Decision Trees and Random Decision Forests

Carlo Tomasi

November 1, 2021

1 Decision Trees

Linear regressors from $X = \mathbb{R}^d$ to $Y = \mathbb{R}^K$ have a limited expressive power, because they fit K *affine* functions to the training set. Functions that are more complex than that are approximated poorly. Similarly, K-class linear classifiers partition the data space X into K *convex* regions. If the true decision regions are not convex, the predictors underfit and perform poorly even on the training data.

This limited expressive power has also a silver lining: The number m of parameters of linear predictors is relatively small, $m = K(d + 1)$, so that a linear predictor requires fewer training samples to train, relative to a more expressive predictor, in order to achieve a certain value for the training risk. In other words, linear predictors have low sample complexity and, as a result, they generalize better than more expressive ones.

Kernel SVMs can combine the best of both worlds to some extent: They can model complex decision boundaries and at the same time generalize well, thanks to the fact that the sample complexity of SVMs with bounded data spaces is independent of d. However, kernels need to be designed so as to adapt well to the true decision boundary geometry, and this problem is typically addressed in an empirical fashion by trial and error.

When we studied logistic-regression classifiers, we also saw one of the advantages of score-based classifiers, namely that scores provide more information than just an estimated label. Specifically, scores for different classes can be the basis for simple extensions of binary classifiers to the multi-class case.

Decision trees combine the advantages of a score-based predictor (for both classifiers and regressors!) with the expressiveness deriving from a very flexible partition of X. Specifically, they recursively split X with hyperplanes, and they assign class scores or target values (depending on whether the problem is one of regression or classification) to each subset of the partition. This recursive splitting leads to effectively arbitrary expressiveness, as long as the partition is fine enough.

Since the splitting is recursive, the resulting partition of the data space X can be represented by a strictly binary tree. As will be seen in more detail below, the root of the tree represents all of X, and contains the parameters of the hyper-plane used for the first split. The two children of the root represent the two half-spaces that that hyper-plane divides X into. Each of the two children in turn contains an additional hyper-plane that splits the child's half-space, and the structure continues recursively.

The tree can then be used for prediction: Given a data point $\mathbf{x}$, check which side of the root partition it belongs to, and send it to the corresponding child. This check is repeated at every internal tree node encountered in this way, until a leaf is reached. The leaf corresponds to a

(possibly small) convex region of X, and predicted scores or values $\hat{y} = h(\mathbf{x})$ are associated to that region. We will see in the next section how the hyper-planes are constructed, and how the prediction values at each leaf are computed.

Staying at a high level for now, decision trees determine each of the hyper-planes greedily, by choosing the hyper-plane (out of a set of choices that is typically restricted, as we will see) that maximizes the predictor's confidence, gauged through a measure called *purity*. Different definitions of purity have been proposed, but they are all based on the distribution $p(y|X_S)$ of the values y for data points $\mathbf{x} \in X_S$, where X_S is one of the regions of the partition. Loosely speaking, $p(y|X_S)$ is *pure* if it is heavily concentrated around a small set of values, so that there is at least an approximate agreement as to what individual value should be assigned to data points $\mathbf{x}$ in X_S.

Of course, $p(y|X_S)$ is unknown, but it can be estimated from the training data:

$$p(y|X_S) \;\approx\; p(y|S)$$

where S is the subset of the training set T whose data points are in X_S:

$$S \stackrel{\mathsf{def}}{=} \{(\mathbf{x}, y) \in T \;:\; \mathbf{x} \in X_S\}$$

and $p(y|S)$ denotes an empirical estimate of a distribution, for instance, a histogram. With this approximation, the empirical probability distributions of values in subsets S of the training set T are interpreted as (estimates of the) statistical probability distributions of values for entire regions X_S of data space X.

In particular, the distributions $p(y|S)$ for the leaves of the tree are used to compute a prediction value through a statistical *summary* of $p(y|S)$:

$$\hat{y}(X_S) = \text{summary}(p(y|X_S)) \;.$$

Typically, the summary is a mode (majority value) for classification, and either a mean or a median for regression. These choices are akin to the ones used for k-nearest-neighbor classification when $k > 1$.

As should be clear from the discussion so far, decision trees also have limitations. First, the partition into regions is greedy, and therefore sub-optimal: It is possible (and indeed likely) that to achieve an optimal partition overall one may have to make partition decisions which, when seen individually and out of context, appear to be sub-optimal.

A second weakness of decision trees results from their very expressiveness, as we have by now come to expect: Flexibility leads to over-fitting (or high sample complexity, if over-fitting is to be avoided). Because of this, a large body of literature has been devoted to ways of *pruning* decision trees, that is, curbing their ability to partition X so as to improve generalization. We will not study pruning methods, because a better way to improve generalization is to build multiple trees with different views of the data (in a sense to be clarified later), and let them vote on a value. This is the basic idea of *random decision forests*, the subject of a later section.

The present section first defines decision trees in general, and then considers how decision trees are trained. Choosing a training method involves defining (i) a measure of purity, (ii) a way to infer a locally optimal partition of a set S into two subsets L and R so as to increase purity as much as possible, and (iii) a criterion for stopping the recursive partition process.

1.1 The Structure of Decision Trees and their Use as Predictors

A *decision tree* is a binary tree that defines a recursive partition of the data space X into subregions. Specifically, the root τ of the tree is associated to all of X, and contains a predicate $\mathcal{P}_\tau(\mathbf{x})$ called a *split rule*. The left child $\tau.L$ of τ is associated to the subset X_L of X of all the points in X that satisfy the predicate. The right child $\tau.R$ of τ is associated to the complement X_R of X_L in X. Each child is then expanded recursively in the same way, by its own predicate.

During training, the predicates in the tree's nodes are applied to the data points in the training set T, rather than in the data space X, with the result of splitting T into subsets. Specifically, the set S at an internal node is split into those samples whose data points are in X_L and those whose data points are in X_R:

$$L \stackrel{\mathsf{def}}{=} \{(\mathbf{x}, y) \in S \mid \mathbf{x} \in X_L\} \quad \text{and} \quad R \stackrel{\mathsf{def}}{=} \{(\mathbf{x}, y) \in S \mid \mathbf{x} \in X_R\} \,.$$

Just as before, we can view $p(y|L)$ and $p(y|R)$ as estimates of the probability distributions of y in X_L and X_R, respectively:

$$p(y|L) \approx \mathbb{P}_T[y \mid \mathbf{x} \in X_L] \quad \text{and} \quad p(y|R) \approx \mathbb{P}_T[y \mid \mathbf{x} \in X_R] \,.$$

The split rules are learned by partitioning the training set T recursively in a way that increases the purity of the subsets formed by each split, in a sense to be made more precise later on.

A popular binary split rule called a *1-rule* partitions a subset $S \subseteq X \times Y$ into the two sets

$$L = \{(\mathbf{x}, y) \in S \,|\, x_j \leq t\} \quad \text{and} \quad R = \{(\mathbf{x}, y) \in S \,|\, x_j > t\} \tag{1}$$

where x_j is the j-th component of $\mathbf{x}$ and t is a real number. Thus, a 1-rule only allows for separating hyper-planes that are aligned with the coordinate axes: Pick a coordinate index j, and check if x_j is above or below a threshold t.

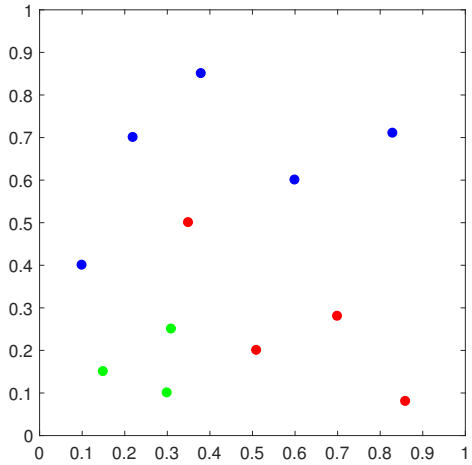
This note considers only binary decision trees[1] with 1-rule splits, and the word "binary" is omitted in what follows.

Concretely, the split rules are placed on the interior nodes of a binary tree and the probability distributions are on the leaves. The tree τ can be defined recursively as either a single (leaf) node with values of the posterior probability $p(y|S)$ collected in a vector $\tau.\mathbf{p}$ or an (interior) node with a split function with parameters $\tau.j$ and $\tau.t$ that returns either the left or the right descendant ($\tau.L$ or $\tau.R$) of τ depending on the outcome of the split. At inference time, a prediction tree τ takes a new data point $\mathbf{x}$, looks up its posterior distribution in the tree by following splits in τ down to a leaf, and returns the value y obtained by summarizing $\tau.\mathbf{p}$. This algorithm is detailed in Algorithm 1 and illustrated in Figure 1 for a three-class classifier (so that the summary is $\arg\max(\tau.\mathbf{p})$).

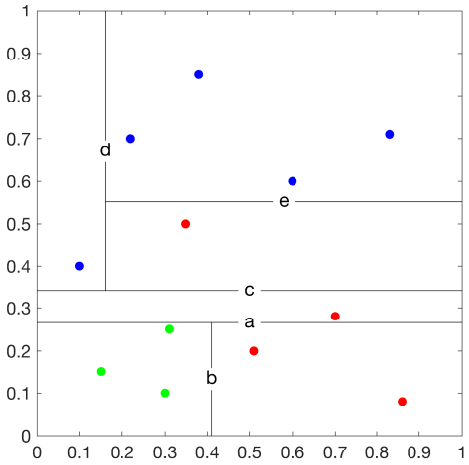
1.2 Training Prediction Trees

Optimal training of a prediction tree would compute the partition of X that leads to the lowest possible generalization error. In addition, a good tree from a computational point of view would use the minimum possible number of splits. The second requirement, optimal efficiency, is unrealistic, since building the most efficient tree is NP-complete, as can be proven by a reduction of the set cover problem [9].

[1]The tree is binary, but the predictor is not necessarily binary. For instance, a binary classification tree can handle more than two classes.



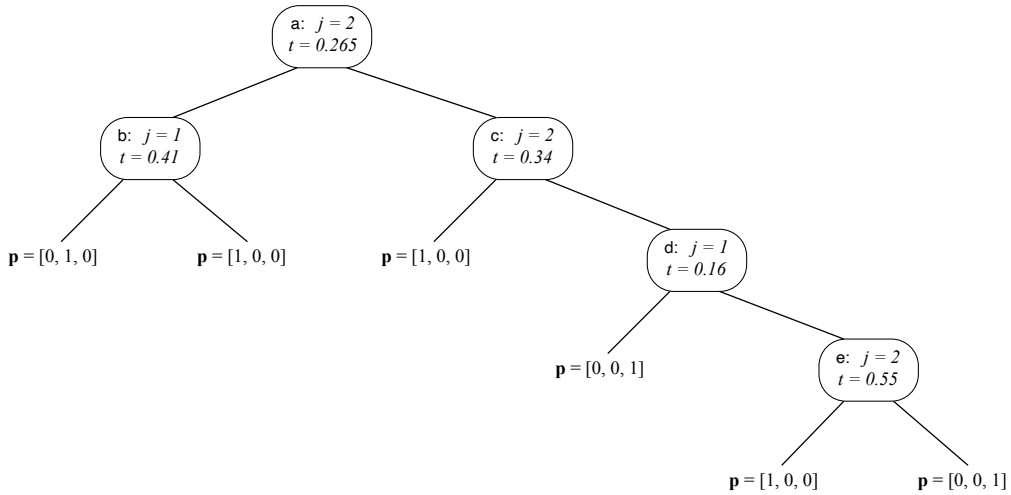
(a)



(b)

node	impurity
a	0.583
b	0.4
c	0.286
d	0.167
e	0.2

(c)



(d)

Figure 1: The training samples in (a) belong to three categories (1: red, 2: green, 3: blue). The partition in (b) is computed in the order a, b, c, d, e, and the impurities of the sets each line splits are shown in (c). The tree in (d) represents the same partition, and each interior node corresponds to one line in (c). All the leaves of the tree contain zero-impurity posterior probabilities $\mathbf{p} = [\,p(\text{red}|\mathbf{x}),\ p(\text{green}|\mathbf{x}),\ p(\text{blue}|\mathbf{x})\,]$. This purity corresponds to the fact that all the points in each region of the partition in (b) have the same label.

The impurity function used in this example is the training error $1 - \max_y p(y|S)$. For example, before the first split, the training set has 4 red samples, 3 green, and 5 blue, so that $p(y|S)$ is $(4, 3, 5)/12$ and the training error is $1 - 5/12 \approx 0.583$ (first row in (c)).

Algorithm 1 Prediction with a decision tree

```
function y ← predict(x, τ, summary)
    if leaf?(τ) then
        return summary(τ.p)
    else
        return predict(x, split(x, τ), summary)
    end if
end function

function τ ← split(x, τ)
    if x_{τ.j} ≤ τ.t then
        return τ.L
    else
        return τ.R
    end if
end function
```

Prediction trees are trained with a greedy procedure, and only ensure optimality—on the training set—at each node separately. The procedure is sketched in Algorithm 2, and is invoked with the call

$$\text{trainTree}(T, 0) \ .$$

The algorithm first determines whether the set S it is given as input is worth splitting.[2] If so, it finds optimal parameters j and t that split S into sets L and R (equation (1)), stores those parameters at the root of a new tree τ, stores as the root's children $\tau.L$ and $\tau.R$ the result of calling itself recursively on sets L and R, and returns τ.

If on the other hand S is not worth splitting, then the new tree τ is a single leaf node that contains an estimate of the posterior distribution of labels in S.

This procedure leads to large trees that overfit and therefore generalize poorly, so a second step of training *prunes* the tree to improve the generalization error. However, random forest classifiers address generalization in a different way, by combining the predictions made by several trees. Because of this, pruning is not performed in random forest predictors, and is not discussed here. Instead, we now show how to split a set (findSplit), how to decide whether to continue splitting (split?), and how to estimate the distribution of labels in a set (distribution).

Splitting. The optimal single split of training data in set S into two sets L and R maximizes the decrease

$$\Delta \mathrm{i}(S, L, R) = \mathrm{i}(S) - \frac{|L|}{|S|}\mathrm{i}(L) - \frac{|R|}{|S|}\mathrm{i}(R) \tag{2}$$

where $\mathrm{i}(S)$ is the *impurity* of S. The so-called *Gini index*

$$\mathrm{i}(S) = 1 - \sum_{y \in Y} p^2(y|S)$$

[2] Read on to find out when a set is worth splitting, how the optimal split parameters are found, and how the posterior distribution is estimated.

Algorithm 2 Training a prediction tree

function $\tau \leftarrow \text{trainTree}(S, \text{depth})$
  if split?(S, depth) **then**
    $[L, R, \tau.j, \tau.t] \leftarrow \text{findSplit}(S)$
    $\tau.L \leftarrow \text{trainTree}(L, \text{depth} + 1)$
    $\tau.R \leftarrow \text{trainTree}(R, \text{depth} + 1)$
  else
    $\tau.\mathbf{p} \leftarrow \text{distribution}(S)$
  end if
  return τ
end function

function $[L, R, j, t] \leftarrow \text{findSplit}(S)$
  $i_S \leftarrow \text{i}(S)$ ▷ $\text{i}(S)$ is the impurity of S. See text.
  $\Delta_{\text{opt}} \leftarrow -1$ ▷ At the end, Δ_{opt} will be the greatest decrease in impurity.
  for $j = 1, \ldots, d$ **do** ▷ Loop on all data dimensions.
    for $\ell = 1, \ldots, u_j$ **do** ▷ Loop on all thresholds for dimension j.
      $L \leftarrow \{(\mathbf{x}, y) \in S \,|\, x_j \leq t_j^{(\ell)}\}$ ▷ The thresholds $t_j^{(\ell)}$ for $j = 1, \ldots, d$ and $\ell = 1, \ldots, u_j$
      $R \leftarrow S \setminus L$ ▷ are assumed to have been precomputed (see text).
      $\Delta \leftarrow i_S - \frac{|L|}{|S|}\text{i}(L) - \frac{|R|}{|S|}\text{i}(R)$ ▷ See text for a faster way to compute Δ
      if $\Delta > \Delta_{\text{opt}}$ **then**
        $[\Delta_{\text{opt}}, L_{\text{opt}}, R_{\text{opt}}, d_{\text{opt}}, t_{\text{opt}}] \leftarrow [\Delta, L, R, j, t]$
      end if
    end for
  end for
  return $[L_{\text{opt}}, R_{\text{opt}}, d_{\text{opt}}, t_{\text{opt}}]$
end function

function answer $\leftarrow$ split?(S, depth)
  return $\text{i}(S) > 0$ and $|S| > s_{\min}$ and $\text{depth} < d_{\max}$ ▷ $s_{\min}$ and $d_{\max}$ are predefined thresholds
end function

function $\mathbf{p} \leftarrow \text{distribution}(S)$
  $\mathbf{p} \leftarrow [0, \ldots, 0]$ ▷ A vector of K zeros
  $n \leftarrow 0$
  for $(\mathbf{x}, y) \in S$ **do**
    $p(y) \leftarrow p(y) + 1$
    $n \leftarrow n + 1$
  end for
  return $\mathbf{p}/n$
end function

is often used for either classifiers or regressors, where

$$p(y|S) = \frac{1}{|S|} \sum_{(\mathbf{x}_i, y_i) \in S} I(y_i \approx y)$$

is the fraction of training values in set S that fall into the same bin as (or are equal to, for classifiers) value y. The Gini index minimizes the training error for the stochastic decision rule

$$\hat{y} = h_{\text{Gini}}(\mathbf{x}) = y \text{ with probability } p(y|S(\mathbf{x}))$$

where $S(\mathbf{x})$ is the region of data space that data point $\mathbf{x}$ falls into. When this predictor returns value y as the answer, it contributes to the training error with probability that is approximately $1 - p(y|S)$ when estimated over the entire training set T. This is because that is the fraction of samples in S whose values do not fall into the same bin as y. So the training error for the Gini classifier is the sum of these error probabilities, weighted by the probability that the predictor returns y:

$$\overline{\text{err}}_{\text{Gini}}(S) = \sum_{y \in Y} p(y|S)(1 - p(y|S)) = 1 - \sum_{y \in Y} p^2(y|S) = \text{i}(S) \ .$$

Note that in this interpretation the number of possible values for y is finite even for regressors, because the true distribution $p(y|X_S)$ is replaced with the empirical histogram $p(y|S)$. This limitation could be overcome by other representations for $p(y|X_S)$ that do not resort to bins.

For classifiers, a simpler alternative measure of impurity is

$$\text{i}(S) = \overline{\text{err}}(S) \ ,$$

the training error accrued for the elements in S if this set were no longer split. This measure of impurity is therefore the fraction of labels in S that are different from the label that occurs most frequently in S, since all these labels would be misclassified:

$$\overline{\text{err}}(S) = 1 - \max_y p(y|S) \ .$$

Both the Gini index and the training error are empirical measures of the impurity of the distribution of the training data in set S, in the sense that when and only when all training data in S have the same value (S is "pure") one obtains

$$\overline{\text{err}}(S) = \overline{\text{err}}_{\text{Gini}}(S) = 0 \ ,$$

and the two measures are otherwise positive. The choice of impurity measure depends on the application domain, and it is difficult to give general rules of thumb for which one is better.

With either measure of impurity, the best split is found in practice by cycling over all values of the component index $j \in 1, \ldots, d$ and all possible choices of threshold t in equation (1). The number of thresholds to be tried is finite because the number of training samples and therefore values of x_j is finite as well: If x_j and x'_j are consecutive values for the j-th component of $\mathbf{x}$ among all the samples in T, there is no need to evaluate more than one threshold between x_j and x'_j.

Specifically, one can build a sorted list

$$x_j^{(0)}, \ldots, x_j^{(u_j)}$$

of the $u_j + 1$ unique values of x_j in T and set the thresholds to be tested as

$$t = t_j^{(1)}, \ldots, t_j^{(u_j)} \quad \text{where} \quad t_j^{(\ell)} = \frac{x_j^{(\ell-1)} + x_j^{(\ell)}}{2} \quad \text{for} \quad \ell = 1, \ldots, u_j$$

to maximize the prediction margin.

The function findSplit in Algorithm 2 summarizes the computation of the optimal split. Efficiency can be improved by sorting the values of x_j and updating $|L|$, $|R|$, $\mathrm{i}(R)$, $\mathrm{i}(L)$ and Δ while traversing the list from left to right, rather than computing Δ from scratch at every iteration.

Stopping Criterion. It is dangerous to stop splits when the change in training error falls below some threshold, because a split that seems useless now might lead to good splits later on. Consider for instance the data space

$$X = \{\mathbf{x} \in \mathbb{R}^2 \mid -1 \leq x_1 \leq 1 \text{ and } -1 \leq x_2 \leq 1\}$$

for a classification problem with $K = 2$ classes and true labels

$$y = c_1 \text{ for } x_1 x_2 > 0 \quad \text{and} \quad y = c_2 \text{ for } x_1 x_2 < 0 \; .$$

Splitting on either x_1 or x_2 once does not change the misclassification rate, but splitting twice leads to a good classifier. In other words, neither dimension is predictive by itself, but the two of them together are.

Instead, one typically stops when the impurity of a set is zero, or when splitting a set would result in too few samples in the resulting subsets, or when the tree has reached a maximum depth. See function split? in Algorithm 2.

Label Distribution. The training algorithm places an estimate of the posterior distribution of labels given the data point at each leaf of the prediction tree. This estimate is simply the empirical estimate from the training set. Specifically, for a classifier the set of possible values is Y. For a regressor, the possible values in Y are binned into a predetermined number of bins, indexed by an index y. Either way, if the leaf set S contains N_y samples with index y, the distribution is

$$p(y|S) = \frac{N_y}{|S|} \; .$$

2 Random Decision Forests

Decision trees can represent arbitrarily complex hypothesis spaces $\mathcal{H}$. For instance, one can subdivide the unit interval $[0, 1]$ on the real line into segments of length ϵ for any $\epsilon > 0$ with a deep enough tree that splits each parent segment in half. In multiple dimensions, to subdivide $[0, 1]^d$ into small hypercubes of side ϵ, build a tree that interleaves d interval-splitting trees, one per dimension. One can then assign a separate value to each hypercube, thereby approximating any desired distribution function to any degree.

Because of their expressiveness, decision trees must be curbed lest they overfit. The complexity of individual decision trees is typically controlled by pruning them after expansion [6]. Empirical evidence shows that a better alternative is to use random forest predictors, which combine the predictions of several trees through a voting scheme.

A *random forest* [5] is a predictor that consists of a collection of decision trees $h_m(\mathbf{x})$ for $m = 1, \ldots, M$ that depend on independent identically distributed sets of random parameters. Each tree is trained on a different view of the data, and casts a unit vote for the label (for classification) or value (for regression) associated to input $\mathbf{x}$. Votes are aggregated by a suitable summary function: Majority for classification, mean or median for regression.

Of course, multiple votes would be useless if they all agreed. Because of this consideration, several ways have been proposed and compared to each other [2, 7] to ensure that the votes that different trees cast are independent of each other. These include the following:

Bagging, that is, applying the bootstrap resampling method we saw in an earlier note to train different trees with different bags out of T. The bags are made of training samples drawn independently and uniformly at random from T with replacement, and have the same size as T [4].

Boosting, in which the random subsets of samples are drawn in sequence, each subset is drawn from a distribution that favors samples on which previous predictors in the sequence failed, and predictors vote with a weight proportional to their performance [8].

Arcing, similar to boosting but without the final weighting of votes [3].

Random forests combine bagging with random feature selection for each node of every tree in the forest [1, 5]. Bagging ensures that different trees have different views of the data. Random feature selection means that instead of picking *the best* dimension d on which to split at any given node of any given tree, the dimension d is chosen at random.

The combination has proven very effective in compromising between expressiveness and generalization. In Breiman's words,

i Its accuracy is as good as boosting and sometimes better.

ii It's relatively robust to outliers and noise.

iii It's faster than bagging or boosting alone.

iv It gives useful internal estimates of statistical risk, strength, correlation and variable importance.

v It's simple and easily parallelized.

Algorithm 3 summarizes how to train a random forest and use it for prediction. Using Breiman's training method, the function findSplit we developed for decision trees is modified to pick the feature component index j on which to split *at random*. This random selection is in contrast with the optimal choice of j, that is, the one that maximizes the decrease in impurity. Randomness is preferred over optimality in the splitting rule, since randomness increases the diversity of the trees in the forest and decreases the statistical risk as a consequence. Typical values of M, the number of trees, are in the tens or hundreds, and this hyper-parameter can be optimized by cross-validation. The size $|S|$ of each subset S is equal to the size N of the entire training set.

2.1 Out-of-Bag Estimate of the Statistical Risk

Recall that the goal of machine learning is not to minimize the training risk, but rather the statistical risk of the predictor, that is, the expected loss over samples drawn out of the (unknown) data model $p(\mathbf{x}, y)$. Interestingly, bagging enables a way to estimate the statistical risk of the random forest predictor as follows.

We saw that when drawing a set (or *bag*) B of N samples uniformly at random and with replacement out of the training set T, about 37% of the samples are left out of B on average (and an equal fraction of samples are repetitions). Let now $B_1, \ldots, B_M$ be the M bags used to train the M predictors $h_1, \ldots, h_M$ in a random forest. As mentioned above, each bag contains N samples drawn out of T with replacement, where $N = |T|$. An *out-of-bag predictor* h_{oob} *that works only on training data* can be constructed by letting predictor h_m provide a value for a training sample if and only if the sample is *not* in B_m, and then taking a summary (majority, mean, median) of the vote:

$$\forall \mathbf{x} \text{ such that } (\mathbf{x}, y) \in T\,, \quad h_{\text{oob}}(\mathbf{x}) = \text{summary}(\{h_m(\mathbf{x}) \,|\, m = 1, \ldots, M \text{ and } (\mathbf{x}, y) \notin B_m\})\,.$$

Notation: Let us unpack this expression. Pick a sample $(\mathbf{x}, y)$ out of the training set T. For prediction, of course, we ignore the true label y. To form the prediction $h_{\text{oob}}(\mathbf{x})$, we first ask the trees in the forest to provide their best estimates $\hat{y}$ of the prediction. However, we only accepts values from trees that were trained without using the sample $(\mathbf{x}, y)$. Therefore, h_{oob} lets tree number m provide a value $h_m(\mathbf{x})$ if and only if there is a sample $(\mathbf{x}, y)$ in the training set T for which

$$(\mathbf{x}, y) \notin B_m\,.$$

Finally, we form the prediction $h_{\text{oob}}(\mathbf{x})$ by summarizing the values provided by all participating trees. For data points $\mathbf{x}$ that do not come from T, the value of $h_{\text{oob}}(\mathbf{x})$ is undefined.

The *out-of-bag risk* is then the risk of h_{oob} estimated on T', the set of samples out of T that were not used to train all the trees:

$$T' = \{t \in T \mid \exists\ m \text{ such that } t \notin B_m\}\,.$$

The set T' may be a proper subset of T, because some of the samples in T may show up in all of the bags, and are therefore not included in T'. These omni-present samples are not used to compute the out-of-bag risk, because no tree is allowed to vote on them. The out-of-bag risk is defined as

$$e_{\text{oob}}(h, T') = \frac{1}{|T'|} \sum_{(\mathbf{x},y)\in T'} \ell(y, h_{\text{oob}}(\mathbf{x}))$$

where ℓ is the loss function. This empirical risk can be shown to be an unbiased estimate of the random forest's statistical risk [5].

Algorithm 3 Training a random forest and using it for prediction

function $\phi \leftarrow \text{trainForest}(T, M)$ ▷ M is the desired number of trees
  $\phi =\leftarrow \emptyset$ ▷ The initial forest has no trees
  for $m = 1, \ldots, M$ **do**
    $S \leftarrow$ set of $|T|$ samples drawn uniformly at random out of T with replacement
    $\phi \leftarrow \phi \cup \{\text{trainTree}(S, 0)\}$
  end for
end function

function $\tau \leftarrow \text{trainTree}(S, \text{depth})$
▷ This function is the same as in the Algorithm used to train a decision tree, except that it calls findSplitR instead of findSplit
end function

function $[L, R, j, t] \leftarrow \text{findSplitR}(S)$
▷ *This function replaces* findSplit *in* trainTree *when training a random forest*
  $i_S \leftarrow \text{i}(S)$ ▷ $\text{i}(S)$ is the impurity of S. See text.
  $\Delta_{\text{opt}} \leftarrow -1$ ▷ At the end, Δ_{opt} will be the greatest decrease in impurity.
  $j \leftarrow$ integer drawn uniformly at random out of $\{1, \ldots, d\}$
  for $\ell = 1, \ldots, u_j$ **do** ▷ Loop on all thresholds for dimension j.
    $L \leftarrow \{\mathbf{x} \,|\, x_j \leq t_j^{(\ell)}\}$ ▷ The splitting thresholds $t_j^{(\ell)}$ for $j = 1, \ldots, d$ and $\ell = 1, \ldots, u_j$
    $R \leftarrow S \setminus L$ ▷ are assumed to have been precomputed (see text).
    $\Delta \leftarrow i_S - \frac{|L|}{|S|}\text{i}(L) - \frac{|R|}{|S|}\text{i}(R)$ ▷ See text for a faster way to compute Δ
    if $\Delta > \Delta_{\text{opt}}$ **then**
      $[\Delta_{\text{opt}}, L_{\text{opt}}, R_{\text{opt}}, d_{\text{opt}}, t_{\text{opt}}] \leftarrow [\Delta, L, R, j, t]$
    end if
  end for
  return $[L_{\text{opt}}, R_{\text{opt}}, d_{\text{opt}}, t_{\text{opt}}]$
end function

function $y \leftarrow \text{forestPredict}(\mathbf{x}, \phi, \text{summary})$
  $V = \{\}$ ▷ A set of values, one per tree, initially empty
  for $\tau \in \phi$ **do**
    $y \leftarrow \text{predict}(\mathbf{x}, \tau, \text{summary})$ ▷ The predict function for decision trees
    $V \leftarrow V \cup \{y\}$
  end for
  return $\text{summary}(V)$
end function

References

[1] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. *Neural Computation*, 9:1545–1588, 1997.

[2] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms. *Machine Learning*, 36(1/2):105–139, 1999.

[3] L. Breiman. Arcing classifiers. Technical report, Statistics Department, University of California, Berkeley, CA, 1996.

[4] L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.

[5] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[6] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[7] T. Dietterich. An experimental comparison on three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000.

[8] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *13th International Conference on Machine Learning*, pages 148–156, 1996.

[9] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees in NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.