Random Forest Classifiers

Carlo Tomasi

1 Classification Trees

A *classification tree* represents the probability space $\mathcal{P}$ of posterior probabilities $p(y|\mathbf{x})$ of label given feature by a recursive partition of the feature space X, where each partition is performed by a test on the feature $\mathbf{x}$ called a *split rule*. To each set of the partition is assigned a posterior probability distribution, and $p(y|\mathbf{x})$ for a feature $\mathbf{x} \in X$ is then defined as the probability distribution associated with the set of the partition that contains $\mathbf{x}$.

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

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

where x_d is the d-th component of $\mathbf{x}$ and t is a real number. This note considers only binary classification 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 $p(y|\mathbf{x})$ collected in a vector $\tau.\mathbf{p}$ or an (interior) node with a split function with parameters $\tau.d$ 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. A classification tree τ then takes a new feature $\mathbf{x}$, looks up its posterior in the tree by following splits in τ down to a leaf, and returns the MAP estimate, as summarized in Algorithm 1.

1.1 Training Classification Trees

Optimal training of a classification 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 was proven by a reduction of the set cover problem [9].

Classification trees are typically 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 d 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 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.

[1] The tree is binary, the classifier is not necessarily: A binary classification tree can handle more than two classes.

[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 1 Classification with a tree

```
function y ← treeClassify(x, τ)
    if leaf?(τ) then
        return arg max_y τ.p
    else
        return treeClassify(x, split(x, τ))
    end if
end function

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

This procedure leads to large trees that 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 classifiers, 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 its labels (distribution).

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

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

where

$$\mathrm{i}(S) = \overline{\mathrm{err}}(S)$$

is also called the *impurity* of S. In this expression, $\overline{\mathrm{err}}(S)$ is the error accrued for the elements in S if this set were no longer split, and 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:

$$\overline{\mathrm{err}}(S) = 1 - \max_y p(y|S) \quad \text{where} \quad p(\cdot|S) = \mathrm{distribution}(S)$$

is the empirical distribution of the labels in the set S.

The so-called *Gini index*

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

is often used instead of the training error $\overline{\mathrm{err}}(S)$, where

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

Algorithm 2 Training a classification tree

```
function τ ← trainTree(S, depth)
    if split?(S, depth) then
        [L, R, τ.d, τ.t] ← findSplit(S)
        τ.L ← trainTree(L, depth + 1)
        τ.R ← trainTree(R, depth + 1)
    else
        τ.p ← distribution(S)
    end if
    return τ
end function

function [L, R, d, t] ← findSplit(S)
    i_S ← i(S)                                   ▷ i(S) is the impurity of S. See text.
    Δ_opt ← −1                                   ▷ At the end, Δ_opt will be the greatest decrease in impurity.
    for d = 1, ..., D do                         ▷ Loop on all data dimensions.
        for ℓ = 1, ..., u_d do                   ▷ Loop on all thresholds for dimension d.
            L ← {x | x_d ≤ t_d^(ℓ)}              ▷ The splitting thresholds t_d^(ℓ) for d = 1, ..., N and ℓ = 1, ..., u_d
            R ← S \ L                            ▷ are assumed to have been precomputed (see text).
            Δ ← i_S − (|L|/|S|) i(L) − (|R|/|S|) i(R)   ▷ See text for a faster way to compute Δ
            if Δ > Δ_opt then
                [Δ_opt, L_opt, R_opt, d_opt, t_opt] ← [Δ, L, R, d, t]
            end if
        end for
    end for
    return [L_opt, R_opt, d_opt, t_opt]
end function

function answer ← split?(S, depth)
    return i(S) > 0 and |S| > s_min and depth < d_max   ▷ s_min and d_max are predefined thresholds
end function

function p ← distribution(S)
    p ← [0, ..., 0]                              ▷ A vector of K zeros
    n ← 0
    for (x, y) ∈ S do
        p(y) ← p(y) + 1
        n ← n + 1
    end for
    return p/n
end function
```

is the fraction of training labels in set S that are equal to y. The Gini index minimizes the training error for a decision rule different from the MAP estimate, namely the stochastic rule

$$y = f_{\text{Gini}}(\mathbf{x}) = y \text{ with probability } p(y|\Sigma)$$

where Σ is the region of feature space that feature $\mathbf{x}$ falls into. When this classifier returns class y as the answer, it contributes to the training error with probability $1 - p(y|S)$, because that is the fraction of samples in S that are not in class y. So the training error for the Gini classifier is the sum of these error probabilities, weighted by the probability that the classifier returns y:

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

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 label (S is "pure") one obtains

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

and the two measures are otherwise positive. Empirical evidence shows that the choice of impurity measure is not critical.

With either measure of impurity, the best split is found in practice by cycling over all values of $d \in 1, \ldots, D$ 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_d is finite as well: If x_d and x'_d are consecutive values for the d-th component of $\mathbf{x}$ among all the samples in T, there is no need to evaluate thresholds that are between x_d and x'_d. As a refinement, one can build a sorted list

$$x_d^{(0)}, \ldots, x_d^{(u_d)}$$

of the u_d unique values of x_d in T and set the thresholds to be tested as

$$t = t_d^{(1)}, \ldots, t_d^{(u_d)} \quad \text{where} \quad t_d^{(\ell)} = \frac{x_d^{(\ell-1)} + x_d^{(\ell)}}{2} \quad \text{for} \quad \ell = 1, \ldots, u_d$$

to maximize the classification margin.

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

Stopping Criterion. Stopping splits when the change in training error crosses some threshold is dangerous, because a seemingly useless split might lead to good splits later on. Consider for instance the feature space

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

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 feature 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 Algorithm 2.

Label Distribution. The training algorithm places an estimate of the posterior distribution of labels given the feature at each leaf of the classification tree. This estimate is simply the empirical estimate from the training set. Specifically, if the leaf set S contains N_y samples with label y, the distribution is

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

2 Random Forest Classifiers

Classification trees can represent arbitrarily complex probability spaces $\mathcal{P}$ and therefore hypothesis spaces $\mathcal{H}$. For instance, one can subdivide the unit interval $[0, 1]$ on the real line into segments of length ϵ with a 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 can then assign separate label distributions to each hypercube, thereby approximating any desired distribution function to any degree.

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

A *random forest classifier* [5] is a classifier that consists of a collection of classifier trees $f_m(\mathbf{x})$ for $m = 1, \ldots, M$ that depend on independent identically distributed sets of parameters and each tree casts a unit vote for the most popular class at input $\mathbf{x}$.

Several ways have been proposed and compared to each other [2, 7] to randomize the trees. These include the following:

Bagging, that is, training each tree on a random subset of training samples drawn independently and uniformly at random from T with replacement [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 classifiers in the sequence failed, and classifiers are given a vote weight proportional to their performance [8].

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

A particularly successful form of randomization combines bagging with random feature selection for each node of every tree in the forest [1, 5]. In Breiman's words,

- i Its accuracy is as good as Adaboost and sometimes better.
- ii It's relatively robust to outliers and noise.
- iii It's faster than bagging or boosting.
- iv It gives useful internal estimates of error, strength, correlation and variable importance.
- v It's simple and easily parallelized.

Algorithm 3 summarizes how to train a random forest classifier and use it for classification. Using Breiman's training method, the function $\mathrm{findSplit}$ is modified to pick the feature component index d on which to split at random rather than by maximizing the decrease (2) in the training error. A version of the algorithm in which several features were split upon at each node was found to be only marginally better. Either way, randomness is preferred over optimality in the splitting rule, since randomness increases the diversity of the trees in the forest and decreases the generalization error as a consequence. Typical values of M, the number of trees, are in the tens or hundreds. 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 Generalization Error

Interestingly, bagging enables a way to estimate the generalization error of the random forest classifier as follows.

When drawing a set S of N samples uniformly at random and with replacement out of the training set T, about 37% of the samples are left out of S (and an equal fraction of samples are repetitions). To see this, consider the experiment of drawing a single element out of T. The probability that any one element is drawn is $1/N$, so the probability that it is not drawn is $1 - 1/N$, and the probability that that same element is not drawn in any of the N draws is

$$\left(1 - \frac{1}{N}\right)^N \; .$$

Since all elements in T are treated the same, this expression is also the expected fraction of elements that do not end up in S. Since

$$\lim_{N\to\infty} \left(1 - \frac{1}{N}\right)^N = \frac{1}{e} \approx 0.37 \; ,$$

for large enough N about 37% of the elements of T are not in S. This approximation is good rather soon. For instance, $(1 - 1/24)^{24} \approx 0.36$.

Let now $T_1, \ldots, T_M$ be the M input sets used for the M classifiers $f_1, \ldots, f_M$ in a random forest. An *out-of-bag classifier* f_{oob} that works only on training data can be constructed by letting classifier f_m vote for a training sample if and only if the sample is *not* in T_m:

$$v(y) = \left|\{m \,|\, (\mathbf{x}, y') \notin T_m \text{ and } f_m(\mathbf{x}) = y\}\right|$$

and then taking the majority vote $\arg\max_y v(y)$ as usual. The *out-of-bag error* is then the training error of f_{oob},

$$\frac{1}{N} \sum_{n=1}^{N} I(y_n \neq f_{\text{oob}}(\mathbf{x}_n)) \; ,$$

which can be shown to be an unbiased estimate of the random forest's generalization error [5].

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.

Algorithm 3 Training a random forest and using it for classification

```
function φ ← trainForest(T)
    φ =← ∅
    for m = 1, ..., M do
        S ← set of N samples drawn uniformly at random out of T with replacement
        φ ← φ ∪ {trainTree(S, 0)}
    end for
end function

function τ ← trainTree(S, depth)
        ▷ This function is the same as in Algorithm 2, except that it calls findSplitR instead of findSplit
end function

function [L, R, d, t] ← findSplitR(S)
▷ This function replaces findSplit in trainTree when training a random forest
    i_S ← i(S)                                  ▷ i(S) is the impurity of S. See text.
    Δ_opt ← −1                                  ▷ At the end, Δ_opt will be the greatest decrease in impurity.
    d ← integer drawn uniformly at random out of {1, ..., D}
    for ℓ = 1, ..., u_d do                      ▷ Loop on all thresholds for dimension d.
        L ← {x | x_d ≤ t_d^(ℓ)}                 ▷ The splitting thresholds t_d^(ℓ) for d = 1, ..., N and ℓ = 1, ..., u_d
        R ← S \ L                               ▷ are assumed to have been precomputed (see text).
        Δ ← i_S − (|L|/|S|) i(L) − (|R|/|S|) i(R)   ▷ See text for a faster way to compute Δ
        if Δ > Δ_opt then
            [Δ_opt, L_opt, R_opt, d_opt, t_opt] ← [Δ, L, R, d, t]
        end if
    end for
    return [L_opt, R_opt, d_opt, t_opt]
end function

function y ← forestClassify(x, φ)
    v ← [0, ..., 0]                             ▷ A vector of K votes, initially all zero
    for m = 1, ..., M do
        y ← treeClassify(x, τ_m)
        v(y) ← v(y) + 1
    end for
    return arg max_y v(y)
end function
```

[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.