Generalized Additive Models

Duke Course Notes
Cynthia Rudin

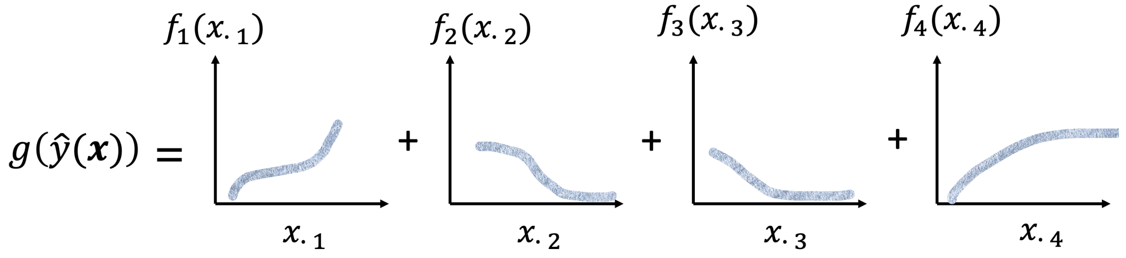
I have long been excited about the work of Rich Caruana and coauthors [Lou et al., 2012, Caruana et al., 2015] on generalized additive models (GAMs). They use a very simple but effective trick to be able to leverage the firepower of algorithms like AdaBoost to create predictive models that are more easily understood by humans.

A generalized additive model is comprised as a sum of p terms, each of which is a nonlinear function of one of the original variables.

$$g(\hat{y}(\mathbf{x})) = \sum_{j=1}^{p} f_j(x_{\cdot,j}).$$

Here, again the $x_{\cdot,j}$ notation means that I am talking about the jth feature, leaving a placeholder so that you know I am referring to feature j rather than datapoint j. The function g is a link function chosen by the user. This function (like in logistic regression) might transform estimated probabilities $\hat{y}$ – that must take on values between 0 and 1 – to the whole real line to be compatible with the right hand side. The term $f_j(x_{\cdot,j})$ is called the component function for feature j.

An illustration is below, where nonlinear functions of each of the four variables are shown. As you can see, there are no cross-terms, i.e., no interaction terms between variables.

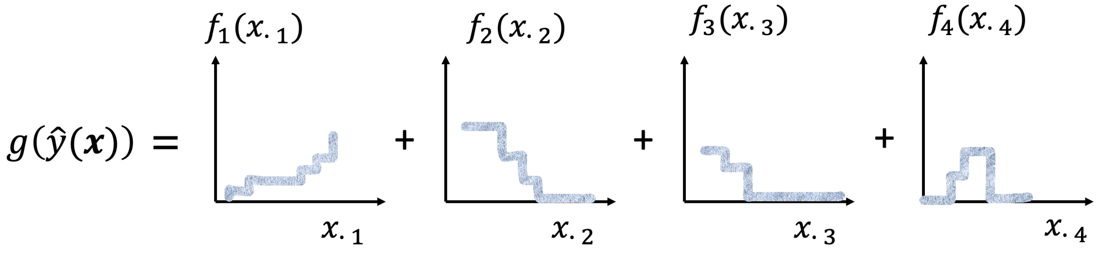


For instance, let us say that we want to predict poor outcomes of COVID-19, and say that "age" is one of the original features in the dataset. In this case, the probability of a poor outcome is low for younger people and could increase rapidly for middle-aged people, and could be constant and high for older people.

Thus, we would hope that the component function f_{age} would be flat, then increase, and flatten out again.

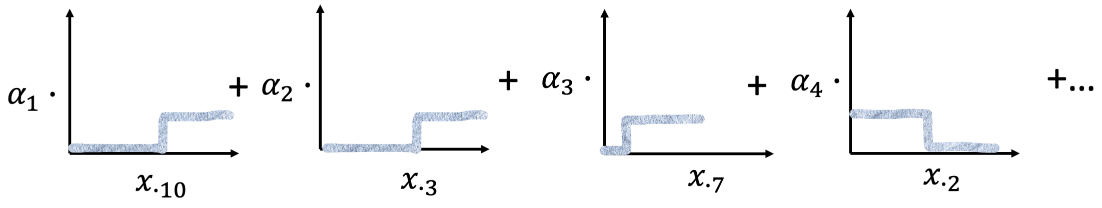
Generalized additive models give us interpretability like that of a linear model since there are no interaction terms between variables, but they also are more powerful than linear models, as you could see from the example using age above where nonlinearities are important.

We will use a sum of step functions to comprise each of the component functions.

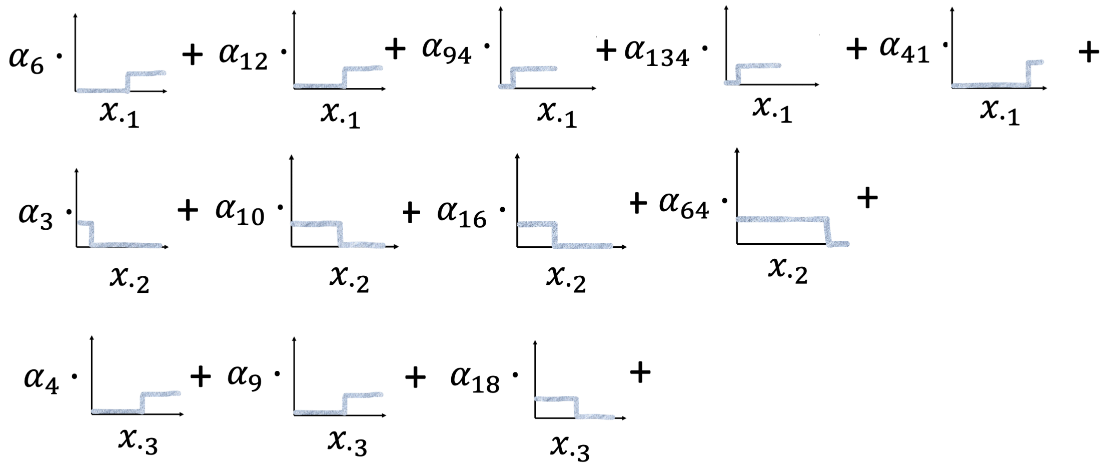


Each of these step functions looks like either $\alpha \cdot 1_{[x_{\cdot j} \geq \theta_{j\ell}]}$ (a step facing right) or $\alpha \cdot 1_{[x_{\cdot j} \leq \theta_{j\ell}]}$ (a step facing left), where α is nonnegative. If you would like component function j to be monotonically decreasing, you could choose it to contain only left-facing steps; if you would like the component function to be monotonically increasing, you could choose only right-facing steps. If you want the component function to be flexible and you do not care about monotonicity, then you could allow it to use both right-facing and left-facing step functions. Equivalently, we could use just upwards-facing steps and allow α to be either positive (for the function to increase) or negative (for the function to decrease).

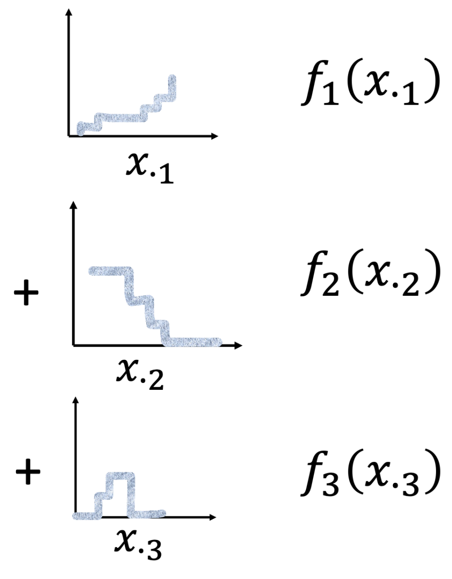
To get the component functions, the easiest way will be to use boosted stumps and rearrange them into component functions. The term "stump" just refers to a step function. It is called a stump because you might also think of a step function as a tree with just one split. At every iteration, AdaBoost will add one weighted step function to one of the component functions.



In the illustration above, AdaBoost added a step function of height α_1 to the component function for the 10th feature, then it added a step function of height α_2 to the component function of the 3rd feature and so on. After we run AdaBoost, we have a lot of stumps. We sort them by feature, so that all the stumps that used feature 1 are together at the top, all the stumps that used feature 2 are together below that, and so on:


$\alpha_6 \cdot$ + $\alpha_{12} \cdot$ + $\alpha_{94} \cdot$ + $\alpha_{134} \cdot$ + $\alpha_{41} \cdot$ +
$x_{\cdot 1}$
$\alpha_3 \cdot$ + $\alpha_{10} \cdot$ + $\alpha_{16} \cdot$ + $\alpha_{64} \cdot$ +
$x_{\cdot 2}$
$\alpha_4 \cdot$ + $\alpha_9 \cdot$ + $\alpha_{18} \cdot$ +
$x_{\cdot 3}$

Now, we add up the stumps for each feature separately to form the component functions.



Notice how, in the illustration above, f_1 is monotonically increasing because it is comprised of only right-facing stumps. f_2 has only left-facing stumps, and f_3 has some of each.

Let us write down this process using AdaBoost notation. For simplicity, I will just use right-facing stumps, but one could easily generalize this to include other kinds of stumps. At iteration t, AdaBoost produces a term like $\alpha_t h_t(\mathbf{x}) = \alpha_t \cdot 1_{[x_{j_t} \geq \theta_{j_t \ell}]}$, where $\theta_{j_t l}$ is the ℓth threshold for feature j_t, which was chosen at iteration t of AdaBoost by the weak learning algorithm. Let us regroup the terms. For notational convenience, I'll define weak classifier $1_{[x_{\cdot j} \geq \theta_{j\ell}]}$ as stump$(j, \ell)(\mathbf{x})$. It is 1 when the jth component of $\mathbf{x}$ is at least $\theta_{j\ell}$, and 0 otherwise.

$$\begin{aligned}
g(\hat{y}(\mathbf{x})) &= \sum_t \alpha_t h_t(\mathbf{x}) \quad \text{(sum over iterations)} \\
&= \sum_{j=1}^{p} \sum_{\text{thresholds } \ell} \sum_t 1[h_t = \text{stump}(j,\ell)] \cdot \alpha_t \cdot \text{stump}(j,\ell)(\mathbf{x}) \\
&= \sum_{j=1}^{p} \sum_{\text{thresholds } \ell} \sum_t 1_{\text{ if classifier } h_t \text{ is a step of feature } j \text{ at threshold } \theta_{j\ell}} \cdot \alpha_t \cdot \\
&\qquad\qquad 1_{\text{if the } j\text{th feature of } \mathbf{x} \text{ exceeds threshold } \theta_{jl}} \\
&= \sum_{j=1}^{p} \sum_{\text{thresholds } \ell} \left[\sum_t 1[h_t = \text{stump}(j,\ell)] \cdot \alpha_t \right] \cdot \text{stump}(j,\ell)(\mathbf{x}) \\
&=: \sum_j \sum_\ell \lambda_{j,\ell} \cdot \text{stump}(j,\ell)(\mathbf{x}) \\
&\qquad \text{(defining } \lambda_{j,\ell} \text{ as sum over } \alpha_t \text{ when the } j,\ell \text{ stump is used)} \\
&= \sum_j \left[\sum_\ell \lambda_{j,\ell} \cdot \text{stump}(j,\ell)(\mathbf{x}) \right] \\
&=: \sum_j f_j(x_{\cdot,j}) \quad \text{(defining the component functions).}
\end{aligned}$$

Thus, we have our component functions.

Training Methods for Generalized Additive Models

First, AdaBoost is a perfectly reasonable way to train a GAM in the way I just described.

Backfitting is a second approach, where you would iteratively model the residual between y and our model, $f = \sum_j f_j$, and add it into f.

It might be beneficial to include pairwise interactions. In the paper Accurate Intelligible Models with Pairwise Interactions [Caruana et al., 2015], they use a functional form:

$$g(\hat{y}(\mathbf{x})) = \sum_j f_j(x_{\cdot,j}) + \sum_{\text{feature pairs } k,j:k\neq j} f_{k,j}(x_{\cdot,k}, x_{\cdot,j})$$

Caruana et al. [2015] call this a GA^{2}M model. To train it, they first fit a generalized additive model (no interaction terms yet). They use forward selection to add in interaction terms. Specifically, until convergence, they add an interaction term that is chosen to minimize the residual between y and the model's predictions $\hat{y}$. Then they refit the model $\hat{y}$ with the new interaction term.

The component functions with lots of steps can model essentially any 1D function, which makes stumps a desirable choice. A problem with the component functions fitted using AdaBoost or the other methods I just discussed is that they can have very wiggly component functions that are caused by overfitting. You can try to fix those manually, but you could also try not to overfit by making the models sparse.

In the next subsection, I'll discuss FastSparse [Liu et al., 2022], which is an adaptation of AdaBoost for sparse GAM models.

FastSparse for Training Sparse GAMs

FastSparse [Liu et al., 2022] produces GAMs with a small number of overall stumps. This is convenient because one can rearrange a component function into a small table. For instance, if we had learned the sparse component function for age: $f_{\text{age}} = 1.5 \cdot 1_{[\text{age}\geq 30]} + 0.5 \cdot 1_{[\text{age}\geq 35]} + 0.3 \cdot 1_{[\text{age}\geq 40]} + 0.1 \cdot 1_{[\text{age}\geq 45]}$, we could plot it as a set of step functions, or it can translate into a table with the same meaning:
Here you would be assigned some number of points based on your age for the "age" component function, and be assigned points for other component functions. The total number of points would translate into a risk as usual from AdaBoost's formula $P(Y=1|\mathbf{x}) = e^{2f(\mathbf{x})}/(1+e^{2f(\mathbf{x})})$ that we derived earlier.

If you have a lot of component functions, it is useful if all of these tables are fairly small. Such tables are often used in medicine and criminal justice.

age	f_{age}
age < 30	0 points
$30 \leq$ age <35	1.5 points
$35 \leq$ age <40	2 points
$40 \leq$ age <45	2.3 points
$45 \leq$ age	2.4 points
Your score	___

FastSparse leverages a lot of AdaBoost's beautiful ideas.

AdaBoost Notation Refresher

We recall some notation from AdaBoost:

$$\begin{aligned} R^{\text{train}}(\boldsymbol{\lambda}) &= \frac{1}{n}\sum_i e^{-y_i f(\mathbf{x}_i)} \quad \text{(exponential loss)} \\ &= \frac{1}{n}\sum_i e^{-\sum_j y_i h_j(\mathbf{x}_i)\cdot\lambda_j} \\ &= \frac{1}{n}\sum_i e^{-(\mathbf{M}\boldsymbol{\lambda})_i} \quad \text{(definition of matrix of margins)} \end{aligned}$$

At time t, the weight vector and normalization are:

$$d_{t,i} = \frac{e^{-(\mathbf{M}\boldsymbol{\lambda}_t)_i}}{Z_t} \text{ where } Z_t = \sum_i e^{-(\mathbf{M}\boldsymbol{\lambda}_t)_i} = nR^{\text{train}}(\boldsymbol{\lambda}_t).$$

Did you notice that the normalization factor is the objective we're trying to minimize? It's so interesting! In any case, after we choose weak classifier j_t, we can define:

$$d_- = \sum_{i:M_{ij_t}=-1} d_{t,i} \quad \text{(weighted error of the weak classifier at time } t\text{)}$$

AdaBoost's update rule is

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-d_-}{d_-}\right).$$

Furthermore, we derived a recursive formula for AdaBoost's objective:

$$R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) = R^{\text{train}}(\boldsymbol{\lambda}_t)2[d_-(1-d_-)]^{1/2}. \quad (1)$$

AdaBoost's coordinate descent algorithm is:

```
$d_{1,i} = 1/n$ for $i = 1...n$
$\boldsymbol{\lambda}_1 = \mathbf{0}$
loop $t = 1...T$
 $j_t$ = weak classifier chosen at iteration $t$, with weighted error $d_-$, where
 $d_- = \sum_{M_{ij_t}=-1} d_{t,i}$
 $\alpha_t = \frac{1}{2}\ln\left(\frac{1-d_-}{d_-}\right)$
 $\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \alpha_t \mathbf{e}_{j_t}$
 $d_{t+1,i} = e^{-(\mathbf{M}\boldsymbol{\lambda}_{t+1})_i}/Z_{t+1}$ for each $i$, where $Z_{t+1} = \sum_{i=1}^n e^{-(\mathbf{M}\boldsymbol{\lambda}_{t+1})_i}$
end
```

Now that we've finished reviewing AdaBoost, I can finally discuss FastSparse. It is different from AdaBoost in two ways: the choice of j_t, and the update rule for $\boldsymbol{\lambda}_{t+1}$. It's objective is also different:

$$R_0^{\text{train}}(\boldsymbol{\lambda}) = R^{\text{train}}(\boldsymbol{\lambda}) + C_0\|\boldsymbol{\lambda}\|_0,$$

which is a sum of the exponential loss from AdaBoost, and a regularization term that counts the nonzero terms of $\boldsymbol{\lambda}$. Including the ℓ_0 term encourages sparse solutions.

I'll give the update rule for $\boldsymbol{\lambda}$ first. The notation is a little simpler than AdaBoost since I won't use the j_t notation. I'll just say that I am updating component j at time t.

FastSparse's update rule for $\boldsymbol{\lambda}_{t+1}$. Here we will update component j.

Case 1 is when $\lambda_{t,j} = 0$.
If the following condition holds:

$$d_- \in \left[\frac{1}{2} - \frac{1}{2R^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0(2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0)},\ \frac{1}{2}\right],$$

then set $\lambda_{t+1,j} = 0$. Otherwise set

$$\lambda_{t+1,j} = \frac{1}{2}\ln\left(\frac{1-d_-}{d_-}\right).$$

Case 2 is when $\lambda_{t,j} \neq 0$.
We need to take away the jth component of f temporarily,

$$\begin{aligned}
f_t^{\backslash j}(\mathbf{x}) &= (\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j)^T\mathbf{x} \quad \text{(set the } j\text{th component to 0).} \\
d_{t,i}^{\backslash j} &:= e^{-y_i f_t^{\backslash j}(\mathbf{x}_i)}/Z^{\backslash j} \quad \text{(where } Z^{\backslash j} \text{ is a normalization constant)} \qquad (2) \\
d_-^{\backslash j} &:= \sum_{i:M_{ij}=-1} d_{t,i}^{\backslash j} \qquad (3) \\
R_{\backslash j}^{\text{train}}(\boldsymbol{\lambda}_t) &= R^{\text{train}}(\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j) \qquad (4)
\end{aligned}$$

If

$$d_-^{\backslash j} \in \left[\frac{1}{2} - \frac{1}{2R_{\backslash j}^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0(2R_{\backslash j}^{\text{train}}(\boldsymbol{\lambda}_t) - C_0)}, \frac{1}{2}\right]$$

then set $\lambda_{t+1,j} = 0$. Otherwise set

$$\lambda_{t+1,j} = \frac{1}{2}\ln\left(\frac{1 - d_-^{\backslash j}}{d_-^{\backslash j}}\right).$$

Let us think about these two cases. They are actually very similar to each other. In both cases, we start from the jth term of $\boldsymbol{\lambda}_t$ being 0 and compute an interval around 1/2. An error rate of 1/2 means feature j is no better than random guessing on the weighted training data. An error rate close to 0 means that weak classifier j is really good and will reduce the error substantially if we use it.

Proof (i.e., derivation) of Case 1 and Case 2 for the update rule.
In Case 1, $\lambda_{t,j} = 0$ and we are considering setting $\lambda_{t+1,j} \neq 0$. Recall

$$R_0^{\text{train}}(\boldsymbol{\lambda}) = R^{\text{train}}(\boldsymbol{\lambda}) + C_0\|\boldsymbol{\lambda}_0\|.$$

If we include a new non-zero term in $\boldsymbol{\lambda}$, the value of R_0^{train} will go up by C_0. Looking at the balance between these terms, if the second term goes up by C_0, in order to reduce R_0^{train}, the first term (which is the exponential loss $R^{\text{train}}(\boldsymbol{\lambda})$), must reduce by at least C_0. So we need to know that $R^{\text{train}}(\boldsymbol{\lambda}_t) - R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) \geq C_0$ at each iteration of our new algorithm for reducing R_0^{train}.

Using the recursive formula (1) for AdaBoost's loss above,

$$R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) = R^{\text{train}}(\boldsymbol{\lambda}_t)2[d_-(1 - d_-)]^{1/2},$$

When does this obey $R^{\text{train}}(\boldsymbol{\lambda}_t) - R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) \geq C_0$? Let's find out.

$$R^{\text{train}}(\boldsymbol{\lambda}_t) - R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) = R^{\text{train}}(\boldsymbol{\lambda}_t)\left[1 - 2[d_-(1-d_-)]^{1/2}\right] \overset{?}{\geq} C_0$$

$$\begin{aligned}
1 - 2[d_-(1-d_-)]^{1/2} &\overset{?}{\geq} \frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)} \\
1 - \frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)} &\overset{?}{\geq} 2[d_-(1-d_-)]^{1/2} \\
\frac{1}{2}\left(1 - \frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)}\right) &\overset{?}{\geq} [d_-(1-d_-)]^{1/2}
\end{aligned}$$

Setting the left side equal to v temporarily, we have:

$$\begin{aligned}
v &\overset{?}{\geq} [d_-(1-d_-)]^{1/2} \\
v^2 &\overset{?}{\geq} d_-(1-d_-) = d_- - d_-^2 \\
d_-^2 - d_- + v^2 &\overset{?}{\geq} 0.
\end{aligned}$$

This is an upwards-facing quadratic, and we know that the inequality is satisfied when d_- is small and large, but possibly not in between. The roots of the quadratic are:

$$\text{roots} = \frac{1}{2} \pm \frac{1}{2}\sqrt{1-4v^2}. \tag{5}$$

Doing a little simplifying on the term within the square root:

$$\begin{aligned}
1 - 4v^2 &= 1 - 4\frac{1}{(2)^2}\left(1 - \frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)}\right)^2 \\
&= 1 - \left[1 - 2\frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)} + \left(\frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)}\right)^2\right] \\
&= 2\frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)} - \left(\frac{C_0}{R^{\text{train}}(\boldsymbol{\lambda}_t)}\right)^2 \\
&= \frac{C_0}{\left(R^{\text{train}}(\boldsymbol{\lambda}_t)\right)^2}[2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0].
\end{aligned}$$

So the roots in (5) become:

$$\text{roots} = \frac{1}{2} \pm \frac{1}{2R^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0[2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0]}. \tag{6}$$

We don't really want to consider error rates above 1/2, so that removes the upper root. Thus, we know that for d_- being below $\frac{1}{2} - \frac{1}{2R^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0[2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0]}$, we will get the improvement in the loss we needed in order to justify adding a nonzero term for $\boldsymbol{\lambda}_t$:

$$\left[d_- \leq \frac{1}{2} - \frac{1}{2R^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0[2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0]}\right] \Longrightarrow \left[R^{\text{train}}(\boldsymbol{\lambda}_t) - R^{\text{train}}(\boldsymbol{\lambda}_{t+1}) \geq C_0\right].$$

In that case, since $\lambda_{t,j} = 0$, we update it using AdaBoost's update rule, setting $\lambda_{t+1,j} = \lambda_{t+1,j} + \alpha_t = 0 + \frac{1}{2}\ln\left(\frac{1-d_-}{d_-}\right)$.

If d_- is close to 1/2, i.e., it's above $\frac{1}{2} - \frac{1}{2R^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0[2R^{\text{train}}(\boldsymbol{\lambda}_t) - C_0]}$, the objective R_0^{train} is smaller if we keep the value of $\lambda_{t+1,j}$ at 0, so we will do that. On to Case 2.

In Case 2, $\lambda_{t,j} \neq 0$. Certainly we would set $\lambda_{t+1,j} = 0$ if this improved the loss. We calculate the objectives for $\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j$ (which is when we set the jth component to 0), compared with $\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j + \lambda^{\text{step}}\mathbf{e}_j$ (where we then updated the jth component afterwards). We need to compare these two options.

For $R^{\text{train}}(\boldsymbol{\lambda}_t - \boldsymbol{\lambda}_{t,j}\mathbf{e}_j + \lambda^{\text{step}}\mathbf{e}_j)$, using Definitions (2) and (3), by AdaBoost's recursive update rule, if we ran one step of AdaBoost,

$$\lambda^{\text{step}} = \frac{1}{2}\ln\left(\frac{1 - d_-^{\backslash j}}{d_-^{\backslash j}}\right), \tag{7}$$

and by the recursive equation (1),

$$R^{\text{train}}(\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j + \lambda^{\text{step}}\mathbf{e}_j) = R^{\text{train}}(\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j) \cdot 2\left[(1 - d_-^{\backslash j})d_-^{\backslash j}\right]^{1/2}.$$

This looks identical to the math from Case 1, so we know that:

$$R^{\text{train}}(\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j) - R^{\text{train}}(\boldsymbol{\lambda}_t - \lambda_{t,j}\mathbf{e}_j + \lambda^{\text{step}}\mathbf{e}_j) \leq C_0 \tag{8}$$

whenever the following happens, using notation from (4):

$$d_-^{\backslash j} \in \left[\frac{1}{2} - \frac{1}{2R_{\backslash j}^{\text{train}}(\boldsymbol{\lambda}_t)}\sqrt{C_0(2R_{\backslash j}^{\text{train}}(\boldsymbol{\lambda}_t) - C_0)}, \frac{1}{2}\right].$$

If $d_-^{\backslash j}$ is in this range, weak classifier j is not very good according to (8) and won't give us an overall improvement in the loss, so we set $\lambda_{t+1,j} = 0$. Otherwise, we set $\lambda_{t+1,j} = \lambda^{\text{step}}$ from (7). We are done with Case 2.

Order of weak classifiers

The only remaining part of FastSparse to discuss is its choice for the j to update at iteration t.

As a preprocessing step, FastSparse sorts the features by their accuracy (that is, their agreement with the labels):

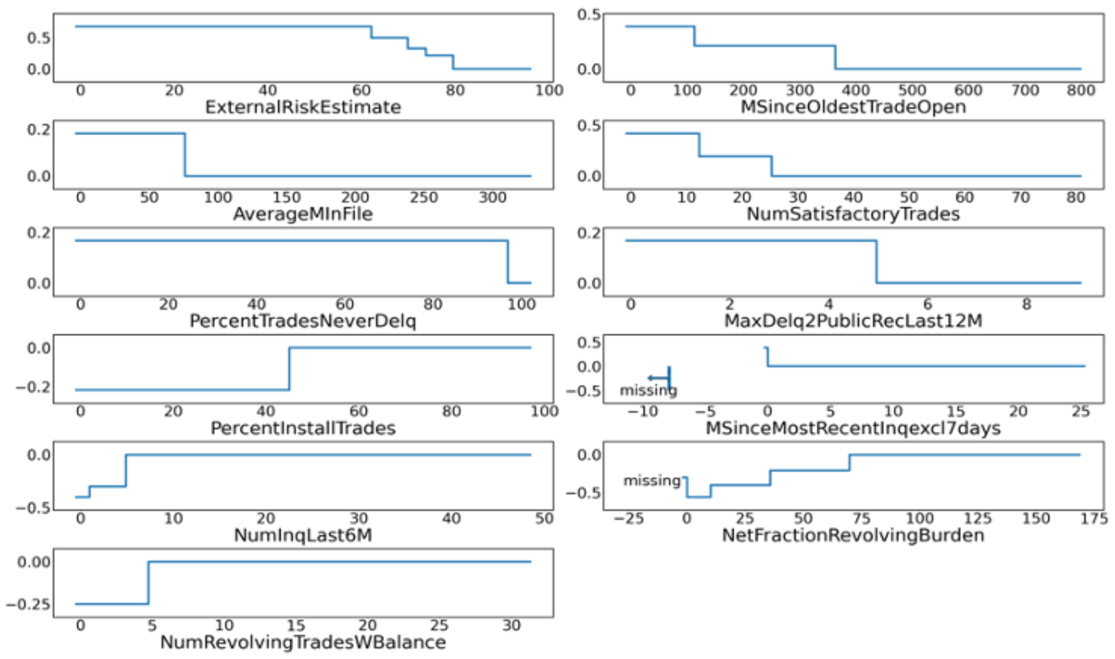
$$\text{acc}(\text{stump}(j,\ell)) = \frac{1}{n}\sum_i \mathbb{1}_{[y_i=\text{stump}(j,\ell)(\mathbf{x}_i)]}.$$

Then it cycles through the j's in order of this sorting for a while (maybe 100 iterations), following the steps above to tell it how far to move in each direction. Thus, FastSparse does not (like AdaBoost) move in the steepest direction, it cycles among all coordinates in order to reduce computation of gradients.

After several iterations (perhaps 100) of coordinate descent following the steps listed above, it performs a "swap step" where it tries to swap one stump that is in the model for another one that is not in the model, by trying to move the coefficient $\lambda_{t,j}$ from the current stump it is "swapping out" to every new feature that is currently not being used in the model to see if there is a better one. FastSparse has a priority queue. If it tries to swap out a stump but there is no better one to swap with, FastSparse doesn't want to try swapping that stump again for a while, and puts it at the back of the priority queue. If the swap is successful, it leaves that stump at the front of the priority queue.

FastSparse alternates between swap steps and more coordinate descent steps until converged.

An example of a FastSparse model on the FICO dataset from the 2018 Explainable Machine Learning Challenge [FICO et al., 2018] is below. Here, you can see all of the component functions. (There are some component functions that have indicator variables for missing values that are shown at the left of the component.) We can see that the feature ExternalRiskEstimate is important, because it takes on a large range of values; individuals with low ExternalRiskEstimate have a very different estimate loan default than individuals with high External-RiskEstimate.



To calculate the total score, you add up each component at the value of each feature, which is $f(\mathbf{x})$, and send it through $P(Y = 1|\mathbf{x}) = e^{2f(\mathbf{x})}/(1 + e^{2f(\mathbf{x})})$.

Remarks

Let us finish by discussing when generalized additive models of the kind we have discussed are a good idea. Clearly they are designed for tabular data where each feature is meaningful (as opposed to computer vision data, for instance).

They create powerful nonlinear component functions, but such functions are only really helpful when the original features are real-valued. You can definitely still use them on binary features, but the component functions are very boring single step functions – either the function f increases when you increase the feature from 0 to 1, or the function f decreases when you change the feature from 0 to 1. Thus, if you have a large number of binary features, f could become a giant mess of step functions. In that case, perhaps a decision tree might be better.

They are good for binary classification (or you could use the same technique for regression since it adapts easily) but it's not clear how to use them for multiclass problems. Decision trees would be better in that case, since each leaf can predict a different class.

GAMs have no interaction terms, whereas decision trees are essentially comprised of interaction terms.

So, as you can probably see by now, generalized additive models are probably best suited for datasets with a relatively small number of continuous variables (including a few binary variables too is fine).

References

Rich Caruana, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pages 1721–1730, 2015.

FICO, Google, Imperial College London, MIT, University of Oxford, UC Irvine, and UC Berkeley. Explainable Machine Learning Challenge. `https://community.fico.com/s/explainable-machine-learning-challenge`, 2018.

Jiachang Liu, Chudi Zhong, Margo Seltzer, and Cynthia Rudin. Fast sparse classification for generalized linear and additive models. In Proceedings of Artificial Intelligence and Statistics (AISTATS), 2022.

Yin Lou, Rich Caruana, and Johannes Gehrke. Intelligible models for classification and regression. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pages 150–158, 2012.