Nonlinear Modelling in R with GAMS

Miao Cai

1/9/2019

1 Introduction to Generalized Additive Models

Trade-offs in Model Building

- Linear models
- GAMs
- Black-Box ML

```
library(mgcv)
gamfit = gam(y ~ s(x), data = dat)
```

The flexible smooths in GAM are constructed of many smaller functions, which are called basis functions. Each smooth is a sum of a number of basis functions and each basis function is multiplied by a coefficient, each of which is a parameter in the model

1.1 Basis Functions and Smoothing

The flexibility of GAM makes it easy to overfit the data.

- Close to the data (avoiding under-fitting)
- Not fitting the noise (avoid overfitting)

1.1.1 Smoothing parameter

The complexity of likelihood, or how much the curve changes its shape is meausred by **wiggliness**

$$\text{Fit} = \text{Likelihood} - \lambda * \text{Wiggliness}$$

The key is balance the trade-off between Likelihood and Wiggliness. The smoothing parameter λ is used to control the balance.

Normally we let the package chooses the smoothing parameter.

```
# Setting a fixed smoothing parameter
gam(y ~ s(x), data = dat, sp = 0.1)
gam(y ~ s(x, sp = 0.1), data = dat)

# Smoothing via restricted maximum likelihood
gam(y ~ s(x), data = dat, method = "REML")
```

`REML` is recommended as the smoothing algorithm. It is most likely to give stable and reliable results.

1.1.2 Number of basis functions

Set the number of basis functions

```
gam(y ~ s(x, k = 3), data = dat, method = "REML")
gam(y ~ s(x, k = 10), data = dat, method = "REML")
```

Use the defaults

```
gam(y ~ s(x), data = dat, method = "REML")
```

We can test if the number of basis functions is adequate using statistical tests.

1.2 Multivariate GAMs

- The `mgcv` package does not use character variables
- Normally we give splines to continuous variables, while keep categorical variables as they are in GAMs.
- Set `by` parameter, we can specify different smoothing in different categories.

```
model4b <- gam(hw.mpg ~ s(weight, by = fuel) + fuel, data = mpg,
               method = "REML")
```

2 Interpreting GAMs

A good way to interpret significant smooth terms in GAMs is: **A significant smooth term is the one where you cannot draw a horizontal line through the 95% confidence interval.**

Note that a high EDF (effective degrees of freedom) doesn't mean significance or vice versa. In the example model, the price term is non-linear but non-significant, meaning that it has some complexity, but there isn't certainty to the shape or direction of its effect.

The plots generated by `mgcv`'s `plot()` function are partical effect plots. That is, they show the component effect of each of the smooth or linear terms in the model, which add up to the overall prediction.

```
plot(gam_model, select = c(2, 3))
plot(gam_model, pages = 1)
plot(gam_model, pages = 1, all.terms = TRUE)
```

The first option we have when making our plots is which partial effect to show.

- The `select` argument chooses which terms we plot, with the default being all of them.
- Normally, each plot gets its own page, but using the pages argument, you can decide how many total pages to spread plots across.
- Finally, by default we only see the smooth plots, but by setting `all.terms = TRUE`, we can display partial effects of linear or categorical terms as well.

```
plot(gam_model, rug = TRUE)
plot(gam_model, residuals = TRUE)
plot(gam_model, rug = TRUE, residuals = TRUE,
     pch = 1, cex = 1)
```

We often want to show data alongside model predictions.

- The `rug` argument puts X-values along the bottom of the plot.
- The residuals argument puts partial residuals on the plot.

Partial residuals are the difference between the partial effect and the data, after all other partial effects have been accounted for.

2.1 Showing standard errors

```
plot(gam_model, se = TRUE)
plot(gam_model, shade = TRUE)
```

By default, plot will put standard errors on your plots. These show the 95% confidence interval for the mean shape of the effect. It is often preferable to use shading rather than lines to show these intervals, which can be achieved using the `shade = TRUE` argument.

2.2 Transforming standard errors

```
plot(gam_model, seWithMean = TRUE)
plot(gam_model, seWithMean = TRUE, shift = coef(gam_model)[1])
```

It is often useful to plot the standard errors of a partial effect term combined with the standard errors of the model intercept. This is because the confidence intervals at the mean value of a variable can be very tiny, and don't reflect the overall uncertainty in our model. Using the `seWithMean` argument adds in this uncertainty.

To make the plot even more interpretable, it is useful to shift the scale so that the intercept is included. Using the `shift` argument, we can shift the scale by value of the intercept, which is the first coefficient of the model. Now, the partial effect plot has a more natural interpretation: it shows us the prediction of the output, assuming other variables are at their average values.

2.3 Model checking

```
gam.check(mod)
```

```
Method: REML   Optimizer: outer newton
full convergence after 9 iterations.
Gradient range [-0.0001467222,0.00171085]
(score 784.6012 & scale 2.868607).
Hessian positive definite, eigenvalue range [0.00014,198.5]
Model rank =  7 / 7

Basis dimension (k) checking results. Low p-value
(k-index<1) may indicate that k is too low, especially
if edf is close to k'.

        k'  edf k-index p-value
s(x1) 3.00 1.00    0.35  <2e-16 ***
s(x2) 3.00 2.88    1.00    0.52
 ---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
```

- Full converge: not convergence occurs when there are too many parameters in the model for not enough data.
- Basic dimension checking results show a statistical test for patterns in model residuals, which should be random. Small p-values indicate that residuals are not randomly distributed, which often means that there are not enough basis functions. Always visualize your results and compare the `k` and `edf` values in addition to looking at the p-value.
- Fixing one problem can reveal another problem, so it is always important to re-run `gam.check` after changing models.

2.4 Concurvity

In GAMs, even if two variables are not collinear, they may have concurvity, one may be a smooth curve of another.

```
concurvity(m1, full = TRUE)
```

```
            para s(X1) s(X2)
worst          0  0.84  0.84
observed       0  0.22  0.57
estimate       0  0.28  0.60
```

`mgcv`'s concurvity function measures concurvity in model variables. Like `gam.check()`, we run this function on a model object to examine the quality of our model.

Concurvity has two modes:

- `full = TRUE` reports overall concurvity for each smooth. Specifically, it shows how much each smooth is predetermined by all the other smooths.

Since concurvity is complex, the function reports three different ways of measuring concurvity. Each is better in some situations. What is important is that you should always look at the worst case, and if the value is high (say, **over 0.8**), you should inpect your model carefully.

```
concurvity(m1, full = FALSE)
```

- If any of the values from the `full` model is high, we will want to also use the second mode, setting `full = FALSE`. With `full = FALSE`, the function returns matrices of pairwise concurvities. These show the degree to which each variable is predicted by each other variable, rather than all the other variables. This can be used to pinpoint which variables have a close relationship. Once again, the function returns three measures, this time as three matrices. Look for the worse-case scenario and see if variables with high values have problematic shapes or confidence intervals.

3 Logistic GAMs

3.1 Visualizing logistic GAMs

When we plot the output of a logistic GAM, we see the partial effect of smooths on the log odds scale. It can be difficult to interpret this. We understand the shape of the effect, but the magnitude of the effect on probability is not immediately apparent.

```
plot(binom_mod, pages = 1, trans = plogis)
```

When plotting, we can convert our output to the probability scale by using the `trans` argument. The `trans` argument takes a function that transforms the output, so we can pass the `plogis()` logistic function to this argument and all values in plot will be transformed from log-odds to proabilities.

When we transform this way, we can see that our partial effects are all centered on an average value of 0.5. This is because we are looking at each partial effect with no intercept. To incorporate the model intercept, we can use the shift argument that we learnt earlier. `shift` adds its value to all model outputs, before the transformation in the function we pass to trans. So we shift our outputs by passing the intercept, extracting it from the model object with the `coef()` function.

```
plot(binom_mod, pages = 1, trans = plogis,
     shift = coef(binom_mod)[1])
plot(binom_mod, pages = 1, trans = plogis,
     shift = coef(binom_mod)[1], seWithMean = TRUE)
```

Now, each partial effect plot can be interpreted as showing the probability of the outcome if all other variables were at their average value. At their own average value, you got only the effect of the intercept.

After adding `seWithMean = TRUE`, we add the intercept uncertainty to the model. Now the confidence interval in our partial effect plots also have a natural interpretation. They may be interpreted as **the range of uncertainty of the probability of the outcome for any value of the variable, holding other variables equal at their average value**.

3.2 Making predictions

```
predict(log_mod2, type = "link")
predict(log_mod2, type = "response")
```

By default, the `predict()` function returns values on the "link" scale, That is, the scale on which the model was fit to the data. For logistic model, this is the log-odds scale. We can have `predict()` return results on the probability scale by using the argument `type = "response"`. This is equivalent of running the `plogis()` logistic function on our predictions.

```
predict(log_mod2, type = "link", se.fit = TRUE)
```

If we set the argument `se.fit = TRUE` in our call, `predict()` returns a list where the first element fit contains our vector of predictions, and the second element named `se.fit` contains standard errors for our predictions.

Standard errors are only approximations when we use the probability scale. This is because errors are non-symmetrical on this scale. If you use standard error to construct confidence intervals for your predictions, you should do so on the log-odds scale and then convert them to probability using the `plogis()` lgositic function.

```
predict(log_mod2, type = "terms")
```

```
      s(n_acts) s(bal_crdt_ratio) s(avg_prem_balance) s(retail_crdt_ratio) s(avg_fin_balance) s(mortgag
1     1.2115213      0.3327855673        -0.135920526         0.0678994892       -0.040572487   -0.2918
2    -0.8850186     -0.4058818961        -0.135920526        -0.0075325272       -0.040572487   -0.0209
3     0.5693622      0.2972364048        -0.135920526         0.0678994892        0.156060412   -0.0209
4     0.8974704      0.3827671103        -0.135920526         0.0626482277        0.032042157   -0.0209
5     0.8974704     -0.0727464938        -0.135920526        -0.0686510698        0.065216680    0.2906
6    -0.6228781      0.1936974771        -0.135920526        -0.0075325272        0.776081676   -0.0209
7     0.3642246      0.3377181800        -0.135920526        -0.0075325272       -0.040572487    0.2849
8    -0.8850186     -0.4058818961        -0.135920526        -0.0075325272       -0.040572487   -0.0209
9     1.0209905      0.3604064595         0.317309246        -0.0253158695       -0.040572487    0.3551
10    1.7675666     -0.4533384774         0.346837355         0.0377046376        0.150927175    0.1269
```

In multiple regression, it is often useful to understand how each term contributes to an individual prediction. We can examine this by setting the type argument to "terms" in the `predict()` function. This will produce a matrix showing the contribution of each smooth to each prediction.

If we were to sum across all the columns of this matrix and add the intercept, we would have our overall prediction on the log-odds scale.

```
predict(log_mod2, type = "terms")[1,]
```

```
s(n_acts)    s(bal_crdt_ratio)
          1.21152126           0.33278557
 s(avg_prem_balance) s(retail_crdt_ratio)
         -0.13592053           0.06789949
  s(avg_fin_balance)      s(mortgage_age)
         -0.04057249          -0.29183903
```

```
s(cred_limit)
  -0.37055621
```

Here we look at the first row of this output to see the role of each term in influencing our prediction probability. This allows us to explain model predictions. For instance, the `n_acts` has about four times the effect in increasing purchase probability prediction than `retail_crdt_ratio`. If we add these terms up, add the intercept, and transform using the `plogis()` function, we get this data point's predicted purchase probability.

4 Course review

1. Chapter 1

- GAM theory
- Fitting GAMs
- Mixing linear and nonlinear terms

2. Chapter 2

- Interpreting GAMs
- Visualizing GAMs
- Model-checking and concurvity

3. Chapter 3

- 2-D interaction and spatial data
- Interactions with different scales
- Continuous-categorical interaction

4. Chapter 4

- Logistic GAMs
- Plotting logistic ouputs
- Making predictions

4.1 GAMs and the tidyverse

```
library(broom)
augment(gam_model)
tidy(gam_model)
glance(gam_model)
```

```
library(caret)
train(x, y, method = "gam")
```

Other types of smooths

```
?smooth.terms
```

Other types of outcomes/distributions

```
?family.mgcv
```

Variable selection

```
?gam.selection
```

Complex model structures

```
?gam.models
```