A partition T_1, \dots, T_N of T .

- 1: $LI = \text{GETLEAVESINDICES}(T)$
 - 2: $T_1, \dots, T_N = \text{PARTITION}(LI, T)$
 - 3: **return** T_1, \dots, T_N
-

$\lambda = 1$, might yield to “similar” top trees and, hence, less randomness in the upper parts of the final trees. Also, the splits conducted might be suboptimal w.r.t. the original information gain criterion $G_{j,\theta}(S)$, which might yield to suboptimal top parts of the trees (e.g., no feature selection is conducted).³

3.1.2 Distribution & Construction of Bottom Trees. Given the top tree, all training instances are distributed to the top tree’s leaves, see Algorithm 5. The top tree is modified in such a way that the leaf index is returned for a query instead of a (label-based) prediction. The resulting indices can then be used to partition all the training data T to the different leaf buckets.

Finally, one or more bottom trees are built for each of the leaf buckets T_1, \dots, T_N . The number $n_b \geq 1$ of bottom trees built per bucket can be defined by the user. For large-scale scenarios, sharing top trees among the different overall trees can be computationally very advantageous since less calls to DISTRIBUTE are needed (hence, passes over the data) and since the construction of all bottom trees can effectively be parallelized (using the same chunk of data fitting in the system’s memory). However, sharing top trees that way generally leads to less randomness in the overall ensemble, which might reduce the model’s quality. From a practical perspective, this depicts a trade-off between runtime and tree diversity.

3.2 Implementation

The wrapper-based construction outlined above yields much smaller partitions associated with the leaves of the top tree, which can be handled more efficiently. Its efficiency, however, depends on a careful implementation of the involved steps.

Some of the steps are conducted simultaneously for the different trees to be built. More precisely, the construction of the top trees as well as the distribution of the patterns are done via two single passes over the training instances. Each pass is conducted by processing all the data in chunks using a certain chunk size C (e.g., $C = 1,000,000$). In the first pass over the data, random subsets are extracted for the top trees. Given the subsets, the top trees are built, which are then used to distribute all instances to the top trees’ leaves. Note that considering random subsets (based on a full pass over the data) might be crucial for obtaining good estimates $\bar{M} = \max(2, M \cdot \frac{R}{n})$. During the distribution phase, a new subset of training instances is created for each leaf bucket, which is either stored in memory or on disk (using HDF5 [24]).

The construction of all top trees can be done using $O(R + C)$ memory, where R is the size of a single random subset and C the

³For large-scale scenarios with millions or even billions of training patterns, these potential drawbacks do not seem to have a significant influence (basically, only a few suboptimal splits are conducted in the upper parts of the generally very deep trees). Note that the adapted node splitting (in case $\lambda = 1$ is used) is related to Mondarian Forests [15], which only resort to label-independent node splits.

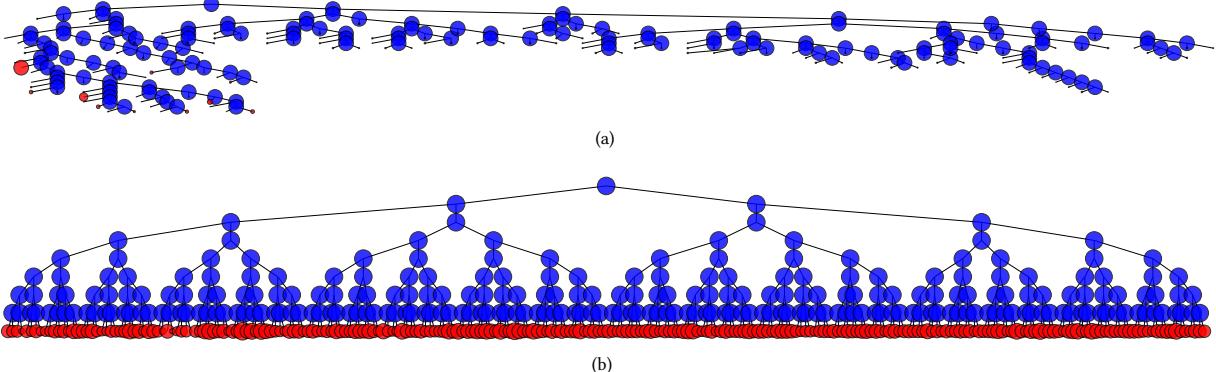


Figure 3: Two top trees built via BUILDTOPTREE using $\bar{G}_{j,\theta}(S)$ with (a) $\lambda = 0$ and (b) $\lambda = 1$, respectively. The size of a leaf (red) is proportional to the number of points assigned to it. The standard construction scheme that stops as soon as a node is pure and which resorts to the normal information gain ($\lambda = 0$) might yield very unbalanced leaves (a few leaves contain almost all instances!). The adapted construction scheme with $\lambda = 1$ yields very balanced partitions. In expectation, all these leaves will contain about M leaves after the distribution phase and are, hence, small enough for the construction of bottom trees.

Table 1: Datasets

Name	n_{train}	n_{test}	d
covtype	464,809	116,203	54
susy	5,000,000	500,000	18
higgs	11,000,000	1,000,000	28
landsat-osm	1,000,000,000	2,964,607	81

chunk size. Further, the distribution of the instances can be done spending $O(R + C)$ additional memory as well. Finally, for the third phase, one needs $O(\hat{M})$ memory with \hat{M} being the maximal size of a leaf bucket after the distribution phase.

The general wrapper-based framework is implemented in Python (version 2.7.11), where the *Numpy* package (version 1.11.2) [14] is used for all matrix/vector-based operations. For the computation of both the top and the bottom trees, we resort to a pure C implementation that follows the construction scheme implemented by the *Scikit-Learn* package (version 0.18.1) [22]; *Swig* [1] is used to generate a Python extension. The overall implementation is made publicly available on <https://github.com/gieseke/woody> under the *GNU General Public License v3.0*.

4 EXPERIMENTS

We consider a standard multi-core machine for all experiments and compare our approach with two state-of-the-art competitors. The experiments can be reproduced using the code made available on <https://github.com/gieseke/woody>.

4.1 Experimental Setup

All experiments were conducted on a standard desktop computer with an Intel(R) Core(TM) i7-3770 CPU at 3.40GHz (4 cores, 8 hardware threads), 16GB RAM, and 16GB swap space. The operating system was Ubuntu 16.04 (64 Bit). In general, we focus on runtimes

for the construction phases as well as on the accuracies obtained on the test set. If not stated otherwise, all results reported are averages over four runs with different seeds being used for the initialization. Our approach can, in principle, be applied to any kind of random forest variant (e.g., extremely randomized trees). For the sake of simplicity, we resort to standard random forests as sketched in Section 2. For the experiments, we consider the following four implementations:

- (1) The first one is the wrapper-based construction scheme proposed in this work, referred to as *woody*.
- (2) The second one, *subsets*, uses subsets drawn uniformly at random from *all* the available training instances. For each such subset, a single classification tree is built. The overall implementation resembles the wrapper-based implementation outlined above, i.e., random subsets are extracted via a pass over all instances. However, instead of top trees, standard trees are built for these subsets. The remaining points are *not* distributed/considered. Intermediate results are stored on disk, as it is done for *woody*.
- (3) The third one, *sklearn*, is the *RandomForestClassifier* implementation provided by the *Scikit-Learn* [22] package.
- (4) The fourth one, *h2o*, is the implementation provided by the *H2O* package (*H2ORandomForestEstimator*).⁴

Most parameters are set to their default values. Some of the parameters are automatically adapted to the specific dataset at hand, see Appendix A for the details. We focus on classification scenarios to assess the classification performances. In all cases, we consider a separate test set for evaluating the classification accuracy. For the different experiments, n_{train} training instances, n_{test} test instances, and d features are considered, see Table 1. Despite the three medium-sized datasets covtype, susy, and higgs [16], we also consider landsat-osm, a large-scale dataset from the field of remote sensing containing up to one billion training instances, see Appendix B for details.

⁴<http://docs.h2o.ai>

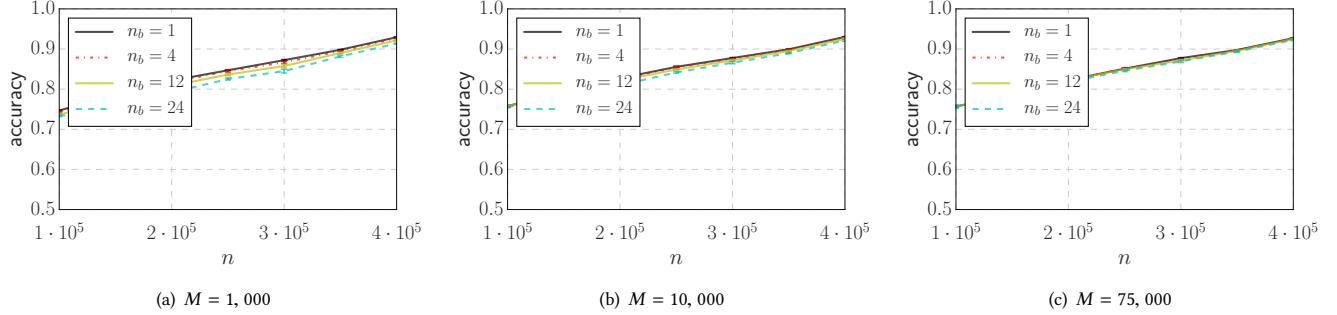


Figure 4: Influence of the number n_b of bottom trees per top tree on the accuracy given the covtype dataset. For each result (line), 24 trees are built in total. Three different woody instances are considered that are induced by different assignments for M .

4.2 Model Parameters

We start by analyzing the influence of two of the main model parameters introduced by the wrapper-based scheme.

4.2.1 Influence of n_b . The initial two phases can consume a significant part of the overall runtime. Especially the distribution phase involves extracting many large subsets that might have to be stored on disk. To reduce the overhead for these two phases, one can construct $n_b > 1$ bottom trees per top tree. This essentially leads to final trees sharing upper parts, which, in turn, might lead to less randomness in the overall ensemble. To investigate the influence of this parameter, we consider the covtype dataset and three different instances of woody induced by different assignments for M : $M = 1,000$, $M = 20,000$, and $M = 75,000$. If M is small, larger top trees will be built. For large M , the top trees will have small sizes. Hence, we expect a slightly worse performance for small M and a competitive performance for large M .

We consider 1, 4, 12 and 24 as assignments for n_b . In all cases, 24 are built in total (e.g., $n_b = 24$ and $n_t = 1$). The results are shown in Figure 4. As expected, the accuracies are slightly worse for large n_b . However, the differences are generally very small, indicating that sharing upper parts does not significantly hurt the performance. Further, the differences between the three instances of woody for varying M are very small as well, which indicates that sharing larger parts does not lead to a significant drop w.r.t. the accuracy as long as sufficiently large bottom trees are built.

4.2.2 Influence of λ . We consider two models to investigate the influence of λ : A standard random forest implementation that resorts to the adapted information gain criterion as well as woody. In both cases, we consider the covtype dataset and different assignments for λ ($\lambda = 0, 0.2, \dots, 1.0$). Both ensembles consider of 24 trees, where $n_t = 6$ and $n_b = 4$ are used for woody.

The outcome is shown in Figure 5: As expected, smaller values for λ generally yield slightly better accuracies. This is especially the case for the standard random forest implementation, which builds the full trees using the adapted criterion. However, the results also indicate that the adapted information gain does not severely affect the accuracy. Also, the accuracies still improve in general the more data are taken into account.⁵ Finally, the influence of λ is even less

⁵These results are in line with the ones reported for Mondarian Forests [15], which resort to a label-independent information gain.

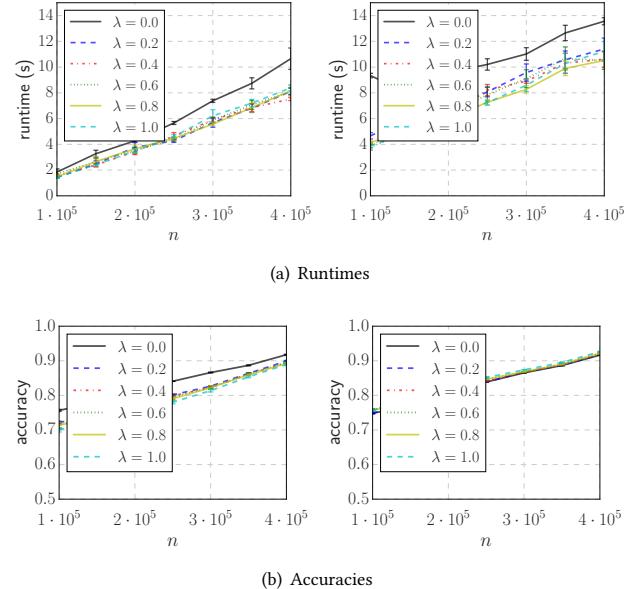


Figure 5: Influence of λ . Left: Standard random forest with adapted information gain G . Right: The wrapper-based approach woody with the top tree being built using \bar{G} .

for woody, which is due to the fact that the wrapper-based approach only resorts to \bar{G} for the top trees. To conclude, we observe that the accuracy does not seem to be severely affected (see also below), especially in case \bar{G} is used for the construction of the top trees only. However, balanced splits are important to quickly reduce the nodes' sizes for woody. This is especially the case for almost pure nodes, which still have to be split to achieve the desired leaf sizes M .

4.3 Small Data

Next, we compare the training times and test accuracies induced by the covtype, susy, and higgs datasets. Again, we consider ensembles consisting of 24 classification trees ($n_t = 6$ and $n_b = 4$ for woody); all other parameters are set to the values described in Appendix A. The results are shown in Figure 6: Except for subsets, all implementations yield similar results, with h2o exhibiting a

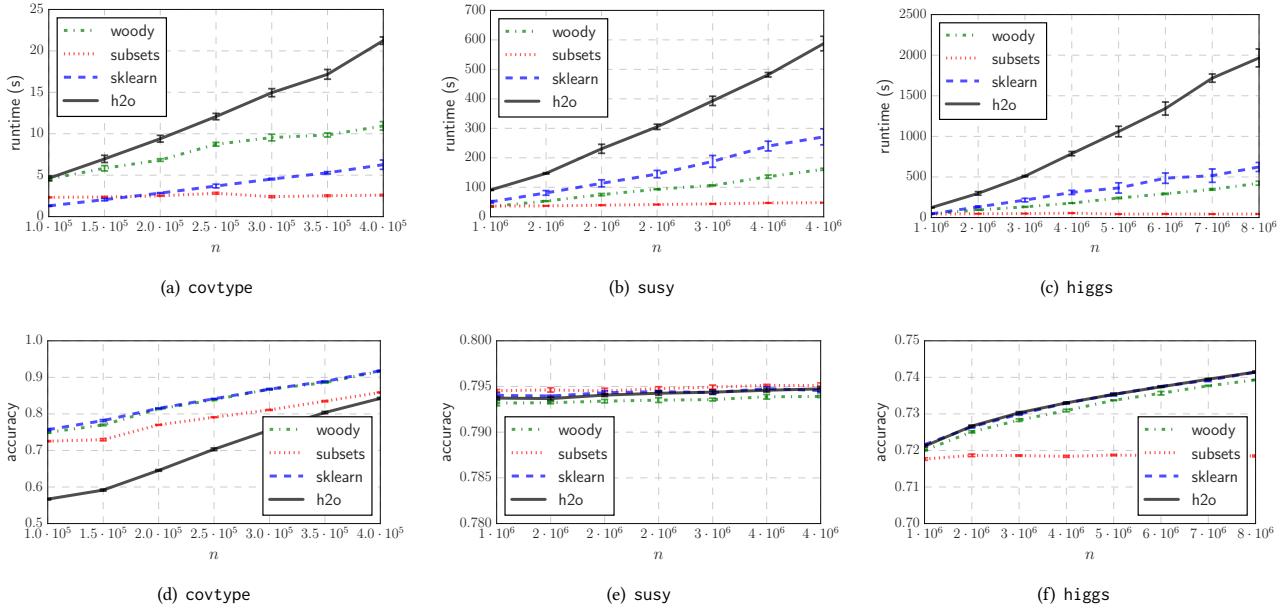


Figure 6: Training runtimes and test accuracies for the three medium-sized datasets (mean runtimes/accuracies as well as one standard deviations based on four runs with different seeds). The runtimes and accuracies are very similar to each other, indicating that the changes made for woody do not significantly affect the outcome.

slightly worse classification accuracy on the covtype dataset (most likely due to the maximum tree depth of 20 not being sufficient for this dataset). The training runtimes are also similar among all implementations, with woody being slightly faster than sklearn on both the susy and the higgs dataset and slightly slower on the covtype dataset. The subsets scheme performs worse on both the covtype and higgs dataset. In both cases, it can be seen that not taking all the available training instances into account leads to a “stagnating” and slightly worse accuracy.⁶

To conclude, all implementations yield very similar accuracies and the modifications incorporated for woody do not seem to severely affect the performance. However, the wrapper-based construction scheme can successfully incorporate significantly larger datasets, which potentially yields much better ensembles (see below). The direct competitor, subsets, which only builds trees for subsets of instances, does not seem to benefit from more data, as it is indicated by the worse performance on both covtype and higgs. Finally, we would like to stress that *fully-grown* trees are obtained via woody, which might be a crucial for dealing with rare instances.

4.4 Big Data

Next, we make use of the landsat-osm dataset described in Appendix B and consider up to 50 million training instances; the classification accuracies are evaluated on about three million test instances. We consider the same parameters for the different implementations as before and consider 12 estimators ($n_t = 3$ and $n_b = 4$ as well as $\lambda = 1.0$ for woody). Intermediate results are now stored on disk instead in main memory for woody and subsets.

⁶While subsets performs slightly better on susy, we do not consider the differences to be relevant (less than 0.2% differences w.r.t. the classification accuracy).

The results are shown in Figure 7. It can be seen that both woody and subsets are able to successfully process all training instances. Further, taking more training instances into account leads to better accuracies. The average accuracy of subsets is also slightly worse than the one of woody. The other two implementations could not handle these large-scale scenarios well: The sklearn implementation was only able to deal with ten million training instances without running into memory problems. For this single dataset instance, it yielded an average test accuracy of 0.72, which was about the same as the one achieved by woody in this case. While the h2o implementation could process dataset instances with up to 30 million instances, the average test accuracy was below 0.24 in all cases (not shown). To conclude, woody is capable of taking all the available training instances into account. While the improvement over subsets w.r.t. the accuracy might be moderate for the dataset at hand, we would like to stress woody’s capability to build full trees while taking all the training instances into account—which can be crucial for correctly classifying rare instances.

Finally, we make use of all one billion training instances given in the landsat-osm dataset and evaluate the runtime behavior of the woody implementation ($n_t = 1$ and $n_b = 4$). This dataset contains very dominant classes (e.g., the ‘water’ class) and, thus, can lead to very unbalanced top trees in case the standard information gain criterion G is used. However, using the modified information gain \tilde{G} with $\lambda = 1$ yields balanced top trees with only few leaves (about 1,000). Hence, such top trees can be used to successfully partition huge datasets into smaller chunks. The runtimes of the different phases (single run) are shown in Figure 8. It can be seen that woody can efficiently handle the one billion training instances with the

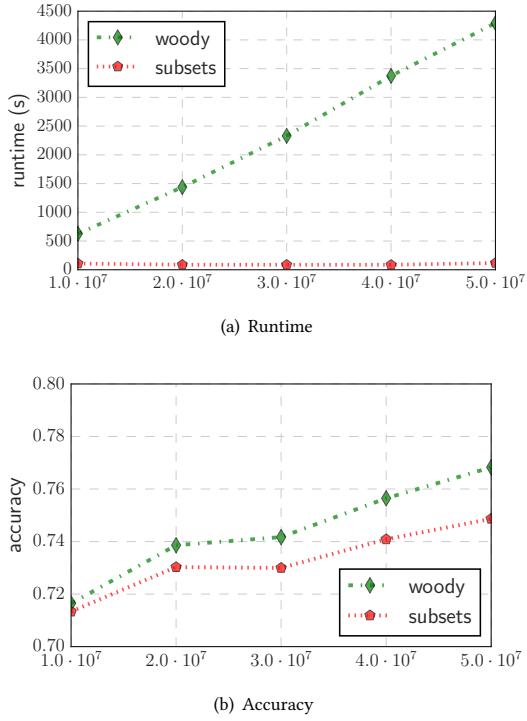


Figure 7: Training runtimes and test accuracies for the landsat-osm dataset with up to 50 million instances. The sklearn implementation could not handle more than 10 million training instances due to memory errors. While the h2o implementation could process the dataset instances, the accuracy on the test set was below 0.24 in all cases.

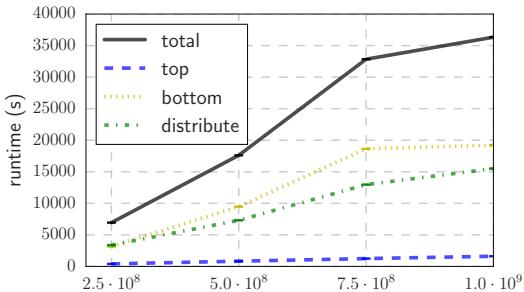


Figure 8: Runtimes for the different phases of woody for the landsat-osm dataset using up to one billion instances.

distribution and bottom tree construction phases dominating the practical runtime.

5 CONCLUSION

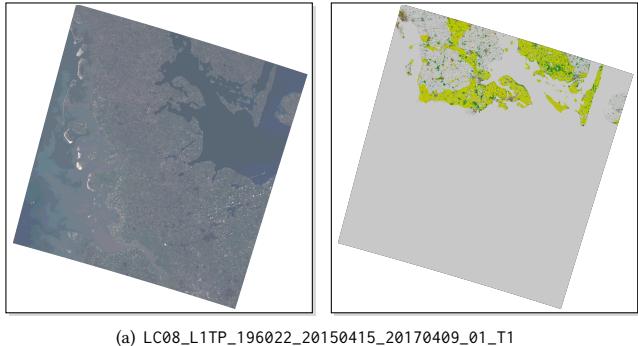
We propose woody, a wrapper-based construction framework for building large random forests for hundreds of millions of training instances. The key idea is to use top trees to partition all the available training data into smaller subsets associated with the top trees' leaves and to build bottom trees for these subsets. While

being conceptually simple, the framework allows the construction of ensembles with fully-grown trees for very large datasets. The practical benefits of our approach were empirically demonstrated on three medium-sized datasets and a large-scale application from the field of remote sensing with up to 10^9 data points.

We expect these results to carry over to other learning tasks making woody a powerful tool for mining big datasets—without requiring expensive compute resources. To the best of our knowledge, woody is the first implementation that renders the construction of random forests possible for datasets containing hundreds of millions of instances using a standard desktop computer. We think that the woody implementation—made publicly available on <https://github.com/gieseke/woody>—will be of significant practical importance for many real-world tasks in future.

REFERENCES

- [1] David M. Beazley. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4 (TCLTK'96)*. USENIX Association, Berkeley, CA, USA, 15–15. <http://dl.acm.org/citation.cfm?id=1267498.1267513>
- [2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used For Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.
- [3] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.
- [4] 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*. ACM, 785–794.
- [5] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [6] Sara del Rio, Victoria López, José Manuel Benítez, and Francisco Herrera. 2014. On the use of MapReduce for imbalanced big data using Random Forest. *Information Sciences* 285 (2014), 112–137.
- [7] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. 2012. Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 232–239.
- [8] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *Journal of Machine Learning Research* 15 (2014), 3133–3181.
- [9] Robin Genier, Jean-Michel Poggi, Christine Tuleau-Malot, and Nathalie Villa-Vialaneix. 2015. Random Forests for Big Data. *eprint arXiv:1511.08327v1* (2015).
- [10] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63, 1 (2006), 3–42.
- [11] Håkan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat. 2011. CudaRF: A CUDA-based implementation of Random Forests. In *The 9th IEEE/ACS International Conference on Computer Systems and Applications*. IEEE Computer Society, 95–101.
- [12] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning*. Springer.
- [13] Karl Jansson, Håkan Sundell, and Henrik Boström. 2014. gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE Computer Society, 1612–1621.
- [14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–2018. SciPy: Open source scientific tools for Python. <http://www.scipy.org>. (2001–2018).
- [15] Balaji Lakshminarayanan, Daniel M. Roy, and Yee Whye Teh. 2014. Mondrian Forests: Efficient Online Random Forests. In *Advances in Neural Information Processing Systems*. 3140–3148.
- [16] Moshe Lichman. 2013. UCI Machine Learning Repository. (2013). <http://archive.ics.uci.edu/ml>
- [17] Gilles Louppe. 2014. *Understanding Random Forests*. Ph.D. Dissertation. University of Liège, Faculty of Applied Sciences, Department of Electrical Engineering & Computer Science.
- [18] Gilles Louppe and Pierre Geurts. 2012. Ensembles on Random Patches. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*. Springer-Verlag, 346–361.
- [19] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. The MIT Press.
- [20] OpenStreetMap contributors. 2017. Planet dump retrieved from <https://planet.osm.org>. <https://www.openstreetmap.org>. (2017).
- [21] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. In *Proceedings of the 35th International Conference on Very Large Data Bases*. VLDB



(a) LC08_L1TP_196022_20150415_20170409_01_T1

Figure 9: Illustration of the first part of the landsat-osm dataset (used for the results shown in Figure 7). The gray pixels are unlabeled data points.

Endowment, 1426–1437.

- [22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard D Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [23] Toby Sharp. 2008. Implementing Decision Trees and Forests on a GPU. In *Computer Vision - ECCV 2008, 10th European Conference on Computer Vision (Lecture Notes in Computer Science)*, Vol. 5305. Springer, 595–608.
- [24] The HDF Group. 2000–2017. Hierarchical data format version 5. (2000–2017). <http://www.hdfgroup.org/HDF5>
- [25] Michael A. Wulder, Jeffrey G. Masek, Warren B. Cohen, Thomas R. Loveland, and Curtis E. Woodcock. 2012. Opening the archive: How free data has enabled the science and monitoring promise of Landsat. *Remote Sensing of Environment* 122, Supplement C (2012), 2 – 10. Landsat Legacy Special Issue.

A PARAMETERS

If not stated otherwise, the parameters are fixed to the following values (using the same notation as for sklearn): `max_features="sqrt"`, `bootstrap=True`, `criterion="gini"`, `n_jobs=4`, `max_depth=None`, `min_samples_leaf=1`, and `min_samples_split=2`. Note that woody uses C code for the construction of bottom trees, which follows the way random forests are built by sklearn. The parameters for h2o are adapted accordingly. In contrast to the other two implementations, the maximum tree depths is set to 20 for h2o (default value); larger tree depths led to memory errors.

For woody, the bottom trees are built in the same way as via the sklearn implementation (using the same parameters), except for the `bootstrap` parameter (bootstrap samples are extracted during the distribution phase). Both, the number of samples for the top trees as well as the desired leaf sizes for the bottom trees are defined via $\min(500000, n, \max(100\sqrt{n}, 100000))$. For the experiments, either a `MemoryStore` or a `DiskStore` is considered. The former one is used for the runtime comparison shown in Figure 6, where both the training data as well as the intermediate results are stored in memory (to obtain a fair comparison with sklearn). For the other large-scale experiments, `DiskStore` is used that loads the data from disk (in chunks) and also stores the intermediate results on disk. For the covtype dataset, the chunk size was fixed to $C = 100,000$, whereas for all other datasets, a chunk size of $C = 1,000,000$ was used. For subsets, we consider subsets of size 50,000 for covtype and of size 500,000 for all other datasets. We refer to <https://github.com/gieseke/woody>.

for the code and the experimental setup.

B LANDSAT-OSM

The features for the landsat-osm dataset are based on satellite data from the *Landsat 8* [25] project, see Figure 9. The associated labels stem from the *OpenStreetMap* (OSM) [20] project. More precisely, we consider 9 bands (grayscale images) and 3×3 image patches, resulting in 81 features. We extract such patches from the following Landsat scenes:⁷

- LC08_L1TP_193022_20170501_20170515_01_T1
- LC08_L1TP_194022_20160606_20170324_01_T1
- LC08_L1TP_195021_20160512_20170324_01_T1
- LC08_L1TP_195022_20160512_20170324_01_T1
- LC08_L1TP_196021_20150821_20170406_01_T1
- LC08_L1TP_196022_20150415_20170409_01_T1
- LC08_L1TP_197020_20150422_20170409_01_T1
- LC08_L1TP_197021_20150422_20170409_01_T1

The LC08_L1TP_196022_20150415_20170409_01_T1 scene is split into two parts, where 5% are used as test set (random subset) and the remaining 95% as training set. The other scenes are attached to the training set for the runtime evaluation provided in Figure 8, yielding one billion training instances. Prior to extracting the patches, we pansharpened all images [25].

Finally, the label for each such image patch is based on an OSM label extracted for the pixel at the center of the patch. The following OSM labels are extracted for all patches [20]: 1: `landuse:forest`, 2: `landuse:meadow`, 3: `waterway:riverbank`, 4: `highway:all`, 5: `building:all`, 6: `landuse:reservoir`, 7: `natural:grassland`, 8: `railway:light_rail`, and 9: `landuse:farmland`. The overall dataset consists of more than one billion labeled patches and provides a realistic classification benchmark scenario.

⁷Downloadable via <https://earthexplorer.usgs.gov/>.