https://multithreaded.stitchfix.com/assets/files/gam.pdf

GAM: The Predictive Modeling Silver Bullet

Author: Kim Larsen

Introduction

Imagine that you step into a room of data scientists; the dress code is casual and the scent of strong coffee is hanging in the air. You ask the data scientists if they regularly use generalized additive models (GAM) to do their work. Very few will say yes, if any at all.

Now let's replay the scenario, only this time we replace GAM with, say, random forest or support vector machines (SVM). Everyone will say yes, and you might even spark a passionate debate.

Despite its lack of popularity in the data science community, GAM is a powerful and yet simple technique. Hence, the purpose of this post is to convince more data scientists to use GAM. Of course, GAM is no silver bullet, but it is a technique you should add to your arsenal. Here are three key reasons:

- Easy to interpret.
- Flexible predictor functions can uncover hidden patterns in the data.
- Regularization of predictor functions helps avoid overfitting.

In general, GAM has the interpretability advantages of GLMs where the contribution of each independent variable to the prediction is clearly encoded. However, it has substantially more flexibility because the relationships between independent and dependent variable are not assumed to be linear. In fact, we don't have to know a priori what type of predictive functions we will eventually need. From an estimation standpoint, the use of regularized, nonparametric functions avoids the pitfalls of dealing with higher order polynomial terms in linear models. From an accuracy standpoint, GAMs are competitive with popular learning techniques.

In this post, we will lay out the principles of GAM and show how to quickly get up and running in R. We have also put together an PDF that gets into more detail around smoothing, model selection and estimation.

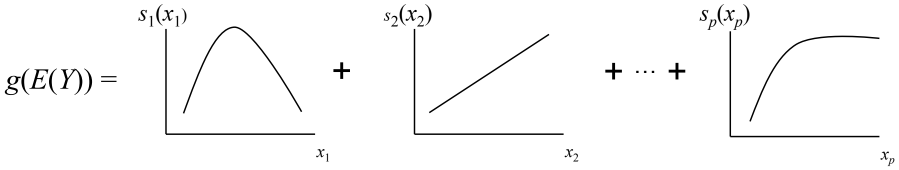
What is GAM?

Generalized additive models were originally invented by Trevor Hastie and Robert Tibshirani in 1986 (see [1], [2]). The GAM framework is based on an appealing and simple mental model:

- Relationships between the individual predictors and the dependent variable follow smooth patterns that can be linear or nonlinear.

- We can estimate these smooth relationships *simultaneously* and then predict $g(E(Y)))$ by simply adding them up.

Mathematically speaking, GAM is an additive modeling technique where the impact of the predictive variables is captured through smooth functions which—depending on the underlying patterns in the data—can be nonlinear:



We can write the GAM structure as:

$$g(E(Y)) = \alpha + s_1(x_1) + \cdots + s_p(x_p),$$

where Y is the dependent variable (i.e., what we are trying to predict), $E(Y)$ denotes the expected value, and $g(Y)$ denotes the *link function* that links the expected value to the predictor variables $x_1, \ldots, x_p$.

The terms $s_1(x_1), \ldots, s_p(x_p)$ denote smooth, *nonparametric* functions. Note that, in the context of regression models, the terminology *nonparametric* means that the shape of predictor functions are fully determined by the data as opposed to *parametric* functions that are defined by a typically small set of parameters. This can allow for more flexible estimation of the underlying predictive patterns without knowing upfront what these patterns look like. For more details on how to create these smooth functions, see the section called “Splines 101” in the PDF.

Note that GAMs can also contain parametric terms as well as two-dimensional smoothers. Moreover, like generalized linear models (GLM), GAM supports multiple link functions. For example, when Y is binary, we would use the logit link given by

$$g(E(Y)) = \log \frac{P(Y=1)}{P(Y=0)}.$$

Why Use GAM?

As mentioned in the intro, there are at least three good reasons why you want to use GAM: interpretability, flexibility/automation, and regularization. Hence, when your model contains nonlinear effects, GAM provides a regularized and interpretable solution – while other methods generally lack at least one of these three features. In other words, GAMs strike a nice balance between the interpretable, yet biased, linear model, and the extremely flexible, “black box” learning algorithms.

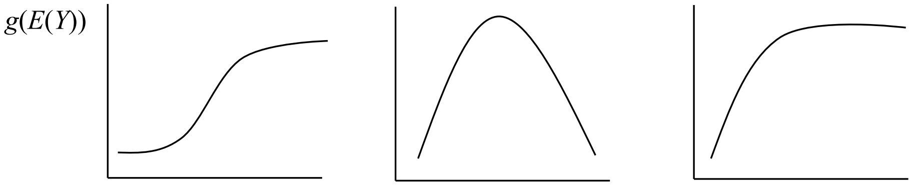
Interpretability

When a regression model is additive, the interpretation of the marginal impact of a single variable (the partial derivative) does not depend on the values of the other variables in the model. Hence, by simply looking at the output of the model, we can make simple statements about the effects of the predictive variables that make sense to a nontechnical person. For example, for the graphic illustration above, we can say that the (transformed) expected value of Y increases linearly as x_2 increases, holding everything else constant. Or, the (transformed) expected value of Y increases with x_p until x_p hits a certain point, etc.

In addition, an important feature of GAM is the ability to control the smoothness of the predictor functions. With GAMs, you can avoid wiggly, nonsensical predictor functions by simply adjusting the level of smoothness. In other words, we can impose the prior belief that predictive relationships are inherently smooth in nature, even though the dataset at hand may suggest a more noisy relationship. This plays an important role in model interpretation as well as in the believability of the results.

Flexibility and Automation

GAM can capture common nonlinear patterns that a classic linear model would miss. These patterns range from “hockey sticks” – which occur when you observe a sharp change in the response variable – to various types of “mountain shaped” curves:



When fitting *parametric* regression models, these types of nonlinear effects are typically captured through binning or polynomials. This leads to clumsy model formulations with many correlated terms and counterintuitive results. Moreover, selecting the best model involves constructing a multitude of transformations, followed by a search algorithm to select the best option for each predictor – a potentially greedy step that can easily go awry.

We don’t have this problem with GAM. Predictor functions are automatically derived *during* model estimation. We don’t have to know up front what type of functions we will need. This will not only save us time, but will also help us find patterns we may have missed with a parametric model.

Obviously, it is entirely possible that we can find parametric functions that look like the relationships extracted by GAM. But the work to get there is tedious, and we do not have 20/20 hindsight prior to model estimation.

Regularization

As mentioned above, the GAM framework allows us to control smoothness of the predictor functions to prevent overfitting. By controlling the wiggliness of the predictor functions, we can directly tackle the bias/variance tradeoff. Moreover, the type of penalties applied in GAMs have connections to Bayesian regression and l_2 regularization (see the PDF for details).

In order to see how this works, let's look at a simple, simulated example in R. We are simulating a dataset with 100 data points and two variables, x and Y. The *true* relationship between x and Y follows the sine function, but our data has normally distributed random errors.

```r
set.seed(3)
x <- seq(0,2*pi,0.1)
z <- sin(x)
y <- z + rnorm(mean=0, sd=0.5*sd(z), n=length(x))
d <- cbind.data.frame(x,y,z)
```

We want to predict Y given x by fitting the simple model:

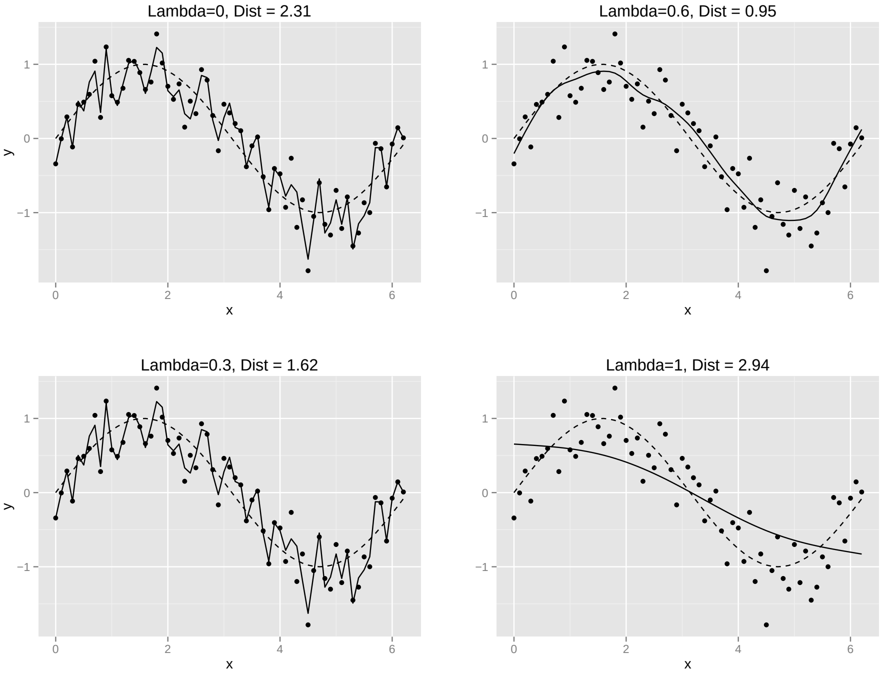
$$y = s_\lambda(x) + e,$$

where $s_\lambda(x)$ is some smooth function. The level of smoothness is determined by the *smoothing parameter*, which we denote by λ. The higher the value of λ, the smoother the curve. In the PDF, you can find a more details on how λ works to create smoothness as well as how to estimate $s(x)$. But for now, let's just think of $s(x)$ as a smooth function. For more details on smoothers, see the section called "Splines 101" in the PDF.

We fit the model above to the simulated data with four different values for λ. For each value of λ, we calculated the distance against the true function (the underlying sine curve). The results are shown in the charts below. The dots represent the actual data points, the punctuated line is the true curve, and the solid line is this smoother.

Clearly, the model with $\lambda = 0$ provides the best fit of the data, but the resulting curve looks very wiggly and would be hard to explain. Moreover, it has the highest distance to the sine curve, which means that it does not do a good job of capturing the true relationship. Indeed, the best choice in this case seems to be some intermediate value, like $\lambda = 0.6$.

Notice how the smoothing parameter allows us to *explicitly* balance the bias/variance tradeoff; smoother curves have more bias (in-sample error), but also less variance. Curves with less variance tend to make more sense and validate better in out-of-sample tests. However, if the curve is too smooth, we may miss an important pattern.



Smoothing 101

Smoothers are the cornerstones of GAM and hence a quick overview is in order before we get into model estimation. At a high level, there are three classes of smoothers used for GAM:

- Local regression (loess)
- Smoothing splines
- Regression splines (B-splines, P-splines, thin plate splines)

In general, regression splines are more practical. They are computationally cheap, and can be written as linear combinations of *basis functions* that do not depend on the dependent variable, Y, which is convenient for prediction and estimation.

Local Regression (loess)

Loess belongs to the class of *nearest neighborhood-based smoothers*. In order to appreciate loess, we have to understand the most simplistic member of this family: the *running mean smoother*.

Running mean smoothers are symmetric, moving averages. Smoothing is achieved by sliding a window based on the nearest neighbors across the data, and computing the average of Y at each step. The level of smoothness is determined by the width of the window. While appealing due to their simplicity, running mean smoothers have two major issues: they're not very smooth and they perform poorly at the boundaries of the data. This is a problem when we build predictive models, and hence we need more sophisticated choices, such as loess.

How Loess Works

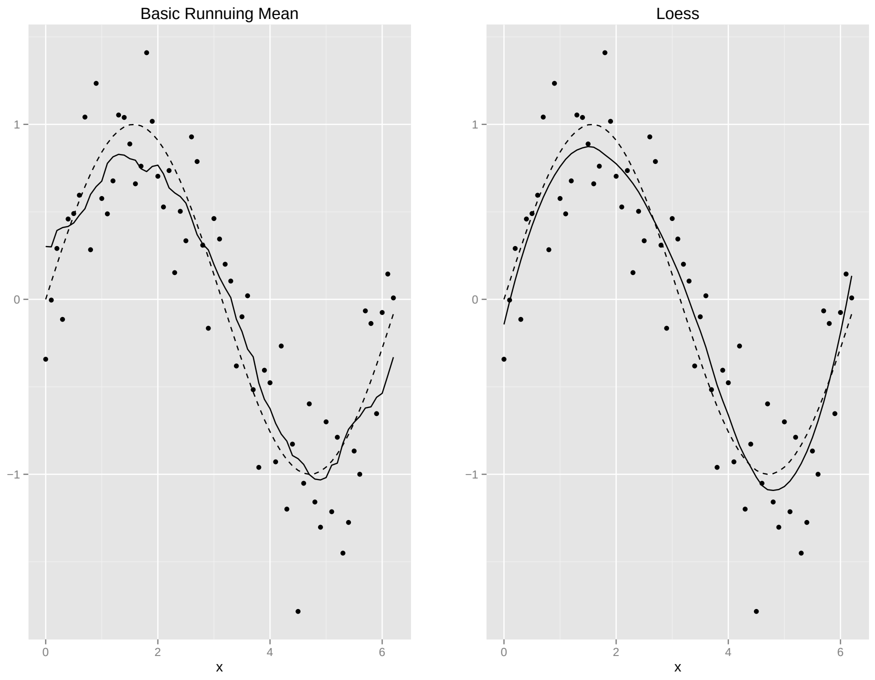
Loess produces a smoother curve than the running mean by fitting a weighted regression within each nearest-neighbor window, where the weights are based on a kernel that suppresses data points far away from the target data point. For example, to produce a loess-smoothed value for target data point x, loess deploys the following steps:

1. Determine smoothness using the *span* parameter. For example, if span = 0.6, each symmetric sliding neighborhood will contain 60% of the data – (30% to the left and 30% to the right).
2. Calculate $d_i = (x_i - x)/h$ where h is the width of the neighborhood. Create weights using the *tri-cube* function $w_i = (1 - d_i^3)^3$, if x_i is inside the neighborhood, and 0 elsewhere.
3. Fit a weighted regression with Y as the dependent variable using the weights from step 3. The fitted value at target data point x is the smoothed value.

Below is a loess smoother applied to the simulated data, loess function in R with a span of 0.6. As we can see, loess overcomes the issues with the running mean smoother:

Smoothing Splines

Smoothing splines take a completely different approach to deriving smooth curves. Rather than using a nearest-neighbor moving window, we estimate the smooth function by minimizing the penalized sum of squares



$$\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda \int (s''(x))^2 dx,$$

where the residual sum of squares

$$\sum_{i=1}^{n}(y_i - s(x_i))^2$$

ensures that we fit the observed data, while the *penalty term*

$$\lambda \int (s''(x))^2 dx$$

imposes smoothness (i.e., penalizes wiggliness).

Note that the penalty term imposes smoothness by calculating the integrated square of the second derivatives. Intuitively, this makes sense: the second derivative measures the slopes of the slopes. Thus, a wiggly curve will have large second derivatives, while a straight line will have second derivatives of 0. Hence we're essentially "adding up" the squared second derivatives to measure the wiggliness of the curve.

The tradeoff between model fit and smoothness is controlled by the *smoothing parameter*, λ. Clearly, the smoothing parameter operates in a different way than the span parameter in loess, which controls the width of the window, although they both serve the same ultimate purpose.

Interestingly, it turns out that the function that minimizes the penalized sum of squares is a *natural cubic spline* with *knots* at every data point, which is also known as a *smoothing spline*. However, for predictive modeling, smoothing splines have a major drawback: it is not practical to have knots at every data point when dealing with large models. Moreover, having knots at every data point only helps us in situations where we want to estimate wiggly functions with small values of λ. This is a rare use case for predictive modeling where we generally want to avoid overfitting. Thus, a smoothing spline is essentially wasteful, as the *effective degrees of freedom* used will be much smaller than the number of knots (due to the penalty).

Regression Splines

Regression splines offer a more practical alternative to smoothing splines. The main advantage of regression splines is that they can be expressed as a linear combination of a finite set of *basis functions* that do not depend on the dependent variable Y, which is practical for prediction and estimation.

We can write a regression spline of order q as

$$s(x) = \sum_{l=1}^{K} B_{l,q}(x)\beta_l = B'\beta,$$

where $B_{p,1}(x), \dots, B_{p,K}(x)$ are basis functions, B is the model matrix of basis functions, and $\beta = [\beta_1 : \beta_2 : \cdots : \beta_p]$ are the coefficients. The number of basis functions depends on the number of *inner knots* – a set of ordered, distinct values of x_j – as well as the order of the spline. Specifically, if we let m denote the number of inner knots, the number of basis functions is given by $K = p + 1 + m$.

How Regression Splines Work

To see how this works, let's try fitting a simple, non-penalized cubic *B-spline* to the data from the example above. This requires a total of $2(q + 1) + m$ knots, where the additional knots are $q + 1$ equal *boundary knots* and $q + 1$ equal lower boundary knots. The boundary knots are arbitrary as long as they are outside the inner knots. The equivalent knots at the boundaries are needed to ensure that the spline passes through the boundary knots.

Given a set of knots, $k_1, \dots, k_{(2(q+1)+m)}$, we can calculate the basis functions using a recursive formula (see [6])

$$
\begin{aligned}
B_{j,0}(x) = I(k_j \leq x < k_{j+1})\, B_{j,q}(x) \\
= \frac{x - k_j}{k_{j+q} - k_j} B_{j,i-1}(x) + \frac{k_{j+q+1} - x}{k_{j+q+1} - k_{j+1}}\, B_{j+1,q-1}(x).
\end{aligned}
$$

In this example we are taking the easy route by using quantiles to define the inner knots. The outer knots are set to the min and max of x:

```
min(x)
> 0
max(x)
> 6.2
quantile(x, probs=c(0.25, .50, .75))
> 1.55 3.10 4.65
```

Since this is a cubic spline, we only need the third order basis functions – i.e., $B_{3,1}, \ldots, B_{3,7}$. But, due to the recursive relationship, we cannot calculate these functions without first calculating the lower order basis functions.

Here is how you can use R to create basis functions and estimate their corresponding, non-penalized coefficients:

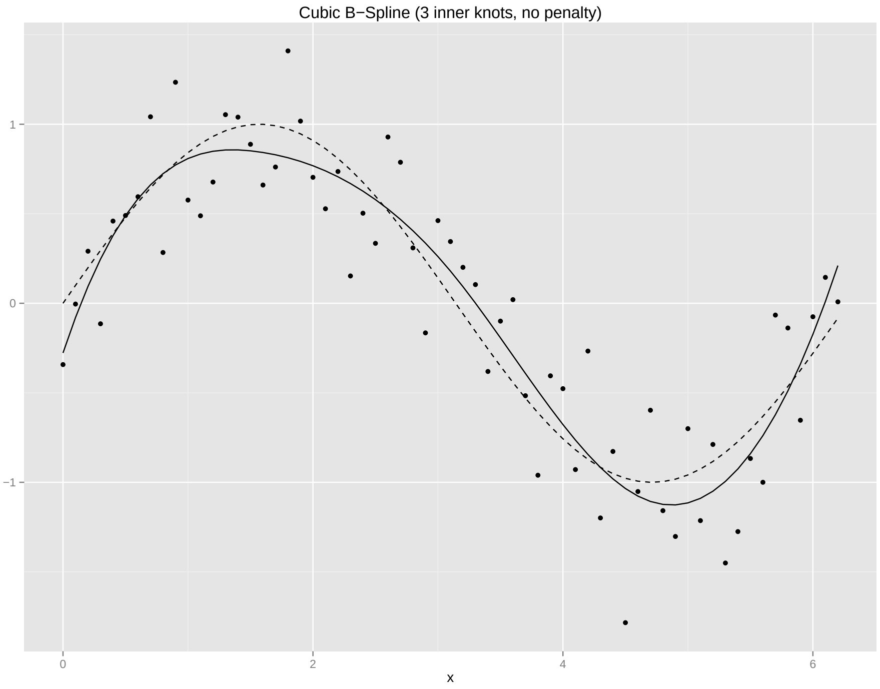
```
### Create basis functions
B <- bs(x, degree=3, intercept=TRUE, Boundary.knots=c(0, 6.2), knots=c(1.55, 3.10, 4.65))

### Get the coefficients
model <- lm(y~0 + B)

### The fitted values from the lm object are the smooth values
smoother <- lm$fitted
```

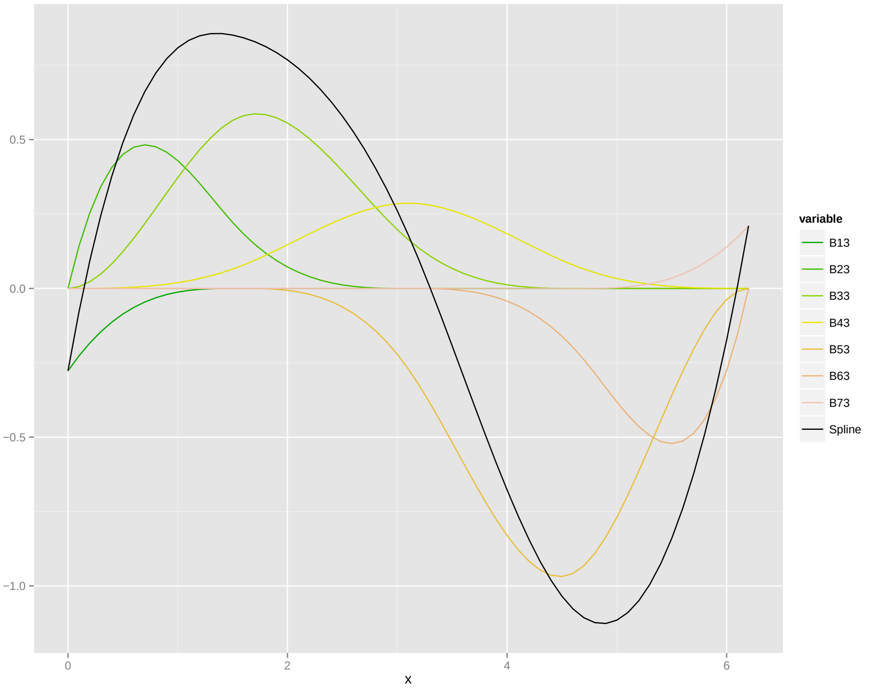
Generally, one does not need to worry too much about knot placement. Quantiles seem to work well in most cases (although more than three knots is usually required). For example, here we are getting a decent fit with only three inner knots based on arbitrarily chosen quantiles, see the graphic on page 10.

Last, but not least, plotting the basis functions, along with the final spline, helps illuminate what is going on behind the scenes. The plot below shows the basis functions multiplied by their respective coefficients – i.e., $B_{3,j}\, \beta_j$ – along with the final spline. It is easy to imagine how more knots, which means more basis functions, create a more flexible curve, see the graphic on page 10.

Penalized Regression Splines

In the simple example above, the only lever we have to control the smoothness of the B-spline is the number of knots – fewer knots translate into more smoothness.


Cubic B−Spline (3 inner knots, no penalty)
1
0
−1
0
2
4
6
x


0.5
0.0
−0.5
−1.0
0
2
4
6
x
variable
B13
B23
B33
B43
B53
B63
B73
Spline

However, we can apply a penalty when estimating the basis function coefficients to promote smoothness, just like we do with full smoothing splines. Thus, instead of solving for β_j with a standard linear model like we did above, we can minimize the *penalized sum of squares* to get the smoother for x

$$\min_{\beta} \left\{ ||y - B'\beta||^2 + \beta' P \beta \right\}.$$

Note that the coefficients applied to the basis functions are essentially amplifiers of the curvature of the spline. Hence, a popular way to penalize B-spline basis functions is to use *P-splines* which efficiently impose smoothness by directly penalizing the differences between adjacent coefficients. For example, for a P-spline, the penalty term can look like this:

$$\beta' P \beta = \sum_{l=1}^{K-1} (\beta_{l+1} - \beta_l)^2.$$

There are many other available choices for regression splines, but that is beyond the scope of this post. Typically, you don't need anything fancier than the splines covered here. For more on penalty matrices and different type of smoothers, see [3].

In the next section we will discuss how to minimize the penalized sum of squares for a model with more than one smoother – which is the ultimate use case we are after.

Estimating GAMs

As mentioned in the intro, GAMs consist of *multiple* smoothing functions. Thus, when estimating GAMs, the goal is to *simultaneously* estimate all smoothers, along with the parametric terms (if any) in the model, while factoring in the covariance between the smoothers. There are two ways of doing this:

- Local scoring algorithm.
- Solving GAM as a large GLM with penalized iterative reweighted least squares (PIRLS).

For details on GAM estimation, see the "Estimation" section in the PDF.

In general, the local scoring algorithm is more flexible in the sense that you can use any type of smoother in the model whereas the GLM approach only works for regression splines (see the "Smoothing 101" section in the PDF). However, the local scoring algorithm is computationally more expensive and it does not lend itself as nicely to automated selection of smoothing parameters as the GLM approach.

When fitting a GAM, the choice of *smoothing parameters* – i.e., the parameters that control the smoothness of the predictive functions – is key for the aesthetics and fit of the model. We can choose to pre-select the smoothing parameters or we may choose to estimate the smoothing parameters from the data. There are two ways of estimating the smoothing parameter for a logistic GAM:

- Generalized cross validation criteria (GCV).
- Mixed model approach via restricted maximum likelihood (REML).

REML only applies if we are casting GAM as a large GLM. Generally the REML approach converges faster than GCV, and GCV tends to under-smooth (see [3], [9]).

Penalized Likelihood

For both local scoring and the GLM approach, the ultimate goal is to maximize the penalized likelihood function, although they take very different routes. The penalized likelihood function is given by

$$2l(\alpha, s_1(x_1), \dots, s_p(x_p)) - \text{penalty},$$

where $l(\alpha, s_1, \dots, s_p)$ is the standard log likelihood function. For a binary GAM with a logistic link function, the penalized likelihood is defined as

$$l(\alpha, s_1(x_1), \dots, s_p(x_p)) = \sum_{i=1}^{n}(y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i)).$$

$$\hat{p}_i = \left(1 + \exp(-\hat{\alpha} - \sum_{j=1}^{p} s_j(x_{ij}))\right)^{-1}.$$

where $\hat{p}_i$ is given by

$$\hat{p}_i = P(Y=1|x_1, \dots, x_p) = \left(1 + \exp(-\hat{\alpha} - \sum_{j=1}^{p} s_j(x_{ij}))\right)^{-1}.$$

The penalty can, for example, be based on the second derivatives

$$\text{penalty} = \sum_{j=1}^{p} \lambda_j \int (s_j''(x_j))^2 dx.$$

The parameters, $\lambda_1, \ldots, \lambda_p$, are the aforementioned *smoothing parameters* which control how much penalty (smoothness) we want to impose on the model. The higher the value of λ_j, the smoother the curve. These parameters can be preselected or trained from the data. See the “Estimation” section of the PDF for more details.

Intuitively, this type of penalty function makes sense: the second derivative measures the slopes of the slopes. This means that wiggly curve will have large second derivatives, while a straight line will have second derivatives of 0. Thus we can quantify the total wiggliness by “adding up” the squared second derivatives.

Local Scoring Algorithm

The local scoring algorithm is an extension of the backfitting algorithm, which in turn is based on the Gauss-Seidel procedure for solving linear systems.

This is an iterative procedure with multiple nested loops. The backfitting/Gauss-Seidel framework can be used to solve a wide array of messy systems, and does not require calculation of derivatives. Here is how it works (see [2]):

Step 1: Set all smooth functions to 0, i.e., $s_j(x_j) \equiv 0$

Step 2: Cycle through variables to get the smooth functions

First, define the estimated log-odds for observation i , $i = 1, \ldots, n$, as

$$\nu_i = \hat{\alpha} + \sum_{j=1}^{p} s_j(x_{ij}),$$

and then construct the *pseudo dependent variable*

$$z_i = \nu_i + \frac{y_i - \hat{p}_i}{\hat{p}_i(1 - \hat{p}_i)},$$

as well as the weights

$$w_i = \hat{p}_i(1 - \hat{p}_i).$$

To get the function $s_1(x_1)$ simply smooth the pseudo dependent variable z against x_1, using the weights defined above. We can then do the same thing for x_2, after adjusting $\nu, \hat{p}, w$ to account for the change to $s_1(x_1)$. Same goes for $x_3, \ldots, x_p$.

Note that cycling through the predictor to get the weighted smoothers requires an extra layer of iterations because the weights change at every iteration. Hence, there are a significant amount of computations to be done.

Step 3: Repeat step 2 until the functions converge.

Solving GAM as a Large GLM

The basic idea here is to recast a GAM as a parametric, penalized GLM. The GLM approach is a more direct approach as it reduces step 2 of the local scoring to a single step where the coefficients are estimated simultaneously. Moreover, it comes with the properties of the battle-tested GLM framework.

Casting GAM as a Large GLM

As discussed above, regression splines can be written as a linear combination of the basis functions and the corresponding coefficients. For example, the spline for predictor x_j can be written as

$$s_j(x_j) = B_j'\beta_j,$$

where B_j is a matrix of basis functions for x_j, and $\beta_j = [\beta_{j,1}, \ldots, \beta_{1,K_j}]$ denotes the corresponding coefficients.

If we create the pseudo dependent variable z as in step 2 of the local scoring algorithm, and the corresponding weighting matrix $W = \text{diag}(w_1, \ldots, w_p)$, we can get the coefficients by minimizing the *penalized sum of squares* given by

$$||\sqrt{W}(z - \boldsymbol{B}'\boldsymbol{\beta})||^2 + \boldsymbol{\beta}' P \boldsymbol{\beta}.$$

Here $\boldsymbol{B} = [B_1 : B_2 : \cdots : B_p]$ is the overall *model matrix*, which contains all basis functions and hence will have $\sum K_l$ columns and n rows. The overall penalty matrix $\boldsymbol{P}$ is a block diagonal matrix defined as

$$\boldsymbol{P} = \text{block diag}(\lambda_1 P_1, \ldots, \lambda_p P_p).$$

Penalized Re-weighted Iterative Least Squares (PIRLS)

On the surface, this looks like a garden-variety, penalized least squares problem. However, z and W are not static since they depend on the fitted values (estimated probabilities). Hence, we need an iterative approach to get $\hat{\boldsymbol{\beta}}$. This approach is called penalized re-weighted iterative least squares (PIRLS) which is an extension of the widely used iterative re-weighted least squares (IRLS) used for GLMs. The coefficient estimates at iteration k are given by

$$\hat{\boldsymbol{\beta}}_{(k+1)} = (\boldsymbol{B}' W_{(k)} \boldsymbol{B} + \boldsymbol{P})^{-1} \boldsymbol{B}' W_{(k)} z_{(k)},$$

and we continue to increment k until we reach convergence. Note that the iteration subscript applied to W and z indicate that they both depend on the estimated probabilities $\hat{p}$ which change at every iteration.

Bayesian Interpretation

It turns out that the penalties imposed by GAM are equivalent to treating the coefficients of the large GLM as random variables with the normally distributed priors (see [9], [10])

$$\boldsymbol{\beta} \sim N\left(0, \sigma^2 \boldsymbol{P}^{-1}\right).$$

This connection makes sense: when we estimate *smooth* functions in a GAM we are essentially imposing the *prior belief* that the predictive functions follow smooth patterns. Moreover, the fact that GAM is a regularized model means that GAMs have some level of built-in fortification against multicollinearity.

Note that if we have no penalty – i.e., if $\lambda_j \to 0$ for all j – the variance goes to infinity and we get a "free" estimate. Conversely, if we set all $\lambda_j = \infty$, all coefficients are shrunk to 0. Moreover, if the penalty matrix is an identity matrix, which means that we are simply "muting" the splines in a blanket fashion, we get ridge regression (L2 regularization).

Effective Degrees of Freedom

For a regular GLM, the degrees of freedom equal the number of parameters to be estimated, which can also be calculated by taking the trace of the hat matrix:

$$\text{model df} = \text{tr}(H) = \text{tr}\left(X(X'X)^{-1}X'\right).$$

We can follow the same idea for a GAM and calculate the *effective degrees of freedom* as

$$\text{model edf} = \text{tr}\left(\boldsymbol{B}(\boldsymbol{B}'W\boldsymbol{B} + \boldsymbol{P})^{-1}\boldsymbol{B}'W\right).$$

If there is no penalty, the model is a regular large GLM and the trace is the number of parameters. However, the number of effective degrees of freedom will decrease as the as the smoothing parameters, $\lambda_1, \ldots, \lambda_p$, increase.

This is important, as GAM software packages will show effective degrees of freedom as part of the model output.

Selecting the Smoothing Parameter

Until now, we have assumed that the smoothing parameters are pre-determined. However, although a λ of 0.6 seems to work quite well for most models, we can determine the smoothing parameters from the data. It is more costly from a computational perspective, but more appealing from a scientific perspective.

At a high level, there are two ways of estimating the smoothing parameter for a logistic GAM:

- Generalized cross validation criteria (CGV).
- Mixed model approach via restricted maximum likelihood (REML).

REML only applies if we are casting GAM as a large GLM. Generally the REML approach converges faster than GCV, and GCV tends to under-smooth (see [3], [9]).

Optimizing the smoothing parameters requires an extra layer of iterations. This is either done through an *inner loop* or an *outer loop*. The inner loop approach optimizes the smoothing parameter within each iteration of PIRLS, while the outer loop runs a full PIRLS for each trial vector $\lambda = \lambda_1, \ldots, \lambda_p$. In the mgcv R package, which we will discuss later, these two approaches are referred to as *performance iterations* and *nested iterations*, respectively.

Generalized Cross Validation Criteria (GCV)

This is based on a "leave one out" cross-validation approach. The strategy is to remove one data point at a time, fit a smoother to the remaining data, and then fit of the smoother against the entire dataset. The goal is to pick the λ_j that minimizes the average error across all n validations.

Fortunately, it turns out that we do not have to fit n smoothers to achieve this (for details see [3], [7]). In fact, for a logistic GAM, we can use the GCV statistic:

$$\text{GCV} = \frac{n||\sqrt{W}(z - \boldsymbol{B}'\boldsymbol{\beta})||^2}{(n - \text{tr}(H))^2},$$

where H is the hat matrix and $\boldsymbol{B}$ is the model matrix consisting of basis functions. This statistic essentially calculates the error of the model and adjusts for the degrees of freedom and is a linear transformation of the AIC statistic. Hence we can use this statistic for model comparison in general, not just selection of smoothing parameters.

When running GAMs in R, you will see the UBRE score in outputs. This is essentially the GCV score when the scale parameter of the distribution of the response variable is known. For example, when Y is binomial where the scale parameter is known, you will see the UBRE score used in outputs. See [3] for more details.

REML

Since GAM has a Bayesian interpretation (see the section on Bayesian Interpretation above), we can treat it like a standard mixed model by separating out the fixed effects and estimating the smoothing parameters as variance parameters. (Note that the variance of the coefficients depend on P, which in turn depends on $\lambda = \lambda_1, \ldots, \lambda_p$.)

Here is how it works: the restricted likelihood function, given the vector of smoothing parameters, $\boldsymbol{\lambda}$, is obtained by integrating out ***beta*** from the joint density of the data and the coefficients

$$l_r(\hat{\boldsymbol{\beta}}, \lambda) = \int f(y|\boldsymbol{\beta}) f(\boldsymbol{\beta}) d\boldsymbol{\beta}.$$

The restricted likelihood function depends on λ and the estimates $\hat{\boldsymbol{\beta}}$ (through the penalty), but not the random parameters $\boldsymbol{\beta}$. Thus we can use this function to derive trial vectors for λ for a nested PIRLS iteration:

1. Given a trial vector λ, estimate $\boldsymbol{\beta}$ using PIRLS.
2. Update $\boldsymbol{\lambda}$ by maximizing the restricted log likelihood.
3. Repeat steps 1 and 2 until convergence.

For more details see [4] and [10].

Variable Selection

GAM is not the type of modeling technique where you can leisurely throw in hundreds of variables to see what "sticks." GAMs are somewhat parameter-heavy, which can carry nontrivial computational costs for large models. Moreover, GAM is not immune to multicollinearity, which means that the "kitchen sink" approach can lead to strange results.

Hence it is advised to pre-screen the candidate predictors before we run a GAM. In other words, we need a way to measure the "univariate" strength of each predictor and use this information to remove variables that are never going to contribute in any meaningful way.

Variable Pre-Screening With the Information Value

Perhaps the most powerful framework for exploring the univariate strength of a predictive variable, within the context of a binary regression model, is an approach that has close ties to the Kullback-Leibler divergence: the *information value* (IV) and *weight of evidence* (WOE).

This framework has a number of appealing features:

- Detect any relationship between a predictor x_j and a dependent variable Y, whether it is nonlinear or linear.

- Visualize the relationship between x_j and Y.
- Assess the predictive value of missing values.
- Seamlessly compare the strength of continuous and categorical variables.

The WOE/IV framework is based on the following relationship:

$$\log \frac{P(Y=1|x_j)}{P(Y=0|x_j)} = \log \frac{P(Y=1)}{P(Y=0)} + \log \frac{f(x_j|Y=1)}{f(x_j|Y=0)}.$$

This is saying that the conditional logit, given x_j, can be written as the overall log-odds (the "intercept") plus the *log-density ratio* – also known as the *weight of evidence* (WOE). Thus, when we are fitting a logit model, we are – whether we like it or not – attempting to estimate WOE. Hence, for pre-modeling visualization, we can plot WOE against x_j in order to visualize how x_j affects Y in a univariate setting.

Measuring Univariate Variable Strength

We can leverage WOE to measure the predictive strength of x_j – i.e., how well it helps us separate cases when $Y = 1$ from cases when $Y = 0$. This is done through the *information value* (IV), which is defined like this:

$$\text{IV} = \int \log \frac{f(x_j|Y=1)}{f(x_j|Y=0)} \left(f(x_j|Y=1) - f(x_j|Y=0)\right) dx.$$

Note that the IV is essentially a weighted sum of all the individual WOE values where the weights incorporate differences between the numerators and the denominators. Generally, if IV < 0.05, the variable has very little predictive power.

Estimating WOE

The most common approach to estimating the conditional densities needed to calculate WOE is to bin x_j and then use a histogram-type estimator.

Here is how it works: create a $k \times 2$ table where k is the number of bins, and the cells within the two columns count the number of records where $Y = 1$ and $Y = 0$, respectively. The conditional densities are then obtained by calculating the "column percentages" from this table. The typical number of bins used is 10-20. If x_j is categorical, no binning is needed and the histogram estimator can be used directly.

If $B_1, \ldots, B_k$ denote the bins for x_j, the WOE for bin i can be written as

$$\text{WOE}(x_j)_i = \log \frac{P(X_j \in B_i|Y=1)}{P(X_j \in B_i|Y=0)},$$

and the IV is the weighted sum of the k WOE values

$$\mathrm{IV}(x_j) = \sum_{i=1} k(P(X_j \in B_i|Y=1) - P(X_j \in B_i|Y=0)) \times \mathrm{WOE}(x_j)_i$$

You can use the Information R package to do these calculations. The package also allows for cross validation of IV and can be downloaded from this repository: https://github.com/klarsen1/Information

Multivariate Variable Selection

Pre-screening of variables is simply a way to reduce the pool of variables we want to search from. It does not dictate the final model. Whether or not we choose to perform variable screening, chances are that we still need to apply some sort of "multivariate" selection of variables inside GAM to select the final model – just like we would with any other type of regression model.

There are two available approaches to variable selection in R:

1. Stepwise selection (forward and backward)
2. Shrinkage

Stepwise selection needs no explanation. The spirit of the shrinkage approach is similar to lasso regression, although the paradigm of these two approaches is quite different. The basic idea is to add additional penalties to the model, by adding constants to the diagonal of the penalty matrix, in order to "shrink" smoothers that are very wiggly (during REML) to 0. Hence, instead of physically removing variables in a stepwise fashion, we are leaving weak smoothers in the model with coefficients near 0. For more information, see [3] and the mgcv documentation.

Fitting GAMs in R

The two main packages in R that can be used to fit generalized additive models are gam and mgcv. The gam package was written by Trevor Hastie and closely follows the theory outlined in [2]. The mgcv package was written by Simon Wood, and, while it follows [2] in many ways, it is much more general because it considers GAM to be any penalized GLM (for more details, see [3]).

The differences are described in detail in the documentation for mgcv. Here is a cheat sheet:

Component	**gam**	**mgcv**
Confidence intervals	Frequentist	Bayesian

Component	**gam**	**mgcv**
Splines	Smoothing splines and loess	Does not support loess or smoothing splines, but supports a wide array of regression splines (P-splines, B-splines, thin plate splines, tensors) + tensors
Parametric terms	Supported	Supported, and you can penalize or treat as random effects
Variable selection	Stepwise selection	Shrinkage
Optimization	Local scoring	PIRLS
Selecting smoothing parameters	No default approach	Finds smoothing parameters by default. Supports both REML and GCV
Large datasets	Can parallelize stepwise variable selection with the doMC package	Special bam function for large datasets. Can also parallelize certain operations in the gam function through openMP
Missing values	Clever approach to dealing with missing values through na.action=gam.replace	No special treatment. Omits observations with missing values
Multi dimensional smoothers	Supported with loess	Supported with tensors and thin plate splines
Model diagnostics	Standard GAM diagnostics	Standard GAM diagnostics + the concurvity measure which is a generalization of collinearity

gam and mgcv do not work well when loaded at the same time. Restart the R session if you want to switch between the two packages – detaching one of the packages is not sufficient.

Here is an example of how to fit a GAM in R:

```
### GAM example using mgcv

library(mgcv)
library(ggplot2)
# fake data
n <- 50
sig <- 2
```

```
dat <- gamSim(1,n=n,scale=sig)

# P-spline smoothers (with lambda=0.6) used for x1 and x2; x3 is parametric.
b1 <- mgcv::gam(y ~ s(x1, bs='ps', sp=0.6) + s(x2, bs='ps', sp=0.6) + x3, data = dat)
summary(b1)
plot(b1)


# plot the smooth predictor function for x1 with ggplot to get a nicer looking graph
p <- predict(b1, type="lpmatrix")
beta <- coef(m)[grepl("x1", names(coef(b1)))]
s <- p[,grepl("x1", colnames(p))] %*% beta
ggplot(data=cbind.data.frame(s, dat$x1), aes(x=dat$x1, y=s)) + geom_line()


# predict
newdf <- gamSim(1,n=n,scale=sig)
f <- predict(b1, newdata=newdf)


# select smoothing parameters with REML, using P-splines
b2 <- mgcv::gam(y ~ s(x1, bs='ps') + s(x2, bs='ps') + x3, data = dat, method="REML")

# select variables and smoothing parameters
b3 <- mgcv::gam(y ~ s(x0) + s(x1) + s(x2) + s(x3) , data = dat, method="REML", select=TRUE)

# loess smoothers with the gam package (restart R before loading gam)
library(gam)
b4 <- gam::gam(y ~ lo(x1, span=0.6) + lo(x2, span=0.6) + x3, data = dat)
summary(b4)
```

Comparing GAM Performance With Other Techniques

Business Problem

We will be using a marketing example from the insurance industry (source undisclosed). The data contains information on customer responses to a historical direct mail marketing campaign. Our goal is to improve the performance of future waves of this campaign by targeting people who are likely to take the offer. We will do this by building a “look-alike” model to predict the probability that a given client will accept the offer, and then use that model to select the target audience going forward[1f].

Obviously, we want a model that is accurate so that we can find the best possible target audience. In addition, we want to be able to provide insights from the model, such as *partial impact* charts, that show how the average propensity changes across various client features. We want to make sure that the relationships we find stand to reason from a business perspective.

Data

The dataset has 68 predictive variables and 20k records. For modeling and validation purposes, we split the data into 2 parts:

- 10k records for training. This dataset will be used to estimate models.
- 10k records for testing. This dataset will be kept in a vault to the very end and used to compare models.

The success of the model will be based on its ability to predict the probability that the customer takes the offer (captured by the PURCHASE indicator), for the validation dataset.

Most variables contain credit information, such as number of accounts, active account types, credit limits, and utilization. The dataset also captures the age and location of the individuals.

Let's return to our marketing case study. Recall that we are trying to predict whether a person takes a direct marketing offer. Hence, we are trying to build a GAM that looks like this:

$$\log \frac{P(\text{convert})}{1 - P(\text{convert})} = s_1(x_1) + \cdots + s_p(x_p) + x'\beta$$

where
(x'β
)$areparametricterms(dummyvariablesinourcase)$.

Model Comparison Strategy

We built six models with six different techniques using the training dataset. The models were then validated against the validation dataset. The area under the ROC curve was used to evaluate model performance.

In order to make the comparison as fair as possible, we used the same set of variables for each model. The variables were selected using the following procedure:

1. Remove all variables with an information value (IV) less than 0.05. (See the PDF for more details on IV.). You can use the Information Package to calculation information values.

2. Eliminate highly correlated variables using variable clustering (ClustOfVar package). We generated 20 clusters and picked the variable with the highest IV within each cluster.

Obviously, we could have used variable selection techniques inside GAM as described above, but we wanted to use the same 20 variables for each model.

List of the seven models tested:

1. Random forest with 100 trees using the openMP enabled randomForestSRC package.
2. GAM (mgcv) using P-splines with smoothing parameters of 0.6 for all variables (except dummy variables).
3. Same as #2, but optimal smoothing parameters are selected with REML (instead of using 0.6 for all variables).
4. Same as #2, but optimal smoothing parameters are selected with REML (see the PDF for details) and weak variables are shrunk towards 0 using selection=TRUE in mgcv.
5. SVM built with the e1071 package, using a Gaussian radial kernel.
6. KNN classifier with k=100. Distance metrics were weighted using an Epanechnikov kernel. See the kknn package for more details.
7. Linear logistic regression model.

The code used can be downloaded from the github repository.

Testing Results

Model	Validation AUROC	Estimation Time	Scoring Time
Random forest	0.809	6.39	39.38
GAM, lambda=0.6	0.807	3.47	0.52
GAM, estimate lambdas	0.815	42.72	0.29
GAM, estimate lambdas, extra shrinkage	0.814	169.73	0.33
SVM	0.755	13.41	1.12
Linear logit	0.800	0.1	0.006
KNN with K=100	0.800	NA	3.34

Note that in order to get the AUROC for the SVM, we used the enhanced version of Platt's method to convert class assignments into probabilities (to get a continuous measure, see [11]). Settings for KNN and SVM were based on trying different combinations.

As we can see, GAM performs well compared to the other methods. Obviously, this test is based on a single dataset, so no universal conclusions can be drawn, but the dataset has enough correlation and "chunky" variables to make the results relevant.

The GAM models where smoothing parameters were automatically selected with REML perform better than the model where we used a flat smoothing parameter of 0.6 across all variables (which tends to work well for most models). However, in this example, the models with automatic selection also tend to produce more wiggly functions than the model with $\lambda = 0.6$ across all variables. For a targeting model, the additional wiggliness is not worth the loss of model intuition.

The biggest surprises in this test are the performances of SVM and the linear logit model. The linear logit model is performing surprisingly well given that the strongest variable in the model (N_OPEN_REV_ACTS) is not linearly correlated with the log odds of success (PURCHASE). The reason could be that, while this relationship is not linear, it is monotonic. Also, the AUROC is based on the estimated probability, which is indeed not linear in the predictive variables due to the Sigmoidal transformation $(1 + exp(-\nu))^{-1}$. SVM on the other hand, is performing surprisingly poorly. However, it should be mentioned that the author of this post has very little experience with SVM, which could be a disadvantage for SVM. Also, the Pratt conversion of SVM classification to probabilities could play a role here.

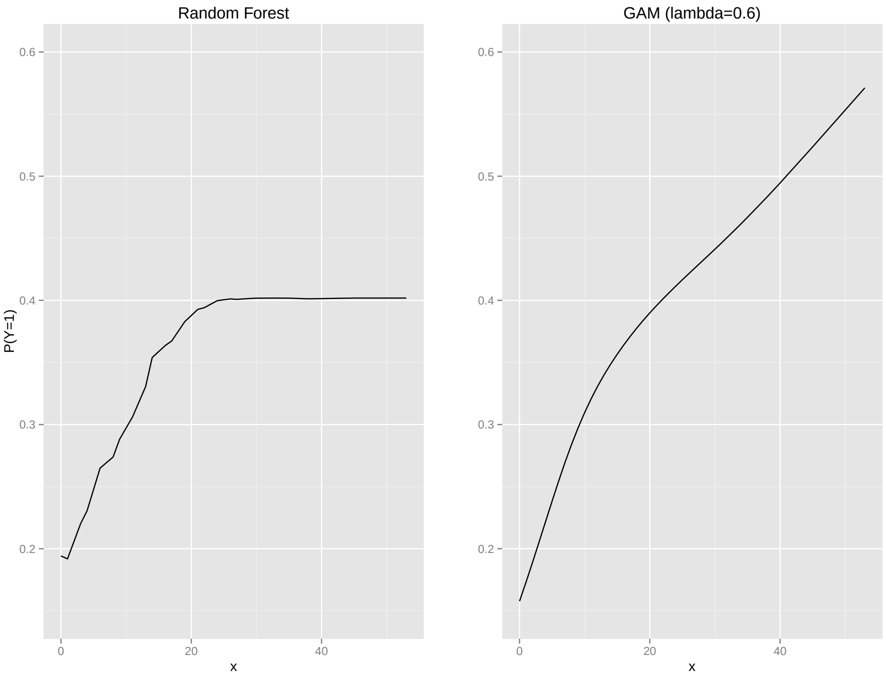
Partial Relationships

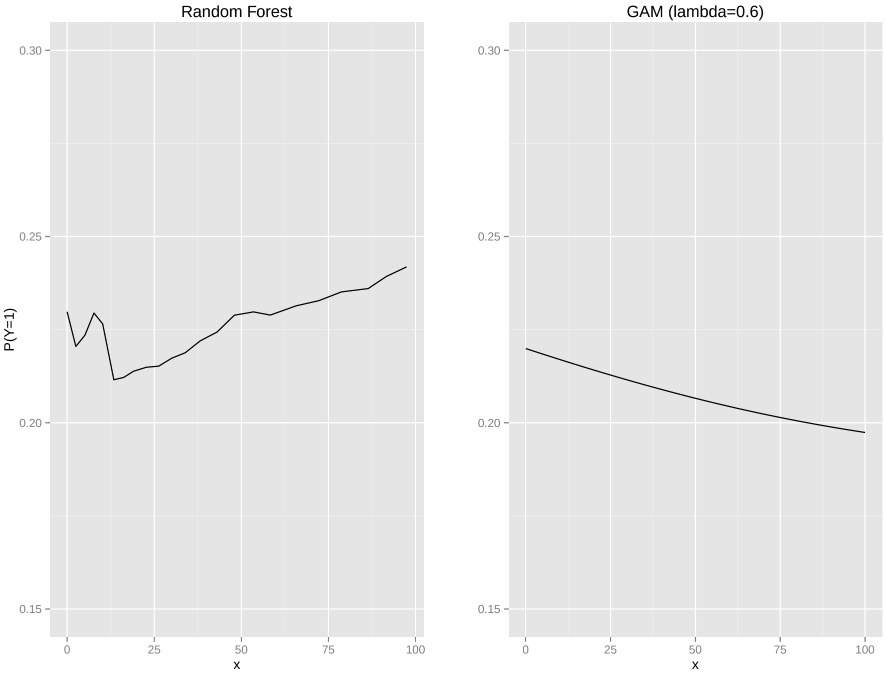
As stated earlier, an important part of this modeling exercise is to look at the partial relationships between the binary dependent variable (PURCHASE) and the predictors.

This is shown below for the variable N_OPEN_REV_ACTS (number of open revolving accounts) for random forest and GAM. Note that for the random forest model, these plots are generated by sending different values of x_j (in our case 20) through the forest and getting the estimated probabilities at each value of x_j. For GAM, we simply plot the final regression spline.

Note that, unlike GAM, random forest does not try to promote smoothness. This clearly shows in the chart below, as the GAM-based predictive function is smoother than the one from random forest. However, the GAM model does some potentially dangerous interpolation beyond $x = 20$ where the data is thin. (only 1.64% of the sample have N_OPEN_REV_ACTS>20 although the conversion rate for this group is 2.3 times higher than the average).

And here are the partial impact plots for one the weakest variables. The random forest curve does not look very intuitive:


Random Forest
GAM (lambda=0.6)
P(Y=1)
0.6
0.5
0.4
0.3
0.2
0
20
40
x


Random Forest
GAM (lambda=0.6)
P(Y=1)
0.30
0.25
0.20
0.15
0
25
50
75
100
x

Final Words

As stated in the introduction, the purpose of this post is to get more data scientists to use GAM. Hopefully, after reading this post, you'll agree that GAM is a simple, transparent, and flexible modeling technique that can compete with other popular methods. The code in the github repository should be sufficient to get started with GAM.

Of course, GAM is no silver bullet; one still needs to think about what goes into the model to avoid strange results. In fact, random forest is probably the closest thing to a silver bullet. However, random forest is much more of a black box, and you cannot control smoothness of the predictor functions. This means that you cannot combat the bias variance tradeoff as directly as with GAMs or ensure interpretable predictor functions. For those reasons, every data scientist should make room in their toolbox for GAM.

References

[1] Hastie, Trevor and Tibshirani, Robert. (1990), Generalized Additive Models, New York: Chapman and Hall.

[2] Hastie, Trevor and Tibshirani, Robert. (1986), Generalized Additive Models, Statistical Science, Vol. 1, No 3, 297-318.

[3] Wood, S. N. (2006), Generalized Additive Models: an introduction with R, Boca Raton: Chapman & Hall/CRC

[4] Wood, S. N. (2004). Stable and efficient multiple smoothing parameter estimation for generalized additive models. Journal of the American Statistical Association 99, 673–686

[5] Marx, Brian D and Eilers, Paul H.C. (1998). Direct generalized additive modeling with penalized likelihood, Computational Statistics & Data Analysis 28 (1998) 193-20

[6] Sinha, Samiran, A very short note on B-splines, http://www.stat.tamu.edu/~sinha/research/note1.pdf

[7] German Rodrıguez (2001), Smoothing and Non-Parametric Regression, http://data.princeton.edu/eco572/smoothing.pd

[8] Notes on GAM By Simon Wood. http://people.bath.ac.uk/sw283/mgcv/tampere/gam.pdf

[9] Notes on Smoothing Parameter Selection By Simon Wood, http://people.bath.ac.uk/sw283/mgcv/tampere/smoothness.pdf

[10] Notes on REML & GAM By Simon Wood, http://people.bath.ac.uk/sw283/talks/REML.pdf

[11] Karatzoglou, Alexandros, Meyer, David and Hornik, Kurt (2006), Support Vector Machines in R, Journal of Statistical Software Volume 15, Issue 9, http://www.jstatsoft.org/v15/i09/paper

[12] “e1071” package, https://cran.r-project.org/web/packages/e1071/e1071.pdf

[13] “mgcv” package, https://cran.r-project.org/web/packages/mgcv/mgcv.pdf

[14] “gam” package, https://cran.r-project.org/web/packages/gam/gam.pdf

[15] “randomForestSRC” package, https://cran.r-project.org/web/packages/randomForestSRC/randomForestSRC.pdf

[1f] When we target clients with the highest propensity, we may end up preaching to the choir as opposed to driving uplift. But that is beyond the scope of this post.