Mondrian Forests: Efficient Online Random Forests

Balaji Lakshminarayanan*
Gatsby Unit
University College London

Daniel M. Roy
Department of Engineering
University of Cambridge

Yee Whye Teh
Department of Statistics
University of Oxford

Abstract

Ensembles of randomized decision trees, usually referred to as *random forests*, are widely used for classification and regression tasks in machine learning and statistics. Random forests achieve competitive predictive performance and are computationally efficient to train and test, making them excellent candidates for real-world prediction tasks. The most popular random forest variants (such as Breiman's random forest and extremely randomized trees) operate on batches of training data. Online methods are now in greater demand. Existing online random forests, however, require more training data than their batch counterpart to achieve comparable predictive performance. In this work, we use Mondrian processes (Roy and Teh, 2009) to construct ensembles of random decision trees we call *Mondrian forests*. Mondrian forests can be grown in an incremental/online fashion and remarkably, the distribution of online Mondrian forests is the same as that of batch Mondrian forests. Mondrian forests achieve competitive predictive performance comparable with existing online random forests and periodically re-trained batch random forests, while being more than an order of magnitude faster, thus representing a better computation vs accuracy tradeoff.

1 Introduction

Despite being introduced over a decade ago, random forests remain one of the most popular machine learning tools due in part to their accuracy, scalability, and robustness in real-world classification tasks [3]. (We refer to [5] for an excellent recent survey of random forests.) In this paper, we introduce a novel type of random forest—called *Mondrian forests* (MF), due to the fact that the underlying tree structure of each classifier in the ensemble is a so-called *Mondrian process*. Using the properties of Mondrian processes, we present an efficient *online* algorithm that agrees with its batch counterpart at each iteration. Not only are online Mondrian forests faster and more accurate than recent proposals for online random forest methods, but they nearly match the accuracy of state-of-the-art batch random forest methods trained on the same dataset.

The paper is organized as follows: In Section 2, we describe our approach at a high-level, and in Sections 3, 4, and 5, we describe the tree structures, label model, and incremental updates/predictions in more detail. We discuss related work in Section 6, demonstrate the excellent empirical performance of MF in Section 7, and conclude in Section 8 with a discussion about future work.

*Corresponding author. Email address: balaji@gatsby.ucl.ac.uk.

2 Approach

Given N labeled examples $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N) \in \mathbb{R}^D \times \mathcal{Y}$ as training data, our task is to predict labels $y \in \mathcal{Y}$ for unlabeled test points $\boldsymbol{x} \in \mathbb{R}^D$. We will focus on multi-class classification where $\mathcal{Y} := \{1, \ldots, K\}$, however, it is possible to extend the methodology to other supervised learning tasks such as regression. Let $\boldsymbol{X}_{1:n} := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, $Y_{1:n} := (y_1, \ldots, y_n)$, and $\mathcal{D}_{1:n} := (\boldsymbol{X}_{1:n}, Y_{1:n})$.

A Mondrian forest classifier is constructed much like a random forest: Given training data $\mathcal{D}_{1:N}$, we sample an independent collection $T_1, \ldots, T_M$ of so-called Mondrian trees, which we will describe in the next section. The prediction made by each Mondrian tree T_m is a distribution $p_{T_m}(y|\boldsymbol{x}, \mathcal{D}_{1:N})$ over the class label y for a test point $\boldsymbol{x}$. The prediction made by the Mondrian forest is the average

$$\frac{1}{M} \sum_{m=1}^{M} p_{T_m}(y|\boldsymbol{x}, \mathcal{D}_{1:N}) \tag{1}$$

of the individual tree predictions. As $M \to \infty$, the average converges at the standard rate to the expectation

$$\mathbb{E}_{T \sim \mathrm{MT}(\lambda, \mathcal{D}_{1:N})}[\, p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})] + \mathcal{O}(M^{-1/2}), \tag{2}$$

where $\mathrm{MT}\,(\lambda, \mathcal{D}_{1:N})$ is the distribution of a Mondrian tree. As the limiting expectation does not depend on M, we would not expect to see overfitting behavior as M increases. A similar observation was made by Breiman in his seminal article [2] introducing random forests. Note that Eq. (2) is ensemble model combination, *not* Bayesian model averaging.

In the online learning setting, the training examples are presented one after another in a sequence of trials. Mondrian forests excel in this setting: at iteration $n+1$, each Mondrian tree $T \sim \mathrm{MT}\,(\lambda, \mathcal{D}_{1:n})$ is updated to incorporate the next labeled example $(\boldsymbol{x}_{n+1}, y_{n+1})$ by sampling an extended tree T' from a distribution $\mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1})$. Using properties of the Mondrian process, we can choose a probability distribution MTx such that $T' = T$ on $\mathcal{D}_{1:n}$ and T' is distributed according to $\mathrm{MT}\,(\lambda, \mathcal{D}_{1:n+1})$, i.e.,

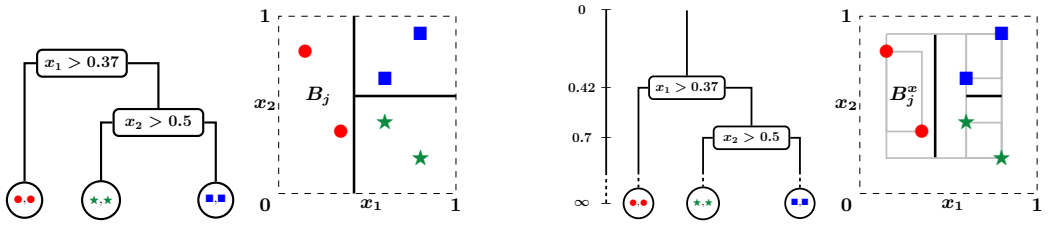
$$\begin{aligned} T &\sim \mathrm{MT}\,(\lambda, \mathcal{D}_{1:n}) \\ T' \mid T, \mathcal{D}_{1:n+1} &\sim \mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1}) \end{aligned} \qquad \textit{implies} \qquad T' \sim \mathrm{MT}\,(\lambda, \mathcal{D}_{1:n+1})\,. \tag{3}$$

Therefore, the distribution of Mondrian trees trained on a dataset in an incremental fashion is the same as that of Mondrian trees trained on the same dataset in a batch fashion, irrespective of the order in which the data points are observed. To the best of our knowledge, none of the existing online random forests have this property. Moreover, we can sample from $\mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1})$ efficiently: the complexity scales with the depth of the tree, which is typically logarithmic in n.

While treating the online setting as a sequence of larger and larger batch problems is normally computationally prohibitive, this approach can be achieved efficiently with Mondrian forests. In the following sections, we define the Mondrian tree distribution $\mathrm{MT}\,(\lambda, \mathcal{D}_{1:N})$, the label distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$, and the update distribution $\mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1})$.

3 Mondrian trees

For our purposes, a **decision tree** on $\mathbb{R}^D$ will be a hierarchical, binary partitioning of $\mathbb{R}^D$ and a rule for predicting the label of test points given training data. More carefully, a **rooted, strictly-binary tree** is a finite tree T such that every node in T is either a leaf or internal



(a) Decision Tree (b) Mondrian Tree

Figure 1: Example of a decision tree in $[0,1]^2$ where x_1 and x_2 denote horizontal and vertical axis respectively: Figure 1(a) shows tree structure and partition of a decision tree, while Figure 1(b) shows a Mondrian tree. Note that the Mondrian tree is embedded on a vertical time axis, with each node associated with a time of split and the splits are committed only within the range of the training data in each block (denoted by gray rectangles). Let j denote the left child of the root: $B_j = (0, 0.37] \times (0, 1]$ denotes the block associated with red circles and $B_j^x \subseteq B_j$ is the smallest rectangle enclosing the two data points.

node, and every node is the child of exactly one parent node, except for a distinguished root node, represented by ϵ, which has no parent. Let $\mathsf{leaves}(\mathsf{T})$ denote the set of leaf nodes in T. For every internal node $j \in \mathsf{T} \setminus \mathsf{leaves}(\mathsf{T})$, there are exactly two children nodes, represented by $\mathsf{left}(j)$ and $\mathsf{right}(j)$. To each node $j \in \mathsf{T}$, we associate a block $B_j \in \mathbb{R}^D$ of the input space as follows: We let $B_\epsilon := \mathbb{R}^D$. Each internal node $j \in \mathsf{T} \setminus \mathsf{leaves}(\mathsf{T})$ is associated with a **split** (δ_j, ξ_j), where $\delta_j \in \{1, 2, \ldots, D\}$ denotes the dimension of the split and ξ_j denotes the location of the split along dimension δ_j. We then define

$$B_{\mathsf{left}(j)} := \{\boldsymbol{x} \in B_j : x_{\delta_j} \leq \xi_j\} \quad \text{and} \quad B_{\mathsf{right}(j)} := \{\boldsymbol{x} \in B_j : x_{\delta_j} > \xi_j\}. \tag{4}$$

We may write $B_j = \left(\ell_{j1}, u_{j1}\right] \times \ldots \times \left(\ell_{jD}, u_{jD}\right]$, where ℓ_{jd} and u_{jd} denote the ℓower and $u$pper limit, respectively, of the rectangular block B_j along dimension d. Put $\boldsymbol{\ell}_j = \{\ell_{j1}, \ell_{j2}, \ldots, \ell_{jD}\}$ and $\mathbf{u}_j = \{u_{j1}, u_{j2}, \ldots, u_{jD}\}$. The decision tree structure is represented by the tuple $T = (\mathsf{T}, \boldsymbol{\delta}, \boldsymbol{\xi})$. We refer to Figure 1(a) for a simple illustration of a decision tree.

Let $\mathsf{parent}(j)$ denote the parent of node j. Let $N(j)$ denote the indices of training data points at node j, i.e., $N(j) = \{n \in \{1, \ldots, N\} : \boldsymbol{x}_n \in B_j\}$. Let $\mathcal{D}_{N(j)} = \{\boldsymbol{X}_{N(j)}, Y_{N(j)}\}$ denote the features and labels of training data points at node j. Let ℓ_{jd}^x and u_{jd}^x denote the lower and upper limit of training data points (hence the superscript x) respectively in node j along dimension d. Let $B_j^x = \left(\ell_{j1}^x, u_{j1}^x\right] \times \ldots \times \left(\ell_{jD}^x, u_{jD}^x\right] \subseteq B_j$ denote the smallest rectangle that encloses the training data points in node j.

3.1 Mondrian process distribution over decision trees

Mondrian processes, introduced by Roy and Teh [14], are families $\{\mathcal{M}_t : t \in [0, \infty)\}$ of random, hierarchical binary partitions of $\mathbb{R}^D$ such that $\mathcal{M}_t$ is a refinement of $\mathcal{M}_s$ whenever $t > s$.[1] Mondrian processes are natural candidates for the partition structure of random decision trees, but Mondrian processes on $\mathbb{R}^D$ are, in general, infinite structures that we cannot represent all at once. Because we only care about the partition on a finite set of observed data, we introduce **Mondrian trees**, which are restrictions of Mondrian processes to a finite set of points. A Mondrian tree T can be represented by a tuple $(\mathsf{T}, \boldsymbol{\delta}, \boldsymbol{\xi}, \boldsymbol{\tau})$, where $(\mathsf{T}, \boldsymbol{\delta}, \boldsymbol{\xi})$ is a decision tree, $\boldsymbol{\tau} = \{\tau_j\}_{j \in \mathsf{T}}$, and $\tau_j \geq 0$ denotes the time of the split associated with node j. The time of split increases with depth, i.e., $\tau_j > \tau_{\mathsf{parent}(j)}$. We abuse notation and define $\tau_{\mathsf{parent}(\epsilon)} = 0$.

[1] Roy and Teh [14] studied the distribution of $\{\mathcal{M}_t : t \leq \lambda\}$ and refered to λ as the *budget*. See [13, Chp. 5] for more details. We will refer to t as time, not be confused with discrete time in the online learning setting.

Given a non-negative *lifetime* parameter λ and training data $\mathcal{D}_{1:n}$, the generative process for sampling Mondrian trees from $\mathrm{MT}(\lambda, \mathcal{D}_{1:n})$ is described in the following two algorithms:

Algorithm 1 $\mathsf{SampleMondrianTree}\big(\lambda, \mathcal{D}_{1:n}\big)$

Initialize: $\mathsf{T} = \emptyset$, $\mathsf{leaves}(\mathsf{T}) = \emptyset$, $\boldsymbol{\delta} = \emptyset$, $\boldsymbol{\xi} = \emptyset$, $\boldsymbol{\tau} = \emptyset$, $N(\epsilon) = \{1, 2, \ldots, n\}$
$\mathsf{SampleMondrianBlock}\big(\epsilon, \mathcal{D}_{N(\epsilon)}, \lambda\big)$ ▷ *Algorithm 2*

Algorithm 2 $\mathsf{SampleMondrianBlock}\big(j, \mathcal{D}_{N(j)}, \lambda\big)$

Add j to T
For all d, set $\ell^x_{jd} = \mathsf{min}(\boldsymbol{X}_{N(j),d}), u^x_{jd} = \mathsf{max}(\boldsymbol{X}_{N(j),d})$ ▷ *dimension-wise* min *and* max
Sample E from exponential distribution with rate $\sum_d (u^x_{jd} - \ell^x_{jd})$
if $\tau_{\mathsf{parent}(j)} + E < \lambda$ **then** ▷ *j is an internal node*
    Set $\tau_j = \tau_{\mathsf{parent}(j)} + E$
    Sample split dimension δ_j, choosing d with probability proportional to $u^x_{jd} - \ell^x_{jd}$
    Sample split location ξ_j uniformly from interval $[\ell^x_{j\delta_j}, u^x_{j\delta_j}]$
    Set $N(\mathsf{left}(j)) = \{n \in N(j) : \boldsymbol{X}_{n,\delta_j} \leq \xi_j\}$ and $N(\mathsf{right}(j)) = \{n \in N(j) : \boldsymbol{X}_{n,\delta_j} > \xi_j\}$
    $\mathsf{SampleMondrianBlock}\big(\mathsf{left}(j), \mathcal{D}_{N(\mathsf{left}(j))}, \lambda\big)$
    $\mathsf{SampleMondrianBlock}\big(\mathsf{right}(j), \mathcal{D}_{N(\mathsf{right}(j))}, \lambda\big)$
else ▷ *j is a leaf node*
    Set $\tau_j = \lambda$ and add j to $\mathsf{leaves}(\mathsf{T})$

The procedure starts with the root node ϵ and recurses down the tree. In Algorithm 2, we first compute the $\boldsymbol{\ell}^x_\epsilon$ and $\mathbf{u}^x_\epsilon$ i.e. the lower and upper limits of B^x_ϵ, the smallest rectangle enclosing $\boldsymbol{X}_{N(\epsilon)}$. We sample E from an exponential distribution whose rate is the so-called linear dimension of B^x_ϵ, given by $\sum_d (u^x_{\epsilon d} - \ell^x_{\epsilon d})$. Since $\tau_{\mathsf{parent}(\epsilon)} = 0$, $E + \tau_{\mathsf{parent}(\epsilon)} = E$. If $E \geq \lambda$, the time of split is not within the lifetime λ; hence, we assign ϵ to be a leaf node and the procedure halts. (Since $\mathbb{E}[E] = 1/\big(\sum_d (u^x_{jd} - \ell^x_{jd})\big)$, bigger rectangles are less likely to be leaf nodes.) Else, ϵ is an internal node and we sample a split $(\delta_\epsilon, \xi_\epsilon)$ in B^x_ϵ from the *uniform split distribution* in B^x_ϵ. More precisely, we first sample the dimension δ_ϵ, taking the value d with probability proportional to $u^x_{\epsilon d} - \ell^x_{\epsilon d}$, and then sample the split location ξ_ϵ uniformly from the interval $[\ell^x_{\epsilon\delta_\epsilon}, u^x_{\epsilon\delta_\epsilon}]$. The procedure then recurses along the left and right children.

Mondrian trees differ from standard decision trees (e.g. CART, C4.5) in the following: (i) the splits are sampled independent of the labels $Y_{N(j)}$; (ii) every node j is associated with a split time τ_j; (iii) the lifetime parameter λ controls the total number of splits (similar to the maximum depth parameter for standard decision trees); (iv) the split represented by an internal node j holds only within B^x_j and not the whole of B_j. No commitment is made in $B_j \setminus B^x_j$. Figure 1 illustrates the difference between Mondrian trees and decision trees.

Consider the family of distributions $\mathrm{MT}(\lambda, F)$, where F ranges over all possible finite sets of data points. Due to the fact that these distributions are derived from that of a Mondrian process on $\mathbb{R}^D$ restricted to a set F of points, the family $\mathrm{MT}(\lambda, \cdot)$ will be *projective*. Intuitively, projectivity implies that the tree distributions possess a type of self-consistency in distribution. In words, if we sample a Mondrian tree T from $\mathrm{MT}(\lambda, F)$ and then restrict the tree T to a subset $F' \subseteq F$ of points, then the restricted tree T' has distribution $\mathrm{MT}(\lambda, F')$. This property follows from a similar property of Mondrian processes [13, 14]. Most importantly, projectivity gives us a consistent way to extend a Mondrian tree on a data set $\mathcal{D}_{1:n}$ to a larger data set $\mathcal{D}_{1:n+1}$. We exploit this property to incrementally grow a Mondrian tree: even though $\mathrm{MT}(\lambda, \mathcal{D}_{1:n})$ is defined on $\mathbb{R}^D$, we instantiate the Mondrian tree just on the regions where we have observed training data points so far; upon observing $\mathcal{D}_{n+1}$, we *extend* the Mondrian by sampling from the conditional Mondrian distribution, referred to as $\mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1})$ in (3), unveiling the Mondrian tree only where we have observed training data.

4 Label distribution: model, hierarchical prior, and predictive posterior

So far, our discussion has been focused only on the tree structure. In this section, we focus on the label distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$. Intuitively, we want the label distribution at a node to be a smoothed estimate of the empirical distribution of labels at a node. We achieve this smoothing via a hierarchical Bayesian approach within each tree. For each tree T, we introduce latent parameters $\mathcal{G}$ which specify a distribution over y at each node, denoted by $p_T(y|\boldsymbol{x}, \mathcal{G})$. Next, we define a hierarchical prior $p_T(\mathcal{G})$ that encourages label distribution at a node to be similar to that of its parent. Finally, we discuss how the likelihood and the hierarchical prior are combined to obtain the label distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$.

Let $\mathsf{leaf}(\boldsymbol{x})$ denote the unique leaf node in T such that $\boldsymbol{x} \in B_{\mathsf{leaf}(\boldsymbol{x})}$. As is common in the decision tree literature, we assume that the probability of labels within each block is independent of $\boldsymbol{X}$ given the tree structure T. Let G_j denote the distribution of labels at node j and $\mathcal{G} = \{G_j : j \in \mathsf{T}\}$ denote the set of label distributions at all the nodes in the tree. Given a tree T, the likelihood for $\boldsymbol{x}$ is defined by the label distribution at the node $\mathsf{leaf}(\boldsymbol{x})$, i.e., $p_T(y|\boldsymbol{x}, \mathcal{G}) = G_{\mathsf{leaf}(\boldsymbol{x})}$. In this paper, we focus on the case of categorical labels taking values in the set $\{1, \ldots, K\}$. Hence, $G_j = [G_{j,1}, G_{j,2}, \ldots, G_{j,K}]$ is the *discrete* distribution, where $G_{j,k}$ is the probability of label k at node j.

We model the collection G_j, for $j \in T$, as a hierarchy of normalized stable processes (NSP). A NSP prior is a distribution over distributions and is a special case of the Pitman-Yor process (PYP) prior where the concentration parameter is taken to zero.[2] The discount parameter $d \in (0, 1)$ controls how much the samples vary around the base distribution; if $G_j \sim \mathrm{NSP}(d, H)$, then $\mathbb{E}[G_{jk}] = H_k$ and $\mathrm{Var}[G_{jk}] = (1 - d)H_k(1 - H_k)$. We use a hierarchical NSP (HNSP) prior over G_j as follows:

$$G_\epsilon | H \sim \mathrm{NSP}(d_\epsilon, H), \qquad \text{and} \qquad G_j | G_{\mathsf{parent}(j)} \sim \mathrm{NSP}(d_j, G_{\mathsf{parent}(j)}). \tag{5}$$

This hierarchical prior was first proposed by Wood et al. [19]. Here we set $d_j = \exp\{-\gamma(\tau_j - \tau_{\mathsf{parent}(j)})\}$, and the base distribution H to be the uniform distribution over the K labels.

Given training data $\mathcal{D}_{1:N}$, the predictive distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$ is obtained by integrating over $\mathcal{G}$, i.e., $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N}) = \mathbb{E}_{\mathcal{G} \sim p_T(\mathcal{G}|\mathcal{D}_{1:N})}[p_T(y|\boldsymbol{x}, \mathcal{G})] = \mathbb{E}_{\mathcal{G} \sim p_T(\mathcal{G}|\mathcal{D}_{1:N})}[G_{\mathsf{leaf}(\boldsymbol{x}),y}] = \overline{G}_{\mathsf{leaf}(\boldsymbol{x}),y}$, where the posterior $p_T(\mathcal{G}|\mathcal{D}_{1:N}) \propto p_T(\mathcal{G}) \prod_{n=1}^{N} p_T(y_n|\boldsymbol{x}_n, \mathcal{G})$. Posterior inference in the HNSP, i.e., computation of the posterior means $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$, is a special case of posterior inference in hierarchical PYP (HPYP). In particular, Teh [17] considers the HPYP with multinomial likelihood (in the context of language modeling). The model considered here is a special case of [17]. Exact inference is intractable and hence we resort to approximations. In particular, we use a fast approximation known as the interpolated Kneser-Ney (IKN) smoothing [17], a popular technique for smoothing probabilities in language modeling [10]. The IKN approximation in [17] can be extended in a straightforward fashion to the online setting, and the computational complexity of adding a new training instance is linear in the depth of the tree. We refer the reader to Appendix A for further details.

5 Online training and prediction

In this section, we describe the distribution $\mathrm{MTx}(\lambda, T, \mathcal{D}_{n+1})$ that incrementally adds the data point $\mathcal{D}_{n+1}$ to a tree T. These updates are based on the conditional Mondrian algorithm [14], specialized to a finite set of points. In general, one or more of the following three operations

[2] Taking the discount parameter to zero leads to a Dirichlet process. Hierarchies of NSPs admit more tractable approximations than hierarchies of Dirichlet processes, hence our choice here.

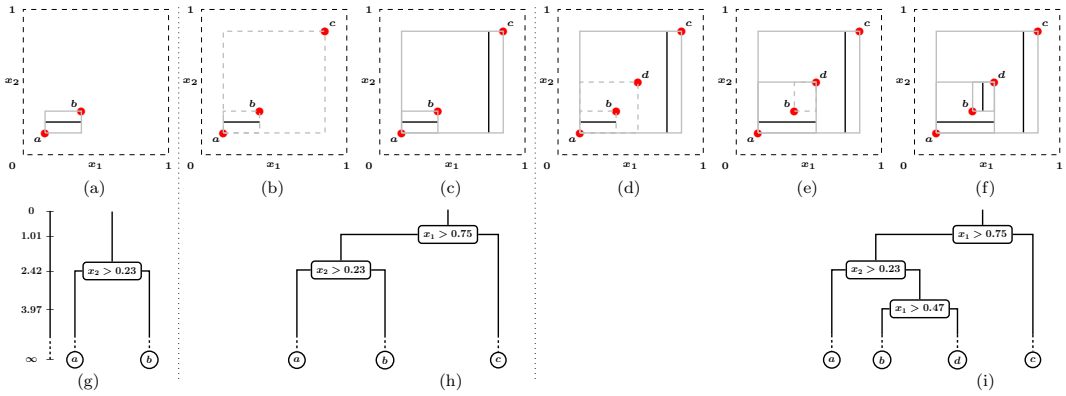


Figure 2: Online learning with Mondrian trees on a toy dataset: We assume that $\lambda = \infty, D = 2$ and add one data point at each iteration. For simplicity, we ignore class labels and denote location of training data with red circles. Figures 2(a), 2(c) and 2(f) show the partitions after the first, second and third iterations, respectively, with the intermediate figures denoting intermediate steps. Figures 2(g), 2(h) and 2(i) show the trees after the first, second and third iterations, along with a shared vertical time axis.

At iteration 1, we have two training data points, labeled as a, b. Figures 2(a) and 2(g) show the partition and tree structure of the Mondrian tree. Note that even though there is a split $x_2 > 0.23$ at time $t = 2.42$, we commit this split only within B_j^x (shown by the gray rectangle).

At iteration 2, a new data point c is added. Algorithm 3 starts with the root node and recurses down the tree. Algorithm 4 checks if the new data point lies within B_ϵ^x by computing the additional extent $\mathbf{e}^\ell$ and $\mathbf{e}^u$. In this case, c does not lie within B_ϵ^x. Let R_{ab} and R_{abc} respectively denote the small gray rectangle (enclosing a, b) and big gray rectangble (enclosing a, b, c) in Figure 2(b). While extending the Mondrian from R_{ab} to R_{abc}, we could either introduce a new split in R_{abc} outside R_{ab} or extend the split in R_{ab} to the new range. To choose between these two options, we sample the time of this new split: we first sample E from an exponential distribution whose rate is the sum of the additional extent, i.e., $\sum_d (e_d^\ell + e_d^u)$, and set the time of the new split to $E + \tau_{\mathsf{parent}(\epsilon)}$. If $E + \tau_{\mathsf{parent}(\epsilon)} \leq \tau_\epsilon$, this new split in R_{abc} can precede the old split in R_{ab} and a split is sampled in R_{abc} outside R_{ab}. In Figures 2(c) and 2(h), $E + \tau_{\mathsf{parent}(\epsilon)} = 1.01 + 0 \leq 2.42$, hence a new split $x_1 > 0.75$ is introduced. The farther a new data point $\boldsymbol{x}$ is from B_j^x, the higher the rate $\sum_d (e_d^\ell + e_d^u)$, and subsequently the higher the probability of a new split being introduced, since $\mathbb{E}[E] = 1/\big(\sum_d (e_d^\ell + e_d^u)\big)$. A new split in R_{abc} is sampled such that it is consistent with the existing partition structure in R_{ab} (i.e., the new split cannot slice through R_{ab}).

In the final iteration, we add data point d. In Figure 2(d), the data point d lies within the extent of the root node, hence we traverse to the left side of the root and update B_j^x of the internal node containing $\{a, b\}$ to include d. We could either introduce a new split or extend the split $x_2 > 0.23$. In Figure 2(e), we extend the split $x_2 > 0.23$ to the new extent, and traverse to the leaf node in Figure 2(h) containing b. In Figures 2(f) and 2(i), we sample $E = 1.55$ and since $\tau_{\mathsf{parent}(j)} + E = 2.42 + 1.55 = 3.97 \leq \lambda = \infty$, we introduce a new split $x_1 > 0.47$.

may be executed while introducing a new data point: (i) introduction of a new split 'above' an existing split, (ii) extension of an existing split to the updated extent of the block and (iii) splitting an existing leaf node into two children. To the best of our knowledge, existing online decision trees use just the third operation, and the first two operations are unique to Mondrian trees. The complete pseudo-code for incrementally updating a Mondrian tree T with new data $\mathcal{D}$ according to $\text{MTx}(\lambda, T, \mathcal{D})$ is described in the following two algorithms. Figure 2 walks through the algorithms on a toy dataset.

Algorithm 3 $\mathsf{ExtendMondrianTree}(T, \lambda, \mathcal{D})$

Input: Tree $T = (\mathsf{T}, \boldsymbol{\delta}, \boldsymbol{\xi}, \boldsymbol{\tau})$, new training instance $\mathcal{D} = (\boldsymbol{x}, y)$
$\mathsf{ExtendMondrianBlock}(T, \lambda, \epsilon, \mathcal{D})$ ▷ *Algorithm 4*

Algorithm 4 $\mathsf{ExtendMondrianBlock}(T, \lambda, j, \mathcal{D})$

Set $\mathbf{e}^\ell = \mathsf{max}(\boldsymbol{\ell}_j^x - \boldsymbol{x}, 0)$ and $\mathbf{e}^u = \mathsf{max}(\boldsymbol{x} - \mathbf{u}_j^x, 0)$ ▷ $\mathbf{e}^\ell = \mathbf{e}^u = \mathbf{0}_D$ *if* $\boldsymbol{x} \in B_j^x$
Sample E from exponential distribution with rate $\sum_d (e_d^\ell + e_d^u)$
if $\tau_{\mathsf{parent}(j)} + E < \tau_j$ **then** ▷ *introduce new parent for node* j
    Sample split dimension δ, choosing d with probability proportional to $e_d^\ell + e_d^u$
    Sample split location ξ uniformly from interval $[u_{j,\delta}^x, x_\delta]$ **if** $x_\delta > u_{j,\delta}^x$ **else** $[x_\delta, \ell_{j,\delta}^x]$.
    Insert a new node $\tilde{j}$ just above node j in the tree, and a new leaf j'', sibling to j, where
        $\delta_{\tilde{j}} = \delta$, $\xi_{\tilde{j}} = \xi$, $\tau_{\tilde{j}} = \tau_{\mathsf{parent}(j)} + E$, $\boldsymbol{\ell}_{\tilde{j}}^x = \mathsf{min}(\boldsymbol{\ell}_j^x, \boldsymbol{x})$, $\mathbf{u}_{\tilde{j}}^x = \mathsf{max}(\mathbf{u}_j^x, \boldsymbol{x})$
        $j'' = \mathsf{left}(\tilde{j})$ **iff** $x_{\delta_{\tilde{j}}} \leq \xi_{\tilde{j}}$
    $\mathsf{SampleMondrianBlock}(j'', \mathcal{D}, \lambda)$
else
    Update $\boldsymbol{\ell}_j^x \leftarrow \mathsf{min}(\boldsymbol{\ell}_j^x, \boldsymbol{x}), \mathbf{u}_j^x \leftarrow \mathsf{max}(\mathbf{u}_j^x, \boldsymbol{x})$ ▷ *update extent of node* j
    if $j \notin \mathsf{leaves}(\mathsf{T})$ **then** ▷ *return if* j *is a leaf node, else recurse down the tree*
        if $x_{\delta_j} \leq \xi_j$ **then** $\mathsf{child}(j) = \mathsf{left}(j)$ **else** $\mathsf{child}(j) = \mathsf{right}(j)$
        $\mathsf{ExtendMondrianBlock}(T, \lambda, \mathsf{child}(j), \mathcal{D})$ ▷ *recurse on child containing* $\mathcal{D}$

In practice, random forest implementations stop splitting a node when all the labels are identical and assign it to be a leaf node. To make our MF implementation comparable, we '*pause*' a Mondrian block when all the labels are identical; if a new training instance lies within B_j of a paused leaf node j and has the same label as the rest of the data points in B_j, we continue pausing the Mondrian block. We '*un-pause*' the Mondrian block when there is more than one unique label in a block. Algorithms 9 and 10 in the appendix discuss versions of $\mathsf{SampleMondrianBlock}$ and $\mathsf{ExtendMondrianBlock}$ for paused Mondrians.

Prediction using Mondrian tree Let $\boldsymbol{x}$ denote a test data point. If $\boldsymbol{x}$ is already "contained" in the tree T, i.e., if $\boldsymbol{x} \in B_j^x$ for some leaf $j \in \mathsf{leaves}(T)$, then the prediction is taken to be $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$. Otherwise, we somehow need to incorporate $\boldsymbol{x}$. One choice is to extend T by sampling T' from $\text{MTx}(\lambda, T, \boldsymbol{x})$ as described in Algorithm 3, and set the prediction to $\overline{G}_j$, where $j \in \mathsf{leaves}(T')$ is the leaf node containing $\boldsymbol{x}$. A particular extension T' might lead to an overly confident prediction; hence, we average over *every* possible extension T'. This integration can be carried out analytically and the computational complexity is linear in the depth of the tree. We refer the reader to Appendix B for further details.

6 Related work

The literature on random forests is vast and we do not attempt to cover it comprehensively; we provide a brief review here and refer to [5] and [7] for a recent review of random forests in

batch and online settings respectively. Classic decision tree induction procedures choose the best split dimension and location from all candidate splits at each node by optimizing some suitable quality criterion (e.g. information gain) in a greedy manner. In a random forest, the individual trees are randomized to de-correlate their predictions. The most common strategies for injecting randomness are (i) bagging [1] and (ii) randomly subsampling the set of candidate splits within each node.

Two popular random forest variants in the batch setting are *Breiman-RF* [2] and *Extremely randomized trees (ERT)* [9]. Breiman-RF uses bagging and furthermore, at each node, a random k-dimensional subset of the original D features is sampled. ERT chooses a k dimensional subset of the features and then chooses one split location each for the k features randomly (unlike Breiman-RF which considers all possible split locations along a dimension). ERT does not use bagging. When $k = 1$, the ERT trees are *totally randomized* and the splits are chosen independent of the labels; hence the ERT-1 method is very similar to MF in the batch setting in terms of tree induction. (Note that unlike ERT, MF uses HNSP to smooth predictive estimates and allows a test point to branch off into its own node.) Perfect random trees (PERT), proposed by Cutler and Zhao [6] for classification problems, produce totally randomized trees similar to ERT-1, although there are some slight differences [9].

Existing online random forests [7, 15] start with an empty tree and grow the tree incrementally. Every leaf of every tree maintains a list of k candidate splits and associated quality scores. When a new data point is added, the scores of the candidate splits at the corresponding leaf node are updated. To reduce the risk of choosing a sub-optimal split based on noisy quality scores, additional hyper parameters such as the minimum number of data points at a leaf node before a decision is made and the minimum threshold for the quality criterion of the best split, are used to assess 'confidence' associated with a split. Once these criteria are satisfied at a leaf node, the best split is chosen (making this node an internal node) and its two children are the new leaf nodes (with their own candidate splits), and the process is repeated. These methods could be memory inefficient for deep trees due to the high cost associated with maintaining candidate quality scores for the fringe of potential children [7].

There has been some work on incremental induction of decision trees, e.g. incremental CART [4], ITI [18], VFDT [8] and dynamic trees [16], but to the best of our knowledge, these are focused on learning decision trees and have not been generalized to online random forests. We do not compare MF to incremental decision trees, since random forests are known to outperform single decision trees.

7 Empirical evaluation

The purpose of these experiments is to evaluate the predictive performance (test accuracy) of MF as a function of (i) fraction of training data and (ii) training time. We divide the training data into 100 mini-batches and we compare the performance of online random forests (MF, ORF-Saffari) to batch random forests (Breiman-RF, ERT-k, ERT-1) which are trained on the same fraction of the training data. Our scripts are implemented in Python. We implemented the ORF-Saffari algorithm as well as ERT in Python for timing comparisons. The scripts will be made publicly available. We did not implement the ORF-Denil algorithm since its performance is very similar to ORF-Saffari [7] and the computational complexity of the ORF-Denil algorithm is worse than that of ORF-Saffari. We used the Breiman-RF implementation in *scikit-learn* [12].[3]

[3]The *scikit-learn* implementation uses highly optimized C code, hence we do not compare our runtimes with the *scikit-learn* implementation. The ERT implementation in *scikit-learn* achieves very similar test accuracy as our ERT implementation, hence we do not report those results here.

We evaluate on four of the five datasets used in [15] — we excluded the *mushroom* dataset as even very simple logical rules achieve $> 99\%$ accuracy on this dataset.[4] We re-scaled the datasets such that each feature takes on values in the range $[0, 1]$ (by subtracting the min value along that dimension and dividing by the range along that dimension, where $\mathsf{range} = \mathsf{max} - \mathsf{min}$).

As common in the random forest literature, we set the number of trees $M = 100$. For Mondrian forests, we set $\lambda = \infty$, $\gamma = 10D$. For ORF-Saffari, we set $\mathsf{num\_epochs} = 20$ (number of passes through the training data) and set the other hyper parameters to the values used in [15]. For Breiman-RF and ERT, the hyper parameters are set to default values. We repeat each algorithm with five random initializations and report the mean performance. The results are shown in Figure 3. (The * in Breiman-RF* indicates *scikit-learn* implementation.)

Comparing test accuracy vs fraction of training data on *usps*, *satimages* and *letter* datasets, we observe that **MF achieves accuracy very close to the batch RF versions** (Breiman-RF, ERT-k, ERT-1) trained on the same fraction of the data. **MF significantly outperforms ORF-Saffari trained on the same fraction of training data**. In batch RF versions, the same training data can be used to evaluate candidate splits at a node and its children. However, in the online RF versions (ORF-Saffari and ORF-Denil), incoming training examples are used to evaluate candidate splits just at a current leaf node and new training data are required to evaluate candidate splits every time a new leaf node is created. Saffari et al. [15] recommend multiple passes through the training data to increase the effective number of training samples. In a realistic streaming data setup, where training examples cannot be stored for multiple passes, MF would require significantly fewer examples than ORF-Saffari to achieve the same accuracy.

Comparing test accuracy vs training time on *usps*, *satimages* and *letter* datasets, we observe that **MF is at least an order of magnitude faster than re-trained batch versions and ORF-Saffari**. For ORF-Saffari, we plot test accuracy at the end of every additional pass; hence it contains additional markers compared to the top row which plots results after a single pass. Re-training batch RF using 100 mini-batches is unfair to MF; in a streaming data setup where the model is updated when a new training instance arrives, MF would be significantly faster than the re-trained batch versions. Assuming trees are balanced after adding each data point, it can be shown that computational complexity of MF scales as $\mathcal{O}(N \log N)$ whereas the re-trained batch RF scales as $\mathcal{O}(N^2 \log N)$ (Appendix C).

It is remarkable that choosing splits independent of labels achieves competitive classification performance. This phenomenon has been observed by others as well—for example, Cutler and Zhao [6] demonstrate that their PERT classifier (which is similar to batch version of MF) achieves test accuracy comparable to Breiman-RF on many real world datasets. However, in the presence of irrelevant features, methods which choose splits independent of labels (MF, ERT-1) perform worse than Breiman-RF and ERT-k (but still better than ORF-Saffari) as indicated by the results on the *dna* dataset. We trained MF and ERT-1 using just the most relevant 60 attributes amongst the 180 attributes[5]—these results are indicated as MF† and ERT-1† in Figure 3. We observe that, as expected, filtering out irrelevant features significantly improves performance of MF and ERT-1.

8 Discussion

We have introduced *Mondrian forests*, a new random forest variant which can be trained incrementally in an efficient manner. MF significantly outperforms existing online random forests in terms of training time as well as number of training instances required to achieve a

[4] https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.names
[5] https://www.sgi.com/tech/mlc/db/DNA.names

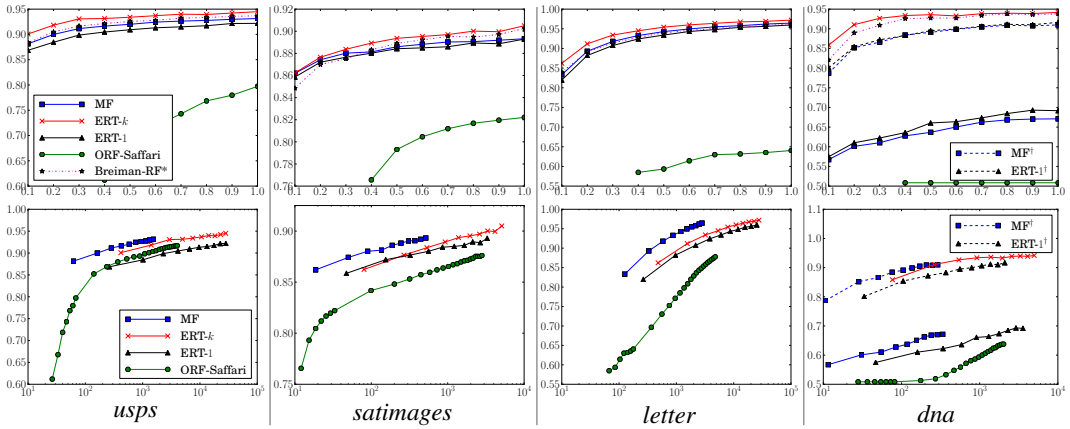


Figure 3: Results on various datasets: y-axis is test accuracy in both rows. x-axis is fraction of training data for the top row and training time (in seconds) for the bottom row. We used the pre-defined train/test split. For *usps* dataset $D = 256, K = 10, N_{\text{train}} = 7291, N_{\text{test}} = 2007$; for *satimages* dataset $D = 36, K = 6, N_{\text{train}} = 3104, N_{\text{test}} = 2000$; *letter* dataset $D = 16, K = 26, N_{\text{train}} = 15000, N_{\text{test}} = 5000$; for *dna* dataset $D = 180, K = 3, N_{\text{train}} = 1400, N_{\text{test}} = 1186$.

particular test accuracy. Remarkably, MF achieves competitive test accuracy to batch random forests trained on the same fraction of the data. MF is unable to handle lots of irrelevant features (since splits are chosen independent of the labels)—one way to use labels to guide splits is via recently proposed Sequential Monte Carlo algorithm for decision trees [11]. The computational complexity of MF is linear in the number of dimensions (since rectangles are represented explicitly) which could be expensive for high dimensional data; we will address this limitation in future work. Random forests have been tremendously influential in machine learning for a variety of tasks; hence lots of other interesting extensions of this work are possible, e.g. MF for regression, theoretical bias-variance analysis of MF, extensions of MF that use hyperplane splits instead of axis-aligned splits.

Acknowledgments

We would like to thank Charles Blundell, Gintare Dziugaite, Creighton Heaukulani, José Miguel Hernández-Lobato, Maria Lomeli, Alex Smola, Heiko Strathmann, and Srini Turaga for helpful discussions and feedback on drafts. BL gratefully acknowledges generous funding from the Gatsby Charitable Foundation. This research was carried out in part while DMR held a Research Fellowship at Emmanuel College, Cambridge, with funding also from a Newton International Fellowship through the Royal Society. YWT's research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) ERC grant agreement no. 617411.

References

[1] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996.

[2] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[3] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2006.

[4] S. L. Crawford. Extensions to the CART algorithm. *Int. J. Man-Machine Stud.*, 31(2): 197–217, 1989.

[5] A. Criminisi, J. Shotton, and E. Konukoglu. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Found. Trends Comput. Graphics and Vision*, 7(2–3):81–227, 2012.

[6] A. Cutler and G. Zhao. PERT - Perfect Random Tree Ensembles. *Comput. Sci. and Stat.*, 33:490–497, 2001.

[7] M. Denil, D. Matheson, and N. de Freitas. Consistency of online random forests. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2013.

[8] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. 6th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. (KDD)*, pages 71–80. ACM, 2000.

[9] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1): 3–42, 2006.

[10] J. T. Goodman. A bit of progress in language modeling. *Comput. Speech Lang.*, 15(4): 403–434, 2001.

[11] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh. Top-down particle filtering for Bayesian decision trees. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2013.

[12] 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. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[13] D. M. Roy. *Computability, inference and modeling in probabilistic programming.* PhD thesis, Massachusetts Institute of Technology, 2011. http://danroy.org/papers/Roy-PHD-2011.pdf.

[14] D. M. Roy and Y. W. Teh. The Mondrian process. In *Adv. Neural Inform. Proc. Syst. (NIPS)*, volume 21, pages 27–36, 2009.

[15] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *Computer Vision Workshops (ICCV Workshops)*. IEEE, 2009.

[16] M. A. Taddy, R. B. Gramacy, and N. G. Polson. Dynamic trees for learning and design. *J. Am. Stat. Assoc.*, 106(493):109–123, 2011.

[17] Y. W. Teh. A hierarchical Bayesian language model based on Pitman–Yor processes. In *Proc. 21st Int. Conf. on Comp. Ling. and 44th Ann. Meeting Assoc. Comp. Ling.*, pages 985–992. Assoc. for Comp. Ling., 2006.

[18] P. E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4(2):161–186, 1989.

[19] F. Wood, C. Archambeau, J. Gasthaus, L. James, and Y. W. Teh. A stochastic memoizer for sequence data. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2009.

A Posterior inference and prediction using the HNSP

Recall that we use a hierarchical Bayesian approach to specify a smooth label distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$ for each tree T. The label prediction at a test point $\boldsymbol{x}$ will depend on where $\boldsymbol{x}$ falls relative to the existing data in the tree T. In this section, we assume that $\boldsymbol{x}$ lies within one of the leaf nodes in T , i.e., $\boldsymbol{x} \in B^x_{\mathsf{leaf}(\boldsymbol{x})}$, where $\mathsf{leaf}(\boldsymbol{x}) \in \mathsf{leaves}(T)$. If $\boldsymbol{x}$ does not lie within any of the leaf nodes in T, i.e., $\boldsymbol{x} \notin \cup_{j \in \mathsf{leaves}(T)} B^x_j$, one could extend the tree by sampling T' from $\mathrm{MTx}(\lambda, T, \boldsymbol{x})$, such that $\boldsymbol{x}$ lies within a leaf node in T' and apply the procedure described below using the extended tree T'. Appendix B describes this case in more detail.

Given training data $\mathcal{D}_{1:N}$, a Mondrian tree T and the hierarchical prior over $\mathcal{G}$, the predictive label distribution $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$ is obtained by integrating over $\mathcal{G}$, i.e.

$$\begin{aligned} p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N}) &= \mathbb{E}_{\mathcal{G} \sim p_T(\mathcal{G}|\mathcal{D}_{1:N})}[p_T(y|\boldsymbol{x}, \mathcal{G})] \\ &= \mathbb{E}_{\mathcal{G} \sim p_T(\mathcal{G}|\mathcal{D}_{1:N})}[G_{\mathsf{leaf}(\boldsymbol{x}),y}] \\ &= \overline{G}_{\mathsf{leaf}(\boldsymbol{x}),y}. \end{aligned} \tag{6}$$

Hence, the prediction is given by $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$, the posterior mean at $\mathsf{leaf}(\boldsymbol{x})$. The posterior mean $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$ can be computed using existing techniques, which we review in the rest of this section.

Posterior inference in the HNSP is a special case of posterior inference in hierarchical PYP (HPYP). Teh [17] considers the HPYP with multinomial likelihood (in the context of language modeling)—the model considered here (HNSP with multinomial likelihood) is a special case of [17]. Hence, we just sketch the high level picture and refer the reader to [17] for further details. We first describe posterior inference given N data points $D_{1:N}$ (batch setting), and later explain how to adapt inference to the online setting. Finally, we describe the computation of the predictive posterior distribution.

Batch setting

Posterior inference is done using the Chinese restaurant process representation, wherein every node of the decision tree is a restaurant; the training data points are the customers seated in the tables associated with the leaf node restaurants; these tables are in turn customers at the tables in their corresponding parent level restaurant; the dish served at each table is the class label. Exact inference is intractable and hence we resort to approximations. In particular, we use the approximation known as the interpolated Kneser-Ney (IKN) smoothing, a popular smoothing technique for language modeling [10]. The IKN smoothing can be interpreted as an approximate inference scheme for the HPYP, where the number of tables serving a particular dish in a restaurant is at most one [17]. More precisely, if $c_{j,k}$ denotes the number of customers at restaurant j eating dish k and $\mathsf{tab}_{j,k}$ denotes the number of tables at restaurant j serving dish k, the IKN approximation sets $\mathsf{tab}_{j,k} = \mathsf{min}(c_{j,k}, 1)$. The counts $c_{j,k}$ and $\mathsf{tab}_{j,k}$ can be computed in a single bottom-up pass as follows: for every leaf node $j \in \mathsf{leaves}(\mathsf{T})$, $c_{j,k}$ is simply the number of training data points with label k at node j; for every internal node $j \in \mathsf{T} \setminus \mathsf{leaves}(\mathsf{T})$, we set $c_{j,k} = \mathsf{tab}_{\mathsf{left}(j),k} + \mathsf{tab}_{\mathsf{right}(j),k}$. For a leaf node j, this procedure is summarized in Algorithm 5. (Note that this pseudocode just serves as a reference; in practice, these counts are updated in an online fashion, as described in Algorithm 6.)

Posterior inference: online setting

It is straightforward to extend inference to the online setting. Adding a new data point $(\boldsymbol{x}, y)$ affects only the counts along the path from the root to the leaf node of that data point. We update the counts in a bottom-up fashion, starting at the leaf node containing the data point,

Algorithm 5 InitializePosteriorCounts(j)

For all k, set $c_{jk} = \#\{i \in N(j) : y_i = k\}$
Initialize $j' = j$
while True **do**
  if $j' \notin \mathsf{leaves}(\mathsf{T})$ **then**
    For all k, set $c_{j'k} = \mathsf{tab}_{\mathsf{left}(j'),k} + \mathsf{tab}_{\mathsf{right}(j'),k}$
  For all k, set $\mathsf{tab}_{j'k} = \mathsf{min}(c_{j'k}, 1)$ ▷ *IKN approximation*
  if $j' = \epsilon$ **then**
    return
  else
    $j' \leftarrow \mathsf{parent}(j')$

$\mathsf{leaf}(\boldsymbol{x})$. Due to the nature of the IKN approximation, we can stop at the internal node j where $c_{j,y} = 1$ and need not traverse up till the root. This procedure is summarized in Algorithm 6.

Algorithm 6 UpdatePosteriorCounts(j, y)

$c_{jy} \leftarrow c_{jy} + 1$
Initialize $j' = j$
while True **do**
  if $\mathsf{tab}_{j'y} = 1$ **then** ▷ *none of the counts above need to be updated*
    return
  else
    if $j' \notin \mathsf{leaves}(\mathsf{T})$ **then**
      $c_{j'y} = \mathsf{tab}_{\mathsf{left}(j'),y} + \mathsf{tab}_{\mathsf{right}(j'),y}$
    $\mathsf{tab}_{j'y} = \mathsf{min}(c_{j'y}, 1)$ ▷ *IKN approximation*
    if $j' = \epsilon$ **then**
      return
    else
      $j' \leftarrow \mathsf{parent}(j')$

Predictive posterior computation Given the counts $c_{j,k}$ and table assignments $\mathsf{tab}_{j,k}$, the predictive probability (i.e., posterior mean) at node j can be computed recursively as follows:

$$\overline{G}_{jk} = \begin{cases} \frac{c_{j,k} - d_j \mathsf{tab}_{j,k}}{c_{j,\cdot}} + \frac{d_j \mathsf{tab}_{j,\cdot}}{c_{j,\cdot}} \overline{G}_{\mathsf{parent}(j),k} & c_{j,\cdot} > 0, \\ \overline{G}_{\mathsf{parent}(j),k} & c_{j,\cdot} = 0, \end{cases} \tag{7}$$

where $c_{j,\cdot} = \sum_k c_{j,k}$, $\mathsf{tab}_{j,\cdot} = \sum_k \mathsf{tab}_{j,k}$, and $d_j := \exp\big(-\gamma(\tau_j - \tau_{\mathsf{parent}(j)})\big)$ is the "discount" for node j, defined in Section 4. Informally, the discount interpolates between the counts c and the prior. If the discount $d_j \approx 1$, then $\overline{G}_j$ is more like its parent $\overline{G}_{\mathsf{parent}(j)}$. If $d_j \approx 0$, then $\overline{G}_j$ weights the counts more. These predictive probabilities can be computed in a single top-down pass as shown in Algorithm 7.

B Prediction using Mondrian tree

Let $\boldsymbol{x}$ denote a test data point. We are interested in the predictive probability of y at $\boldsymbol{x}$, denoted by $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$. As in typical decision trees, the process involves a top-down tree traversal, starting from the root. If $\boldsymbol{x}$ is already "contained" in the tree T, i.e., if $\boldsymbol{x} \in B_j^x$

Algorithm 7 ComputePosteriorPredictiveDistribution$(T, \mathcal{G})$

▷ *Description of top-down pass to compute posterior predictive distribution given by (7)*
▷ $\overline{G}_{jk}$ *denotes the posterior probability of* $y = k$ *at node* j
Initialize the ordered set $J = \{\epsilon\}$
while J not empty **do**
  Pop the first element of J
  if $j = \epsilon$ **then**
    $\overline{G}_{\mathsf{parent}(\epsilon)} = H$
  Set $d = \exp\big(-\gamma(\tau_j - \tau_{\mathsf{parent}(j)})\big)$
  For all k, set $\overline{G}_{jk} = c_{j,\cdot}^{-1}\Big(c_{j,k} - d\ \mathsf{tab}_{j,k} + d\ \mathsf{tab}_{j,\cdot}\ \overline{G}_{\mathsf{parent}(j),k}\Big)$
  if $j \notin \mathsf{leaves}(\mathsf{T})$ **then**
    Append $\mathsf{left}(j)$ and $\mathsf{right}(j)$ to the end of the ordered set J

for some leaf $j \in \mathsf{leaves}(T)$, then the prediction is taken to be $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$, which is computed as described in Appendix A. Otherwise, we somehow need to incorporate $\boldsymbol{x}$. One choice is to extend T by sampling T' from $\mathrm{MTx}(\lambda, T, \mathcal{D}_{1:n}, \boldsymbol{x})$ as described in Algorithm 3, and set the prediction to $\overline{G}_j$, where $j \in \mathsf{leaves}(T')$ is the leaf node containing $\boldsymbol{x}$. A particular extension T' might lead to an overly confident prediction; hence, we average over *every* possible extension T'. This expectation can be carried out analytically, using properties of the Mondrian process, as we show below.

Let $\mathsf{ancestors}(j)$ denote the set of all ancestors of node j. Let $\mathsf{path}(j) = \{j\} \cup \mathsf{ancestors}(j)$, that is, the set of all nodes along the ancestral path from j to the root. Recall that $\mathsf{leaf}(\boldsymbol{x})$ is the unique leaf node in T such that $\boldsymbol{x} \in B_{\mathsf{leaf}(\boldsymbol{x})}$. If the test point $\boldsymbol{x} \in B^x_{\mathsf{leaf}(\boldsymbol{x})}$ (i.e., $\boldsymbol{x}$ lies within the 'gray rectangle' at the leaf node), it can never branch off; else, it can branch off at one or more points along the path from the root to $\mathsf{leaf}(\boldsymbol{x})$. More precisely, if $\boldsymbol{x}$ lies outside B^x_j at node j, the probability that $\boldsymbol{x}$ will branch off into its own node at node j, denoted by[6] $p^s_j(\boldsymbol{x})$, is equal to the probability that a split exists in B_j outside B^x_j, which is

$$p^s_j(\boldsymbol{x}) = 1 - \exp\big(-\Delta_j \eta_j(\boldsymbol{x})\big), \quad \text{where } \eta_j(\boldsymbol{x}) = \sum_d \big(\mathsf{max}(x_d - u^x_{jd}, 0) + \mathsf{max}(\ell^x_{jd} - x_d, 0)\big),$$

and $\Delta_j = \tau_j - \tau_{\mathsf{parent}(j)}$. Note that $p^s_j(\boldsymbol{x}) = 0$ if $\boldsymbol{x}$ lies within B^x_j (i.e., if $\ell^x_{jd} \le x_d \le u^x_{jd}$ for all d). The probability of $\boldsymbol{x}$ not branching off before reaching node j is given by $\prod_{j' \in \mathsf{ancestors}(j)}(1 - p^s_{j'}(\boldsymbol{x}))$.

If $\boldsymbol{x} \in B^x_{\mathsf{leaf}(\boldsymbol{x})}$, the prediction is given by $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$. If there is a split in B_j outside B^x_j, let $\tilde{j}$ denote the new parent of j and $\mathsf{child}(\tilde{j})$ denote the child node containing just the test data point,; in this case, the prediction is $\overline{G}_{\mathsf{child}(\tilde{j})}$. Averaging over the location where the test point branches off, we obtain

$$p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N}) = \sum_{j \in \mathsf{path}(\mathsf{leaf}(\boldsymbol{x}))} \Big(\prod_{j' \in \mathsf{ancestors}(j)} (1 - p^s_{j'}(\boldsymbol{x})) \Big) F_j(\boldsymbol{x}), \tag{8}$$

where

$$F_j(\boldsymbol{x}) = p^s_j(\boldsymbol{x}) \mathbb{E}_{\Delta_{\tilde{j}}}\Big[\overline{G}_{\mathsf{child}(\tilde{j})}\Big] + \mathbb{1}[j = \mathsf{leaf}(\boldsymbol{x})](1 - p^s_j(\boldsymbol{x}))\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}. \tag{9}$$

The second term in $F_j(\boldsymbol{x})$ needs to be computed only for the leaf node $\mathsf{leaf}(\boldsymbol{x})$ and is simply the posterior mean of $G_{\mathsf{leaf}(\boldsymbol{x})}$ weighted by $1 - p^s_{\mathsf{leaf}(\boldsymbol{x})}(\boldsymbol{x})$. The posterior mean of $G_{\mathsf{leaf}(x)}$, given

[6] The superscript s in $p^s_j(\boldsymbol{x})$ is used to denote the fact that this split 'separates' the test data point $\boldsymbol{x}$ into its own leaf node.

by $\overline{G}_{\mathsf{leaf}(\boldsymbol{x})}$, can be computed using (7). The first term in $F_j(\boldsymbol{x})$ is simply the posterior mean of $G_{\mathsf{child}(j)}$, averaged over Δ_j, weighted by $p_j^s(\boldsymbol{x})$. Since no labels are observed in $\mathsf{child}(j)$, $c_{\mathsf{child}(j),\cdot} = 0$, hence from (7), we have $\overline{G}_{\mathsf{child}(j)} = \overline{G}_j$. We compute $\overline{G}_j$ using (7). We average over Δ_j due to the fact that the discount in (7) for the node j depends on $\tau_j - \tau_{\mathsf{parent}(j)} = \Delta_j$. To average over all valid split times τ_j, we compute expectation w.r.t. Δ_j which is distributed according to a truncated exponential with rate $\eta_j(\boldsymbol{x})$, truncated to the interval $[0, \Delta_j]$.

The procedure for computing $p_T(y|\boldsymbol{x}, \mathcal{D}_{1:N})$ for any $\boldsymbol{x} \in \mathbb{R}^D$ is summarized in Algorithm 8.

Algorithm 8 $\mathsf{Predict}(T, \boldsymbol{x})$

▷ *Description of prediction using a Mondrian tree, given by (8)*
Initialize $j = \epsilon$ and $p_{\mathsf{NotSeparatedYet}} = 1$
Initialize $\mathbf{s} = \mathbf{0}_K$ ▷ $\mathbf{s}$ *is* K*-dimensional vector where* $s_k = p_T(y = k|\boldsymbol{x}, \mathcal{D}_{1:N})$
while True **do**
  Set $\Delta_j = \tau_j - \tau_{\mathsf{parent}(j)}$ and $\eta_j(\boldsymbol{x}) = \sum_d \big(\mathsf{max}(x_d - u_{jd}^x, 0) + \mathsf{max}(\ell_{jd}^x - x_d, 0)\big)$
  Set $p_j^s(\boldsymbol{x}) = 1 - \exp\big(-\Delta_j \eta_j(\boldsymbol{x})\big)$
  if $p_j^s(\boldsymbol{x}) > 0$ **then**
    ▷ *Let* $\boldsymbol{x}$ *branch off into its own node* $\mathsf{child}(j)$*, creating a new node* $\tilde{j}$ *which is the parent of* j *and* $\mathsf{child}(j)$*.* $\overline{G}_{\mathsf{child}(j)} = \overline{G}_{\tilde{j}}$ *from (7) since* $c_{\mathsf{child}(j),\cdot} = 0$*.*
    Compute expected discount $\bar{d} = \mathbb{E}_\Delta[\exp(-\gamma\Delta)]$ where Δ is drawn from a truncated exponential with rate $\eta_j(\boldsymbol{x})$, truncated to the interval $[0, \Delta_j]$.
    For all k, set $c_{\tilde{j},k} = \mathsf{tab}_{\tilde{j},k} = \mathsf{min}(c_{j,k}, 1)$
    For all k, set $\overline{G}_{\tilde{j}k} = c_{\tilde{j},\cdot}^{-1}\Big(c_{\tilde{j},k} - \bar{d}\ \mathsf{tab}_{\tilde{j},k} + \bar{d}\ \mathsf{tab}_{\tilde{j},\cdot}\ \overline{G}_{\mathsf{parent}(\tilde{j}),k}\Big)$ ▷ *Algorithm 7 and (9)*
    For all k, update $s_k \leftarrow s_k + p_{\mathsf{NotSeparatedYet}}\ p_j^s(\boldsymbol{x})\overline{G}_{\tilde{j}k}$
  if $j \in \mathsf{leaves}(\mathsf{T})$ **then**
    For all k, update $s_k \leftarrow s_k + p_{\mathsf{NotSeparatedYet}}(1 - p_j^s(\boldsymbol{x}))\overline{G}_{jk}$ ▷ *Algorithm 7 and (9)*
    return $\hat{y} = \arg\mathsf{max}_k\ s_k$
  else
    $p_{\mathsf{NotSeparatedYet}} \leftarrow p_{\mathsf{NotSeparatedYet}}(1 - p_j^s(\boldsymbol{x}))$
    if $x_{\delta_j} \leq \xi_j$ **then** $j \leftarrow \mathsf{left}(j)$ **else** $j \leftarrow \mathsf{right}(j)$ ▷ *recurse to the child where* $\boldsymbol{x}$ *lies*

C Computational complexity

We discuss the computational complexity associated with a single Mondrian tree. The complexity of a forest is simply M times that of a single tree; however, this computation can be trivially parallelized since there is no interaction between the trees. Assume that the N data points are processed one by one. Assuming the data points form a balanced binary tree after each update, the computational cost of processing the n^{th} data point is at most $\mathcal{O}(\log n)$ (add the data point into its own leaf, update posterior counts for HNSP in bottom-up pass from leaf to root). The overall cost to process N data points is $\mathcal{O}(\sum_{n=1}^N \log n) = \mathcal{O}(\log N!)$, which for large N tends to $\mathcal{O}(N \log N)$ (using Stirling approximation for the factorial function). For offline RF and ERT, the expected complexity with n data points is $\mathcal{O}(n \log n)$. The complexity of the re-trained version is $\mathcal{O}(\sum_{n=1}^N n \log n) = \mathcal{O}(\log \prod_{n=1}^N n^n)$, which for large N tends to $\mathcal{O}(N^2 \log N)$ (using asymptotic expansion of the hyper factorial function).

D Pseudocode for paused Mondrians

Algorithm 9 $\mathsf{SampleMondrianBlock}\big(j, \mathcal{D}_{N(j)}, \lambda\big)$ version that depends on labels

Add j to T
$\forall d$, set $\ell^x_{jd} = \mathsf{min}(\boldsymbol{X}_{N(j),d}), u^x_{jd} = \mathsf{max}(\boldsymbol{X}_{N(j),d})$ ▷ *dimension-wise* min *and* max
if $\mathsf{AllLabelsIdentical}(Y_{N(j)})$ **then**
    Set $\tau_j = \lambda$ ▷ *pause Mondrian*
else
    Sample E from exponential distribution with rate $\sum_d (u_{jd} - \ell_{jd})$
    Set $\tau_j = \tau_{\mathsf{parent}(j)} + E$
if $\tau_j < \lambda$ **then**
    Sample δ_j with probability of choosing d proportional to $u^x_{jd} - \ell^x_{jd}$
    Sample split location ξ_j along dimension δ_j from an uniform distribution over $\mathcal{U}[\ell^x_{jd}, u^x_{jd}]$
    Set $N(\mathsf{left}(j)) = \{n \in N(j) : x_{n\delta_j} \leq \xi_j\}$ and $N(\mathsf{right}(j)) = \{n \in N(j) : x_{n\delta_j} > \xi_j\}$
    $\mathsf{SampleMondrianBlock}\big(\mathsf{left}(j), \mathcal{D}_{N(\mathsf{left}(j))}, \lambda\big)$
    $\mathsf{SampleMondrianBlock}\big(\mathsf{right}(j), \mathcal{D}_{N(\mathsf{right}(j))}, \lambda\big)$
else
    Set $\tau_j = \lambda$ and add j to $\mathsf{leaves}(\mathsf{T})$ ▷ *j is a leaf node*
    $\mathsf{InitializePosteriorCounts}(j)$ ▷ *Algorithm 5*

Algorithm 10 ExtendMondrianBlock$(T, \lambda, j, \mathcal{D})$ version that depends on labels

if AllLabelsIdentical$(Y_{N(j)})$ **then** ▷ *paused Mondrian leaf*
  Update extent $\boldsymbol{\ell}_j^x \leftarrow \min(\boldsymbol{\ell}_j^x, \boldsymbol{x}), \mathbf{u}_j^x \leftarrow \max(\mathbf{u}_j^x, \boldsymbol{x})$
  Append $\mathcal{D}$ to $\mathcal{D}_{N(j)}$ ▷ *append* $\boldsymbol{x}$ *to* $X_{N(j)}$ *and* y *to* $Y_{N(j)}$
  if $y = \text{unique}(Y_{N(j)})$ **then**
    UpdatePosteriorCounts(j, y) ▷ *Algorithm 6*
    return ▷ *continue pausing*
  else
    Remove j from leaves(T)
    SampleMondrianBlock$\big(j, \mathcal{D}_{N(j)}, \lambda\big)$ ▷ *un-pause Mondrian*
else
  Set $\mathbf{e}^\ell = \max(\boldsymbol{\ell}_j^x - \boldsymbol{x}, 0)$ and $\mathbf{e}^u = \max(\boldsymbol{x} - \mathbf{u}_j^x, 0)$ ▷ $\mathbf{e}^\ell = \mathbf{e}^u = \mathbf{0}_D$ *if* $\boldsymbol{x} \in B_j^x$
  Sample $E \sim \text{Exp}(\sum_d (e_d^\ell + e_d^u))$
  if $E + \tau_{\text{parent}(j)} < \tau_j$ **then** ▷ *introduce new parent for node* j
    Create new Mondrian block $\tilde{\jmath}$ where $\boldsymbol{\ell}_{\tilde{\jmath}}^x = \min(\boldsymbol{\ell}_j^x, \boldsymbol{x})$ and $\mathbf{u}_{\tilde{\jmath}}^x = \max(\mathbf{u}_j^x, \boldsymbol{x})$
    Sample $\delta_{\tilde{\jmath}}$ with $\Pr(\delta_{\tilde{\jmath}} = d)$ proportional to $e_d^\ell + e_d^u$
    if $x_{\delta_{\tilde{\jmath}}} > u_{j,\delta_{\tilde{\jmath}}}^x$, **then** sample $\xi_{\tilde{\jmath}}$ from $\mathcal{U}[u_{j,\delta_{\tilde{\jmath}}}^x, x_{\delta_{\tilde{\jmath}}}]$, **else** sample $\xi_{\tilde{\jmath}}$ from $\mathcal{U}([x_{\delta_{\tilde{\jmath}}}, \ell_{j,\delta_{\tilde{\jmath}}}^x])$
    if $j = \epsilon$ **then** ▷ *set* $\tilde{\jmath}$ *as the new root*
      $\epsilon \leftarrow \tilde{\jmath}$
    else ▷ *set* $\tilde{\jmath}$ *as child of* parent(j)
      if $j = \text{left}(\text{parent}(j))$, **then** $\text{left}(\text{parent}(j)) \leftarrow \tilde{\jmath}$, **else** $\text{right}(\text{parent}(j)) \leftarrow \tilde{\jmath}$
    if $x_{\delta_{\tilde{\jmath}}} > \xi_{\tilde{\jmath}}$ **then**
      Set $\text{left}(\tilde{\jmath}) = j$ and SampleMondrianBlock$\big(\text{right}(\tilde{\jmath}), \mathcal{D}, \lambda\big)$ ▷ *create new leaf for* x
    else
      Set $\text{right}(\tilde{\jmath}) = j$ and SampleMondrianBlock$\big(\text{left}(\tilde{\jmath}), \mathcal{D}, \lambda\big)$ ▷ *create new leaf for* x
  else
    Update $\boldsymbol{\ell}_j^x \leftarrow \min(\boldsymbol{\ell}_j^x, \boldsymbol{x}), \mathbf{u}_j^x \leftarrow \max(\mathbf{u}_j^x, \boldsymbol{x})$ ▷ *update extent of node* j
    if $j \notin$ leaves(T) **then** ▷ *return if* j *is a leaf node, else recurse down the tree*
      if $x_{\delta_j} \leq \xi_j$ **then** $\text{child}(j) = \text{left}(j)$ **else** $\text{child}(j) = \text{right}(j)$
      ExtendMondrianBlock$(T, \lambda, \text{child}(j), \mathcal{D})$ ▷ *recurse on child containing* x