Random Forest and Boosting

Random Forest

One of the most famous and useful bagged algorithms is the Random Forest! A Random Forest is essentially nothing else but bagged decision trees, with a slightly modified splitting criteria. The algorithm works as follows:

1. Sample m data sets $D_1, \ldots, D_m$ from $D$ with replacement.
2. For each $D_i$ train a full decision tree $h_i$ (max-depth=$\infty$) with one small modification: before each split randomly subsample $k \leq d$ features (without replacement) and only consider these for your split. (This further increases the variance of the trees.)
3. The final classifier is $h(x) = \frac{1}{m} \sum_{i=1}^m h_i(x)$.

The Random Forest is one of the best, most popular and easiest to use out-of-the-box classifier. There are two reasons for this:

- The RF only has two hyper-parameters, $m$ and $k$. It is extremely insensitive to both of these. A good choice for $k$ is $k = \sqrt{d}$ (where $d$ denotes the number of features). You can set $m$ as large as you can afford.

- Decision trees do not require a lot of preprocessing. For example, the features can be of different scale, magnitude, or slope. This can be highly advantageous in scenarios with heterogeneous data, for example the medical settings where features could be things like blood pressure, age, gender, ..., each of which is recorded in completely different units.

Useful variants of Random Forests:

- Split each training set into two partitions $D_k = D_k^A \cup D_k^B$, where $D_k^A \cap D_k^B = \emptyset$. Build the tree on $D_k^A$ and estimate the leaf labels on $D_k^B$. You must stop splitting if a leaf has only a single point in $D_k^B$ in it. This has the advantage that each tree and also the RF classifier become consistent (this roughly means that it approaches the Bayes optimal classifier in the limit of large training set).

- Do not grow each tree to its full depth, instead prune based on the leave out samples. This can further improve your bias/variance trade-off.

Scenario: Hypothesis class $H$, whose set of classifiers has large bias and the training error is high (e.g. CART trees with very limited depth.)

Famous question: In his machine learning class project in 1988 Michael Kearns famously asked the question: Can weak learners ($H$) be combined to generate a strong learner with low bias?

Famous answer: Yes! (Robert Schapire in 1990.

Solution: Create ensemble classifier

$$H_t(\mathcal{F}) = \sum_{i=1}^t \alpha_i h_i(\mathcal{F})$$

This ensemble classifier is built in an iterative fashion. In iteration $t$ we add the classifier $\alpha_i h_i(\mathcal{F})$ to the ensemble. At test time we evaluate all classifier and return the weighted sum.

The process of constructing such an ensemble in a stage-wise fashion is very similar to gradient descent. However, instead of updating the model parameters in each iteration, we add functions to our ensemble.

Let $t$ denote a (convex and differentiable) loss function. With a little abuse of notation we write

$$\ell(H) = \frac{1}{n} \sum_{i=1}^n \ell(H(x_i), y_i).$$

Assume we have already finished $t$ iterations and already have an ensemble classifier $H_t(\mathcal{F})$. Now in iteration $t+1$ we want to add one more weak learner $h_{t+1}$ to the ensemble. To this end we search for the weak learner that minimizes the loss most, i.e.

$$h_{t+1} = \arg\min_{h \in H} \ell(H_t + \alpha h).$$

Once $h_{t+1}$ has been found, we add it to our ensemble, i.e. $H_{t+1} := H_t + \alpha h$.

How can we find such $h \in H$?

Answer: Use gradient descent in function space. In function space, inner product can be defined as $\langle h, g \rangle = \int x \, h(x)g(x)dx$. Since we only have training set, we define $\langle h, g \rangle := \sum_{i=1}^n h(x_i)g(x_i)$.

Gradient descent in functional space

Given $H$, we want to find the step-size $\alpha$ and (weak learner) $h$ to minimize the loss $\ell(H + \alpha h)$. Use Taylor Approximation on $\ell(H + \alpha h)$,

$$\ell(H + \alpha h) \approx \ell(H) + \alpha \langle \nabla \ell(H), h \rangle.$$

This approximation (of $\ell$ as a linear function) only holds within a small region around $\ell(H)$, i.e. as long as $\alpha$ is small. We therefore fix it to a small constant (e.g. $\alpha \approx 0.1$). With the step-size $\alpha$ fixed, we can use the approximation above to find an almost optimal $h$:

$$h_{t+1} = \arg\min_{h \in H} \ell(H + \alpha h) = \arg\min_{h \in H} \langle \nabla \ell(H), h \rangle = \arg\min_{h \in H} \frac{1}{n} \sum_{i=1}^n \frac{\partial \ell}{\partial H(x_i)} h(x_i).$$

We can write $\ell(H) = \sum_{i=1}^n \ell(H(x_i)) = \ell(H(x_1), \ldots, H(x_n))$ (each prediction is an input to the loss function)

$$\frac{\partial \ell}{\partial H(x_i)} = \left. \frac{\partial \ell}{\partial H(z)} \right|_{z=x_i} h(x).$$

So we can do boosting if we have an algorithm $A$ to solve

$$h_{t+1} = \arg\min_{h \in H} \sum_{i=1}^n \left. \frac{\partial \ell}{\partial H(z)} \right|_{z=x_i} h(x).$$
In order to find the best weak learner, we need to solve the optimization problem: (in the following, we will make use of the fact that the function is convex. Each weight $r_t$ is to be determined.)

For notational convenience (and for reason that will become clear in a little bit), let us define $w_t = \frac{1}{r_t} e^{-r_t h(x_t)}$, where $Z = \sum_{t=1}^T e^{-r_t h(x_t)}$ is a normalizing factor so that $\sum_{t=1}^T w_t = 1$. Note that the normalizing constant $Z$ is identical to the loss function. Each weight $w_t$ therefore has a very nice interpretation. It is the relative contribution of the training point $(x_t, y_t)$ towards the overall loss.

In order to find the best next weak learner, we need to solve the optimization problem: (in the following, we will make use of the fact that $h(x_t) \in \{1, -1\}$.)

\[ h = \arg \min_{h} \sum_{t=1}^T r_t h(x_t) \]
Let us examine each one of these updates.

**Further analysis**

Following multiplicative update rule:

When we are given that minimizes \( \sum w_i^t h(x_i) \) = \( \epsilon \)

The weight update:

Further analysis

Finding the stepsize \( \alpha \)

In the previous example, GBRT, we set the stepsize \( \alpha \) to be a small constant. As it turns out, in the AdaBoost setting we can find the optimal stepsize (i.e. the one that minimizes \( \ell \)) in closed form every time we take a ‘gradient’ step.

When we are given \( \ell, H, h \), we would like to solve the following optimization problem:

\[
\alpha = \arg\min_\alpha \langle \ell, H + \alpha h \rangle
\]

We differentiate w.r.t. \( \alpha \) and equate with zero:

\[
-\sum_{y \in \{+1,-1\}} \sum_{h \in H} w_i e^{-\langle y, h(x_i) \rangle} = 0
\]

It is unusual that we can find the optimal step-size in such a simple closed form. One consequence is that AdaBoost converges extremely fast.

**Re-normalization**

After you take a step, i.e. \( H_{t+1} = H_t + \alpha h \), you need to re-compute all the weights and then re-normalize. It is however straightforward to show that the unnormalized weight \( \hat{w}_i \) is updated as \( \hat{w}_i \leftarrow \hat{w}_i * e^{-\langle h(x_i), y_i \rangle} \) and that the normalizer \( Z \) becomes \( Z \leftarrow Z * 2\sqrt{1 - \epsilon} \). Putting these two together we obtain the following multiplicative update rule: \( w_i \leftarrow w_i * e^{\langle h(x_i), y_i \rangle} / Z \).

A few remarks:

- As long as \( H \) is negation closed (this means for every \( h \in H \) we must also have \( -h \in H \)), it cannot be that the error \( \epsilon > \frac{1}{2} \). The reason is simply that if \( h \) has error \( \epsilon \), it must be that \( -h \) has error \( 1 - \epsilon \). So you could just flip \( h \) to \( -h \) and obtain a classifier with smaller error. As \( h \) was found by minimizing the error, this is a contradiction.

- The inner loop can terminate as the error \( \epsilon = \frac{1}{2} \), and in most cases it will converge to \( \frac{1}{2} \) over time. In that case the latest weak learner \( h \) is only as good as a coin toss and cannot benefit the ensemble (therefore boosting terminates). Also note that if \( \epsilon = \frac{1}{2} \) the step-size \( \alpha \) would be zero.

**Adaboost Pseudo-code**

```
Input: \( \ell, \alpha, \{ (x_i, y_i) \} \), \( H \)
\( H_0 = 0 \)
\forall i : w_i = \frac{1}{2}
for \( t=0:T \) do
  \( h = \arg\min_{h \in H} \sum_{i \in H} w_i \langle y, h(x_i) \rangle \)
  \( \epsilon = \sum_{i \in (H \setminus \{ h \})} w_i \)
  if \( \epsilon < \frac{1}{2} \) then
    \( \alpha = \frac{1}{2} \log \frac{1}{\epsilon} \)
    \( H_{t+1} = H_t + \alpha h \)
    \( \forall i : w_i \leftarrow \frac{w_i}{w_i * \epsilon / \alpha} \)
  else
    return \( H_t \)
end
return \( H_T \)
```

Algorithm 1: AdaBoost in Pseudo-Code

Let us examine each one of these updates.

- The weight update:
focused on learning ranking functions, for example LogitBoost. Boosting is a great way to

\[ w_t = w_{t-1} \cdot e^{-\beta h(x)_t} \]

as, \( h(x)_t \) is either +1 (if classified correctly by this weak learner) or -1 (otherwise), it follows that this weight update multiplies the weight \( w_0 \) either by a factor \( e^{-\beta} > 1 \) if it was classified incorrectly (i.e. increases the weights), or by a factor \( e^{-\beta} < 1 \) if it was classified correctly (i.e. decreases the weight).

- **Normalization update:**
  \[ Z \leftarrow Z \cdot 2 \sqrt{\epsilon (1-\epsilon)}. \]

Previously we established that the normalizer \( Z \) is identical to the loss. We can therefore use it to bound the loss function after \( T \) iterations:

\[ \ell(H) = Z = n \prod_{t=1}^{T} 2 \sqrt{\epsilon_t (1-\epsilon_t)}, \]

(the factor \( n \) comes from the fact that the initial \( Z_0 = n \), when all weights are \( \frac{1}{n} \).) If we define \( \epsilon = \max_t \epsilon_t \), we can establish

\[ \ell(H) \leq n \left( 2 \sqrt{\epsilon (1-\epsilon)} \right)^T. \]

The function \( \epsilon (1-\epsilon) \) is maximized at \( \epsilon = \frac{1}{2} \). But we know that each \( \epsilon_t \leq \frac{1}{2} \) (or else the algorithm would have terminated). Therefore \( \epsilon (1-\epsilon) < \frac{1}{2} \) and we can re-write it as \( \epsilon (1-\epsilon) = \frac{1}{2} - \gamma^2 \), for some \( \gamma \). This leaves us with

\[ \ell(H) \leq n (1 - 4 \gamma^2 )^T. \]

In other words, the training loss is decreasing exponentially.

In fact, we can go even further and compute after how many iterations we must have zero training error. Note that the training loss is an upper bound on the training error (defined as \( \sum_{i=1}^{n} \delta(y_i, H(x)_i) \)) simply because \( \delta(y_i, H(x)_i) \leq e^{-\beta H(x)_i} \) in all cases. We can then compute the number of steps required until the loss is less than 1, which would imply that not a single training input is misclassified.

\[ n (1 - 4 \gamma^2 )^T < 1 \Rightarrow T > \frac{2 \log(n)}{\log(\frac{1}{1-4 \gamma^2})} \]

This is an amazing result. It shows that after \( O(\log(n)) \) iterations your training error must be zero. In practice it often makes sense to keep boosting even after you make no more mistakes on the training set.

**Summary**

Boosting is a great way to turn a weak classifier into a strong classifier. It defines a whole family of algorithms, including Gradient Boosting, AdaBoost, LogitBoost, and many others. Gradient Boosted Regression Trees is one of the most popular algorithms for Learning to Rank, the branch of machine learning focused on learning ranking functions, for example for web search engines. A few additional things to know:

- The step size \( \alpha \) is often referred to as shrinkage.

- Some people do not consider gradient boosting algorithms to be part of the boosting family, because they have no guarantee that the training error decreases exponentially. Often these algorithms are referred to as stage-wise regression instead.

- Inspired by Breiman’s Bagging, stochastic gradient boosting subsamples the training data for each weak learner. This combines the benefits of bagging and boosting. One variant is to subsample only \( n/2 \) data points without replacement, which speeds up the training process.

- One advantage of boosted classifier is that during test time the computation \( H(x) = \sum_{t=1}^{T} \alpha_t h_t(x) \) can be stopped prematurely if it becomes clear which way the prediction goes. This is particularly interesting in search engines, where the exact ranking of results is typically only interesting for the top 10 search results. Stopping the evaluation of lower ranked search results can lead to tremendous speed ups. A similar approach is also used by the Viola-Jones algorithm to speed up face detection in images. Here, the algorithm scans regions of an image to detect possible faces. As almost all regions and natural images do not contain faces, there are huge savings if the evaluation can be stopped after just a few weak learners are evaluated. These classifiers are referred to as cascades, that spend very little time on the common case (no face), but more time on the rare interesting case (face). With this approach Viola and Jones were the first to solve face recognition in real-time on low performance hardware (e.g. cameras).

- AdaBoost is an extremely powerful algorithm, that turns any weak learner that can classify any weighted version of the training set with below 0.5 error into a strong learner whose training error decreases exponentially and that requires only \( O(\log(n)) \) steps until it is consistent.