Introduction to Nonparametric Regression: Basis Expansions, Regularization, Local Regression

Arnab Maity

# Introduction

So far, many of our regression and classification methods used linear combinations of the predictors variables. Both linear and logistic regression models as well as LDA rely on linear model. In general, the association between and may be nonlinear and non-additive.

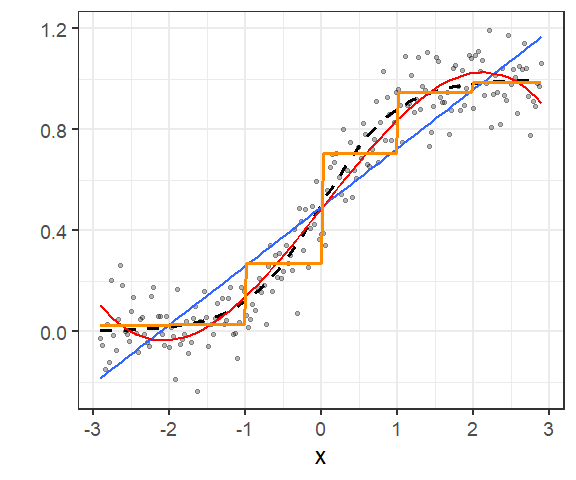
There are some straightforward ways to incorporate nonlinearity in the models. For example, we can include higher order terms such as , , etc. as predictors, and the resulting regression functions/classification boundaries will be nonlinear in the predictors. Polynomial regression is one such example where we posit

for a linear regression model, or

for a logistic regression model, where

Thus we are assuming that the true function is a *linear combination* of the monomial terms . These terms are examples of *basis functions.* In general, given a set of predictors , we may assume that there are there are functions such that the function can be written as or can be approximated as,

where are unknown coefficients. The functions are called *basis functions*. and the representation above is called a *linear basis expansion* in . Once we have specified the basis functions, we can simply use the linear model approach to fit this model.



Examples of a linear (blue), cubic (red) and piecewise constant (orange) fit to a simulated data set with one predictor. The true function is shown in black.

Some examples of basis representations are shoes below:

* Linear model in each predictors:
* Polynomial regressions: takes forms such as or .
* Nonlinear transformations: can take form such as or .
* Piecewise constant: . By breaking the range of up into such non-overlapping regions, we can model with a piecewise constant function.

The figure above shows examples of a linear, polynomial (3rd degree) and piecewise constant fit to a simulated data set with one predictor.

Depending on the problem at hand, we might use specific basis functions such as logarithms or power functions. Often we use the basis to achieve more flexible representations for – polynomial regression is an example of this. However, a drawback of polynomial regression is their global nature, that is, changing the coefficients to obtain a form in one region of the data might result in wild variations in the remote/boundary regions. A more useful approach is to consider families of *piecewise-polynomials* and *splines* that allow for local polynomial representations. Note that piecewise-constants are a special case of piecewise-polynomials with degree being set to zero.

In each of the approaches above, the number of basis functions has to be determined by the user. Thus the *number of basis functions* can be viewed as a tuning parameter. An alternative approach is to produce a *dictionary* consisting of typically a very large number of basis functions, along with the a method for *controlling the complexity* of our model. There are three common approaches:

* *Restriction methods*, where we decide before-hand to limit the class of functions. Generalized additive models (GAM) is an example, where we assume that our model has the form
* where are assumed to be smooth non-linear functions. It is called an additive model because we calculate a separate for each , and then add together all of their contributions. Each of these functions are then models using basis expansion:
* where is the number of basis functions used to model . The size of the model is limited by the number of basis functions used for each component function. Notice that the final form of is still linear in terms of .
* *Selection methods*, which adaptively scan the dictionary and include only those basis functions that contribute significantly to the fit of the model. Here the variable selection techniques discussed before (e.g., LASSO etc) are useful. There are other stage-wise greedy approaches as well – examples are classification and regression trees (CART), multivariate adaptive regression splines (MARS) and boosting.
* *Regularization methods*, where we use the entire dictionary but restrict the coefficients. Ridge and lasso regressions are a simple examples of a regularization approach. We will also discuss more sophisticated methods for regularization.

# Piecewise Polynomials and Splines

For simplicity, let us assume that we have only one predictor – we will relax this assumption in later sections when appropriate. Let us start with piecewise-constant basis functions: given a set of points in the domain of , define

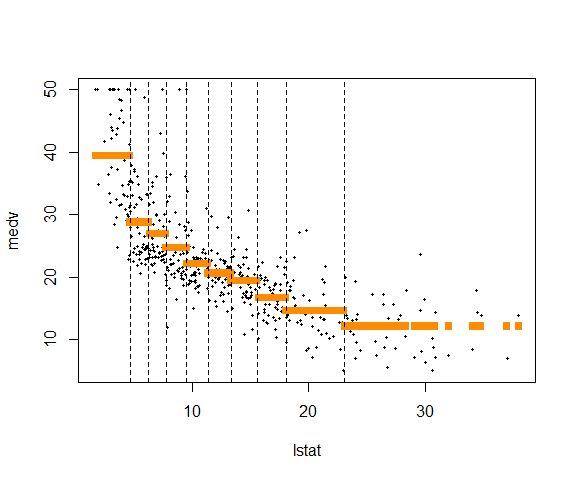
The boundaries of each regions, , are called *knots*.

For linear regression, we will then fit a model

Note that we have omitted from the model above. This is because it is redundant with the intercept. Each can be in only one region . Thus, only one of the will equal to one, and every other functions will be zero. Therefore, . Including the intercept will thus introduce multicollinearity. Since the regions are disjoint, the least squares estimate of is simply the sample mean of those ’s that have ’s in the -th region. Also note that when , then . Thus is the mean of the ’s with values in the first region.

To see the piecewise-constant fit in practice, let us revisit the Boston data, where we regress medv of lstat – see the figure below.

# Define region boundaries  
kn <- quantile(Boston$lstat,   
 probs = seq(.10, .90, by = .1))  
# Basis functions  
basis <- cut(Boston$lstat,   
 breaks = c(-Inf, kn, Inf))  
# Linear model fit and plot  
out <- lm(Boston$medv ~ basis)  
plot(Boston$lstat, Boston$medv,   
 pch = 19, cex = 0.4,  
 xlab = "lstat", ylab = "medv")  
points(Boston$lstat, out$fitted.values,   
 pch =15, col="darkorange")  
abline(v = kn, lty = 2)



Piecewise-constant fit for Boston data.

The vertical lines are the the knots. We can see that in each region, the estimated is constant.

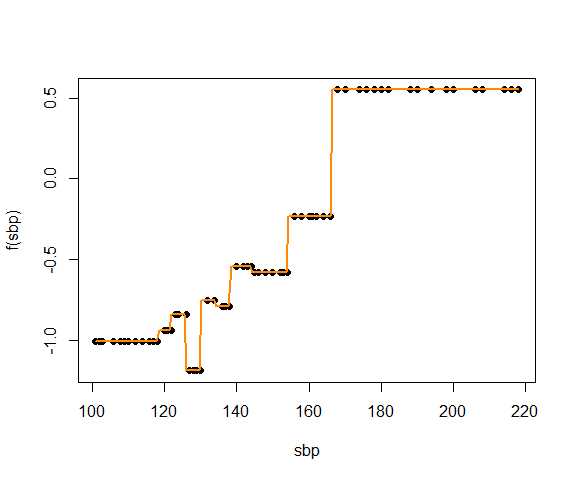
We can similarly fit a logistic regression model using the same basis functions as covariates. Consider the [South African Heart Disease data](https://raw.githubusercontent.com/harpreetSinghGuller/Data-Science-R/refs/heads/master/SAHeart.csv). The dataset is a subset of the Coronary Risk-Factor Study (CORIS) baseline survey. The study was carried out in three rural areas of the Western Cape, South Africa. The aim of the study was to establish the intensity of ischemic heart disease risk factors in that high-incidence region. The data represent white males between 15 and 64, and the response variable is the *presence or absence of myocardial infarction (MI)* at the time of the survey.

heart <- read\_csv("https://raw.githubusercontent.com/harpreetSinghGuller/Data-Science-R/refs/heads/master/SAHeart.csv")  
heart |>  
 dplyr::select(sbp, chd) |>  
 slice(1:4) |>  
 kable()

| sbp | chd |
| --- | --- |
| 160 | 1 |
| 144 | 1 |
| 118 | 0 |
| 170 | 1 |

We use chd as response and sbp (systolic blood pressure) as predictor. The figure below shows the fitted function.

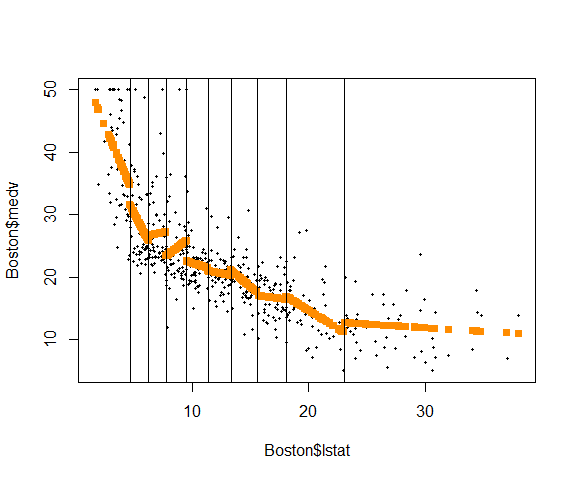
# Define region boundaries  
kn\_logit <- quantile(heart$sbp,   
 probs = seq(.1, .9, by = .1))  
# Basis functions  
basis\_logit <- cut(heart$sbp,   
 breaks = c(-Inf, kn\_logit, Inf))  
# Logistic model fit and plot  
out\_logit <- glm(heart$chd ~ basis\_logit,   
 family = binomial())  
# Grid for prediction  
xgrid <- seq(min(heart$sbp),   
 max(heart$sbp),  
 len = 201)  
ps <- cut(xgrid,   
 breaks = c(-Inf, kn\_logit, Inf))  
pred <- predict(out\_logit,   
 newdata = data.frame(basis\_logit = ps))  
# Plot the estimated function  
plot(heart$sbp, out\_logit$linear.predictors,   
 pch = 19,  
 xlab = "sbp", ylab = "f(sbp)")  
points(xgrid, pred,   
 pch =15, col="darkorange", type = "l", lwd=2)  
abline(v = kn, lty = 2)



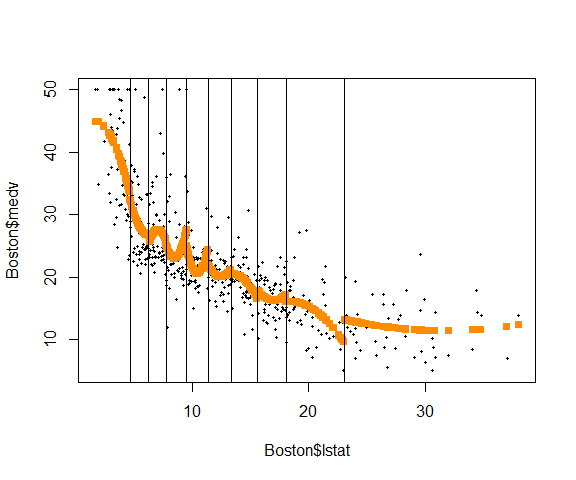
Piecewise-constant fit for SA heart data.

It is easy to see that we can extend this concept to piecewise-polynomial by positing a polynomial model for in each region. For example, the two figures below show piecewise-linear and piecewise-quadratic fits for Boston data.

# piecewise-linear fit  
outlin <- lm(Boston$medv ~ basis +   
 basis:Boston$lstat)  
# piecewise-quadratic fit  
outquad <- lm(Boston$medv ~ basis +   
 basis:Boston$lstat +   
 basis:I(Boston$lstat^2))



Piecewise-linear fit for Boston data.



Piecewise-quadratic fit for Boston data.

One drawback of this approach is that the estimated function is discontinuous at the boundaries of the regions, . Except in special cases, we would typically prefer piecewise-polynomials that are restricted to be continuous at the knots. For example, if we have only two regions, that is, only one knot , a piecewise-linear functions have the form

With the restriction that the overall function is continuous at , we have an additional condition that has the same value as at , that is,

Thus, instead of four original parameters , we will have *three* free parameters with the additional restriction of continuity.

It can be shown that a more direct way to proceed in this case is to use a basis that incorporates the continuity constraints,

where if , otherwise. In the general case with knots, , it can be shown that the continuity constraints can be incorporated by adding terms to the basis – for piecewise-linear functions we can use

We often prefer smoother functions, and these can be achieved by increasing the order of the local polynomial. For a -th degree piecewise-polynomial model with knots, we use the basis functions,

The terms are known as *truncated power basis functions*.

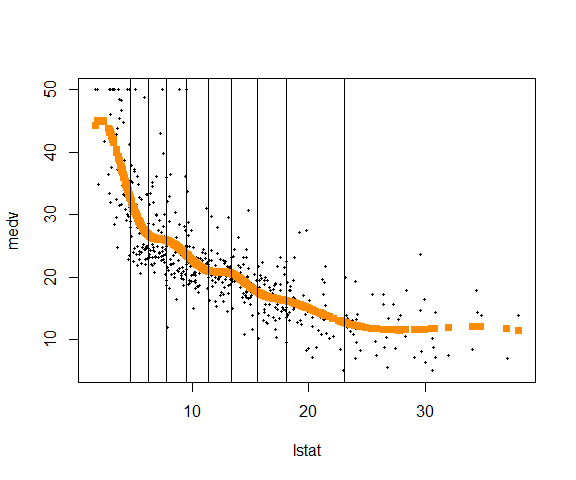
In spline literature, we also call the formulation above to have *order* . Thus piecewise-constant and linear splines have orders 1 and 2, respectively. Typically, we use piecewise-constant, linear and cubic (order-1, 2, and 4) splines in practice. Thus a order- spline with knots needs basis functions including intercept. The number (the total number of basis functions) is called *degrees of freedom*. For example, a cubic spline with 4 knots has degrees of freedom .

These splines with fixed knots are also known as *regression splines*. One needs to select the *order*/*degree* of the spline, the *number of knots* and their *placement*. One simple approach is to parameterize a family of splines by the number of basis functions, and have the observations determine the positions of the knots. Placing the knots at specific quantiles of observed values is often a reasonable approach.

Since the space of spline functions of a particular order and knot sequence is a vector space, there are many equivalent bases for representing them.[[1]](#footnote-1) While the truncated power basis is conceptually simple, it is not too attractive numerically: powers of large numbers can lead to severe rounding problems. The *B-spline basis* allows for efficient computations even when the number of knots is large.

In R, we can use the function bs() in the splines library. Below we fit a cubic spline to Boston data.

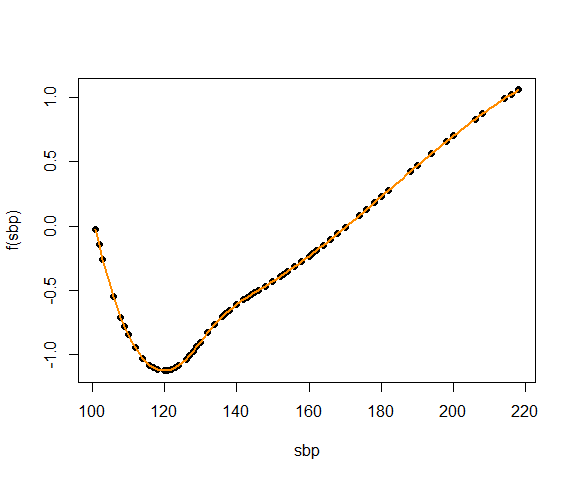
library(splines)  
# Define region boundaries  
kn <- quantile(Boston$lstat,   
 probs = seq(.10, .90, by = .1))  
# Cubic splines basis functions  
basis <- bs(Boston$lstat,  
 degree = 3,  
 knots = kn)  
# Linear model fit  
outbs <- lm(Boston$medv ~ basis)



Cubic spline fit of Boston data.

Similarly, we fit a cubic spline to the SA heart data below. We use two internal knots for this fit. The figure below shows the estimated function.

# Basis with 3 knots  
basis\_cubic <- bs(heart$sbp, df = 6)  
# logistic fit  
heart\_logit <- glm(heart$chd ~ basis\_cubic,   
 family = binomial())  
# Plot the estimated f of the observed data  
plot(heart$sbp, heart\_logit$linear.predictors,  
 xlab = "sbp", ylab = "f(sbp)", pch=19)  
# grid for prediction  
xnew <- seq(min(heart$sbp),   
 max(heart$sbp),   
 by = 1)  
ps <- predict(basis\_cubic, newx = xnew)  
# Estimated function  
fest <- cbind(1, ps) %\*% heart\_logit$coefficients  
lines(xnew, fest, lwd = 2, col = "darkorange")



Cubic spline fit with two knots of the SA heart data.

## Natural Cubic Splines

It is well known that polynomial fits can be erratic near boundaries. Also extrapolating beyond observed data range is generally not advisable. These issues are even more pronounced in spline fits. The polynomials fit beyond the boundary knots behave even more wildly than the corresponding global polynomials in that region.

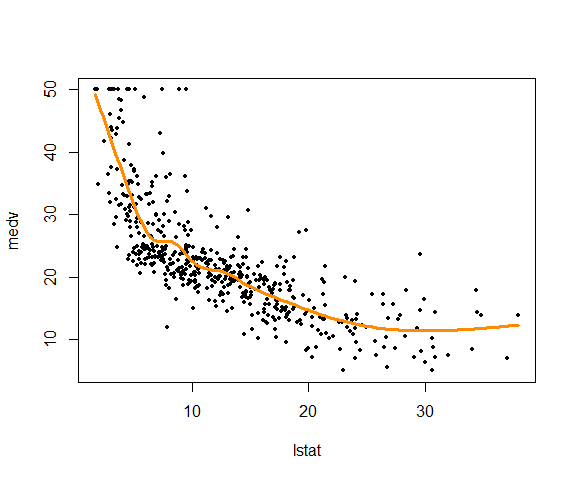
A *natural cubic spline* adds additional constraints: the function is *linear* beyond the boundary knots (in the region where is smaller than the smallest knot, or larger than the largest knot). This constraint frees up four degrees of freedom (two constraints each in both boundary regions). This can be used to include more knots in the interior region. This additional constraint means that natural splines generally produce more stable estimates at the boundaries.

A natural cubic spline with knots is represented by basis functions (equivalently, degrees of freedom is ). One can start from a basis for cubic splines, and derive the reduced basis by imposing the boundary constraints. For example, starting from the truncated power basis, the basis functions for natural cubic spline are

for , where

In R, we can use ns() in splines library to create natural cubic spline basis functions. For the Boston data, we fit natural cubic spline as follows.

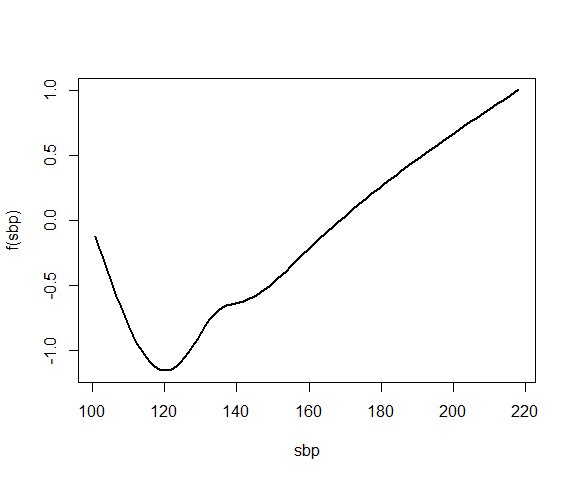
# Linear model fit  
boston\_ns <- lm(medv ~ ns(lstat, df = 9),   
 data = Boston)  
# Grid for prediction  
xgrid <- seq(min(Boston$lstat), max(Boston$lstat),   
 len = 101)  
pred <- predict(boston\_ns,   
 newdata = data.frame(lstat = xgrid))  
# Estimated function plot  
plot(Boston$lstat, Boston$medv,  
 xlab = "lstat", ylab = "medv", pch=19, cex = 0.5)  
lines(xgrid, pred,   
 lwd = 3, col = "darkorange")



Natural cubic spline fit for Boston data.

For the SA heart data, logistic regression can be fit in a similar way.

# logistic fit  
heart\_ns <- glm(chd ~ ns(sbp, df = 6),   
 data = heart,  
 family = binomial())  
# Grid for prediction  
xgrid <- seq(min(heart$sbp), max(heart$sbp),   
 len = 101)  
pred <- predict(heart\_ns,   
 newdata = data.frame(sbp = xgrid))  
# Estimated function plot  
plot(xgrid, pred,   
 lwd = 2, type = "l",  
 xlab = "sbp", ylab = "f(sbp)")



Natural cubic spline fit for SA heart data.

## Choosing the Number and Locations of the Knots

We have so far placed knots at equally spaced quantiles of the observed predictor values. Another option is to place more knots in places where we suspect the function might vary most rapidly, and to place fewer knots where it seems more stable. In R, bs() and ns() uses the first option (quantiles) by default.

To choose the number of knots, , we may test a few values of , and choose one value depending on how the final fitted functions looks. Objectively, we can use cross-validation to select number of knots/degrees of freedom. For linear regression, we minimize prediction MSE/MAE, and for logistic regression, we minimize misclassification error.

For least squares, leave-one-out CV (LOOCV) becomes an viable option. An amazing shortcut makes the cost of LOOCV the same as that of a *single model fit*. The following formula holds for LOOCV prediction MSE:

where where is the -th fitted value from the original least squares fit, and is the leverage, for a given . This is like the ordinary MSE, except the -th residual is divided by . Thus we can compute LOOCV prediction error for each from the original model fit for the whole data using that value of , without fitting models.

# Smoothing Splines

Another approach of creating splines is to completely avoid knot selection problem by using a maximal set of knots. The complexity of the fit is controlled by regularization. Let us assume is continuous. Consider the following problem: among all functions with two continuous derivatives, find one that minimizes the penalized residual sum of squares

where is a fixed non-negative *tuning/smoothing parameter*. The first term measures closeness to the data, while the second term penalizes curvature in the function, and establishes a trade-off between the two terms.

The notation indicates the second derivative of the function . The first derivative measures the slope of a function at , and the second derivative corresponds to the amount by which the slope is changing. Hence, broadly speaking, the second derivative of a function is a measure of its roughness: it is large in absolute value if is very wiggly near , and it is close to zero otherwise. Note that the second derivative of a straight line is zero; note that a line is perfectly smooth.

Note that without the second term (equivalently ), the optimal choice of is a function that interpolates the data, and thus giving . On the other hand, if , we can not tolerate any second derivative. Thus the optimal form of would be linear (i.e., zero second derivative). This would result in a ordinary least squares fit. Thus, as varies from zero to infinity, the fitted function varies from very rough to very smooth, and the hope is that indexes an interesting class of functions in between.

The criterion above is defined on an infinite-dimensional function space. It can be shown that the minimization problem has an explicit, finite-dimensional, unique minimizer: a *natural cubic spline* with knots at the unique values of , . At face value it seems that the family is still over-parametrized, since there are as many as knots/degrees of freedom! However, the penalty term translates to a penalty on the spline coefficients, which are shrunk some of the way toward the linear fit. The value of the tuning parameter controls the level of shrinkage.

An equivalent way to specify smoothing is via *effective degrees of freedom*. Recall, even if we have knots, the regression coefficients are shrunk, and controls the amount of shrinkage. So is not quite the degrees of freedom here. Usually degrees of freedom refer to the number of free parameters, such as the number of coefficients fit in a polynomial or cubic spline. Instead, we define effective degrees of freedom, . It is possible to show that as increases from 0 to infinity, decrease from to 2 (degrees of freedom for a linear fit). Hence is a measure of the flexibility of the smoothing spline – the higher it is, the more flexible (and the lower-bias but higher-variance) the smoothing spline. Formally, it can be shown that the estimated function is a linear combination of the response values: for a fixed

where is an matrix. The effective degrees of freedom is

Thus, instead of specifying , one can specify effective degrees of freedom.

In R, we can use the function smooth.spline() to fit smoothing splines.

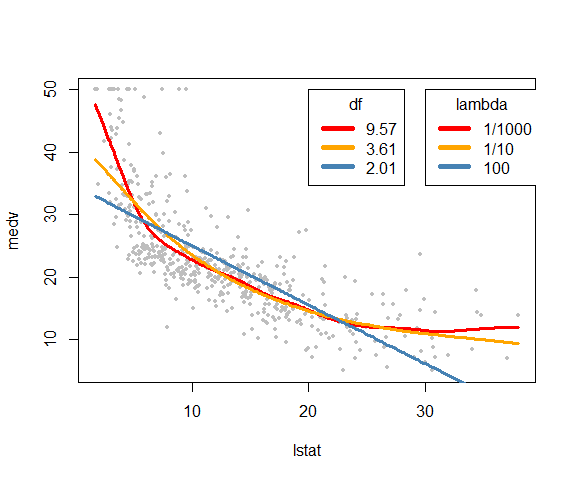
boston\_ss <- smooth.spline(Boston$lstat, Boston$medv,   
 lambda = 1/1000)  
boston\_ss$df

[1] 9.56832

boston\_ss <- smooth.spline(Boston$lstat, Boston$medv,   
 df = 5)  
boston\_ss$lambda

[1] 0.01968027

The figure below shows fitted function for Boston data for three different values of , equivalently different effective degrees of freedom.



Smoothing spline fit for three different lambda values, and the corresponding effective degrees of freedom.

## Selection of smoothing parameter

We can use cross-validation again to select – we can find the value of that makes the cross-validated prediction MSE as small as possible. It turns out that the leave-one-out cross-validation (LOOCV) RSS can be computed very efficiently for smoothing splines, with essentially the same cost as *computing a single fit*, using the following formula:

where is the estimated function based on all of the training observations *except* for the -th observation . In contrast, indicates the smoothing spline function fit to all of the training observations. Therefore, that we can compute each of these leave-one-out fits using only the original fit to all of the data.

In R, we can use the option CV = TRUE without specifying df or lambda to get the LOOCV results.

boston\_ss\_cv <- smooth.spline(Boston$lstat, Boston$medv,  
 cv = TRUE)  
boston\_ss\_cv$spar #chosen sparsity

[1] 0.8705834

boston\_ss\_cv$lambda #chosen lambda

[1] 0.0004655975

boston\_ss\_cv$df #effective df

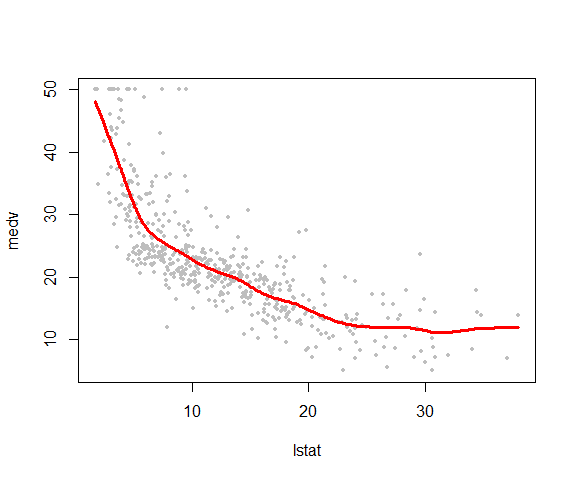
[1] 11.3742

boston\_ss\_cv$pen.crit #RSS

[1] 11524.64

boston\_ss\_cv$cv.crit #loocv

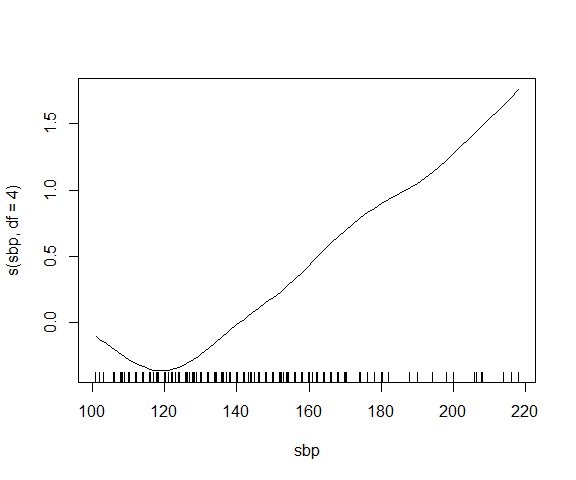
[1] 27.38213



Smoothing spline fit with lambda/df chosen by LOOCV

We can fit smoothing splines for logistic regression using the gam() function in the gam library. The smoothing splines can be specified using the s() function as follows. Note: We could have used gam() in the Boston data as well.

library(gam)  
out <- gam(chd ~ s(sbp, df = 4),   
 data = heart,   
 family = binomial())  
plot(out)



Smoothing spline fit with logistic regression in SA heart data.

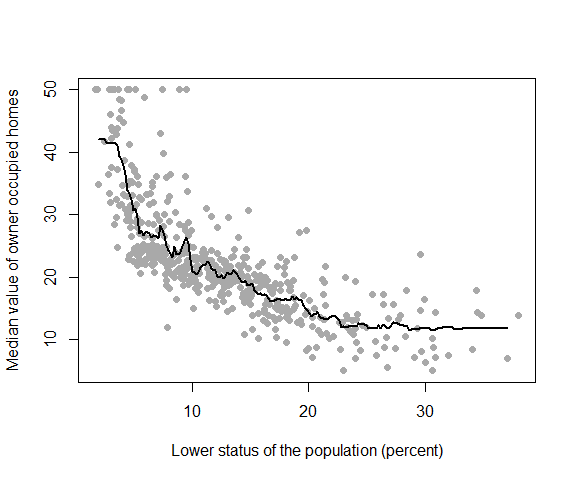
Here we have fit a smoothing spline to sbp with 4 degrees of freedom – by default gam() will exclude intercept when computing degrees of freedom.

# Local Regression

Local regression refers to a class of regression techniques that achieve flexibility in estimating the regression function by fitting a different but simple models separately at each target point . We have already seen one example of such a method – the KNN regression. Recall, KNN takes only values “nearest” to , and estimates based on those points. Observations far away from have no impact on estimation of , thus making KNN a *local* regression method. In general, local regression uses only those observations close to the target point to fit the simple model, and in such a way that the resulting estimated function is smooth.

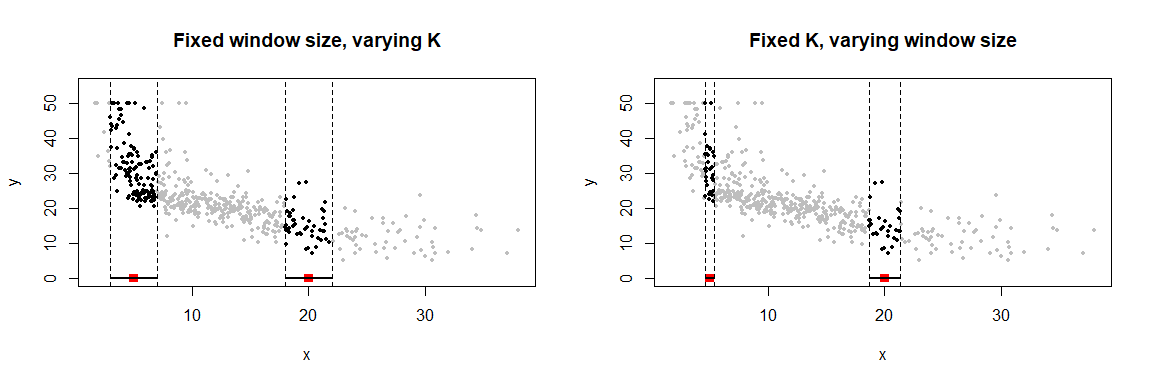
Suppose we have only one predictor, and observed data for . Recall that KNN estimates as

where is the set of observations with values nearest to . As we have seen before, the resulting estimated function is rough and not continuous – see the figure below for an example of 20-NN fit to the Boston data with medv as response and lstat as predictor. The estimated function is plotted on a equally spaced grid of points in .



20-NN fit for Boston data.

This discontinuity is unnecessary and often unwanted. It turns out the source of the problem is that all the points nearest to the target have the same weight, , when computing . Rather than giving them the same, equal, weight, we can give more weight to points closer to , and less for distant points. Another approach could be, rather than using nearest points, we can pre-set a window , and use observations only within this window to form our estimator with same weighting scheme as mentioned before. The figure below shows the two different methods.



Examples of local weighting schemes. The left plot uses a fixed window with , and thus the number of points in the window may vary depending on the value of . The right panel uses fixed number of neighbors, . Thus depending on , the window size changes.

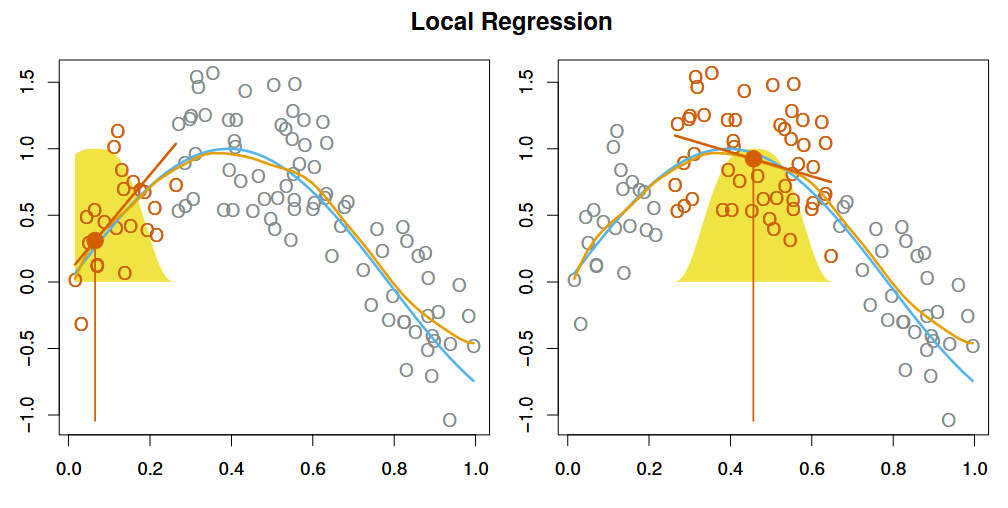
In general, for both the methods above, we can consider a general form of the estimator:

where are pre-specified wights designed in such a way that points nearest to get more weight than points further from , that is, we can assign weights that die off smoothly with distance from the target point.

In general, we specify the weights using *kernel function*, , as

where is a smoothing parameter that determines the the width of the local neighborhood. The kernel function is usually a positive and symmetric function that decays at the tails. Thus, when is closer to zero (i.e., is closer to ), we have higher value of . In contrast, for larger values of , the weight would be lower.

The smooth kernel fit still has problems, however, as exhibited in the figure above. When we take a weighted average of the values for each window, we are essentially assuming the for all the values in the window. This approximation might lead to large bias, especially near the boundaries. For example, in the left panel of the figure above, a local average (also called local *constant*) fit might be reasonable for , but not quite reasonable for . Fitting a straight line, rather than a constant, is much more reasonable here. In other words, we want to approximate in *each* window . Hence . The figure below shows examples of two target points, one near boundary and one in the middle of data scatter – this is for fixed window approach. This approach is called *local linear regression*.



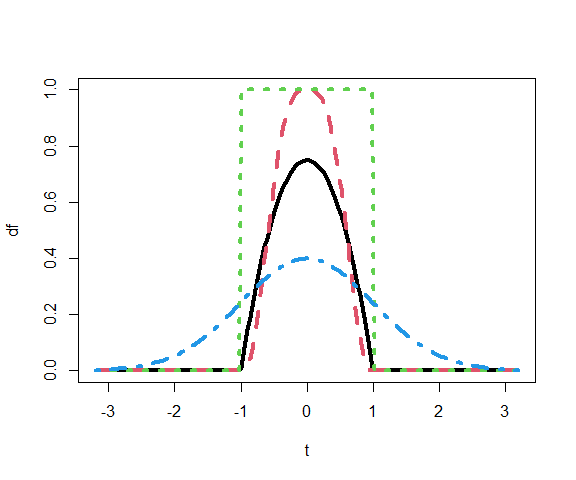
Example of local weights for two target points, one near boundary (left) and one in the middle (right) of data scatter. The orange colored points are local to the target point , represented by the orange vertical line. The yellow bell-shape superimposed on the plot indicates weights assigned to each point, decreasing to zero with distance from the target point. The blue curve represents from which the data were generated, and the light orange curve corresponds to the local regression estimate .

It can be shown that, formally, we can fit a *weighted least squares* regression by minimizing

with respect to and , and estimate accordingly. We can easily extend the local fit using polynomial of any degree. Such a general method is called *local polynomial regression*.

There are quite a few popular kernel functions – examples of four kernel functions are shown in the next figure in the text:

The uniform, Epanechnikov and tri-cube kernels have compact support, and thus assign when is outside . The Gaussian kernel function on the other hand is a popular noncompact kernel that assigns weights to all data points (even outside ). The uniform kernel assigns same weights to every point in , but zero outside. Thus uniform kernel is essentially performing KNN regression.

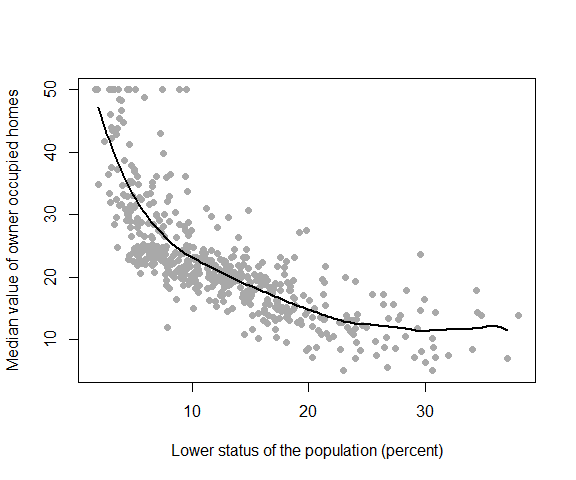


Form of four kernel functions: Uniform (green dotted), Epanechnikov (black solid), Tri-cube (red dashed), and Gaussian (blue dash-dotted).

In order to perform local regression, there are a number of choices to be made: the kernel function , the degree of polynomial to fit, the window size or the number of neighbors . The choices of and degree of polynomial has some impact on the fit. The most important choice is that of or : it controls the flexibility of the non-linear fit. The smaller the value of or , the more local and wiggly will be our fit; alternatively, a very large value of or will lead to a global fit to the data using all of the training observations. We can again use cross-validation to choose or .

An example of a local linear fit of the Boston data is shown below.

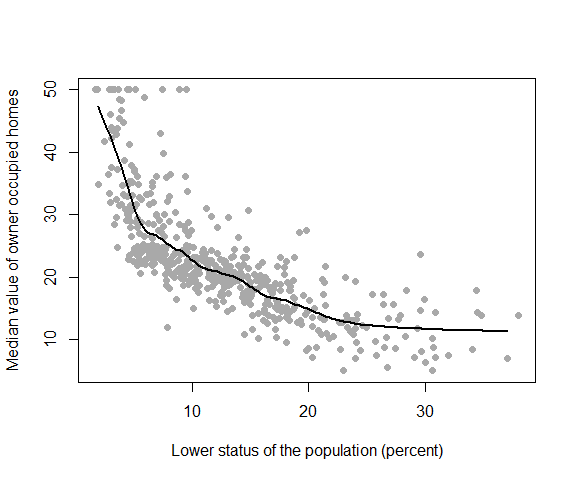
## Epanechnikov kernel  
epan <- function(t){  
 0.75\*(1 - t^2)\*I(abs(t) <= 1)  
}  
## Local linear fitting function  
kfit\_linear <- function(x0, x, y, h){  
 # kernel weights  
 w <- epan(abs(x0 - x)/h)  
 w <- w / sum(w)  
 # Linear fit  
 out <- lm(y ~ x, weights = w)  
 # f(x0)  
 fx0 <- predict(out, newdata = data.frame(x = x0))  
 return(fx0)  
}  
## Grid for estimation of f  
xgrid <- seq(2, 37, len=201)  
## Compute fhat over grid with h = 5   
fhat <- sapply(xgrid, kfit\_linear,   
 x = Boston$lstat,   
 y = Boston$medv,   
 h = 5)  
plot(Boston$lstat, Boston$medv,   
 pch=19,   
 col = "darkgray",  
 xlab = "Lower status of the population (percent)",  
 ylab = "Median value of owner occupied homes")  
lines(xgrid, fhat, lwd=2)



Local linear fit of Boston data with .

Alternatively, We can also use the lo() function in gam library. It uses fixed number of neighbors () as the smoothing parameter. The relevant argument is span which is .

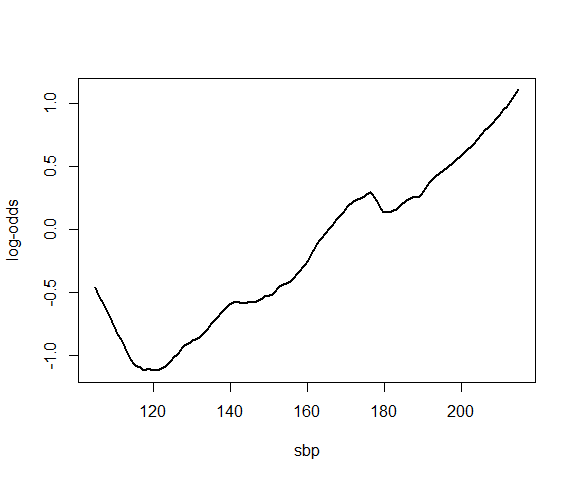
out <- gam(medv ~ lo(lstat, span = 0.2), data = Boston)  
xgrid <- seq(2, 37, len=201)  
fhat <- predict(out, newdata = data.frame(lstat = xgrid))  
plot(Boston$lstat, Boston$medv,   
 pch=19, col = "darkgray",  
 xlab = "Lower status of the population (percent)",  
 ylab = "Median value of owner occupied homes")  
lines(xgrid, fhat, lwd=2)



Local regression fit of Boston data using gam.

We can also fit a local linear logistic regression model by adding the argument family = binomial(). An example using the SA hear data is shown below.

out <- gam(chd ~ lo(sbp, span = 0.3),   
 data = heart,  
 family = binomial())  
xgrid <- seq(105, 215, len=201)  
fhat <- predict(out,   
 newdata = data.frame(sbp = xgrid))  
plot(xgrid, fhat, lwd=2, type = "l",  
 xlab = "sbp",  
 ylab = "log-odds")



Local regression fit of SA heart data using gam.

Local regression can be generalized to incorporate multiple features as well. One very useful generalization involves fitting a multiple linear regression model that is global in some variables (e.g., linear effects), but local in another, such as time. Such models are called *varying coefficient models*. Local regression can be also generalized for bivariate problems, with a pair of variables and , rather than one. We can simply use two-dimensional neighborhoods, and fit bivariate linear regression models using the observations that are near each target point in two-dimensional space. Theoretically the same approach can be implemented in higher dimensions, using linear regressions fit to -dimensional neighborhoods. However, local regression can perform poorly if is much larger than about 3 or 4 because there will generally be very few training observations close to .

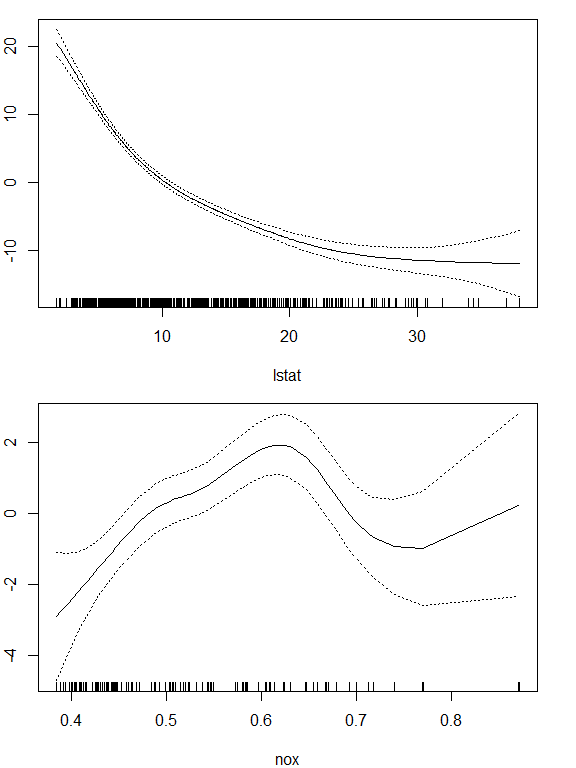
# Generalized Additive Models

Generalized additive models (GAMs) provide a general framework for extending a standard linear model by allowing non-linear functions of each of the variables, while maintaining additivity. In this framework, we assume that we have predictors . In GAM, we model

where are assumed to be smooth non-linear functions. then we can use any of the previously discussed methods (polynomial regression, piecewise-polynomial splines, natural cubic splines, or smoothing splines) to model .

In R, we can use the gam() function in the gam library to fit GAMs using smoothing splines. Let us use Boston data again but with two covariates, lstat, and nox. The figure below shows the estimated functions.

out <- gam(medv ~ s(lstat, df = 4) + s(nox, df = 4),   
 data = Boston)  
par(mfrow = c(2,1), mar = c(4,2,1,2))  
plot(out, terms = "s(lstat, df = 4)", se = TRUE)  
plot(out, terms = "s(nox, df = 4)", se = TRUE)

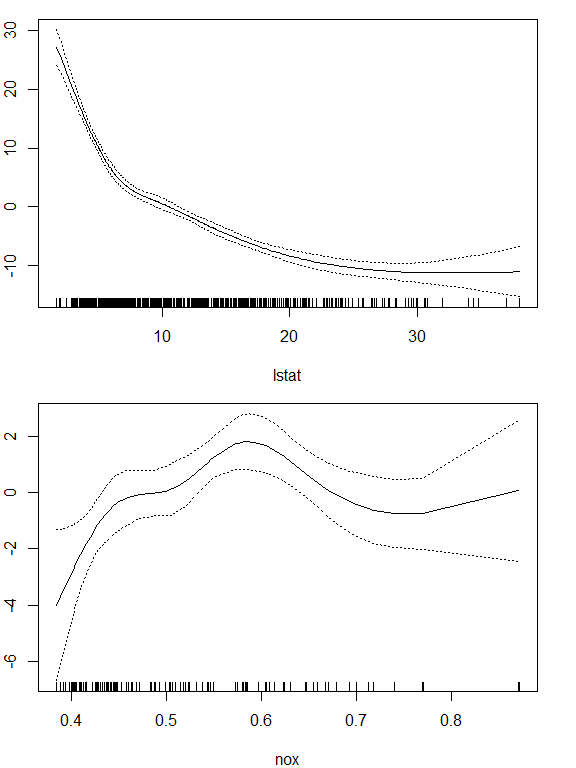


Estimated effects of lstat and nox in the Boston data using GAM.

We call the plot() function with argument se = TRUE. This produces point-wise error bands (estimate two-SE) for the estimated functions.

If we want to fit natural cubic splines instead, we can simply use the lm() function again but with two sets of natural spline basis, as follows. The figure below shows the estimated functions. Even though we are using lm() we can still use plot.Gam() to plot the estimated functions.

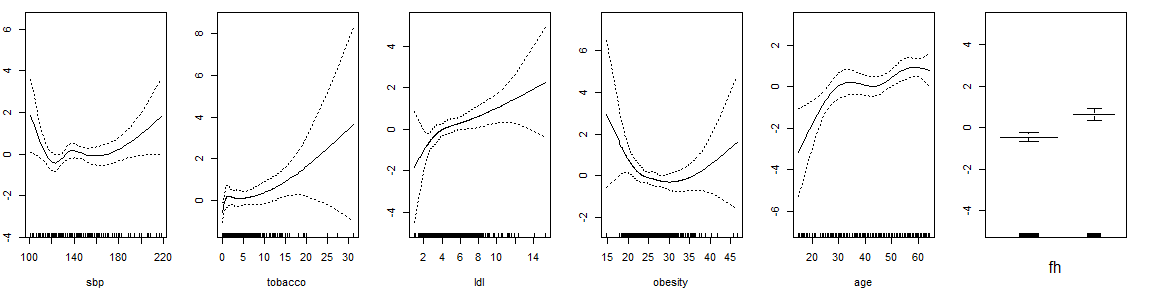
out <- lm(medv ~ ns(lstat, df = 5) + ns(nox, df = 5),  
 data = Boston)  
par(mfrow = c(2,1), mar = c(4,2,1,2))  
plot.Gam(out, se = TRUE)



Estimated effects of lstat and nox in the Boston data using GAM.

To demonstrate GAM in logistic regression, we use sbp, tobacco, ldl, famhist, obesity and age as covariates. Here famhist is a categorical variable with two levels, it is coded by a simple binary or dummy variable, and is associated with a single coefficient in the fit of the model. The effects of the other predictors are modeled using natural splines with 4 degrees of freedom.

fh <- as.factor(ifelse(heart$famhist == "Present", 1, 0))  
out <- glm(chd ~ ns(sbp, df = 4) +   
 ns(tobacco, df = 4) +  
 ns(ldl, df = 4) +  
 ns(obesity, df = 4) +  
 ns(age, df = 4) +   
 fh,  
 data = heart,  
 family = binomial())  
par(mfrow = c(1,6), mar = c(4,2,1,2))  
plot.Gam(out, se = TRUE, scale = 10)



Estimated effects of the predictors in the SA heart data using GAM.

These effects at first may come as a surprise, but an explanation lies in the nature of the retrospective data. These measurements were made sometime after the patients suffered a heart attack, and in many cases they had already benefited from a healthier diet and lifestyle, hence the apparent increase in risk at low values for obesity and sbp.

1. Just as there are for ordinary polynomials. [↑](#footnote-ref-1)