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

Arnab Maity

NCSU Statistics ~ 5240 SAS Hall ~ amaity[at]ncsu.edu

Contents

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 Y and X 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 X^2, X^3, X_1X_2 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

$$E(Y_i|X_i) = f(X)$$

for a linear regression model, or

$$log\left[\frac{P(Y_i = 1|X_i)}{P(Y_i = 2|X_i)}\right] = f(X)$$

for a logistic regression model, where

$$f(X) = \beta_0 + X_i\beta_1 + \ldots + X_i^d\beta_d.$$

Thus we are assuming that the true function $f(X)$ is a *linear combination* of the monomial terms $X, \ldots, X^d$. These terms are examples of *basis functions.* In general, given a set of predictors $\mathbf{X}_i = (X_{i1}, \ldots, X_{ip})$, we may assume that there are there are functions $h_1(\cdot), \ldots, h_M(\cdot)$ such that the function $f(\mathbf{X}_i)$ can be written as or can be approximated as,

$$f(\mathbf{X}_i) = \beta_0 + h_1(\mathbf{X}_i)\beta_1 + \ldots + h_M(\mathbf{X}_i)\beta_M,$$

where $\beta_0, \ldots, \beta_M$ are unknown coefficients. The functions $h_1(\cdot), \ldots, h_M(\cdot)$ are called *basis functions.* and the representation above is called a *linear basis expansion* in X. Once we have specified the basis functions, we can simply use the linear model approach to fit this model.

Some examples of basis representations are shoes below:

- Linear model in each predictors: $h_m(\mathbf{X}_i) = X_{im}$
- Polynomial regressions: $h_m(\mathbf{X})$ takes forms such as X_{im}^2 or $X_{ij}X_{im}$.
- Nonlinear transformations: $h_m(\mathbf{X})$ can take form such as $log(X_{im})$ or $\sqrt{X_{im}}$.
- Piecewise constant: $h_m(\mathbf{X}) = I(L_m \leq X_k \leq U_m)$. By breaking the range of X_k up into such non-overlapping regions, we can model $f(\mathbf{X})$ with a piecewise constant function.

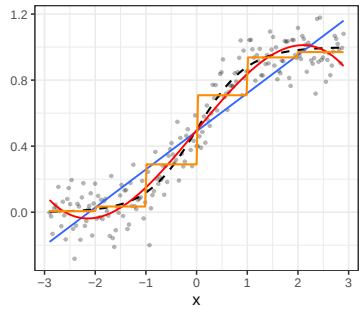

Figure 1: Examples of a linear (blue), cubic (red) and piecewise constant (orange) fit to a simulated data set with one predictor. The true function is shown in black.

Figure 1 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 $f(X)$ – 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*[1] and *splines* that allow for local polynomial representations.

[1] Piecewise-constants are special case 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

$$f(\mathbf{X}_i) = \beta_0 + f_1(X_{i1}) + \ldots + f_p(X_{ip}),$$

  where $f_1(\cdot), \ldots, f_p(\cdot)$ are assumed to be smooth non-linear functions. It is called an additive model because we calculate a separate f_j for each X_j, and then add together all of their contributions. Each of these functions are then models using basis expansion:

$$f_j(X_{ij}) = \beta_{j1} h_{j1}(X_{ij}) + \ldots + \beta_{jM_j} h_{jM_j}(X_{ij}),$$

  where M_j is the number of basis functions used to model f_j. The size of the model is limited by the number of basis functions used for each component function. Notice that the final form of $f(\mathbf{X})$ is still linear in terms of $h_{11}, \ldots, h_{pM_p}$.

- *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.[2] We will also discuss more sophisticated methods for regularization.

[2] In fact, lasso is both a regularization and selection method.

Piecewise Polynomials and Splines

For simplicity, let us assume that we have only one predictor X – we will relax this assumption in later sections when appropriate. Let us start with piecewise-constant basis functions: given a set of points $c_1, \ldots, c_M$ in the domain of X, define

$$h_0(X) = I(X < c_1), \;\; h_1(X) = I(c_1 \leq X < c_2), \;\; \ldots, \;\; h_{M-1} = I(c_{M-1} \leq X < c_M), \;\; h_M(X) = I(X > c_M).$$

The boundaries of each regions, $c_1, \ldots, c_M$, are called *knots*.

For linear regression, we will then fit a model

$$Y_i = \beta_0 + h_1(X_i)\beta_1 + \ldots + h_M(X_i)\beta_m + \epsilon_i.$$

Note that we have omitted $h_0(\cdot)$ from the model above. This is because it is redundant with the interceptEach X_i can be in only one region $[c_{m-1}, c_m)$. Thus, only one of the $h_m(X_i)$ will equal to one, and every other functions will be zero. Therefore, $h_0(X_i) + h_1(X_i) + \ldots + h_M(X_i) = 1$. Including the intercept will thus introduce multicollinearity. Since the regions $[c_{m-1}, c_m)$ are disjoint, the least squares estimate of β_m is simply the sample mean of those Y's that have X's in the m-th region. Also note that when $X < c_1$, then $h_1(X_i) = \ldots = h_M(X_i) = 0$. Thus β_0 is the mean of the Y's with X 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 Figure 2.

```
# 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)
```

The vertical lines are the the knots. We can see that in each region, the estimated $f(X)$ is constant.

We can similarly fit a logistic regression model using the same basis functions as covariates. Consider the South African Heart Disease data[3]. The dataset is a subset of the Coronary Risk-Factor Study

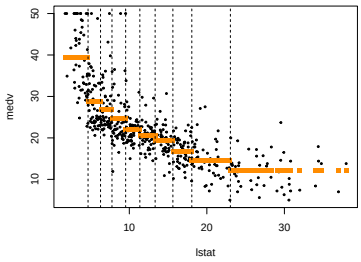

Figure 2: Piecewise-constant fit for Boston data.

[3] Available at `https://web.stanford.edu/~hastie/ElemStatLearn/`

(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.table("http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/SAheart.data",
                    sep = ",",
                    header = TRUE,
                    row.names = 1)
head(heart)
```

```
##   sbp tobacco  ldl adiposity famhist typea obesity alcohol age chd
## 1 160   12.00 5.73     23.11 Present    49   25.30   97.20  52   1
## 2 144    0.01 4.41     28.61  Absent    55   28.87    2.06  63   1
## 3 118    0.08 3.48     32.28 Present    52   29.14    3.81  46   0
## 4 170    7.50 6.41     38.03 Present    51   31.99   24.26  58   1
## 5 134   13.60 3.50     27.78 Present    60   25.99   57.34  49   1
## 6 132    6.20 6.47     36.21 Present    62   30.77   14.14  45   0
```

```
dim(heart)
```

```
## [1] 462  10
```

We use `chd` as response and `sbp` (systolic blood pressure) as predictor. Figure 3 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)
```

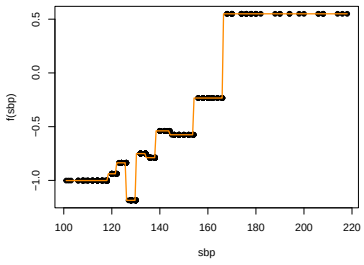

Figure 3: Piecewise-constant fit for SA heart data.

It is easy to see that we can extend this concept to piecewise-polynomial by positing a polynomial model for $h_m(\cdot)$ in each region. For example, Figures 4 and 5 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))
```

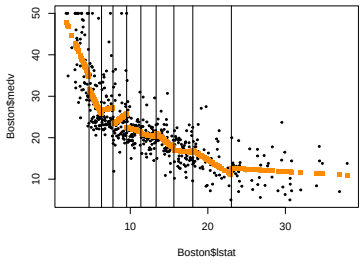

Figure 4: Piecewise-linear fit for Boston data.

One drawback of this approach is that the estimated function is discontinuous at the boundaries of the regions, $c_1, \ldots, c_M$. 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 c, a piecewise-linear functions have the form

$$h_1(X) = (\beta_0 + X\beta_1)I(X < c), \;\; h_1(X) = (\beta_2 + X\beta_3)I(c \leq X).$$

With the restriction that the overall function is continuous at c, we have an additional condition that $h_1(\cdot)$ has the same value as $h_2(\cdot)$ at $X = c$, that is,

$$\beta_0 + c\beta_1 = \beta_2 + c\beta_3.$$

Thus, instead of four original parameters $\beta_0, \ldots, \beta_3$, we will have *three* free parameters with the additional restriction of continuity.

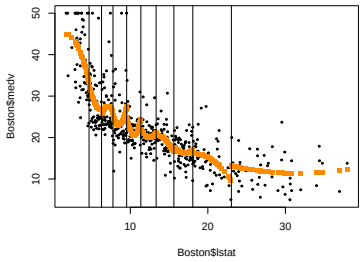

Figure 5: Piecewise-quadratic fit for Boston data.

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

$$h_1(X) = 1, \;\; h_2(X) = X, \;\; h_3(X) = (X - c)_+,$$

where $t_+ = 0$ if $t \leq 0$, t otherwise. In the general case with M knots, $c_1, \ldots, c_M$, it can be shown that the continuity constraints can be incorporated by adding $(X - c_m)_+$ terms to the basis – for piecewise-linear functions we can use

$$h_1(X) = 1, \;\; h_2(X) = X, \;\; h_3(X) = (X - c_1)_+, \;\; \ldots, \;\; h_{M+2}(X) = (X - c_M)_+,$$

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

$$h_1(X) = 1, \;\; h_2(X) = X, \;\; \ldots, \;\; h_{d+1}(X) = X^d, \;\; h_{d+2}(X) = (X - c_1)_+^d, \;\; \ldots, \;\; h_{d+M+1}(X) = (X - c_M)_+^d.$$

The terms $(X - c_k)_+^d$ are known as *truncated power basis functions*.

In spline literature, we also call the formulation above to have *order* $d + 1$. 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-$(d + 1)$ spline with M knots needs $d + M + 1$ basis functions including intercept. The number $d + M + 1$ (the total number of basis functions) is called *degrees of freedom*. For example, a cubic spline with 4 knots has degrees of freedom $3 + 4 + 1 = 8$.

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 X observations determine the positions of the knots. Placing the knots at specific quantiles of observed X 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.[4] 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.

[4] Just as there are for ordinary polynomials.

```
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)
```

Similarly, we fit a cubic spline to the SA heart data below. We use two internal knots for this fit. Figure 7 shows the estimated function.

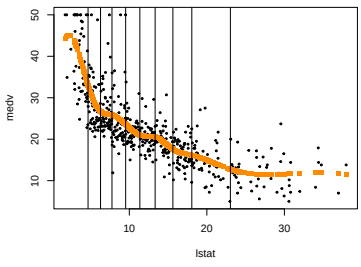

Figure 6: Cubic spline fit of Boston data.

```r
# 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")
```

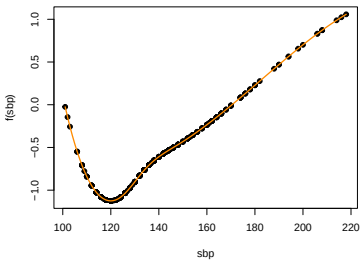

Figure 7: Cubic spline fit with two knots of the SA heart data.

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 X 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 M knots is represented by M basis functions (equivalently, degrees of freedom is M). 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

$$N_1(X) = 1, \ \ , N_2(X) = X, \ \ , N_{m+2} = d_k(X) - d_{M-1}(X),$$

for $m = 1, \ldots, M$, where

$$d_m(X) = \frac{(X - c_m)_+^3 - (X - c_M)_+^3}{C_M - c_m}.$$

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")
```

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)")
```

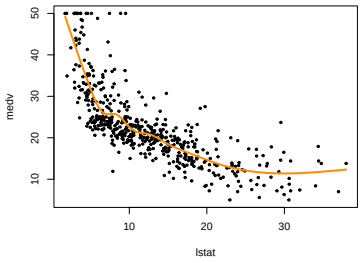

Figure 8: Natural cubic spline fit for Boston data.

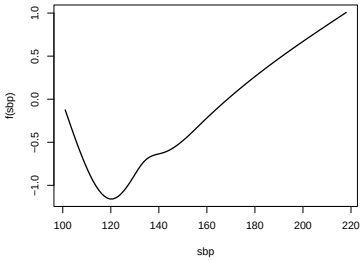

Figure 9: Natural cubic spline fit for SA heart data.

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, M, we may test a few values of M, 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:

$$PMSE(\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - \widehat{Y}_{i,M}}{1 - h_{i,M}}\right)^2,$$

where where $\widehat{Y}_{i,M}$ is the i-th fitted value from the original least squares fit, and $h_{i,M}$ is the leverage, for a given M. This is like the ordinary MSE, except the i-th residual is divided by $1 - h_i$. Thus we can compute LOOCV prediction error for each M from the original model fit for the whole data using that value of M, without fitting n 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 Y is continuous. Consider the following problem: among all functions $f(\cdot)$ with two continuous derivatives, find one that minimizes the penalized residual sum of squares[5]

$$\sum_{i=1}^{n}\{Y_i - f(X_i)\}^2 + \lambda\int\{f''(t)\}^2\,dt,$$

where λ 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 λ establishes a trade-off between the two terms.

The notation $f''(t)$ indicates the second derivative of the function f. The first derivative $f'(t)$ measures the slope of a function at t, 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 $f(t)$ is very wiggly near t, and it is close to zero otherwise.[6]

Note that without the second term (equivalently $\lambda = 0$), the optimal choice of $f(\cdot)$ is a function that interpolates the data, and thus giving $RSS = 0$. On the other hand, if $\lambda = \infty$, we can not tolerate any second derivative. Thus the optimal form of f would be linear (i.e., zero second derivative). This would result in a ordinary least squares

[5] The expression takes the "Loss + Penalty" formulation that we encounter in the context of ridge regression and the lasso. The first term is "squared error" loss function. The second term is a penalty term that penalizes the variability in f.

[6] The second derivative of a straight line is zero; note that a line is perfectly smooth.

fit. Thus, as λ varies from zero to infinity, the fitted function varies from very rough to very smooth, and the hope is that λ indexes an interesting class of functions in between.

The criterion above is defined on an infinite-dimensional function space.[7] 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 X_i, $i = 1, \ldots, n$. At face value it seems that the family is still over-parametrized, since there are as many as n 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 λ controls the level of shrinkage.

[7] In fact, a Sobolev space of functions for which the second term is defined.

An equivalent way to specify smoothing is via *effective degrees of freedom*. Recall, even if we have n knots, the regression coefficients are shrunk, and λ controls the amount of shrinkage. So n 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, df_λ. It is possible to show that as λ increases from 0 to infinity, df_λ decrease from n to 2 (degrees of freedom for a linear fit). Hence df_λ 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 f is a linear combination of the response values: for a fixed λ

$$[\hat{f}(X_1), \ldots, \hat{f}(X_n)]^T = S_\lambda \mathbf{Y},$$

where S_λ is an $n \times n$ matrix. The effective degrees of freedom is

$$df_\lambda = trace(S_\lambda) = \{S_\lambda\}_{11} + \ldots + \{S_\lambda\}_{nn}.$$

Thus, instead of specifying λ, 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
```

Figure 10 shows fitted function for Boston data for three different values of λ, equivalently different effective degrees of freedom.

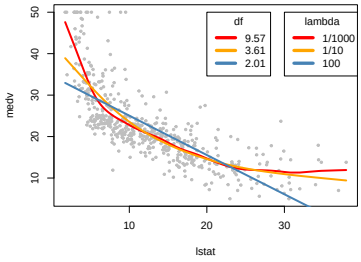

Figure 10: Smoothing spline fit for three different lambda values, and the corresponding effective degrees of freedom.

Selection of smoothing parameter λ

We can use cross-validation again to select λ – we can find the value of λ 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:

$$RSS_{\text{loocv}}(\lambda) = \sum_{i=1}^{n} \{Y_i - \widehat{f}_{\lambda}^{(-i)}(X_i)\}^2 = \sum_{i=1}^{n} \left[\frac{Y_i - \widehat{f}_{\lambda}(X_i)}{1 - \{S_{\lambda}\}_{ii}} \right]^2,$$

where $\widehat{f}_{\lambda}^{(-i)}(\cdot)$ is the estimated function based on all of the training observations *except* for the i-th observation (X_i, Y_i). In contrast, $\widehat{f}_{\lambda}(\cdot)$ 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
```

```
## Call:
## smooth.spline(x = Boston$lstat, y = Boston$medv, cv = TRUE)
##
## Smoothing Parameter  spar= 0.8705834  lambda= 0.0004655975 (12 iterations)
## Equivalent Degrees of Freedom (Df): 11.3742
## Penalized Criterion (RSS): 11524.64
## PRESS(l.o.o. CV): 27.38213
```

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.[8]

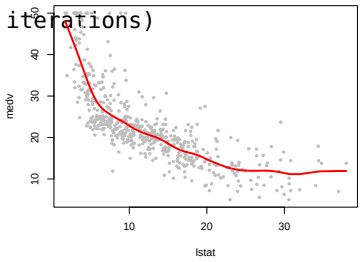

Figure 11: Smoothing spline fit with lambda/df chosen by LOOCV

[8] 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)
```

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.

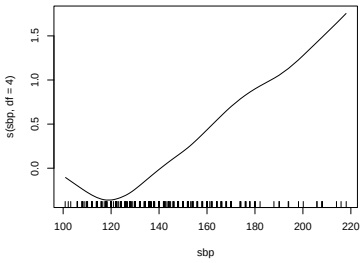

Figure 12: Smoothing spline fit with logistic regression in SA heart data.

Local Regression

Local regression refers to a class of regression techniques that achieve flexibility in estimating the regression function $f(X)$ by fitting a different but simple models separately at each target point x_0. We have already seen one example of such a method – the KNN regression. Recall, KNN takes only x values "nearest" to x_0, and estimates $f(x_0)$ based on those points. Observations far away from x_0 have no impact on estimation of $f(x_0)$, thus making KNN a *local* regression method. In general, local regression uses only those observations close to the target point x_0 to fit the simple model, and in such a way that the resulting estimated function $\widehat{f}(\cdot)$ is smooth.

Suppose we have only one predictor, and observed data (Y_i, X_i) for $i = 1, \ldots, n$. Recall that KNN estimates $f(x_0)$ as

$$\widehat{f}(x_0) = \frac{1}{K} \sum_{X_i \in \mathcal{N}_K(x_0)} Y_i,$$

where $\mathcal{N}_K(x_0)$ is the set of K observations with X_i values nearest to x_0. As we have seen before, the resulting estimated function is rough and not continuous – see Figure 13 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 $[2, 37]$.

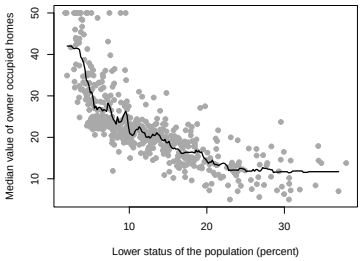

Figure 13: 20-NN fit for Boston data.

This discontinuity is unnecessary and often unwanted. It turns out the source of the problem is that all the K points nearest to the target x_0 have the same weight, $1/K$, when computing $\widehat{f}(x_0)$. Rather than giving them the same, equal, weight, we can give more weight to points closer to x_0, and less for distant points. Another approach could be, rather than using K nearest points, we can pre-set a window $[x_0 \pm h]$, and use observations only within this window to form our estimator $\widehat{f}(x_0)$ with same weighting scheme as mentioned before. Figure 14 shows the two different methods.

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

$$\widehat{f}(x_0) = \frac{\sum_{i=1}^n w_i Y_i}{\sum_{i=1}^n w_i},$$

where w_i are pre-specified wights designed in such a way that points nearest to x_0 get more weight than points further from x_0, that is,

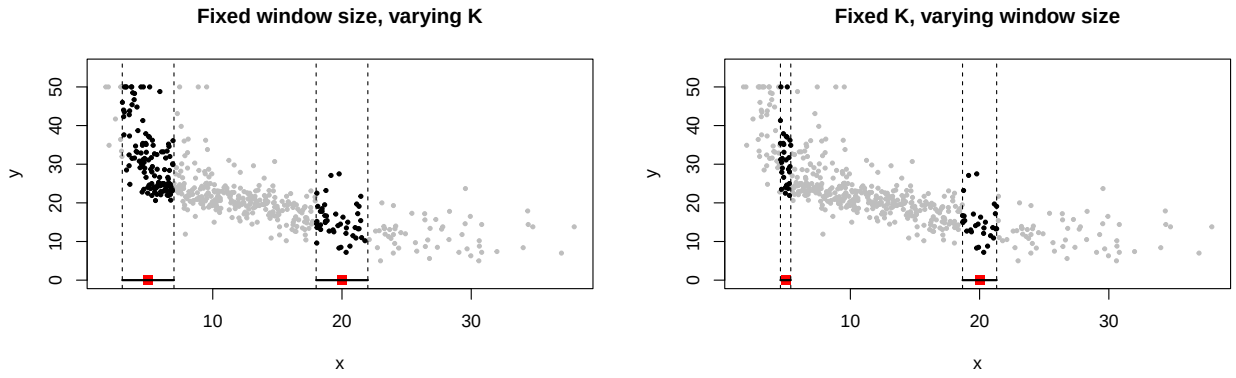

Figure 14: Examples of local weighting schemes. The left plot uses a fixed window $[x_0 - h, x_0 + h]$ with $h = 2$, and thus the number of points in the window may vary depending on the value of x_0. The right panel uses fixed number of neighbors, $K = 30$. Thus depending on x_0, the window size changes.

we can assign weights that die off smoothly with distance from the target point.

In general, we specify the weights using *kernel function*, $D(\cdot)$, as

$$w_i = D(|x_0 - X_i|/h),$$

where h 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 $|x_0 - X_i|/h$ is closer to zero (i.e., X_i is closer to x_0), we have higher value of w_i. In contrast, for larger values of $|x_0 - X_i|/h$, the weight w_i would be lower.

The smooth kernel fit still has problems, however, as exhibited in Figure 14. When we take a weighted average of the Y values for each window, we are essentially assuming the $f(x) \approx f(x_0)$ for all the x values in the window. This approximation might lead to large bias, especially near the boundaries. For example, in the left panel of Figure 14, a local average (also called local *constant*) fit might be reasonable for $x_0 = 20$, but not quite reasonable for $x_0 = 5$. Fitting a straight line, rather than a constant, is much more reasonable here. In other words, we want to approximate $f(x) \approx a + xb$ in *each* window $[x_0 - h, x_0 + h]$. Hence $\widehat{f}(x_0) = \widehat{a} + x_0\widehat{b}$. Figure 15 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*.

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

$$\sum_{i=1}^{n} w_i(Y_i - a - X_i b)^2,$$

with respect to a and b, and estimate $f(\cdot)$ accordingly. We can easily

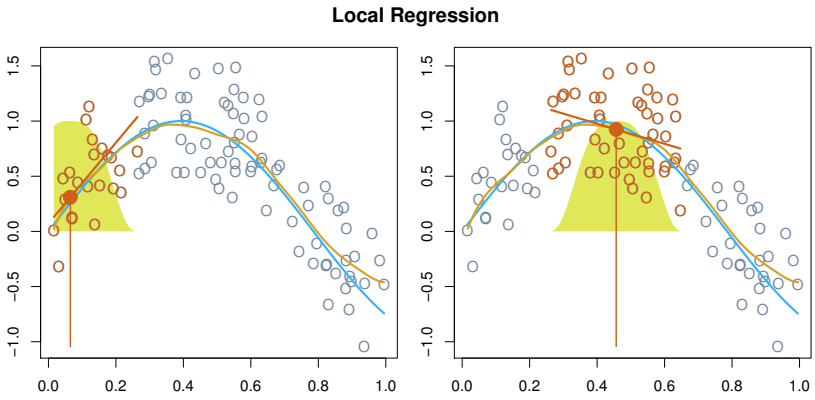

Figure 15: Example of local weights for for two target points, one near boundary (left) and one in the middle (right) of data scatter. The orange colored points are local to the target point x_0, represented by the orange vertical line. The yellow bell-shape superimposed on the plot indicates weights assigned to each point, decreasing to zero with distance from the target point. The blue curve represents $f(x)$ from which the data were generated, and the light orange curve corresponds to the local regression estimate $\widehat{f}(x)$.

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 Figure 16:

$$\text{Uniform: } D(t) = I(-1 \leq t \leq 1)$$

$$\text{Epanechnikov: } D(t) = \frac{3}{4}(1 - t^2)\, I(-1 \leq t \leq 1)$$

$$\text{Tri-cube: } D(t) = (1 - |t|^3)^3\, I(-1 \leq t \leq 1)$$

$$\text{Gaussian: } D(t) = exp(-t^2/2)$$

The uniform, Epanechnikov and tri-cube kernels have compact support, and thus assign $w_i = 0$ when X_i is outside $[x_0 - h, x_0 + h]$. The Gaussian kernel function on the other hand is a popular non-compact kernel that assigns weights to all data points (even outside $[x_0 - h, x_0 + h]$). The uniform kernel assigns same weights to every point in $[x_0 - h, x_0 + h]$, but zero outside. Thus uniform kernel is essentially performing KNN regression.

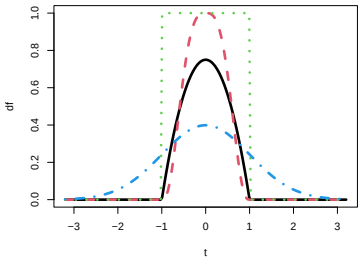

Figure 16: Form of four kernel functions: Uniform (green dotted), Epanechnikov (black solid), Tri-cube (red dashed), and Gaussian (blue dash-dotted).

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

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

```r
## 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)
```

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

```r
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)
```

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.

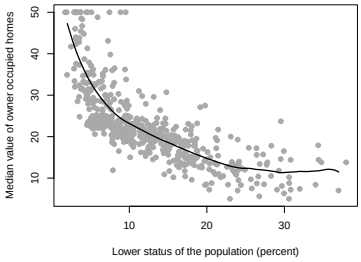

Figure 17: Local linear fit of Boston data with $h = 5$.

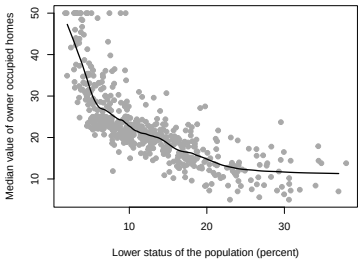

Figure 18: Local regression fit of Boston data using gam.

```
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")
```

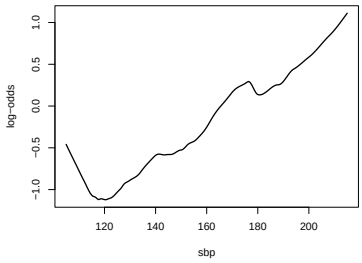

Figure 19: Local regression fit of SA heart data using gam.

Local regression can be generalized to incorporate multiple features $X_1, X_2, \ldots, X_p$ 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 X_1 and X_2, 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 p-dimensional neighborhoods. However, local regression can perform poorly if p is much larger than about 3 or 4 because there will generally be very few training observations close to x_0.

An example of a bivariate local regression is shown below using `Boston` data.

```
out <- gam(medv ~ lo(lstat , nox , span = 0.5),
           data = Boston)
grid <- expand.grid(lstat = seq(2, 37, len=201),
                    nox = seq(0.4, 0.85, len = 101))
fhat <- predict(out, newdata = grid)
contour(seq(2, 37, len=201), seq(0.4, 0.85, len = 101),
        fhat, nlevels = 50,
        xlab = "lstat", ylab = "nox")
```

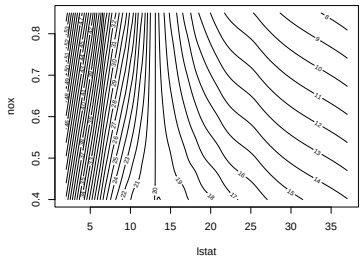

Figure 20: Contour plot of a bivariate local regression fit of the Boston data.

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 p predictors $\mathbf{X}_i = (X_{i1}, \ldots, X_{ip})$.

In GAM, we model

$$f(\mathbf{X}_i) = \beta_0 + \sum_{j=1}^{p} f_j(X_{ij}),$$

where $f_j(\cdot)$ 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 $f_j(\cdot)$.

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`. Figure 21 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)
```

We call the `plot()` function with argument `se = TRUE`. This produces point-wise error bands (estimate $\pm$ 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. Figure 22 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)
```

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) +
```

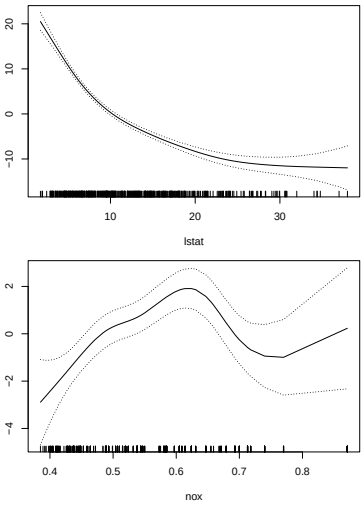

Figure 21: Estimated effects of lstat and nox in the Boston data using GAM.

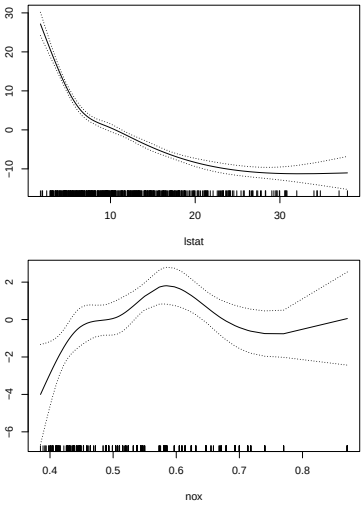

Figure 22: Estimated effects of lstat and nox in the Boston data using GAM.

```
                fh,
          data = heart,
          family = binomial())
par(mfrow = c(1,6), mar = c(4,2,1,2))
plot.Gam(out, se = TRUE, scale = 10)
```

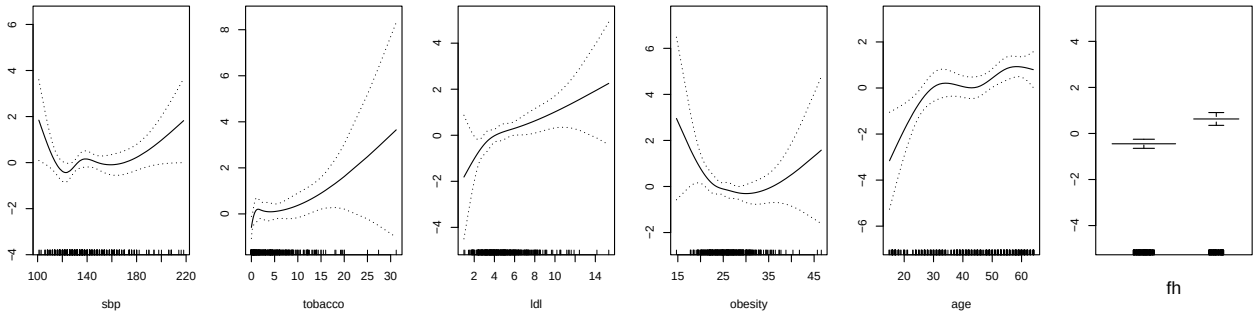

Figure 23: Estimated effects of the predictors in the SA heart data using GAM.

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.