Interpretable Generalized Additive Models via Boosting

Duke Course Notes
Cynthia Rudin

I have long been excited about the work of Rich Caruana and coauthors on generalized additive models using AdaBoost. They use a very simple but effective trick to be able to leverage the firepower of 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.

\[
g(\hat{y}(x)) = \sum_{j=1}^{p} f_j(x_{-j}).
\]

Here, again the \( x_{-j} \) notation means that I am talking about the \( j \)th 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.

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 function \( f_{\text{age}} \) for 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 \( \alpha \cdot 1[x_{j} \geq \theta_{j\ell}] \) (a step facing right) or \( \alpha \cdot 1[x_{j} \leq \theta_{j\ell}] \) (a step facing left). 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.

To get the component functions, we will just use boosted stumps. 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 \( \alpha_{1} \) to the component function for the 10th feature, then it added a step function of height \( \alpha_{2} \) 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:

\[
\begin{align*}
\alpha_6 \cdot x_1 + \alpha_{12} \cdot x_1 + \alpha_{94} \cdot x_1 + \alpha_{134} \cdot x_1 + \alpha_{41} \cdot x_1 + \\
\alpha_3 \cdot x_2 + \alpha_{10} \cdot x_2 + \alpha_{16} \cdot x_2 + \alpha_{64} \cdot x_2 + \\
\alpha_4 \cdot x_3 + \alpha_9 \cdot x_3 + \alpha_{18} \cdot x_3 + 
\end{align*}
\]

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

\[
\begin{align*}
\begin{array}{c}
\text{a} \quad \text{stump} \\
\alpha_6 \cdot x_1 + \alpha_{12} \cdot x_1 + \alpha_{94} \cdot x_1 + \alpha_{134} \cdot x_1 + \alpha_{41} \cdot x_1 + \\
\alpha_3 \cdot x_2 + \alpha_{10} \cdot x_2 + \alpha_{16} \cdot x_2 + \alpha_{64} \cdot x_2 + \\
\alpha_4 \cdot x_3 + \alpha_9 \cdot x_3 + \alpha_{18} \cdot x_3 + 
\end{array}
\end{align*}
\]

\[
f_1(x_1) = \alpha_6 \cdot x_1 + \alpha_{12} \cdot x_1 + \alpha_{94} \cdot x_1 + \alpha_{134} \cdot x_1 + \alpha_{41} \cdot x_1
\]

\[
f_2(x_2) = \alpha_3 \cdot x_2 + \alpha_{10} \cdot x_2 + \alpha_{16} \cdot x_2 + \alpha_{64} \cdot x_2
\]

\[
f_3(x_3) = \alpha_4 \cdot x_3 + \alpha_9 \cdot x_3 + \alpha_{18} \cdot x_3
\]

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 \(h_t(x) = \alpha_t \cdot 1[x_{j_t} \geq \theta_{j_t \ell}]\), where \(\theta_{j_t \ell}\) is the \(\ell\)th threshold for feature \(j_t\), which was chosen at iteration \(t\) of
AdaBoost by the weak learning algorithm. Let us regroup the terms.

\[ g(\hat{y}(\mathbf{x})) = \sum_t \alpha_t h_t(\mathbf{x}) \quad \text{(sum over iterations)} \]

\[ = \sum_j \sum_{\ell} \sum_t 1[h_t(\mathbf{x}) = 1[x_{.j} \geq \theta_{j\ell}] \cdot \alpha_t \cdot 1[x_{.j} \geq \theta_{j\ell}] \]

\[ = \sum_j \sum_{\ell} \left[ \sum_t 1 \text{ if classifier } h_t \text{ is a step of feature } j \text{ at threshold } \theta_{j\ell} \cdot \alpha_t \right] \cdot \]

\[ 1 \text{ if the } j\text{th feature exceeds threshold } \theta_{j\ell} \]

\[ =: \sum_j \sum_{\ell} \lambda_{j,\ell} \cdot 1[x_{.j} \geq \theta_{j\ell}] \quad \text{(defining } \lambda_{j,\ell} \text{ as sum over } \alpha_t \text{ when the } j, \ell \text{ stump is used)} \]

\[ = \sum_j \left[ \sum_{\ell} \lambda_{j,\ell} \cdot 1[x_{.j} \geq \theta_{j\ell}] \right] \]

\[ =: \sum_j f_j(x_{.j}) \quad \text{(defining the component functions)} \]

Thus, we have our component functions.

There are some important notes on generalized additive models.

First, AdaBoost is not the only way to train generalized additive models, such as backfitting, where you would iteratively model the residual between \( y \) and our model, \( f = \sum_j f_j \), and add it into \( f \).

Second, it might be beneficial to include pairwise interactions. In the paper Accurate Intelligible Models with Pairwise Interactions by Lou, Caruana, et al. (2013), they use a functional form:

\[ g(\hat{y}(\mathbf{x})) = \sum_j f_j(x_{.j}) + \sum_{\text{feature pairs } k, j: k \neq j} f_{k,j}(x_{.k}, x_{.j}) \]
In their paper, Lou et al. (2013) call this a GA²M 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.

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 feature is 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, \( f \) would be a giant mess of step functions. In that case, perhaps a decision tree might be better. 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).