Random Forests for Supervised Classification

Greg Holste

November 22, 2019

Contents

1 Introduction

Suppose we want to automatically determine whether an email is spam or legitimate. For $N = 4601$ emails, we have recorded whether or not they were spam as well as some quantitative **features** of each email. For example, we recorded the frequency of the word "free", the frequency of the character "!", and even the length of the longest sequence of capital letters. Put more formally, we are interested in learning the relationship between these $p = 57$ features or *predictors* and the categorical response $Y = \{\text{spam, legitimate}\} = \{1, 0\}$ using the data collected below.

	$FreqMake$ (X_1)	$FreqAddress$ (X_2)	$\cdots$	$Freq!$ (X_{56})	$LongestCapSeq$ (X_{57})	$Spam$ (Y)
1	0.00	0.64	$\cdots$	0.778	61	1
2	0.21	0.28	$\cdots$	0.372	101	1
3	0.06	0.00	$\cdots$	0.276	485	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
4599	0.30	0.00	$\cdots$	0.000	6	0
4600	0.96	0.00	$\cdots$	0.000	5	0
4601	0.00	0.00	$\cdots$	0.125	5	0

Table 1: Spam email data, courtesy of the UCI Machine Learning Repository [4]. Variables of the form *Freq*CHARS = "percentage of email made up by the word/character CHARS (not case-sensitive)" and *LongestCapSeq* = "longest uninterrupted sequence of capital letters."

This is a binary **classification** problem, as we are interested in assigning incoming observations (emails) to one of two *discrete* output classes: "spam" or "legitimate." Our *feature vectors* $\boldsymbol{x_1}, \ldots, \boldsymbol{x_{4601}}$ (the rows of Table 1 excluding the last column) live in the universe we will denote X, the cross-product of our 57 features: $X = X_1 \times \cdots \times X_{57}$. Similarly, our outputs or *labels* of interest $y_1, \ldots, y_{4601}$ each live in the universe Y, which is simply the finite set $\{0, 1\}$. The essence of supervised learning is to use these $(\boldsymbol{x}, y)$ pairs to *learn* the true relationship between X and Y. In this paper, I will further develop the supervised classification setting, present a tree-based method for solving problems like the one above, and eventually arrive at random forests – a powerful statistical learning technique that builds upon this tree-based method.

1.1 Supervised Classification

Observe that we can represent our data from the above example in the form

$$\{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_{4061}}, y_{4061})\},$$

where $\{\boldsymbol{x_i}\}_{i=1}^{4061} \in \mathbb{R}^{57}$ and $\{y_i\}_{i=1}^{4061} \in \mathbb{R}$. Then we would say the task of making predictions about y is a **supervised learning** task. This simply means we aim to make predictions from *labeled* data; the idea of "supervision" comes from our having the true outcomes $y_1, \ldots, y_{4061}$ to guide, or supervise, the learning process.

Definition 1.1.1

A **training set** or **learning set** $\mathcal{L}$ is a set of n pairs of p-element feature vectors and scalar labels $\{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$ [3][9].

When fitting a model to our data in this supervised setting, we say that we "train" the model on $\mathcal{L}$ in the hopes that, from it, we can learn the relationship between X and Y. When the output of interest is continuous, the task of making predictions about y is called regression; when the output is discrete, this task is called classification. So if we have all quantitative features, then supervised classification requires a learning set $\mathcal{L}$ with $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n} \in \mathbb{R}^p$ and categorical labels $y_1, \ldots, y_n \in \mathbb{Z}$ usually represented by integers. For the spam example, we represented "spam" as 1 (presence of condition) and "legitimate" as 0 (absence of condition); that is, $Y = \{\text{spam, legitimate}\} = \{1, 0\}$. Then we can formally define a classifier as follows.

Definition 1.1.2

A **classifier** is a function $\varphi : X \mapsto Y$ that is "learned" from the training set $\mathcal{L}$, where

1. X is the p-dimensional "input space" or "feature space"
   - Formally, X is the cross-product of predictors $X_1, \ldots, X_p$
2. Y is a finite set of C output classes [13].

1.2 Classification Performance

Once we have a trained classifier, how do we measure the quality of its predictions? Before actually fitting a model, it is common to randomly partition our N observations into n training observations and $m = N - n$ testing observations. If we use these n training observations to learn the relationship between X and Y, then we use the m testing observations to measure the performance of the model trained on $\mathcal{L}$. The fact that we have "held out" these m observations from the training set enables us to compare our predicted outcomes to the true outcomes for "novel" observations (those not seen during training).

Definition 1.2.1

A **testing set** or **test set** $\mathcal{T}$ is a set of m pairs of p-element feature vectors and scalar labels $\{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_m}, y_m)\}$ [3][9].

There is no "hard-and-fast" rule to finding the proportions of N allotted for training and testing, though in general we would like to train on as much data as possible while leaving a sufficiently large test set. In practice, the proportion of N set aside for training is virtually always over 50%, often higher with particularly large N. For the spam example, let us choose a $75:25$ "train-test split." We can use the Python programming language to read in the spam data and randomly partition the $N = 4061$ observations into the training set $\mathcal{L} = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_{3450}}, y_{3450})\}$ and testing set $\mathcal{T} = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_{1151}}, y_{1151})\}$. Letting $\varphi(\boldsymbol{x}) = \hat{y}$ be a classifier trained on $\mathcal{L}$, we then compare our predictions on the test set $\hat{y}_1, \ldots, \hat{y}_{1151}$ to the true outcomes $y_1, \ldots, y_{1151}$.

Probably the simplest metric of classification performance is **accuracy**, the proportion of testing examples classified correctly. Our model's accuracy on the test set is given by $\frac{1}{m}\sum_{i=1}^{m} I(\hat{y}_i = y_i)$, where $I(\cdot)$ is the indicator function, which takes on the value 1 when the condition $\cdot$ is true and 0 when $\cdot$ is false. We could also call this metric *mean accuracy* since it is the weighted mean of the test accuracies over all output classes, where the weights are the frequencies of each output class in the test set. Letting $|c| = \sum_{i=1}^{m} I(y_i = c)$, this means

$$\begin{aligned}
Accuracy &= \sum_{c \in Y} \frac{|c|}{m} \,(Accuracy \text{ for class c}) \\
&= \sum_{c \in Y} \frac{|c|}{m} \left(\frac{1}{|c|} \sum_{i:y_i = c} I(\hat{y}_i = y_i) \right) \\
&= \sum_{c \in Y} \frac{1}{m} \sum_{i:y_i = c} I(\hat{y}_i = y_i) \\
&= \frac{1}{m} \sum_{i=1}^{m} I(\hat{y}_i = y_i).
\end{aligned}$$

This is important to realize because sometimes mean accuracy alone can be misleading. If a classifier achieves 85% mean accuracy, this does *not* mean the model achieves around 85% accuracy on each class individually; perhaps the model does quite well with most classes, but only correctly classifies 40% of a particularly important class. To illustrate this concept, suppose we use the following simple classifier for the spam data set:

$$\varphi(\boldsymbol{x}) = \begin{cases} 1, & LongestCapSeq > 20 \\ 0, & LongestCapSeq \leq 20. \end{cases}$$

On the test set $\mathcal{T}$ defined above, this model has approximately 73% mean accuracy. This may seem surprisingly good for such a simple, rule-based model; however, we find that it achieves 77.9% accuracy for legitimate emails and only 65.9% accuracy for spam. Put another way, this model would incorrectly mark around 22% of legitimate emails as if they were spam. This single mean accuracy value of 0.73 ignores one of the most important concepts in predictive modeling: *not all errors are equal.* A more nuanced assessment of model performance must consider the *types* of errors made. One way to visualize this is through a **confusion matrix**, where we tabulate each testing example in a $C \times C$ matrix according to its true and predicted labels (Figure 1).

This allows us to visualize which classes are being "confused" for one another (and how frequently). For example, the top left entry of Figure 1 represents the number of test set emails correctly classified as legitimate. The parenthetical value under this count represents the proportion of truly legitimate emails classified as legitimate. This can be interpreted as the accuracy for class 0 or the **true negative** rate. Similarly, the lower right entry represents the number of spam emails correctly classified as spam, and the proportion 0.66 is the **true positive** rate. Note that a perfect classifier (100% mean accuracy) would produce a *diagonal* confusion matrix since every prediction would be a true positive or true negative.

For the spam data, observe that there are only two types of errors that can be made: spam can be misclassified as legitimate and a legitimate email can be marked spam. Neither

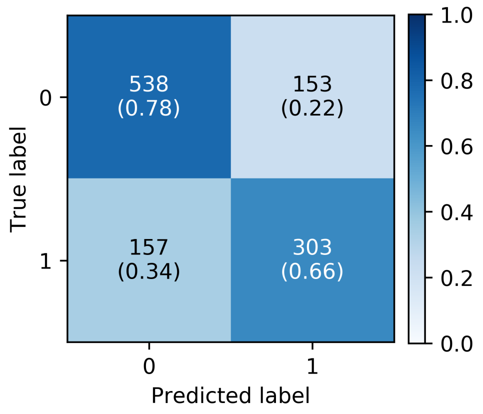


Figure 1: Confusion matrix for single-rule classifier on spam data. Each integer represents the number of test set observations with true and predicted label as specified. The proportions in parentheses are row-wise (out of the number of observations for which the true label is as specified).

error is desirable, but clearly we are more interested in minimizing the latter; that is, we would much rather let in some spam than incorrectly throw out legitimate emails [9]. In other terms, we would like to minimize **false positives**. These are represented in the upper right entry of Figure 1, while **false negatives** are found in the lower left entry. There exist many performance metrics based on the values present in a confusion matrix – such as precision, recall, and F1 score – but we will focus on mean accuracy and false positive rate, as they are most relevant to the spam detection problem.

Lastly, while perhaps not intuitive, we do *not* necessarily want a model that perfectly fits our training data. If we train a model that achieves near-perfect accuracy on the training set, but only 65% accuracy on unseen test data, then we have simply not done our job as responsible statisticians; this is called **overfitting**. Recall that the goal of supervised learning is to use our collected data to learn the true relationship between feature space X and output space Y. If we build a model that does this successfully, then it should perform almost equally well on *any* sufficiently large data set from the X, Y universe (not just our particular set of training data!). Overfitting is typically a symptom of unnecessary model complexity and is a topic we will return to in several places throughout this paper. Now that we understand supervised classification, we can introduce a particular type of tree-based classifier called a decision tree.

2 Decision Trees

2.1 Motivating Tree-Based Methods

To understand the intuition behind decision trees, consider the training data presented below in Figure 2. We have collected two features X_1 and X_2 for $n = 12$ observations, each of which belongs to one of three output classes. Now, if we were to collect a new observation with $X_1 = 1$ and $X_2 = 6$ (equivalently, $\boldsymbol{x_{13}} = (1, 6)$), to what class would you predict it belongs?

If you predicted this new point should belong to class 2 ($\hat{y}_{13} = 2$), not only were you correct, but you just implemented a classification tree!

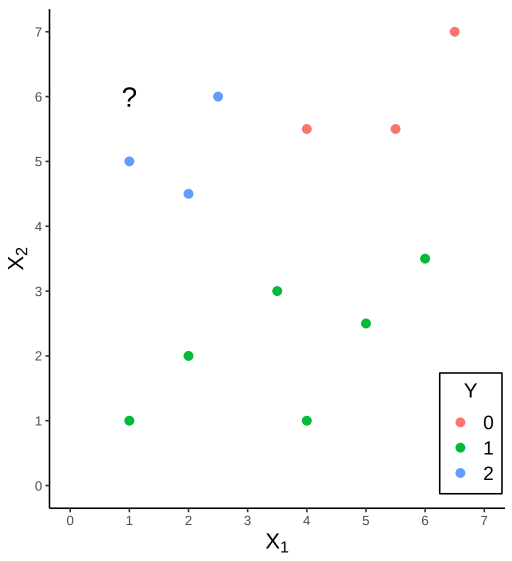


Figure 2: A toy example of a classification setting. "?" represents a new, unlabeled observation.

Even if you didn't consciously make the decisions outlined in Figure 3b to arrive at your prediction, you probably went through a similar process. One of the major benefits of decision trees is their tractability; they are a great example of a "white box" model – we know *exactly* how the model arrives at a prediction. While convenient to illustrate the concept of tree-based methods for classification, this is an unrealistically well-behaved example. How would we construct such a tree for a large ($n \gg 12$), high-dimensional ($p \gg 2$) dataset? How do we best split the data when there is no obvious single boundary between classes? In the following section, we will walk through the Classification and Regression Trees (CART) procedure for growing arbitrarily large decision trees for supervised classification [3].

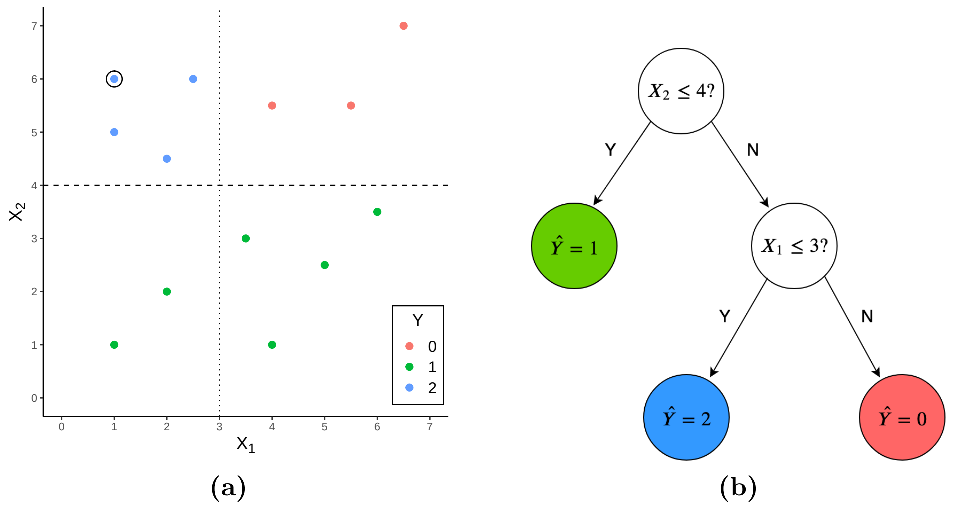


Figure 3: Best binary "splits" for classification (a) and equivalent decision tree representation (b) for data presented in Figure 2. The circled observation in (a) is a new observation that we predict belongs to class 2 (blue).

Broadly, a tree is a type of directed graph in which **nodes** are connected by **edges** (arrows). In the context of a decision tree, each node t represents a subset of the feature space X satisfying some condition(s). For example, the *root node* at the top of Figure 3b contains the entire input space, while its "left child" t_L contains the subset of X satisfying $X_2 \leq 4$. This means we can write $t_L = \{X : X_2 \leq 4\}$; of course, it also follows (and will be useful) that t_L contains all training samples satisfying this condition as well: $\{\mathcal{L} : X_2 \leq 4\} \in t_L$. Lastly, predictions are made at the *terminal nodes*, or "leaves," where there are no more outgoing edges. If a new (unlabeled) observation $\boldsymbol{x}$ falls into a terminal node t, then we predict it belongs to the most frequent class among training observations in node t.

For consistency, we will adhere to the following rules when describing decision trees:

1. Conditions will be phrased in the form $X_j \leq s$ for some $j \in \{1, \ldots, p\}$ and splitting point $s \in \mathbb{R}$.
2. The subset of X satisfying the above condition will be sent to the *left* child of t.
3. Nodes will be labeled $t_{l,h}$, where $l \in \mathbb{Z}_{\geq 0}$ is the node's "level" (from top to bottom) and $h \in \mathbb{N}$ is the node's horizontal position at level l (from left to right).
   - Exception: the root node will always be denoted t_0.
4. For convenience, $|t|$ will represent the number of training samples in node t.

To illustrate these rules, the blue terminal node in Figure 3b would be called $t_{2,1}$. We can see that $t_{2,1}$ contains the subset of X satisfying $X_1 \leq 3$ and $X_2 > 4$, or equivalently,

$$t_{2,1} = \{t_{1,2} : X_1 \leq 3\} = \{t_0 : X_2 > 4, X_1 \leq 3\} = \{X : X_2 > 4, X_1 \leq 3\}.$$

Since, before making our prediction for this new observation, three of our twelve training observations satisfy these conditions, we would also write

$$|\{\mathcal{L} : X_2 > 4, X_1 \leq 3\}| = |t_{2,1}| = 3.$$

2.2 Growing a Tree: Theory

Decision trees are grown by recursively making optimal binary splits in the feature space; these splits are orthogonal partitions of the feature space into two nonempty groups enforced by a condition $X_j \leq s$ for some $j \in \{1, \ldots, p\}$ and $s \in \mathbb{R}$. Put simply, we want to choose the predictor X_j and splitting point s that maximize the quality of a split $X_j \leq s$. In other words, we want to separate output classes as "purely" as possible. It is conventional to frame this as "minimizing the impurity" of a node rather than maximizing its purity, though the two are equivalent.

Definition 2.2.1

The **impurity** $Q(t)$ of a node t measures the *homogeneity* of labels among all training observations in node t [13].

The "purer" a node (the smaller the impurity), the more one label dominates the observed distribution of labels in node t. For example, let $\hat{p}_{t,c}$ be the proportion of training observations in node t that belong to class $c \in Y$; that is,

$$\hat{p}_{t,c} = \frac{1}{|t|} \sum_{i:\boldsymbol{x_i} \in t} I(y_i = c).$$

With this, we can now introduce one of the most popular measures of impurity.

Definition 2.2.2

The **Gini impurity** $Q_G(t)$ of a node t is given by

$$Q_G(t) = \sum_{c \in Y} \hat{p}_{t,c}(1 - \hat{p}_{t,c}) \text{ [7][9].}$$

We see that the term inside the sum is maximized when $\hat{p}_{t,c} = 0.5$ and minimized at $\hat{p}_{t,c} = 0, 1$. This should match our intuition that the most *impure* node would occur when classes are represented equally (meaning we have failed to separate classes) and the purest node would be one in which all samples belong to one class (perfect separation). Note that we do not need to consider the case of $\hat{p}_{t,c} = 0$ for all c, as this would mean we have a terminal node with zero training samples ($|t| = 0$) since we will not allow this to happen.

Equipped with a measure of impurity $Q(t)$, we now want to choose j and s that make the optimal split at node t. We can accomplish this by minimizing the weighted sum of the impurities of the resulting children t_L and t_R; this sum is weighted by $\frac{|t_L|}{|t|}$ and $\frac{|t_R|}{|t|}$, the proportions of training samples sent, respectively, to the left and right child of t. Then the optimal split $X_{j^\star} \leq s^\star$ at node t is given by

$$(j^\star, s^\star) = \min_{j,s} \left\{ \frac{|t_L|}{|t|} Q(t_L) + \frac{|t_R|}{|t|} Q(t_R) \right\}. \tag{1}$$

Now how do we actually solve this optimization problem? It is clear we need to consider all $j = 1, \ldots, p$, but what splitting points s must we consider? It would of course be impossible to query all values of s, and it turns out to not even be necessary to consider all observed values, or *levels*, of $X_1, \ldots, X_p$. The answer is simply to consider all values of s that produce *unique* splits. Letting $\{X_j\}$ be the set of unique observed levels of predictor X_j, we then consider all (j, s) pairs of the form

$$(j, s) \text{ for } j = 1, \ldots, p \text{ and } s \in \{X_j\} \cap t.$$

We could also write that our *parameter space*, or set of possible parameters to consider, is $\{(j, s) : j \in \{1, \ldots, p\}, s \in \{X_j\} \cap t\}$. In words, for a given j, we consider all unique values of X_j falling in node t as potential splitting points. Conveniently, it is often computationally feasible to exhaustively compute our optimization criterion for all such pairs of j and s.[1]

We then solve this same optimization problem for the children of t, and their children, and so on... until some stopping condition. Before we reach the full algorithm for growing

[1]See [11] and [10] for proposed alternatives to exhaustively searching this parameter space.

a decision tree, we lastly must consider when to *stop* growing a tree. Ideally, we would stop when each terminal node is perfectly pure, containing training observations from one class only. This is rarely achievable in practice though, since output classes will often overlap in the feature space; this makes perfect separation of classes difficult and even undesirable (this can cause overfitting). Instead, it is common to pre-specify a minimum terminal node size d, meaning we prohibit a leaf node to contain fewer than d training examples. To incorporate this into the tree-growing process, we simply specify this additional constraint in our optimization problem. Then the optimal split $X_{j^\star} \leq s^\star$ is now given by

$$(j^\star, s^\star) = \min_{j,s} \left\{ \frac{|t_L|}{|t|} Q(t_L) + \frac{|t_R|}{|t|} Q(t_R) \right\}, \text{ subject to } |t_L| \geq d, |t_R| \geq d. \quad (2)$$

Finally, the algorithm in full for growing a decision tree is as follows.

Algorithm 2.2.1 Growing a Decision Tree

```
1:  function DecisionTree($\mathcal{L}$, $d$)
2:      function FindBestSplit($t$)
3:          if $Q(t) = 0$ then                                  ▷ If node is already perfectly pure...
4:              return $\text{argmax}_c\, \hat{p}_{t,c}$        ▷ Predict most frequent (only) class
5:          end if
6:          $(j^\star, s^\star) \leftarrow (None, None)$
7:          $(j^\star, s^\star) \leftarrow \min_{j,s} \left\{ \frac{|t_L|}{|t|} Q(t_L) + \frac{|t_R|}{|t|} Q(t_R) \right\}$, subject to $|t_L| \geq d, |t_R| \geq d$
8:          if $(j^\star, s^\star) = (None, None)$ then          ▷ If reached min. node size...
9:              return $\text{argmax}_c\, \hat{p}_{t,c}$        ▷ Predict most frequent class[2]
10:         else
11:             $t_L, t_R \leftarrow \{\mathcal{L} : X_{j^\star} \leq s^\star\},\ \{\mathcal{L} : X_{j^\star} > s^\star\}$
12:             FindBestSplit($t_L$)
13:             FindBestSplit($t_R$)
14:         end if
15:         return DecisionNode($j^\star$, $s^\star$, $t_L$, $t_R$)   ▷ Root node of trained tree
16:     end function
17:
18:     return FindBestSplit($\mathcal{L}$)
19: end function
```

To fit a decision tree, we would simply call DecisionTree($\mathcal{L}$, d), where $\mathcal{L}$ is our training set and d is the minimum terminal node size. Observe that this algorithm takes advantage of *recursion*: in lines 12 and 13, the function FindBestSplit calls itself for the left and right child of current node t. This gives a chain of recursive function calls such that the returned DecisionNode[3] object in line 15 is the root node, whose attributes t_L and t_R are either a prediction (via lines 4 or 9) or DecisionNode object (via line 15), whose attributes t_L and t_R are either a prediction or DecisionNode object with attributes t_L and t_R, and so on...

[2]In line 9, $\text{argmax}_c\{\cdot\}$ is the value of c for which $\cdot$ is maximized.

[3]While not explicitly defined, DecisionNode is simply an object (or "class" in many programming languages) with attributes $j^\star, s^\star, t_L, t_R$.

As the name Classification *and Regression* Trees suggests, we can also grow a decision tree to predict a continuous response variable. The only modifications we need to make are (1) to choose a different impurity (instead now called a "loss" or "objective" function) and (2) to alter how we make predictions. A common loss function L is mean-squared error (MSE); we then replace $Q(t)$ with $L(t) = \frac{1}{|t|}\sum_{i:\boldsymbol{x_i}\in t}(y_i - \hat{f}(t))^2$, where $\hat{f}(t) = \frac{1}{|t|}\sum_{i:\boldsymbol{x_i}\in t} y_i$ is simply the mean response of all training samples in node t. Similarly, for a terminal node t, instead of predicting the most frequent observed response, we now predict the *mean* observed response $\frac{1}{|t|}\sum_{i:\boldsymbol{x_i}\in t} y_i$.

2.3 Growing a Tree: An Example

Now let us fit a classification tree to the toy data first presented in Figure 2, choosing minimum node size $d = 1$. To make the first split at t_0, we must consider all $\sum_j |\{X_j\}| = 9 + 10 = 19$ possible splits $X_j \leq s$, computing the Gini impurities of their resulting children as well as our optimization criterion. All such values are provided in Table 2a, with the optimal split $X_2 \leq 3.5$ marked in red. While $X_2 \leq 4$ might visually make more sense to us as an initial split, observe that $X_2 \leq 3.5$ provides *exactly* the same split as $X_2 \leq 4$ would (Figure 4a). This should illustrate why it is not necessary to consider $s \notin \{X_j\} \cap t$, or, equivalently, splitting points that are not observed values of X_j falling in node t.

j	s	$Q_G(t_L)$	$Q_G(t_R)$	Criterion
1	1.0	0.500	0.620	0.600
1	2.0	0.500	0.594	0.562
1	2.5	0.480	0.490	0.486
1	3.5	0.500	0.500	0.500
1	4.0	0.594	0.500	0.562
	⋮	⋮	⋮	⋮
2	3.5	0.000	0.500	0.250
2	4.5	0.245	0.480	0.343
2	5.0	0.375	0.375	0.375
2	5.5	0.560	0.500	0.550
2	6.0	0.595	0	0.545
2	7.0	0.625	0	0.625

(a)

j	s	$Q_G(t_L)$	$Q_G(t_R)$	Criterion
1	1.0	0	0.480	0.400
1	2.0	0	0.375	0.250
1	2.5	0	0	0
1	4.0	0.375	0	0.250
1	5.5	0.480	0	0.400
1	6.5	0.500	0	0.500
2	4.5	0	0.480	0.400
2	5.0	0	0.375	0.250
2	5.5	0.500	0.500	0.500
2	6.0	0.480	0	0.400
2	7.0	0.500	0	0.500

(b)

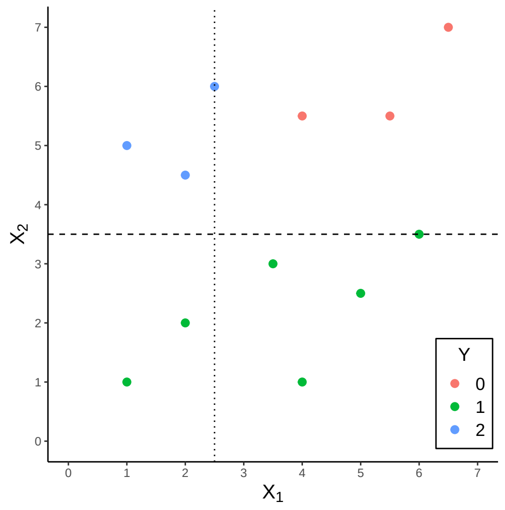
Table 2: Table of Gini impurities and optimization criteria for all unique splits at nodes t_0 (a) and $t_{1,2}$ (b) when fitting a decision tree to the toy data seen in Figure 2. The optimization criterion is the weighted sum defined in Equation 1. Splits (rows) with the minimum observed criterion are presented in red.

Now we would find the best split at $t_{1,1}$, the left child of the root node, except

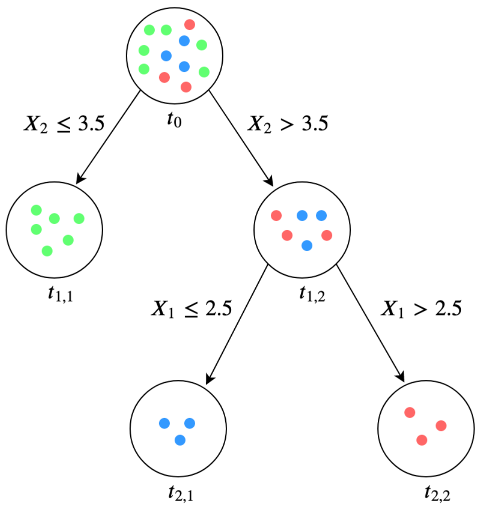
$$Q(t_{1,1}) = \sum_c \hat{p}_{t_{1,1},c}(1 - \hat{p}_{t_{1,1},c}) = 0(1) + 1(0) + 0(1) = 0.$$

This means $t_{1,1}$ is perfectly pure, containing only green ($Y = 1$) training observations, and thus becomes our first terminal node. Now we must find the best split for the right child of the root, $t_{1,2}$. Figure 4b shows clearly that $t_{1,2}$ contains a mixture of red and blue samples, meaning $Q(t_{1,2})$ exceeds 0 – this is an impure node that can be split further. After considering all $\sum_j |\{X_j\} \cap t_{2,1}| = 6 + 5 = 11$ splits, we find that $X_1 \leq 2.5$ provides the best split (see Table 2b). Since the table also shows $Q(t_L) = Q(t_{2,1}) = 0$ and $Q(t_R) = Q(t_{2,2}) = 0$, both $t_{2,1}$ and $t_{2,2}$ become terminal nodes. Since no nonterminal nodes remain, we are finished!

Now armed with a trained classifier $\varphi(\boldsymbol{x})$, we can formally show how in Section 2.1 we predicted $\boldsymbol{x_{13}}$, an observation with $X_1 = 1$ and $X_2 = 6$, would be classified as blue ($\hat{y}_{13} = 2$). Starting at the root, we first consider whether $X_2 \leq 3.5$. Of course, since $X_2 = 6 > 3.5$, we send $\boldsymbol{x_{13}}$ to nonterminal node $t_{1,2}$. Similarly, since $X_1 = 1 < 2.5$, we send $\boldsymbol{x_{13}}$ to $t_{2,1}$, a terminal node. Then we simply predict $\boldsymbol{x_{13}}$ belongs to the most frequent class in $t_{2,1}$; that is, $\varphi(\boldsymbol{x_{13}}) = \text{argmax}_c \, \hat{p}_{t_{2,1},c} = 2$.



(a)



(b)

Figure 4: Optimal binary splits (a) and full decision tree (b) for toy classification example in Figure 2. Training observations belonging in each node are pictured inside the node.

Note that this is a very simple example to illustrate the tree-growing process; we did not even consider the minimum node size $d = 1$, as all terminal nodes were perfectly pure. In practice, it is much more common to reach leaves by the minimum node size constraint than by acheiving zero impurity. Now that we understand the inner workings of decision trees, we can fit a tree to the (more complex) spam data set and consider some shortcomings of such trees for classification.

2.4 Limitations

Specifying a minimum node size of $d = 1$, we can use the Python package sci-kit learn [14] to fit a tree to the training data described in Section 1.2. We find that – amazingly – it correctly classifies all 3,450 training examples (100% training accuracy). However, when applied to the "unseen" test data, the model achieves mean test accuracy of around 90.1%; the resulting confusion matrix can be seen in Figure 5. While 90% accuracy is quite good –

perhaps nearly good enough to reasonably use as a spam filter – our model clearly "caters" to the training set more than it should.

This is a classic example of overfitting. Remember, we do not necessarily want a model that perfectly fits our training data; rather, we seek a classifier that performs just as well on any set of data from our X, Y universe. As mentioned earlier, overfitting is usually the result of an unnecessarily complex model. For example, suppose we are in a simple linear regression (SLR) setting and fit an $(n-1)$-degree polynomial to data that share a moderate positive association. This model would fit the training data extremely well, essentially connecting each observation, but would likely perform poorly when presented with new data. The issue, of course, is that this high-degree polynomial provides an inappropriate amount of complexity, capturing fine-grained patterns in the training data that will not necessarily appear in new sets of data. The decision tree we fit suffers from this exact issue, having 213 terminal nodes and a *depth* of 30 (with levels $l = 0, 1, \ldots, 29$). Perhaps if we fit a less complex ("shallower") tree to the training data, we might produce a classifier that sacrifices training performance, but ultimately *generalizes* better to previously unseen data.

Decision trees are generally prone to overfitting like this without careful tuning of d, which indirectly controls tree depth. In fact, there exist fine-grained methods for controlling tree depth via *cost-complexity pruning*. While we will not explore this topic further, the broad idea is to grow a large decision tree then progressively collapse nodes until we reach a tree that is at an "acceptable" level of both complexity (depth) and predictive power. Another drawback to decision trees is the fact that they can only make splits that are *orthogonal* to the axes. Imagine that the best split is at a 45-degree angle between two dimensions of the feature space; this is a particularly difficult pattern to replicate with only right-angled, single-predictor binary splits. Lastly, the algorithm to grow a decision tree is what is known as a **greedy** algorithm. This means the algorithm arrives at its solution by considering the *locally* optimal choice at each iteration (split). This means that a decision tree is not guaranteed to give a classifier that is *globally* optimal. Contrast this with, say, ordinary least squares (OLS) linear regression, where the desired parameters can be solved for analytically; that is, we can find the exact parameters that provide the best solution globally. In the following sections, we will explore methods to overcome these limitations that prove to be cheaper and more effective than techniques like pruning.

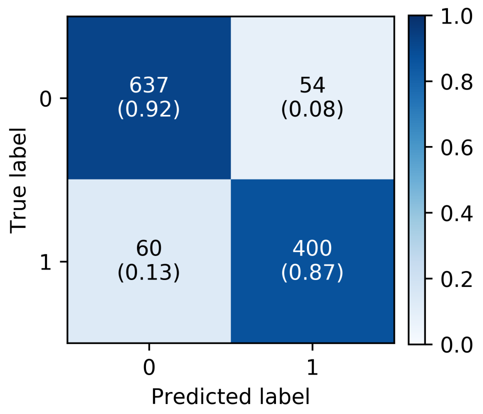
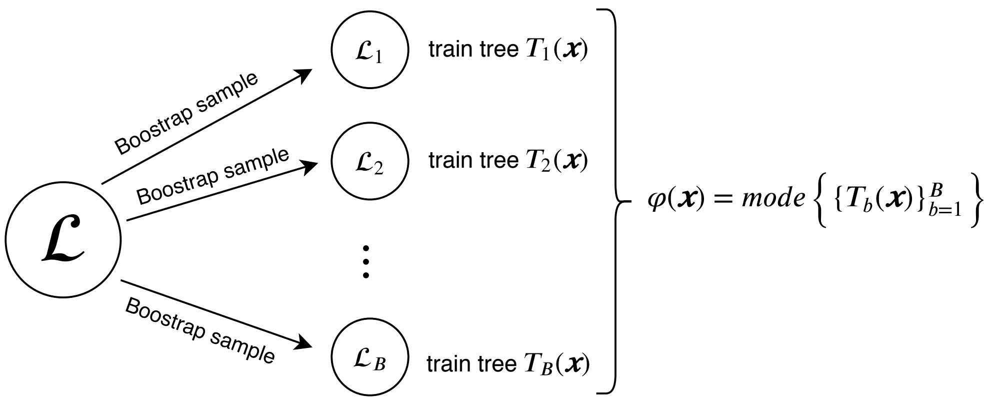


Figure 5: Confusion matrix for decision tree with minimum node size $d = 1$ fit to spam data.

3 Toward Random Forests

In 1979, statistician Bradley Efron first described the **bootstrap** resampling technique [5]. To illustrate the basic idea, suppose we have measured the wingspans of n adults and are interested in finding the (true) population mean wingspan. From our sample, we can calculate an estimate of this population mean – the average of our n observed wingspans – but to actually describe the underlying population, we need a measure of the *variability* of our sample mean. Efron proposed we take a random sample *with replacement* of size n from our initial sample. We then calculate the observed mean of this "bootstrap sample" and repeat this process hundreds or thousands of times so that we produce a distribution of observed means. This allows us to reason about the variability of the sample mean, and would enable us to construct a confidence interval for the population mean. More broadly, the boostrap allows us to estimate properties of any statistic by *resampling* our sample.[4]

We can co-opt this idea with a method called ***b**ootstrap **agg**regat**ing*** (*bagging*) [1], first described by Leo Breiman, the inventor of CART. What if we generated many bootstrapped training sets and somehow combined or "aggregated" predictions made from each training set?



$\mathcal{L}$
Boostrap sample
Boostrap sample
Boostrap sample
$\mathcal{L}_1$
$\mathcal{L}_2$
$\mathcal{L}_B$
train tree $T_1(\boldsymbol{x})$
train tree $T_2(\boldsymbol{x})$
train tree $T_B(\boldsymbol{x})$
$\varphi(\boldsymbol{x}) = mode\left\{ \{T_b(\boldsymbol{x})\}_{b=1}^{B} \right\}$

Figure 6: Diagram of bagging B classification trees to form a single classifier φ. Bootstrap samples are random samples of size $n = |\mathcal{L}|$ with replacement, so they may contain repeat observations.

3.1 Bagging Classifiers

Just as we took bootstrap samples from our initial collection of wingspans, we can take bootstrap samples of our training set $\mathcal{L}$ to form B bootstrapped training sets $\mathcal{L}_1, \ldots, \mathcal{L}_B$. While before we were interested in estimating the population mean wingspan, we are now interested in predicting the true outcome (e.g., spam *vs.* legitimate). Then instead of taking the observed mean of each bootstrap sample, let us fit a decision tree $T_i(\boldsymbol{x})$ to each training set $\mathcal{L}_i$ for all $i \in \{1, \ldots, B\}$. Now that we have a collection, or *ensemble*, of classification trees, how do we aggregate these B classifiers to make a single prediction for some $\boldsymbol{x}$?

[4]This works so long as we assume our sample is representative of the population it came from.

Just as we considered the most frequent class in a terminal node to be our predicted output, we now consider the most frequent class prediction among our B trees to be our predicted output. As summarized in Figure 6, we unify our ensemble of trees into the single *bagged classifier* $\varphi(\boldsymbol{x}) = mode\left\{\{T_b\}_{b=1}^{B}\right\}$. Observe that the number of trees, B, is another value we must pre-specify (in addition to minimum node size d for each tree). Like when choosing n and m in Section 1.2, there is no "default" number of trees, though a typical choice of B is at least 100; we will see, however, that in general performance improves as B increases. We can now apply this very powerful technique to the spam data set.

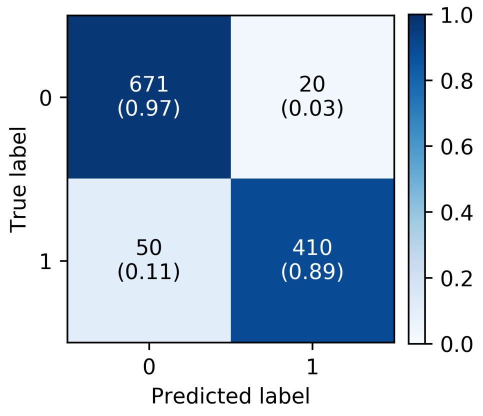


Figure 7: Confusion matrix for classifier resulting from bagging $B = 500$ decision trees, each with $d = 1$, fit to the spam data.

Choosing $B = 500$ and $d = 1$ gives a bagged classifier that achieves (again) 100% accuracy on the training set $\mathcal{L}$ and 93.9% mean accuracy on the test set $\mathcal{T}$. While we are still overfitting the training data, we have notably improved our test accuracy from the 90.1% achieved by a single tree.[5] Perhaps more important than the increase in mean accuracy is the decrease in false positive rate, dropping from about 8% with one tree to $\frac{20}{20+671} \approx 2.89\%$ upon bagging 500 trees (Figure 7). Impressively, this model incorrectly lets in only around 3% of spam.

It is important to note that bagging is a nondeterministic process due to the randomness of boostrapping; the results we obtained are predicated on the particular sequence of B random samples from $\mathcal{L}$. Typically we would fit a bagged classifier multiple times and report summary statistics for our metrics of interest. For instance, if we fit our same bagged classifier $(B = 500, d = 1)$ thirty times, we find a mean test accuracy of 0.9397 with a standard deviation of 0.0012 and a mean false positive rate of 0.0309 with a standard deviation of 0.00095.[6]

The bootstrapped sets $\mathcal{L}_1, \ldots, \mathcal{L}_B$ serve as replicate training sets – cheaper surrogates for collecting entirely new training sets. This allows each classifier (decision tree) to see a slightly different "version" of the original $\mathcal{L}$. Even though each $\mathcal{L}_i, i = 1, \ldots, B$ likely contains repeated observations from $\mathcal{L}$, this is still better than if we were to train B classifiers on the

[5]Also, this jump in performance comes at little computational cost – it took about 14 seconds to train this bagged classifier in Python.

[6]Given the small variability estimates, though, we can see that a single "run" is usually indicative of average model performance.

same training set $\mathcal{L}$; of course this would be a silly approach, yielding B identical (perfectly correlated) classifiers trained on B identical training sets. However, this issue of correlation between trees actually persists since all our bootstrapped training sets are ultimately pulled from the same $\mathcal{L}$; intuitively, classifiers trained on similar training sets $\mathcal{L}_1, \ldots, \mathcal{L}_B$ may make many of the same splits. What if we could take an extra step to reduce the correlation between these trees?

3.2 Random Feature Selection

A **random forest** is an ensemble of *de-correlated* decision trees. What distinguishes random forests from bagging is the addition of *random feature selection* at each split. Going back to Figure 6, let us replace each tree $T_i(\boldsymbol{x})$ with the modified $T_i^\star(\boldsymbol{x})$. This modified tree is grown exactly like $T_i(\boldsymbol{x})$ with the additional step of randomly selecting $k < p$ predictors to consider at each split and proceeding as usual. That is, at each split, we simply consider a smaller parameter space, or set of possible solutions, $\{(j, s) \mid j \in \Theta, s \in \{X_j\} \cap t\}$, where Θ is a set of k integers randomly sampled (*without* replacement) from $\{1, \ldots, p\}$.

This extra injection of randomness into the tree-growing process controls the correlation between trees by preventing them from making the same splits. Observe that bagging can be interpreted as a random forest with $k = p$. In this case, even with slightly different training sets, perhaps every bagged tree makes the same initial split based on the most important predictor; in the case of random forests, with $k < p$, this possibility is made less probable by the new chance that said important variable is left out at one or more splits.

Breiman suggests a default value of $k = \lfloor \sqrt{p} \rfloor$ for classification [2], but this becomes another pre-specified parameter to be tuned. Now we can finally fit a random forest to the spam data set. For the sake of comparison with bagging, let us choose $B = 500$ trees and minimum node size $d = 1$ for each tree; let us also choose, for now, the default value of $k = \lfloor \sqrt{57} \rfloor = 7$. Again using the sci-kit learn Python package to fit this model to the training data, we produce a classifier with test set results seen in Figure 8.

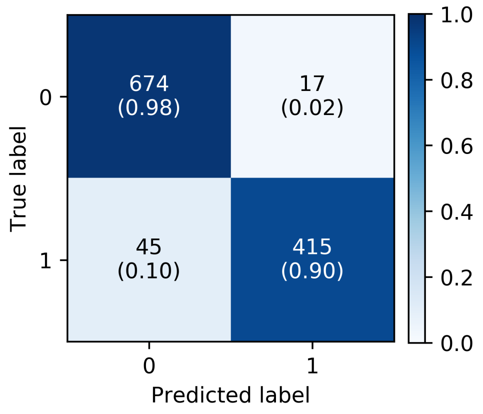


Figure 8: Confusion matrix for random forest of $B = 500$ decision trees, each with $d = 1$, and the default $k = 7$ fit to the spam data.

We find that our random forest acheives around 94.6% mean accuracy on the test set, a slight improvement over bagging at 93.9%. Again we see a drop in false positive rate, arguably our primary goal in this task, down to $\frac{17}{17+674} \approx 2.46\%$. On average, for every 100 legitimate emails, we could expect this model to incorrectly mark only 2-3 of them as spam. For a mostly automatic method with little computational cost,[7] we have constructed a remarkably effective spam filter.

3.3 Why Random Forests Work

We can of course also bag regression trees, meaning we can construct a random forest of regression trees as well. To do this, we only need to modify how we aggregate the B regression trees. Again, instead of taking the most frequent prediction among the B trees to be our predicted value, we take the mean of the B predictions to be our single predicted value; that is, $\varphi(\boldsymbol{x}) = \frac{1}{B}\sum_{i=1}^{B} T_i(\boldsymbol{x})$. Broadly, bagging regression trees improves generalization accuracy by reducing the variance of our predictions. Individual trees are extremely flexible (so much so that they can perfectly fit a 57-dimensional data set such as the spam example); this also, however, means that they are very sensitive to small changes in the data. As we will see, averaging the predictions of many trees serves to stabilize this variability.

Observe that our B bootstrapped training sets $\mathcal{L}_1, \ldots, \mathcal{L}_B$ come from the same sampling distribution; that is, they are identically distributed according to some continuous distribution with mean μ and variance σ^2. We can then show that the variance of a random forest of regression trees, as well as bagged regression trees, is $V(\varphi) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$, where $\rho \in (0, 1]$ is the correlation coefficient between two trees in the forest (see Appendix 7.1 for proof). Now observe that

$$\lim_{B\to\infty}\left(\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2\right) = \rho\sigma^2. \tag{3}$$

Assuming $\rho \neq 1$ (since $\rho = 1$ only when we have B identical regression trees), we have the strict inequality

$$\rho\sigma^2 < \sigma^2\left(\rho + \frac{1-\rho}{B}\right) \implies \rho < \rho + \frac{1-\rho}{B} \implies 0 < \frac{1-\rho}{B},$$

as both numerator and denominator are positive. We have then shown that $(\lim_{B\to\infty} V(\varphi)) < V(\varphi)$. In other words, asymptotically, bagging many regression trees reduces the variance of our predictions! Generally speaking, the more trees we bag, the less variable (and thus typically the more accurate) our predictions.

Now let us return to the Equation 3, which states that $\lim_{B\to\infty} V(\varphi) = \rho\sigma^2$. This means that, asymptotically, the variance of our random forest prediction is solely a function of the correlation between its trees. While bagging reduces variance by virtually eliminating the second term of $V(\varphi)$, random forests even *further* reduce variance by directly reducing the between-tree correlation ρ. Intuitively, when we randomly select a subset of k features to consider at each split, we lessen the chance that different trees make the *same* split according to some particularly important feature. This practice of randomly selecting $k < p$ features at each split practically guarantees that we reduce the correlation between trees compared to if we used $k = p$ (as in bagging).

[7] It took about 1.6 seconds in Python to fit this model to the training data and about 0.02 seconds to make a prediction for a single new email.

We can see this in practice by comparing model performance of bagged classifiers *vs.* random forests for increasing values of B. Running this comparison again for the spam data, we indeed see that once B is sufficiently large (here around $B = 200$ trees), random forests consistently outperform bagged classifiers (Figure 9). This difference in performance must then be attributed to the extra randomness introduced by random feature selection, as it is the only methodological difference between bagging and random forests.

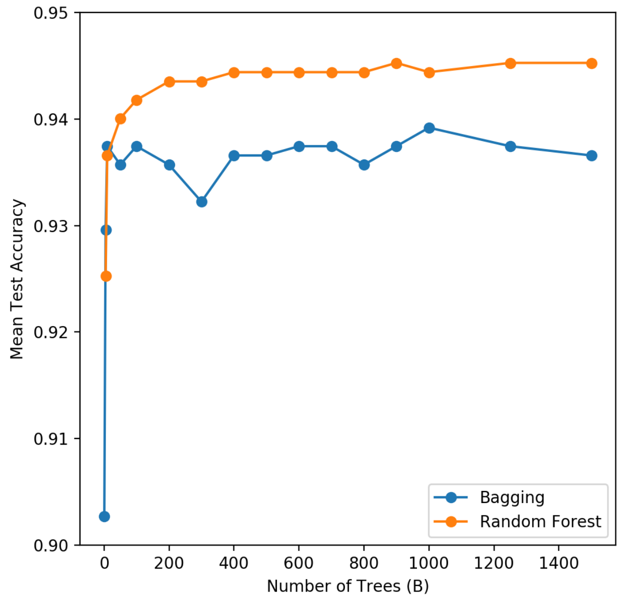


Figure 9: Test accuracy by number of trees B for a bagged classifier and random forest model ($k = 7$) fit to the spam data. All trees were fit with minimum node size $d = 1$. The left-most point is a single classification tree (a bagged classifier of $B = 1$ tree) for comparison.

While we only showed that random forests of *regression* trees (and bagged regression trees) work by reducing the variance of predictions, the same is true for random forests of classification trees. This will go without proof here since it is more difficult to show simply because the expression for the variance of our ensembled predictions becomes very messy. However, the argument and conclusions are identical to those presented above: (1) aggregating many trees – for regression or classification – reduces the variance of our predictions, and (2) injecting randomness into the tree-growing process further reduces variance.

Geurts *et al.* [8] take the variance-reducing approach of random forests to its logical extreme with the Extremely Randomized Trees ("Extra-Trees") algorithm. This procedure is essentially a random forest that *additionally* randomly selects splitting points. To make a split at some node t with the Extra-Trees algorithm, we randomly select $k < p$ predictors $\{X_\theta\}_{\theta\in\Theta}$ like in a random forest. We then randomly select k splitting points $\{s_\theta\}_{\theta\in\Theta}$, where each s_θ is a random point drawn uniformly from the interval $[\min\{X_\theta \cap t\},\ \max\{X_\theta \cap t\}]$; put another way, at each split, we consider the parameter space $\{(j, s) : j \in \Theta, s \in \{s_\theta\}_{\theta\in\Theta}\}$. This even further reduces the correlation between trees by drastically lowering the probability that two trees make the same split, as both predictors *and* splitting points are randomly selected. Results when applying Extra-Trees to the spam data can be seen in Appendix 7.2.

4 Details of Applying Random Forests

4.1 Hyperparameter Tuning

While we have written $\varphi(\boldsymbol{x})$ to refer to a trained classifier, we might technically represent a random forest as $\varphi(\boldsymbol{x}; B; k; d)$. That is, a random forest does not depend upon an input alone; it also depends on the number of trees (B), the number of features to consider at each split (k), and the minimum node size of an individual tree (d).

Definition 4.1.1

A **hyperparameter** or **meta-parameter** of a model or algorithm is a parameter that is not *learned* by the model and instead must be pre-specified.

The values of B, k, and d are examples of hyperparameters – the knobs we can dial to squeeze out extra performance and arrive at a final model. The process of choosing these parameters is sometimes called *hyperparameter tuning*, and there are many approaches ranging from automatic algorithms to manual trial-and-error. One tuning approach, which lies somewhere in the middle of this spectrum, is a **grid search with cross-validation**.

A *grid search* involves first creating a parameter grid. We create a list of values we want each hyperparameter to take on and exhaust the grid; that is, we fit a random forest according to every combination of parameters on the grid. This can be a very computationally intensive task, so it is common to pick only a few "reasonable" values of each hyperparameter to test. For the spam example, let us consider values of $B = (100, 250, 500, 1000, 2000)$, $k = (1, \dots, 7)$, and $d = (1, 2, 3)$. This means our parameter grid specifies $7 \cdot 5 \cdot 3 = 105$ unique combinations of parameters, or unique random forest models, to consider.

The second component to this process is *cross-validation*, or *K-fold cross-validation*, a more general technique to estimate generalization error by making clever use of the training data. We first divide the training set into K folds (groups) $F_1, \dots, F_K$. Then, for each fold, we fit a random forest to the training set *excluding* that fold and evaluate the model on that fold; symbolically, for $i \in \{1, \dots, K\}$, we fit a random forest to $\mathcal{L} \setminus F_i$ and evaluate this model's predictions on F_i. This technique provides K estimates of any performance metric of interest for a given model. More broadly, this approach allows us to obtain a sense of variability in the performance metric(s) we use to ultimately settle on a final model.

To illustrate this process, let us perform a grid search with 10-fold cross-validation using the parameter grid specified above. Observe that we must fit $105 \cdot 10 = 1050$ random forest models, since we must train each model 10 times; this is why even a small parameter grid can be computationally expensive to exhaust. Again using sci-kit learn, we find that the model with the highest mean validation accuracy uses hyperparameters $B = 1000$, $k = 2$, and $d = 1$; this model, $\varphi(\boldsymbol{x}; B = 1000; k = 2; d = 1)$, achieved 95.9% mean accuracy across the $K = 10$ folds during cross-validation with a standard deviation of 0.0123. We then proceed to apply this tuned model to the test set, on which it achieves 94.9% mean accuracy (see Figure 10 for confusion matrix). Note that this model *very narrowly* outperforms our original random forest (94.6% accuracy) with manually chosen $B = 500$, $k = 7$, and $d = 1$. So was all that computing time even worth it?

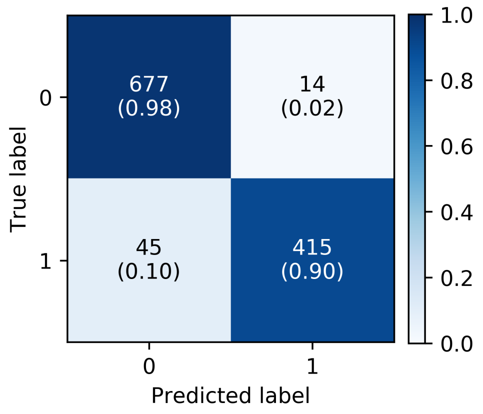


Figure 10: Confusion matrix for tuned random forest with $B = 1000$, $k = 2$, and $d = 1$. Hyperparamters were found by an exhaustive grid search of 105 models with 10-fold cross-validation.

In general, tuning procedures do not see substantial gains in overall performance over manually choosing reasonable hyperparameters. Instead, the purpose of such procedures is to serve as a model selection technique, providing justification for parameter choices. Lastly, note that we do *not* choose our model based on how it performs on the test set. The purpose of the test set is to serve as an "independent" set of data on which we evaluate our final model. In a perfect world, we would collect three large, independent data sets: one for training, one for testing, and one for model validation; cross-validation is simply a cheaper way to validate our model – *before* testing – in lieu of setting aside a validation set.

4.2 Feature Importance Values

While we saw that individual decision trees can be alluringly tractible, it may seem as though we lose that interpretability when combining potentially thousands of trees. It is certainly true that bagged trees and random forests are more of a "black box" than a single decision tree; it is easier to understand the inner workings of an individual tree than the composite of thousands. That being said, we have strategies to extract information about how a random forest model arrives at its predictions, primarily by aiming to quantify **feature importance**. While a random forest prohibits us from visualizing a convenient sequence of rules to arrive at a prediction, we can still ask the question "Which features are most important?"

One approach to answering this question is to accumulate, for each predictor, the *improvement in split-criterion* (impurity or loss) over all splits and all trees in the forest. For a random forest of classification trees, this would mean keeping p running sums $\mathcal{I}_1, \ldots, \mathcal{I}_p$ throughout the entire random forest-fitting process. If we make a split $X_j \leq s$ for some $j \in \{1, \ldots, p\}$ at some node t, then we would add $(Q(t) - Q(t_L)) + (Q(t) - Q(t_R)) = 2Q(t) - Q(t_L) - Q(t_R)$ to $\mathcal{I}_j$ and continue [9]. This gives an assessment of both how frequently each predictor is used in a split and how fruitful those splits are. For example, feature importance values for the random forest fit to the spam data (as in Figure 8) can be seen below (Figure 11).

We can see that the five most "important" features according to cumulative Gini gain are the frequency of "!", frequency of "remove", frequency of "$", frequency of "free", and the average length of uninterrupted sequences of capital letters. This intuitively makes sense

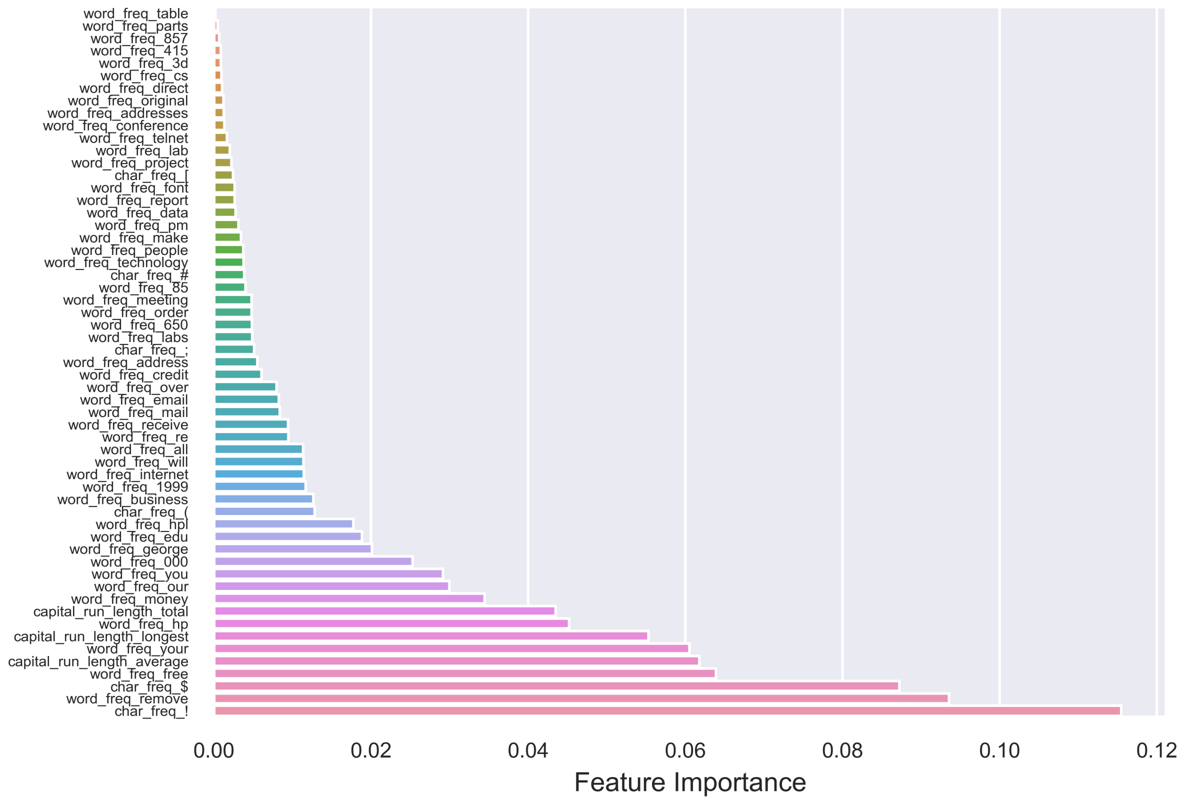


Figure 11: Feature importance values for random forest fit to spam data with $B = 500$, $k = 7$, and $d = 1$. Importance values were "normed" so that they sum to 1.

that these qualities are associated with spam; spam is designed to be attention-grabbing (via capital letters and exclamation marks) and is usually financially motivated (hence, dollar signs and the word "free"). It also should match our intuition that the frequency of unmistakably neutral words such as "table" and "parts" are not very important to predicting whether an email is spam.

4.3 Categorical Features

One of the qualities that makes CART and random forests so popular is the ability to handle heterogeneous data with no distributional assumptions. So far we have only considered the case of continuous features, but CART (and thus bagging and random forests) can also accomodate categorical predictors. If a discrete feature has q unordered levels, then there exist $2^{q-1} - 1$ binary splits; that is, there are $2^{q-1} - 1$ unique ways to split q levels into two nonempty groups.[8] For categorical features with many levels, it can become computationally infeasible to consider all possible splits; for example, a predictor with only $q = 10$ levels would require us to consider 511 possible splits [9]. However, in the case of a binary response (such

[8] This value – the number of ways to partition a q-element set into k nonempty groups – is called a *Stirling number of the second kind* (denoted $S(q,k)$ or $\left\{ {q \atop k} \right\}$). We could then say there are $\left\{ {q \atop 2} \right\} = 2^{q-1} - 1$ unique binary splits of a categorical feature with q levels.

as the spam example), we can order the q levels of our categorical predictor by increasing proportion of associated labels falling into class 1 (say, spam). We can then proceed as if this ranking were a continuous feature, considering all q splits of the form $X_j \leq s$ for all $s \in \{1, \ldots, q\}$. It can be shown that this process gives the optimal split among all possible $2^{q-1} - 1$ splits when growing a classification *or* regression tree.[9]

In a regression setting, it is common to incorporate a categorical predictor by creating $q - 1$ binary indicator variables – one for each level minus a redundant "pivot" level – and simply using these indicators as features. While it may be tempting to do the same when fitting a decision tree with categorical features, this approach is actually misguided. When a discrete feature is truly binary ($q = 2$), it is of course reasonable to represent it with a binary indicator (note that such a split would always be of the form $X_j \leq 0$ for some $j \in \{1, \ldots, p\}$). However, for a categorical feature with $q > 2$, *every* split involving an indicator X_j for some level of this feature would partition the data into 1 level of X_j and the other $q - 1$ levels of X_j. What if a better split is provided by dividing the data into a particular set of 4 levels and the other $q-4$ levels? This is the issue with encoding a multi-level categorical feature as many indicator variables: it means CART will only consider a fraction of possible partitions of its levels. Thus, the procedure outlined in the previous paragraph is preferred.

4.4 Missing Data & Distributional Assumptions

As mentioned briefly in the previous section (and implicitly by omission throughout this paper), decision trees do not require us to make any distributional assumptions about our data. This is a remarkable freedom when compared with more traditional methods for classification; for example, linear discriminant analysis assumes the data are multivariate normal [7] and naïve Bayes classifiers assume features are mutually independent for a fixed level of the outcome [9]. This distribution-free property of CART, which will go without proof here, of course also holds true for bagging, random forests, and Extra-Trees.

Additionally, decision trees – unlike many traditional methods – automatically handle missing values. Since we only consider observed levels as potential splitting points, we simply proceed with the data we have! Methods such as logistic regression, a technique for binary classification, would require us to either discard observations with missing variables or somehow *impute* (fill in) the missing data before training. Fitting a tree to incomplete data can, however, introduce bias against predictors with many missing values. Predictors with fewer levels present (thus fewer potential splitting points) are simply less likely to be chosen as splitting variables.[10] A better approach is to proceed normally, but for every split find one or more *surrogate splits* – essentially "backup" splits for when an incoming observation is missing the splitting variable required to make a prediction.[11] In summary, if we have many missing values, then we should find surrogate splits to potentially use during prediction; otherwise, CART comfortably handles low-to-moderate amounts of missing data.

[9]See [3] for a proof when the outcome is binary and [6] when the output is continuous. Additionally, an approximate procedure for multi-level outcomes is proposed in [12].

[10]See [11] and [10] for splitting methods designed to overcome this bias.

[11]See [15] for details on finding surrogate splits. Essentially they are chosen to mimic the original split $X_j \leq s$ by finding splits involving *different* predictors that best separate the categories $X_j \leq s$ and $X_j > s$ instead of the original categories of Y.

5 Concluding Remarks

In *Elements of Statistical Learning*, Robert Tibshirani[12] *et al.* write that decision trees "come closest meeting the requirements for serving as an off-the-shelf procedure for data mining" [9]. Here, "off-the-shelf" refers to a method that can be applied more or less immediately to data without time-consuming pre-processing or transformation. They explain – as we have seen throughout this paper – that decision trees are fast to build and intepretable (especially when small), they incorporate a mixture of continuous and discrete data, they do not require distributional assumptions about the data, and they easily handle missing values. While not mentioned in this paper, they also cite that decision trees automatically perform feature selection and are invariant under most transformations of predictors. The one area in which decision trees often fall short is predictive accuracy, which is where bagging and random forests pick up the slack. These ensemble methods overcome the general inaccuracy of individual trees "while [maintaining] most of their desirable properties" [9], making random forests one of the most powerful and interpretable "off-the-self" learning techniques.

6 References

[1] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[2] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[3] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees.* Wadsworth, 1984.

[4] Dheeru Dua and Casey Graff. UCI Machine Learning Repository, 2019.

[5] Bradley Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[6] Walter D Fisher. On Grouping for Maximum Homogeneity. *Journal of the American Statistical Association*, 53(284):789–798, 1958.

[7] James Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning.* Springer Series in Statistics. Springer, New York, 2009.

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely Randomized Trees. *Machine Learning*, 63(1):3–42, 2006.

[9] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition.* Springer Series in Statistics. Springer, New York, 2009.

[12] Tibshirani was an advisee of Bradley Efron, inventor of the bootstrap.

[10] Torsten Hothorn, Kurt Hornik, and Achim Zeileis. Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, 2006.

[11] Wei-Yin Loh and Yu-Shan Shih. Split Selection Methods for Classification Trees. *Statistica Sinica*, pages 815–840, 1997.

[12] Wei-Yin Loh and Nunta Vanichsetakul. Tree-Structured Classification via Generalized Discriminant Analysis. *Journal of the American Statistical Association*, 83(403):715–725, 1988.

[13] Gilles Louppe. *Understanding Random Forests: From Theory to Practice.* PhD thesis, University of Liege, 2014.

[14] Fabian Pedregosa et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[15] Terry M Therneau and Elizabeth J Atkinson. An Introduction to Recursive Partitioning Using the RPART Routines, 1997.

[16] John L Weatherwax and David Epstein. A Solution Manual and Notes for: The Elements of Statistical Learning by Jerome Friedman, Trevor Hastie, and Robert Tibshirani, 2013.

7 Appendix

7.1 Variance of Bagged Regression Trees

Since each bootstrapped training set is drawn from the same set $\mathcal{L}$, each regression tree $T_i(\boldsymbol{x}) \sim F$ for all $i = 1, \dots, B$, where F is a shared continuous distribution with mean μ and variance σ^2. Note that these trees are *not* i.i.d (independent and identically distributed); they are only identically distributed. This means, for some $i, j \in \{1, \dots, B\}$, trees T_i and T_j are correlated with correlation coefficient ρ.

So far we know $E(T_i) = \mu$ for all $i = 1, \dots, B$ and, by the definition of correlation coefficient,

$$\rho = \frac{1}{\sigma^2} E\left[(T_i - \mu)(T_j - \mu)\right].$$

This expression also gives

$$\sigma^2\rho = E\left[T_iT_j - \mu T_i - \mu T_j + \mu^2\right] \implies E[T_iT_j] = \sigma^2\rho + E[\mu T_i] + E[\mu T_j] - E[\mu^2],$$

meaning

$$\begin{aligned} E[T_iT_j] &= \sigma^2\rho + \mu E[T_i] + \mu E[T_j] - \mu^2 \\ &= \sigma^2\rho + \mu(\mu) + \mu(\mu) - \mu^2 \\ &= \rho\sigma^2 + \mu^2 = \begin{cases} \sigma^2 + \mu^2, & i = j \\ \rho\sigma^2 + \mu^2, & i \neq j \end{cases}. \end{aligned}$$

Suppose φ is a model that results from bagging these B regression trees. Then we can write

$$\begin{aligned} V(\varphi) = V\left(\frac{1}{B}\sum_{i=1}^{B} T_i\right) &= \frac{1}{B^2} V\left(\sum_{i=1}^{B} T_i\right) \\ &= \frac{1}{B^2}\left[E\left[\left(\sum_i T_i\right)^2\right] - \left[E\left[\sum_i T_i\right]\right]^2\right]. \end{aligned}$$

The second term in the largest brackets above is given by

$$\left[E\left[\sum_i T_i\right]\right]^2 = \left[\sum_i E\left[T_i\right]\right]^2 = \left(\sum_i \mu\right)^2 = (B\mu)^2 = B^2\mu^2.$$

The first term is a little more involved to unpack; first note that

$$\left(\sum_i T_i\right)^2 = \sum_{i=1}^{B}\sum_{j=1}^{B} T_i T_j = \sum_{i=1}^{B} T_i^2 + \sum_{i\neq j} T_i T_j.$$

Then

$$\begin{aligned} E\left[\left(\sum_i T_i\right)^2\right] &= E\left[\sum_{i=1}^{B} T_i^2\right] + E\left[\sum_{i\neq j} T_i T_j\right] \\ &= E[BT_i^2] + E[(B^2 - B)T_iT_j] \\ &= BE[T_i^2] + (B^2 - B)E[T_iT_j] \\ &= B(\sigma^2 + \mu^2) + (B^2 - B)(\rho\sigma^2 + \mu^2) \\ &= B\sigma^2 + B\mu^2 + B^2\rho\sigma^2 + B^2\mu^2 - B\rho\sigma^2 - B\mu^2 \\ &= B\sigma^2 + B^2\rho\sigma^2 + B^2\mu^2 - B\rho\sigma^2. \end{aligned}$$

Finally, via substitution, we have

$$\begin{aligned} V(\varphi) &= \frac{1}{B^2}\left[B\sigma^2 + B^2\rho\sigma^2 + B^2\mu^2 - B\rho\sigma^2 - B^2\mu^2\right] \\ &= \frac{1}{B^2}\left[B(\sigma^2 + B\rho\sigma^2 - \rho\sigma^2)\right] \\ &= \frac{\sigma^2 + B\rho\sigma^2 - \rho\sigma^2}{B} = \frac{\sigma^2 - \rho\sigma^2}{B} + \rho\sigma^2 \\ &= \frac{\sigma^2(1-\rho)}{B} + \rho\sigma^2 = \boxed{\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.} \end{aligned}$$

This result holds true for random forests of regression trees as well because simply substituting every tree T with the modified $T^\star$ does not impact the conclusion. This was left as an exercise in [9] and the solution presented above was aided by [16].

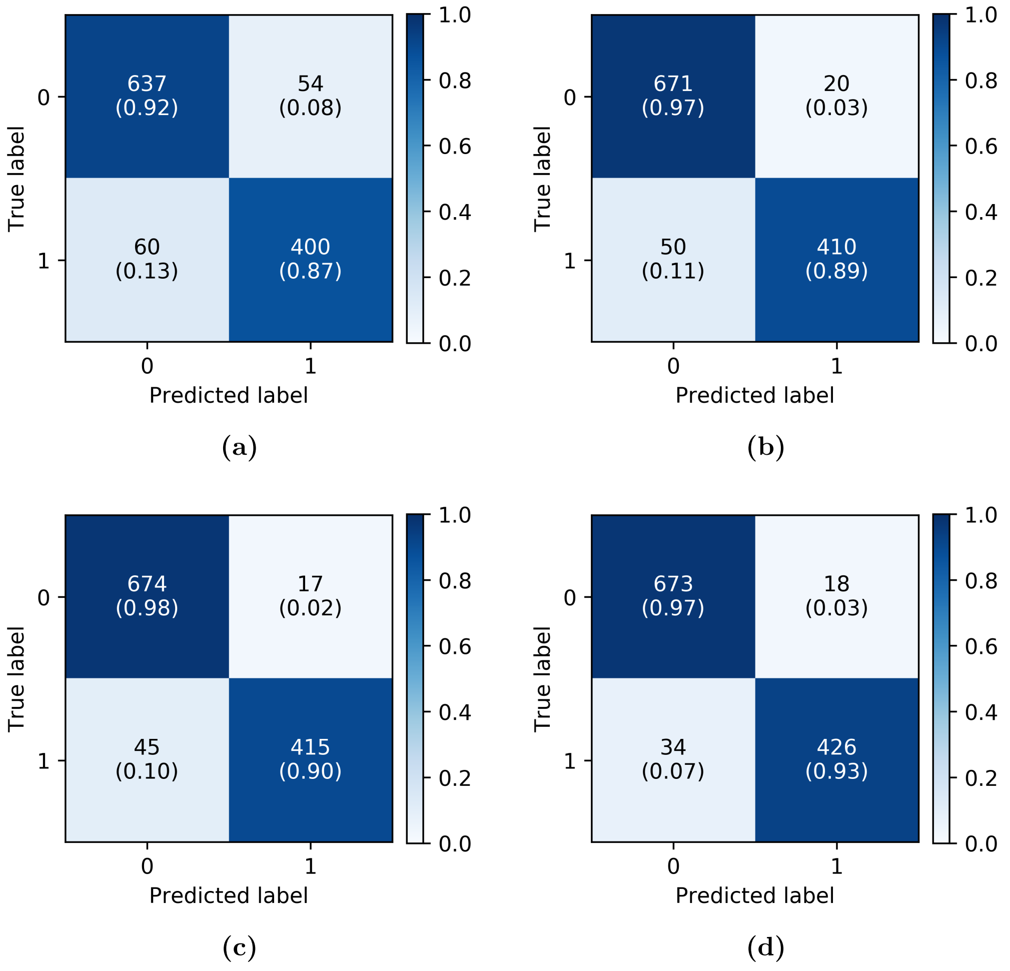


Figure 12: Confusion matrices for classification tree (a), $B = 500$ bagged trees (b), random forest with $B = 500$ and $k = 7$ (c), and Extra-Trees classifier with $B = 500$ and $k = 7$ (d) when applied to the spam test data. All trees were grown with minimum node size $d = 1$.

7.2 Detecting Spam with Extra-Trees

For the sake of comparison with bagging and random forests, let us apply the Extra-Trees algorithm to the spam data with $B = 500$ trees, the default $k = 7$ features at each split, and minimum node size $d = 1$. When applied to the test set, this model achieves 95.5% mean accuracy – an improvement over both bagging (93.9%) and random forests (94.6%) (see Figure 12d for confusion matrix). Interestingly, Extra-Trees produced one more false positive than the random forest model, but eleven fewer false negatives, thus increasing the mean accuracy. If we were to deploy this model as a spam filter, we would expect that, on average, 2-3 out of every 100 spam email would be let in the inbox and 6-7 of every 100 legitimate emails would incorrectly be marked spam.

To illustrate the variance-reducing property of Extra-Trees, we can extend Figure 9 by plotting test accuracy against increasing values of B for bagging *vs.* random forests *vs.* Extra-Trees (see Figure 13). Indeed we see that, once B is sufficiently large (around $B = 250$), random forests consistently outperform bagging and *Extra-Trees consistently outperforms random forests*. We can then conclude this must be due to the extra randomness introduce by randomly selecting splitting points, as it is the only methodological difference between Extra-Trees and traditional random forests. This empirically demonstrates what we have already proven in Section 3.3 and Appendix 7.1: randomness in the tree-growing process mitigates the correlation between trees in the forest, which decreases the variance of our ensembled predictions and generally improves accuracy so long as B is large.

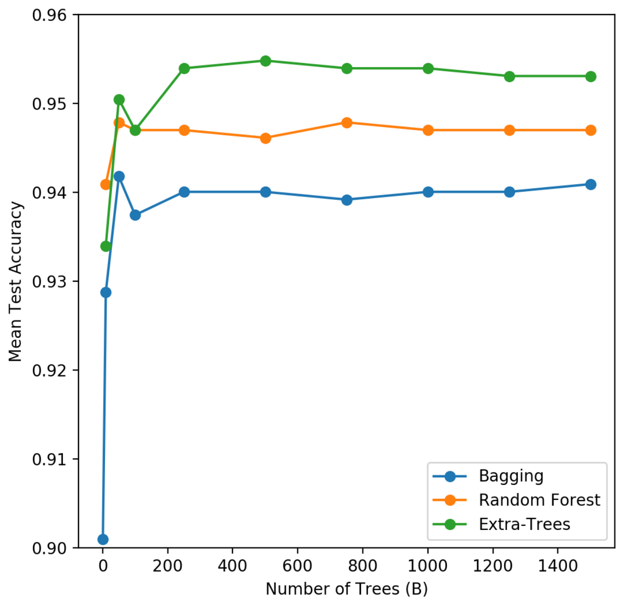


Figure 13: Test accuracy by number of trees B for a bagged classifier, random forest ($k = 7$), and Extra-Trees classifier fit to the spam data. All trees were fit with minimum node size $d = 1$. The left-most point is a single classification tree (a bagged classifier of $B = 1$ tree) for comparison.

7.3 Code Repository

Python and R code used to generate the figures and results presented here can be found in the following Github repository: `https://github.com/holste1/RandomForestsCapstone`.