Non-Linear Modeling

Laura Cline

24/10/2021

Contents

In this lab we analyze the `Wage` data considered in the examples throughout this chapter, in order to illustrate the fact that many of the complex non-linear fitting procedures dicscussed can be easily implemented in `R`. We begin by loading the `ISLR` library, which contains the data.

```
rm(list = ls(all=TRUE))

libs <- c("tidyverse", "ISLR", "modelr", "broom", "splines", "gam", "akima")
invisible(lapply(libs, library, character.only=TRUE))
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
##
## Attaching package: 'broom'
## The following object is masked from 'package:modelr':
##
##     bootstrap
## Loading required package: foreach
##
## Attaching package: 'foreach'
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
## Loaded gam 1.20
```

Polynomial Regression and Step Functions

Polynomial regression is super easy to implement in R. We follow the basic steps required to do linear regression with `lm()` and adjust some small things to include polynomial expressions in the formula.

Before we begin, just a quick aside - normally I don't use `attach()` in my own work because it can get confusing what data is currently attached and which columns you are calling. In this scenario, because we are only working with one dataset through the majority of this lab attaching the data is much less problematic.

We first fit the model using the following command:

```
# Bring in the Wage data and run a fourth-degree polynomial regression
attach(Wage)
fit <- lm(wage ~ poly(age,4), data=Wage)
coef(summary(fit))
```

```
##                 Estimate Std. Error    t value     Pr(>|t|)
## (Intercept)    111.70361  0.7287409 153.283015 0.000000e+00
## poly(age, 4)1  447.06785 39.9147851  11.200558 1.484604e-28
## poly(age, 4)2 -478.31581 39.9147851 -11.983424 2.355831e-32
## poly(age, 4)3  125.52169 39.9147851   3.144742 1.678622e-03
## poly(age, 4)4  -77.91118 39.9147851  -1.951938 5.103865e-02
```

This syntax fits a linear model, using the `lm()` function, in order to predict `wage` using a fourth degree polynomial in `age:poly(age,4)`. The `poly()` command allows us to avoid having to write out a long formula with powers of `age`. The function returns matrix whose columns are a basis of *orthogonal polynomials*, which essentially means that each column is a linear combination of the variables `age`, `age^2`, `age^3` and `age^4`.

By default, `poly()` returns orthogonal polynomials from degree 1 to degree 4. That might seem a little confusing, so we can use the `raw = TRUE` option to specify that we want `age`, `age^2`, `age^3` and `age^4` directly.

However, we can also use `poly()` to obtain `age`, `age^2`, `age^3` and `age^4` directly, if we prefer. We can do this by using the `raw = TRUE` argument to the `poly()` function. Later we see that this does not affect the model in a meaningful way - though the choice of basis clearly effects the coefficient estimates, it does not affect the fitted values obtained.

```
fit2 <- lm(wage ~ poly(age, 4, raw=TRUE), data=Wage)
coef(summary(fit2))
```

```
##                                Estimate   Std. Error   t value     Pr(>|t|)
## (Intercept)                -1.841542e+02 6.004038e+01 -3.067172 0.0021802539
## poly(age, 4, raw = TRUE)1   2.124552e+01 5.886748e+00  3.609042 0.0003123618
## poly(age, 4, raw = TRUE)2  -5.638593e-01 2.061083e-01 -2.735743 0.0062606446
## poly(age, 4, raw = TRUE)3   6.810688e-03 3.065931e-03  2.221409 0.0263977518
## poly(age, 4, raw = TRUE)4  -3.203830e-05 1.641359e-05 -1.951938 0.0510386498
```

There are several other ways to do the same thing - you can either use the wrapper `I()` (as `^` is interpreted as a call to interact variables), or simply just use `cbind()`. We have included the output from these extra methods, but have inclined the code so you can see how they are implemented.

There are several other equivalent ways of fitting this model, which showcase the flexibility of the formula language in R. For example,

```
#Using I()
lm(wage ~ I(age^2)+I(age^3)+I(age^4), data=Wage) %>% coef()
```

```
##   (Intercept)      I(age^2)      I(age^3)      I(age^4)
##  3.117328e+01  1.762538e-01 -4.051882e-03  2.495627e-05
```

This simply creates the polynomial basis functions on the fly, taking care to protect terms like `age^2` via the *wrapper* function `I()` (the `^` symbol has a special meaning in formulas).

```
#Using cbind()
lm(wage ~ cbind(age, age^2, age^3, age^4), data=Wage) %>% coef()
```

```
##                         (Intercept) cbind(age, age^2, age^3, age^4)age
##                       -1.841542e+02                        2.124552e+01
##    cbind(age, age^2, age^3, age^4)    cbind(age, age^2, age^3, age^4)
##                       -5.638593e-01                        6.810688e-03
##    cbind(age, age^2, age^3, age^4)
##                       -3.203830e-05
```

This does the same more compactly, using the `cbind()` function for building a matrix from a collection of vectors; any function call such as `cbind()` inside a formula also serves as a wrapper.

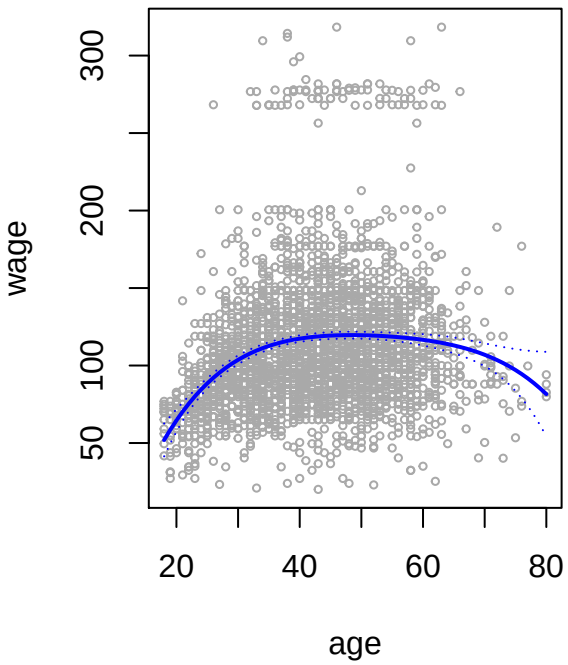
we now create a grid of values for `age` at which we want predictions, and then call the generic `predict()` function, specifying that we want standard errors as well.

Let's take a look at the fitted model to see how well it performs. We'll be using the `modelr` package to help us add a confidence interval around the fit. Since there isn't any difference in outcome between the `poly(..., raw = TRUE)` and previous two models we will just stick with `fit2` for the time being.

```
agelims=range(age)
age.grid=seq(from=agelims[1],to=agelims[2])
preds=predict(fit,newdata=list(age=age.grid),se=TRUE)
se.bands=cbind(preds$fit+2*preds$se.fit,preds$fit-2*preds$se.fit)
```

Finally, we plot the data and add the fit from the degree-4 polynomial.

```
par(mfrow = c(1,2), mar=c(4.5, 4.5, 1, 1), oma = c(0, 0, 4, 0))
plot(age, wage, xlim=agelims, cex=0.5, col='darkgrey')
title("Degree - 4 Polynomial", outer=4)
lines(age.grid, preds$fit, lwd=2, col="blue")
matlines(age.grid, se.bands, lwd=1, col="blue", lty=3)
```



We kind of arbitrarily set up this 4-degree polynomial fit, so how do we know if its the right one to use? We can use hypothesis testing to determine which degree polynomial is most appropriate.

Here the `mar` and `oma` arguments to `par()` allow us to control the margins of the plot, and the `title()` function creates a figure creates a figure title that spans both subplots.

we mentioned earlier that whether or not an orthagonal basis functions is produced in the `poly()` function will not affect the model obtained in a meaningful way. What do we mean by this? The fitted values obtained in this case are identical.

In performing a polynomial regression we must decide on the degree of polynomial to use. One way tto do this is by using hypothesis tests. We now fit models ranging from linear to a degree-5 polynomial and seek to determine the simplest model which is sufficient to explain the relationship between `wage` and `age`. We use the `anova()` function, which performs an *analysis of variance* (ANOVA, using an F-Test) in order to test the null hypothesis that a model M1 is sufficient to explain the data against the alternative hypothesis that a more complex model of M2 is required. In order to use the `anova()` function, M1 and M2 must be *nested* models: the predictors in M1 must be a subset of the predictors in M2. In this case, we fit five different models and sequentially compare the simpler model to the more complex model.

```
fit1 <- lm(wage ~ age, data = Wage)
fit2 <- lm(wage ~ poly(age,2), data=Wage)
fit3 <- lm(wage ~ poly(age,3), data=Wage)
fit4 <- lm(wage ~ poly(age,4), data=Wage)
fit5 <- lm(wage ~ poly(age,5), data=Wage)

anova(fit1, fit2, fit3, fit4, fit5)
```

```
## Analysis of Variance Table
##
## Model 1: wage ~ age
## Model 2: wage ~ poly(age, 2)
```

```
## Model 3: wage ~ poly(age, 3)
## Model 4: wage ~ poly(age, 4)
## Model 5: wage ~ poly(age, 5)
##   Res.Df     RSS Df Sum of Sq        F    Pr(>F)
## 1   2998 5022216
## 2   2997 4793430  1    228786 143.5931 < 2.2e-16 ***
## 3   2996 4777674  1     15756   9.8888  0.001679 **
## 4   2995 4771604  1      6070   3.8098  0.051046 .
## 5   2994 4770322  1      1283   0.8050  0.369682
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It's clear that a linear fit is not sufficient, and that even a cubic fit would outperform a quadratic. The quartic fit is nearing significance, but it can be up to you whether the cubic or the quartic will be better. You could also find the best degree polynomial fit by using Cross-Validation, but that is not covered there.

The p-value comparing the linear `Model 1` to the quadratic `Model 2` is essentially zero, indicating that a linear fit is not sufficient. Similarly, the p-value comparing the quadratic `Model 2` to the cubic `Model 3` is very low (0.001679), so the quadratic fit is also insufficient. The p-value comparing the cubic and degree-4 polynomials, `Model 3` and `Model 4`, is approx. 5% while degree-5 polynomial `Model 5` seems unnecessary because its p-value is 0.37. Hence, either a cubic or a quartic polynomial appear to provide a reasonable fit to the data, but lower- or higher-order models are not justified.

In order to fit a step function, we use the `cut()` function.

Step functions are rather easy to set up; we basically only need `cut()` to help us split up the predictor that we want to create our 'steps' from. It works by covering a numeric into an ordered factor, which will then be 'dummified' when we fit our model.

```
# Example how cut() works
table(cut(age,4))
```

```
##
## (17.9,33.5]   (33.5,49]   (49,64.5] (64.5,80.1]
##         750        1399         779          72
```

```
# Step Function Estimation
lm(wage ~ cut(age,4), data=Wage) %>% tidy()
```

```
## # A tibble: 4 x 5
##   term                    estimate std.error statistic  p.value
##   <chr>                      <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)                94.2       1.48     63.8  0.
## 2 cut(age, 4)(33.5,49]       24.1       1.83     13.1  1.98e-38
## 3 cut(age, 4)(49,64.5]       23.7       2.07     11.4  1.04e-29
## 4 cut(age, 4)(64.5,80.1]      7.64      4.99      1.53 1.26e- 1
```

Here `cut()` automatically picked the cutpoints 33.5, 49 and 64.5 years of age. We could also have specified our own cutpoints directly using the `breaks` option. The function `cut()` returns an ordered categorical variable; the `lm()` function then creates a set of dummy variables for use in the regression. The `age<33.5` category is left out, so the intercept coefficient of $94,158, can be interpreted as the average salary for those user 33.5 years of age, and the other coefficients can be interpreted as the average additional salary for those in the other age groups. We can produce predictions and plots just as we did in the case of the polynomial fit.

Splines

In this section, we will be using the `splines` package. We'll be using `bs()` to help us create the cubic spline basis on which the model will be estimated.

In order to fit **regression splines** in `R`, we use the `splines` library. Regression splines can be fit by constructing an appropriate matrix of basis functions. The `bs()` function generates an entire matrix of basis functions for splies with the specified set of knots. By default, cubic splines are produced. Fitting `wage` to `age` using a regression spline is simple:

```
# Fit our model (a cubic spline)
fit <- lm(wage ~ bs(age, knots=c(25, 48, 60)), data=Wage)
```

```
# Observe coefficients
tidy(fit)
```

```
## # A tibble: 7 x 5
##   term                              estimate std.error statistic  p.value
##   <chr>                                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)                          61.2       9.35     6.54  7.17e-11
## 2 bs(age, knots = c(25, 48, 60))1      2.09     11.8       0.178 8.59e- 1
## 3 bs(age, knots = c(25, 48, 60))2     59.8       9.66     6.19  6.80e-10
## 4 bs(age, knots = c(25, 48, 60))3     56.2      10.7      5.27  1.44e- 7
## 5 bs(age, knots = c(25, 48, 60))4     60.0      10.4      5.78  8.06e- 9
## 6 bs(age, knots = c(25, 48, 60))5     40.4      15.8      2.55  1.09e- 2
## 7 bs(age, knots = c(25, 48, 60))6     21.1      19.4      1.09  2.77e- 1
```

Just to be super clear, this is the model that we just fit:

$$age = \beta_0 + \beta_1(age) + \beta_2(age^2) + \beta_3(age^3) + \beta_4(age - \xi_1)^3 + \beta_5(age - \xi_2)^3 + \beta_6(age - \xi_3)^3 + \epsilon$$

Recall that the last three terms (excluding the error) represent the truncated power basis for each knot. These ensure the continuity of the function at each of the knots through the first and second derivative.

Here we have prespecified the knots at ages 25, 40, and 60. This produces a spline with six basis functions. Recall that a cubic spline with three knots has seven degrees of freedom; these degrees of freedom are used up by an intercept, plus six basis functions. We could also use the `df` option to produce a spline with knots at uniform quantiles of the data.

```
attr(bs(age, df=6), "knots")
```

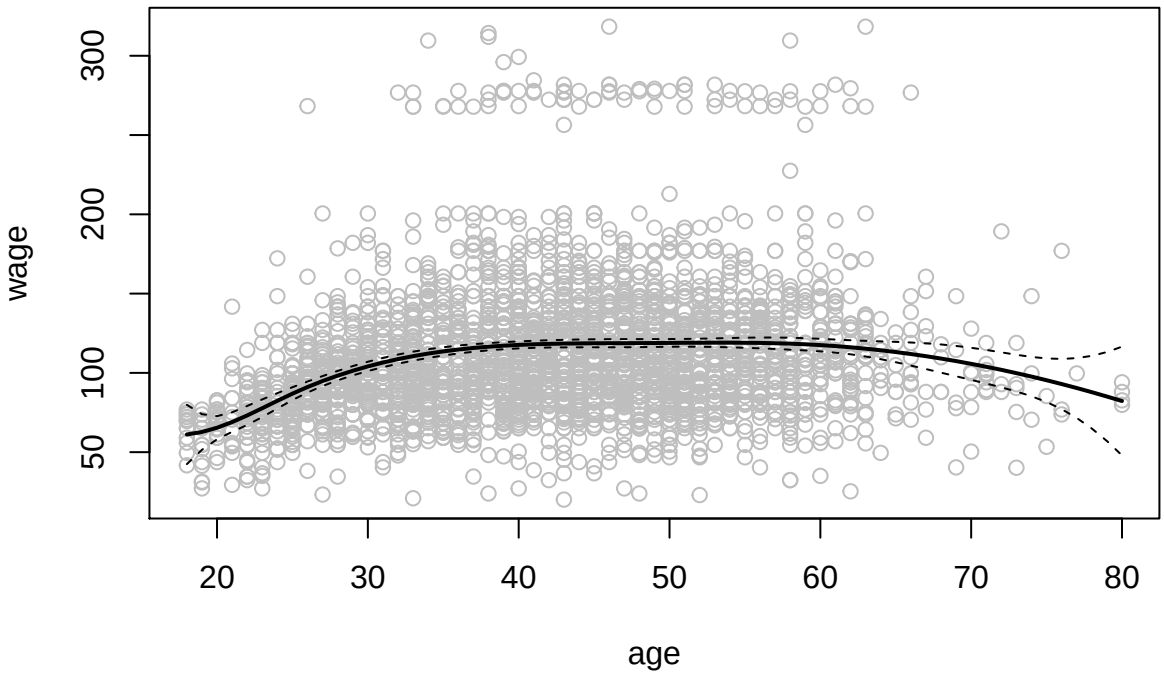
```
##   25%   50%   75%
## 33.75 42.00 51.00
```

In this case, `R` chooses knots at ages 33.75, 42.0, and 51.0 which correspond to the 25th, 50th, and 75th percentiles of `age`. The function `bs()` also has a `degree` argument, so we can fit splines of any degree, rather than the default degree of 3 (which yields a cubic spline).

We will now take a look at the fitted curve in relation to the `Wage` data.

```
agelims = range(age)
age.grid = seq(from=agelims[1], to=agelims[2])
```

```
pred = predict(fit, newdata=list(age=age.grid), se=T)
plot(age, wage, col="gray")
lines(age.grid, pred$fit, lwd=2)
lines(age.grid, pred$fit + 2 * pred$se, lty="dashed")
lines(age.grid, pred$fit - 2 * pred$se, lty="dashed")
```



Notice in the plot how the confidence intervals start straying out from the actual fit. This is a common problem with cubic splines, and to remedy that we can enforce constraints on the form of the function outside of the boundary knots. These are called **natural splines**. We use `ns()` to help us create the basic matrix for our natural cubic spline before plotting the results an comparing the performance of the two models.

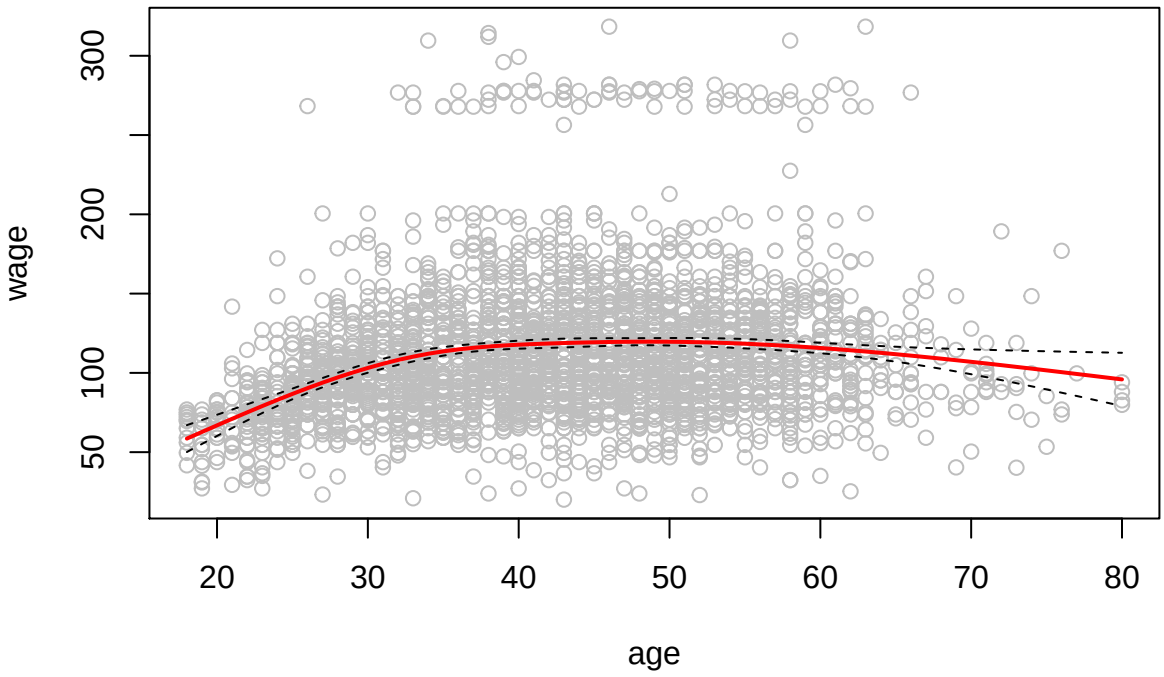
In order to instead fit a natural spline, we use the `ns()` function. Here we fit a natural spline with four degrees of freedom.

```
natural_spline <- lm(wage ~ ns(age, df=4), data=Wage)
```

```
# Observe coefficients
tidy(natural_spline)
```

```
## # A tibble: 5 x 5
##   term              estimate std.error statistic  p.value
##   <chr>                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)         58.6       4.24     13.8  3.39e-42
## 2 ns(age, df = 4)1    60.5       4.19     14.4  1.11e-45
## 3 ns(age, df = 4)2    42.0       4.37      9.60 1.67e-21
## 4 ns(age, df = 4)3    97.0      10.4       9.34 1.80e-20
## 5 ns(age, df = 4)4     9.77      8.66      1.13 2.59e- 1
```

```
pred2 = predict(natural_spline, newdata=list(age=age.grid), se=T)
plot(age, wage, col="gray")
lines(age.grid, pred2$fit, col="red", lwd=2)
lines(age.grid, pred2$fit + 2 * pred2$se, lty="dashed")
lines(age.grid, pred2$fit - 2 * pred2$se, lty="dashed")
```



As with the `bs()` function, we could instead specify the knots directly using the `knots` option.

As you can see, there is not a whole lot of visual difference other than the fact that the confidence intervals are near the boundary and are a little more under control. This is fine, as that is all we set out to accomplish.

We finish this section with a quick implementation of `smooth.spline()`, which as the name suggests, will help us fit a smooth spline to `wage`. This function allows for cross-validation for the best selection of the penalty parameter λ, so we will do an example with and without it.

In order to fit a **smoothing spline**, we use the `smooth.spline()` function.

```
ss <- smooth.spline(age, wage, df=16)
ss_cv <- smooth.spline(age, wage, cv=TRUE)
```

```
## Warning in smooth.spline(age, wage, cv = TRUE): cross-validation with non-unique
## 'x' values seems doubtful
```

```
# Get degrees of freedom from the cross validation
ss_cv$df
```

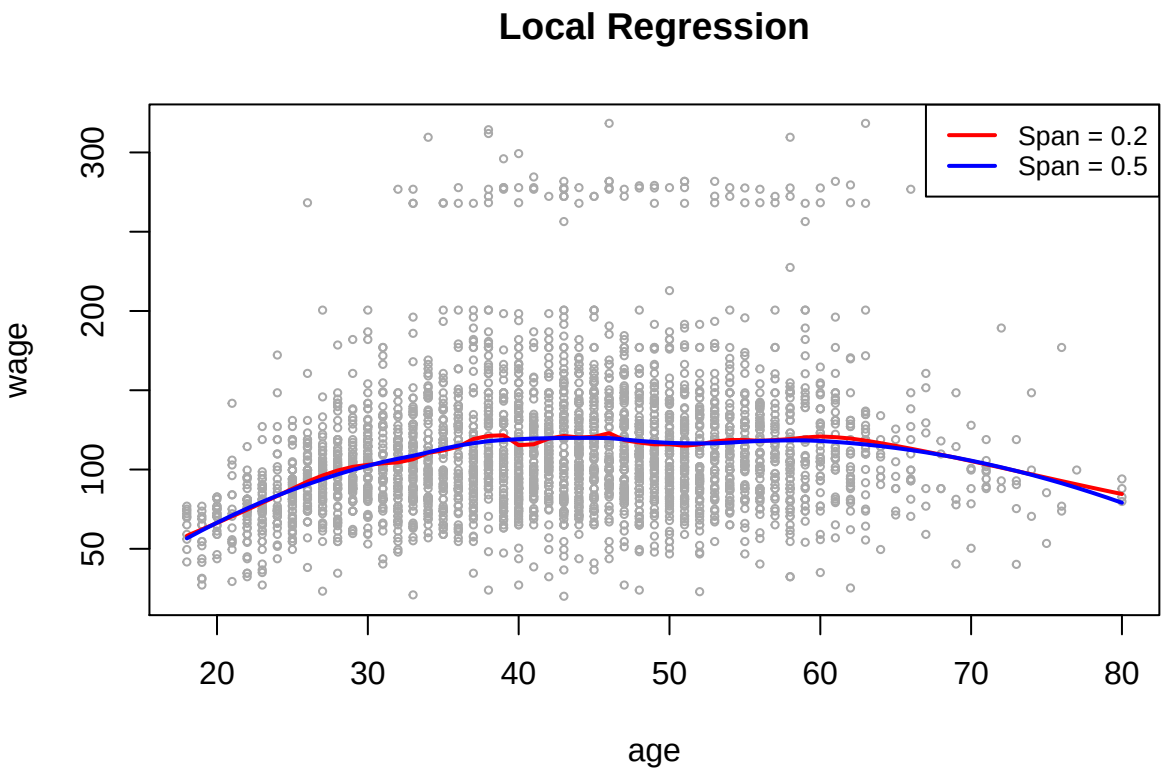
```
## [1] 6.794596
```

```
plot(Wage$age, Wage$wage, xlim = Wage$agelims, cex = 0.5, col="darkgrey")
title("Smoothing Splines")
lines(ss, col='red', lwd=2)
lines(ss_cv, col='blue', lwd=2)
legend('topright', legend=c("16 DF", "6.8 DF"),
       col = c("red", "blue"), lty=1, lwd=2, cex=0.8)
```

Notice that in the first call to `smooth.spline()`, we specified `df=16`. The function that determines which value of λ leads to 16 degrees of freedom. In the second call to `smooth.spline()`, we select the smoothness level by cross-validation; this results in a value of λ that yields 6.8 degrees of freedom.

In order to perform **local regression**, we use the `loess()` function.

```
plot(age, wage, xlim=agelims, cex=0.5, col="darkgrey")
title("Local Regression")
fit = loess(wage ~ age, span = 0.2, data=Wage)
fit2 <- loess(wage ~ age, span = 0.5, data=Wage)
lines(age.grid, predict(fit, data.frame(age = age.grid)), col='red', lwd=2)
lines(age.grid, predict(fit2, data.frame(age = age.grid)), col='blue', lwd=2)
legend("topright", legend=c("Span = 0.2", "Span = 0.5"), col=c("red", "blue"), lty=1, lwd=2, cex=0.8)
```



Here we have performed local linear regression using spans of 0.2 and 0.5: that is, each neighbourhood consists of 20% or 50% of the observations. The larger the span, the smoother the fit. The `locfit()` library can also be used for fitting local regression models in `R`.

Generalized Additive Models (GAMs)

We now fit a GAM to predict `wage`using natural splines functions of `year` and `age`, treating `education` as a qualitative predictor. Since this is just a big linear regression model using an appropriate choice of basis functions, we can simply do this using the `lm()` function.

Our first example will build off the previous section. That is, we will use natural splines of `age` and `year`, along with `education` to fit a model for `wage`. We can accomplish this with the tools we learned so far.

```
g1 <- lm(wage ~ ns(year, 4) + ns(age, 5) + education, data=Wage)

tidy(g1)
```

```
## # A tibble: 14 x 5
##    term                    estimate std.error statistic   p.value
##    <chr>                      <dbl>     <dbl>     <dbl>     <dbl>
##  1 (Intercept)                46.9       4.70     9.98  4.26e- 23
##  2 ns(year, 4)1                8.62      3.47     2.49  1.29e-  2
##  3 ns(year, 4)2                3.76      2.96     1.27  2.04e-  1
##  4 ns(year, 4)3                8.13      4.21     1.93  5.37e-  2
##  5 ns(year, 4)4                6.81      2.40     2.84  4.55e-  3
##  6 ns(age, 5)1                45.2       4.19    10.8   1.44e- 26
##  7 ns(age, 5)2                38.4       5.08     7.57  4.78e- 14
##  8 ns(age, 5)3                34.2       4.38     7.81  7.69e- 15
##  9 ns(age, 5)4                48.7      10.6      4.60  4.31e-  6
## 10 ns(age, 5)5                 6.56      8.37     0.784 4.33e-  1
```

```
## 11 education2. HS Grad            11.0       2.43     4.52  6.43e-  6
## 12 education3. Some College       23.5       2.56     9.16  9.12e- 20
## 13 education4. College Grad       38.3       2.55    15.0   2.40e- 49
## 14 education5. Advanced Degree    62.6       2.76    22.7   5.57e-105
```

We now fit a model using smoothing splines rather than natural splines. In order to fit more general sorts of GAMs, using smoothing splines or other components that cannot be expressed in terms of basis functions and then fit using least squares regression, we will need to use the `gam` library in `R`.

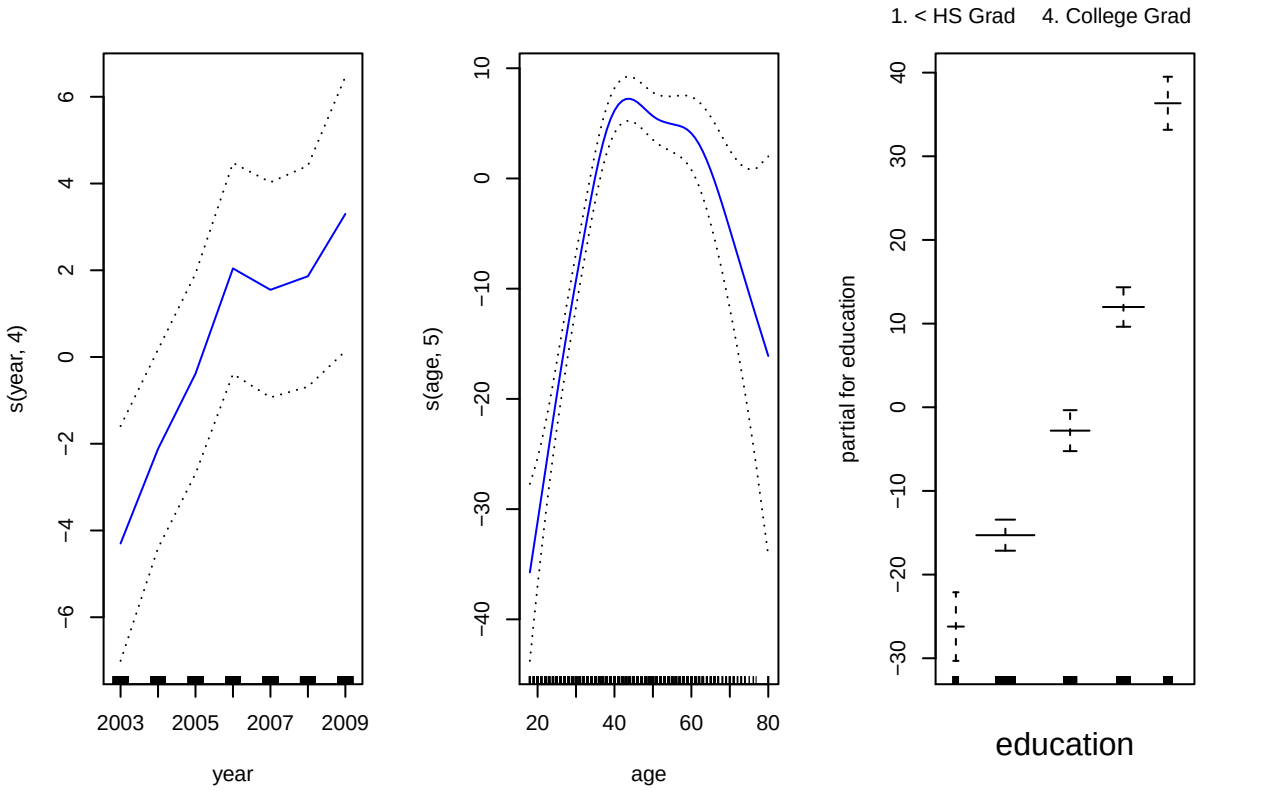
The `s()` function, which is part of the `gam` library, is used to indicate that we would like to use a smoothing spline. We specify that the function of `year` that should have 4 degrees of freedom, and that the function of `age` will have 5 degrees of freedom. Since `education` is qualitative, we leave it as is, and it is converted into four dummy variables. We use the `gam()` function in order to fit a GAM using these components. All of the terms are fit simultaneously, taking each other into account to explain the response.

Next, we use smoothing splines instead of natural cubic ones. To do this, we will use a new library `gam`. Note that `s()` is used to specify a smoothing spline instead of `smooth.spline()` in the previous section. `s()` is local to the `gam` library and is specifically used within the `gam()` function.

```
g2 <- gam(wage ~ s(year, 4) + s(age, 5) + education, data=Wage)
```

Let's check out the results of the natural spline (GAM) (there are both pretty similar):

```
par(mfrow =c(1,3))
plot(g2, se=TRUE, col='blue')
```



The generic `plot()` function recognizes that `g2` is an object of class `gam`, and invokes the appropriate `plot.gam()` method. Conveniently, even though `g1` is not a class of `gam` but rather a class of `lm`, we can *still* use `plot.gam()` on it.

Notice here we had to use `plot.gam()` rather than the *generic* `plot()` function.

In these plots, the function of `year` looks rather linear. We can perform a series of ANOVA tests in order to

determine which three models is best: a GAM that excludes `year` (M1), a GAM that uses a linear function of `year` (M2), or a GAM that uses a spline function of `year` (M3).

`year` appears to be approximately linear in `wage`. Is it even necessary to create a natural spline for it and we can just model a linear function of year? Let's test to see which variation will perform the best:

```
g1 <- gam(wage ~ s(age,5) + education, data=Wage) # Exclude year
g2 <- gam(wage ~ year + s(age, 5) + education, data=Wage) # Wage linear in 'year'
g3 <- gam(wage ~ s(year,4) + s(age, 5) + education, data=Wage) # Natural spline for 'year'

anova(g1, g2, g3)
```

```
## Analysis of Deviance Table
##
## Model 1: wage ~ s(age, 5) + education
## Model 2: wage ~ year + s(age, 5) + education
## Model 3: wage ~ s(year, 4) + s(age, 5) + education
##   Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
## 1      2990    3711731
## 2      2989    3693842  1  17889.2 0.0001419 ***
## 3      2986    3689770  3   4071.1 0.3483897
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

According to these results, we do have evidence to suggest that excluding `year` would not be a good choice. In addition, we do not have enough evidence to suggest that including the natural spline function for `year` would lead to a better performing model.

We find that there is compelling evidence that a GAM with a linear function of `year` is better than a GAM that does not include `year` (p-value: 0.00014). However, there is no evidence that a non-linear function of `year` is needed (p-value = 0.3483). In other words, based on the results of this ANOVA, M2 is preferred.

The `summary()` function produces a summary of the GAM fit:

```
summary(g3)
```

```
##
## Call: gam(formula = wage ~ s(year, 4) + s(age, 5) + education, data = Wage)
## Deviance Residuals:
##     Min      1Q  Median      3Q     Max
## -119.43  -19.70   -3.33   14.17  213.48
##
## (Dispersion Parameter for gaussian family taken to be 1235.69)
##
##     Null Deviance: 5222086 on 2999 degrees of freedom
## Residual Deviance: 3689770 on 2986 degrees of freedom
## AIC: 29887.75
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##              Df  Sum Sq Mean Sq F value    Pr(>F)
## s(year, 4)    1   27162   27162  21.981 2.877e-06 ***
## s(age, 5)     1  195338  195338 158.081 < 2.2e-16 ***
## education     4 1069726  267432 216.423 < 2.2e-16 ***
## Residuals  2986 3689770    1236
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##             Npar Df Npar F  Pr(F)
## (Intercept)
## s(year, 4)        3  1.086 0.3537
## s(age, 5)         4 32.380 <2e-16 ***
## education
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

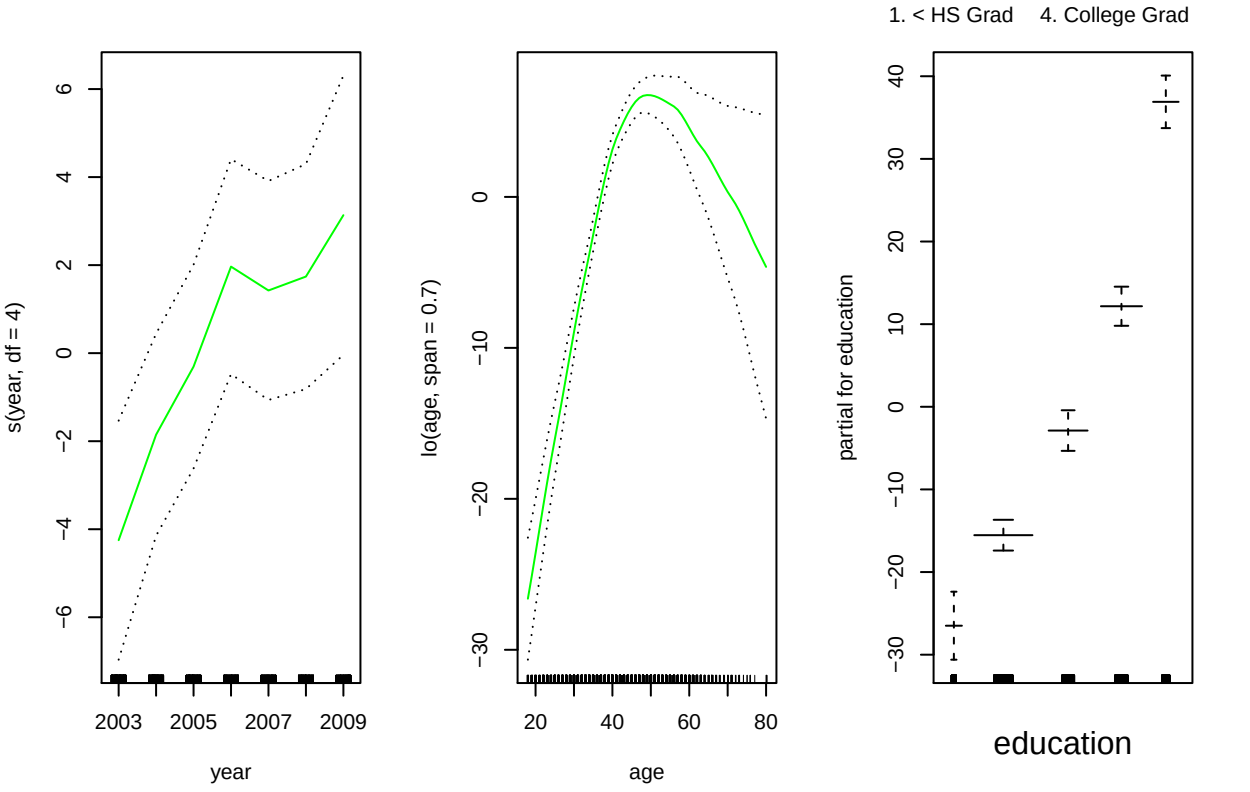
The p-values for `year` and `age` correspond to a null hypothesis of a linear relationship versus the alternative of a non-linear relationship. The large p-value for `year` reinforces our conclusion from the ANOVA test that a linear function is adequare for this term. However, there is very clear evidence that a non-linear term is required for `age`.

We can make predictions from `gam` objects, just like from `lm` objects using the `predict()` method for the class `gam`. Here we make predictions on the training set.

```
preds = predict(g2, newdata=Wage)
```

We can also use local regression fits as building blocks in a GAM, using the `lo()` function.

```
gam_lo <- gam(wage ~ s(year, df = 4) + lo(age, span = .7) + education, data = Wage)
par(mfrow = c(1,3))
plot.Gam(gam_lo, se = TRUE, col = 'green')
```



Here we have used local regression for the `age` term, with a span of 0.7. We can also use the `lo()` function to create interactions before calling a `gam()` function.

`lo()` can also be used to create interactions between variables. The `akima` package can be used to help visualize the results of these interactions, but we do not explore that here.

```
library(akima)
gam_lo_i <- gam(wage ~ lo(year, age, span=0.5) + education, data=Wage)
```

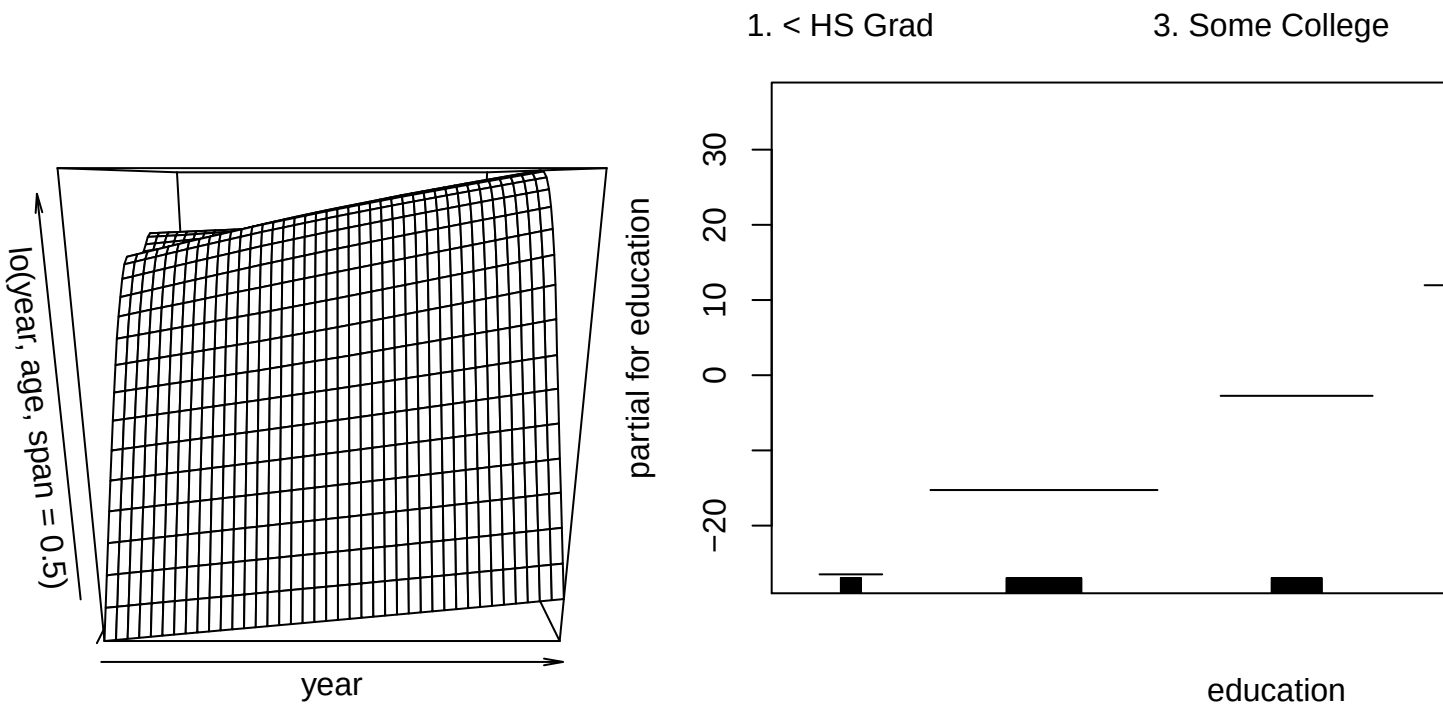
```
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : liv
## too small. (Discovered by lowesd)
```

```
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : lv
## too small. (Discovered by lowesd)
```

```
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : liv
## too small. (Discovered by lowesd)
```

```
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : lv
## too small. (Discovered by lowesd)
```

```
plot(gam_lo_i)
```

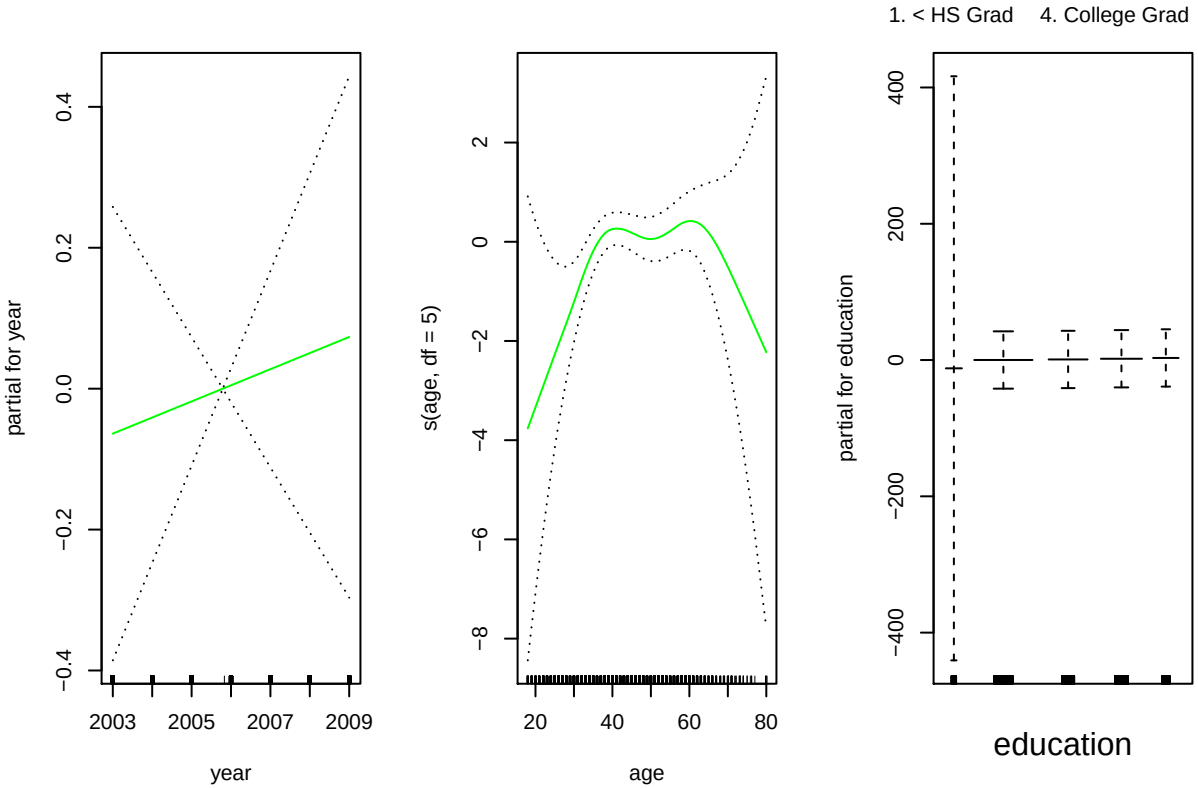


As a final example, we will estimate a logistic regression GAM using an indicator for `Wage > 250` as our dependent variable.

In order to fit a logistic regression GAM, we once again use the `I()` function in constructing the binary response variable, and set `family=binomial`.

```
gam_lr <- gam(I(wage>250) ~ year + s(age, df=5) + education, family=binomial, data=Wage)

par(mfrow = c(1,3))
plot(gam_lr, se=TRUE, col='green')
```



It is easy to see there are no high earners in the <HS category.

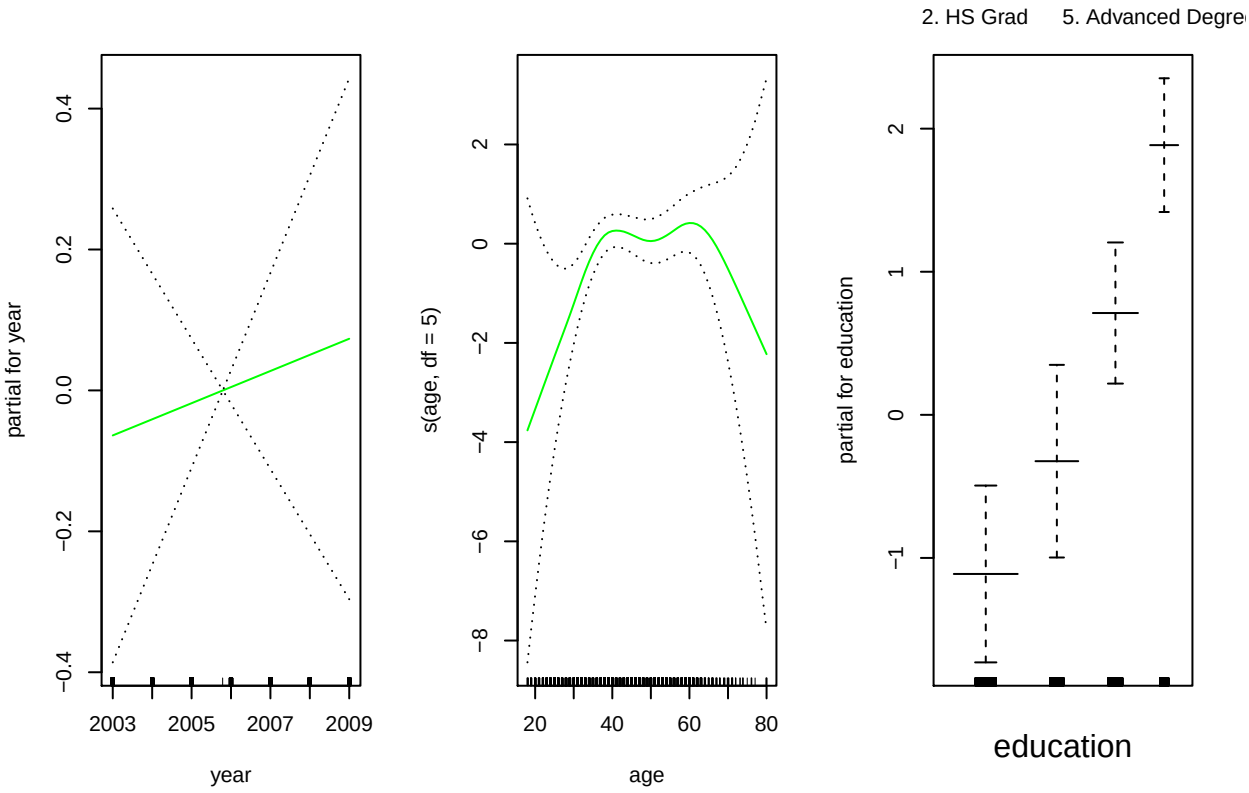
```
table(education, I(wage>250))
```

```
##
## education             FALSE TRUE
##   1. < HS Grad          268    0
##   2. HS Grad            966    5
##   3. Some College       643    7
##   4. College Grad       663   22
##   5. Advanced Degree    381   45
```

Hence, we fit a logistic regression GAM using all but this category. This provides more sensible results.

```
gam_lr <- gam(I(wage>250) ~ year + s(age, df=5) + education, family=binomial, data=Wage, subset = (educa

par(mfrow = c(1,3))
plot(gam_lr, se=TRUE, col='green')
```


partial for year
0.4
0.2
0.0
−0.2
−0.4
2003
2005
2007
2009
year
s(age, df = 5)
2
0
−2
−4
−6
−8
20
40
60
80
age
2. HS Grad
5. Advanced Degre
partial for education
2
1
0
−1
education