**Mixed model review**

Let’s review what these things means. Mixed effects models, apparently the main focus of this blog over the years, are used to estimate “random” or “varying” effects. Here is the classic equation set up:

\[\mathbf{y} = \mathbf{X\beta} + \mathbf{Zb} + \mathbf{\epsilon} \\ \mathbf{b} \sim \textsf{Normal}(0, \sigma\_b) \\ \mathbf{\epsilon} \sim \textsf{Normal}(0, \sigma\_y) \\ \mathbf{X}: \textrm{fixed effects model matrix} \\ \mathbf{Z}: \textrm{random effects model matrix} \\ \sigma\_b, \sigma\_y : \textrm{variance components} \\ \sigma\_b : \textrm{where the magic happens} \\\]

The magic here is the *σb*, as it ties all of the individual effects in **b** under a common distribution. If *σb* were replaced with a fixed number like 10, then all of the effects in **b** would be independent and unaware of each other: There would be no pooling of information between the groups.

Consider the [radon dataset](https://mc-stan.org/rstanarm/reference/rstanarm-datasets.html) example from [Gelman and Hill (2007)](https://amzn.to/3aVa9tB). Radon measurements were taken in Minnesota counties. We would like to estimate the average radon measurement for each county. We have a repeated measures situation, and some counties have more observations than others. We use a mixed effects model to estimate a population distribution of county estimates, and the county-level estimates are randomly varying effects. They are drawn from a random distribution, the scale of which we estimate from the data.

library(tidyverse)

theme\_set(theme\_grey(base\_size = 14))

library(brms)

radon <- rstanarm::radon

b\_radon <- brm(

log\_radon ~ 1 + (1 | county),

radon,

family = gaussian,

file = "radon"

)

b\_radon

#> Family: gaussian

#> Links: mu = identity; sigma = identity

#> Formula: log\_radon ~ 1 + (1 | county)

#> Data: radon (Number of observations: 919)

#> Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;

#> total post-warmup samples = 4000

#>

#> Group-Level Effects:

#> ~county (Number of levels: 85)

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> sd(Intercept) 0.30 0.05 0.22 0.40 1.00 1782 2894

#>

#> Population-Level Effects:

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> Intercept 1.35 0.05 1.26 1.45 1.00 2749 3198

#>

#> Family Specific Parameters:

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> sigma 0.77 0.02 0.73 0.80 1.00 7374 3105

#>

#> Samples were drawn using sampling(NUTS). For each parameter, Bulk\_ESS

#> and Tail\_ESS are effective sample size measures, and Rhat is the potential

#> scale reduction factor on split chains (at convergence, Rhat = 1).

Here sd(Intercept) corresponds to *σb*.

We can plot the observed county means alongside the model estimated ones. First, I do some wrangling so that the difference between observed means and estimated means are computed for use later on.

radon\_aug <- radon %>%

# add ns and means

group\_by(county) %>%

mutate(

observed\_mean = mean(log\_radon),

county\_n = n()

) %>%

ungroup() %>%

# add fitted values

tidybayes::add\_fitted\_draws(b\_radon) %>%

mutate(

observed\_minus\_model = observed\_mean - .value

) %>%

# summarize fitted values

ggdist::median\_qi(.value, observed\_minus\_model)

radon\_aug$type <- "mixed model estimates"

radon$type <- "observed means"

ggplot(radon\_aug) +

aes(x = fct\_infreq(county), y = log\_radon) +

stat\_summary(

aes(color = type, shape = type),

data = radon,

fun = mean,

geom = "point"

) +

geom\_point(

aes(y = .value, color = type, shape = type)

) +

# want to include 0 in the figure

geom\_blank(aes(y = 0)) +

labs(

x = "county (in decreasing order by sample size)",

y = "log(radon)"

) +

geom\_hline(yintercept = fixef(b\_radon)[1]) +

scale\_color\_manual(values = c("blue", "grey40")) +

labs(color = NULL, shape = NULL) +

theme(

axis.text.x = element\_blank(),

axis.ticks.x = element\_blank(),

panel.grid.major.x = element\_blank(),

panel.grid.minor.x = element\_blank(),

legend.title = element\_blank(),

legend.position = "top",

legend.direction = "horizontal",

legend.justification = "left",

)

We see a classic example of partial pooling. For counties with many observations, the estimate mean is hardly adjusted. For counties with less data, the estimate is pulled towards the population mean (Intercept in the summary above).

The following plot shows difference between the observed means and the estimated means, subtracting the grey triangles from the blue squares in the plot above.

ggplot(radon\_aug) +

aes(x = county\_n, y = observed\_minus\_model) +

geom\_point() +

labs(

x = "Number of observations in county",

y = "Observed mean - estimated mean"

)

The contention behind the *smooths = random effects* claim is that what we just did is a case of *smoothing*. These random effects are, in a way, smoothed fixed effects.

**But what’s smoothing?**

Now let’s walk through a generalized additive model to demonstrate a penalized smoothing spline. That was a mouth full, but basically additive models are like the smoothing expansion pack for the standard linear model. We’re still doing regression, but we have some new syntax and our models can do nonlinear relationships more easily now.

I will walk through a basic example of how a spline’s basis functions are weighted to approximate a nonlinear trend, but this is not going to be a full tutorial.

We use the [mcycle](https://rdrr.io/pkg/MASS/man/mcycle.html) dataset which gives the head acceleration in a simulated motorcycle accident. We are going to fit a model, plot the smooth from it, and then we are going to work through what the model did.

library(mgcv)

mcycle <- MASS::mcycle %>%

tibble::rowid\_to\_column()

# Fit the model

gam\_20 <- gam(

accel ~ 1 + s(times, bs = "cr", k = 20),

data = mcycle,

method = "REML"

)

mcycle$.fitted <- fitted(gam\_20)

ggplot(mcycle) +

aes(x = times, y = accel) +

geom\_point(alpha = .5) +

geom\_line(aes(y = .fitted), color = "blue") +

labs(x = "time after impact [ms]", y = "acceleration [g]")

So what happened here? We will cover it visually.

**Splines are the sums of weighted wiggles**

Let’s look at the regression formula.

formula(gam\_20)

#> accel ~ 1 + s(times, bs = "cr", k = 20)

We told gam() to estimate accel using an intercept term and a smooth term on the time predictor (s(times, ...)). Specifically, we created our smooth using a cubic regression spline basis (bs = "cr") with k = 20 - 1 curves. Our model is estimating a function by adding up smaller components called *basis functions*, and the space that defines those components is the *basis*. These basis functions are weighted and summed together to produce a smooth trend called a *spline*. The name *splines* is inspired by drafting splines which are flexible strips of wood that can be weighted and anchored in place to make a nice curve.

To reiterate, conceptually, we are decomposing the times predictor into a bunch of individual wiggly lines (basis functions), and these are weighted and summed together to approximate some nonlinear function. One line I appreciate from his description is that with splines, we replace a predictor variable, like times, with a set of “synthetic” predictor variables.

An easy way to pull the wiggles is to use the model matrix. We have 20 columns for the intercept and 19 basis functions.

model.matrix(gam\_20) %>%

tibble::trunc\_mat(width = 72)

#> # Description: dbl[,20] [133 x 20]

#> `(Intercept)` `s(times).1` `s(times).2` `s(times).3` `s(times).4`

#>

#> 1 1 -0.143 -0.165 -0.154 -0.311

#> 2 1 -0.0317 -0.193 -0.133 -0.293

#> 3 1 0.292 -0.270 -0.0716 -0.242

#> 4 1 0.492 -0.309 -0.0357 -0.208

#> 5 1 0.669 -0.332 -0.00603 -0.175

#> 6 1 0.933 0.0792 -0.0455 -0.0280

#> 7 1 0.808 0.278 -0.0969 -0.0106

#> 8 1 0.730 0.383 -0.122 -0.00387

#> 9 1 0.286 0.851 -0.182 0.00169

#> 10 1 0.123 0.957 -0.143 -0.0130

#> # ... with 123 more rows, and 15 more variables: `s(times).5` ,

#> # `s(times).6` , `s(times).7` , `s(times).8` ,

#> # `s(times).9` , `s(times).10` , `s(times).11` ,

#> # `s(times).12` , `s(times).13` , `s(times).14` ,

#> # `s(times).15` , `s(times).16` , `s(times).17` ,

#> # `s(times).18` , `s(times).19`

To visualize the matrix, I am using a helper function from my personal R package for plotting matrices in ggplot2. What we see is times on the *x* axis and one line for the intercept and for each of the basis functions.

# Helper function to plot the lines of a matrix

ggmatplot <- tjmisc::ggmatplot

# Helper function to label on a theme\_grey() plot

annotate\_grey <- tjmisc::annotate\_label\_grey

ggmatplot(cbind(mcycle$times, model.matrix(gam\_20)), x\_axis\_column = 1) +

annotate\_grey("intercept", 0, 1.02, size = 5) +

annotate\_grey("individual\nbasis\nfunctions", 0, .16, size = 5) +

expand\_limits(y = 1.2) +

labs(x = "time [ms]", title = NULL)

Now we can weight these by multiplying by the model coefficients. Here we use the diag(coef()) trick to prevent the weighted predictors from being summed together.

weighted\_coefs <- model.matrix(gam\_20) %\*% diag(coef(gam\_20))

ggmatplot(cbind(mcycle$times, weighted\_coefs), x\_axis\_column = 1) +

annotate\_grey("weighted intercept", 35, -40, size = 5) +

annotate\_grey("weighted basis functions", 0, 26, size = 5) +

labs(x = "time [ms]", title = NULL)

We can see the two main inflections points in the dataset now. The basis functions around 20 ms and 30 ms become very active in order to push the spline away from 0 at those times.

If we sum the lines together, we get the regression line (the intercept plus the smoothing spline).

ggmatplot(

cbind(mcycle$times, weighted\_coefs),

x\_axis\_column = 1,

n\_colors = 1

) +

stat\_summary(

aes(group = 1),

color = "maroon",

fun = sum,

geom = "line",

size = 1.5

) +

annotate\_grey("sum", 10, -70, size = 5, color = "maroon") +

labs(x = "time [ms]", title = NULL)

Our plots so far demonstrate regression with basis functions, but smoothing splines go one step further: They penalize wiggliness to prevent overfitting. The idea is as follows: For the above demonstration, we chose a 20-dimension spline basis (19 curves because 1 is removed for identifiability, apparently). But where did that number 20 come from? Thin air. What if we specified a dimension of 50? That’s 50 predictors (1 intercept and 49 basis functions). Isn’t it really easy to overfit the data with this approach?

Well, let’s look at a 50-dimension version.

gam\_50 <- gam(

accel ~ s(times, bs = "cr", k = 50),

data = mcycle,

method = "REML"

)

mcycle$.fitted50 <- fitted.values(gam\_50)

ggplot(mcycle) +

aes(x = times, y = accel) +

geom\_point(alpha = .5) +

geom\_line(aes(y = .fitted, color = "20"), size = 1) +

geom\_line(aes(y = .fitted50, color = "50"), size = 1) +

labs(

x = "time after impact [ms]",

y = "acceleration [g]",

color = "Dimension"

)

Huh, they hardly look any different. There is *no* overfitting. What’s going on? I already said it. **Wiggliness is being penalized.**

**Fit versus smoothness**

Behind the scenes, the model is trying to balance two competing goals. On the one hand we want to maximize the fit to the data. In linear regression, this goal amounts to minimizing the sum of squared errors. On the other hand, we want to minimize wiggliness (overfitting). In penalized smoothing splines, this is done by first specifying a penalty matrix that defines *wiggliness* for that spline basis. These two features are pitted against each other in the following equation:

\[\begin{align\*} \mathbf{\hat{β}} &= \operatorname{arg min}\_\mathbf{β}\ \|\mathbf{y} − \mathbf{Xβ}\|^2 + \lambda\mathbf{β}^\intercal\mathbf{Sβ} \\ &\ \text{(want }\mathbf{\hat{β}} \text{ that minimizes fitting error and wiggliness)}\\ \|\mathbf{y} − \mathbf{Xβ}\|^2 &: \text{sum of squared errors (minimize error to improve fit)} \\ \lambda\beta^\intercal\mathbf{Sβ} &: \text{wiggliness penalty} \\ \mathbf{Xβ} &: \text{spline basis times weights} \\ \mathbf{S} &: \text{penalty matrix (defines wiggliness for the spline)} \\ \lambda &: \text{smoothness parameter (increase to make penalty stronger)} \\ \end{align\*}\]

Don’t worry about the exact mathematics here: Just appreciate that error is now paired with wiggliness, and wiggliness is controlled by a penalty matrix **S** and a smoothness parameter *λ*. And yes, *wiggliness* is the technical term.

For our purposes, we won’t worry too much about the penalty matrix. I’ll briefly describe it. For this model, wiggliness is defined by using the second derivative of the estimated spline. The first derivative measures the slope/steepness of the curve along *x*, and the second derivatives measures how much the slope/steepness changes with *x*. Thus, wiggliness is the change in slope, and the penalty matrix provides penalties for each of the model coefficients related to this wiggliness. The excellent gratia will plot the penalty matrix as a heat map. *x* and *y* represent model coefficients (weights for the basis functions), so along the main diagonal we see a penalty applied to each coefficient. In the two off-diagonals, we see neighboring basis functions have their weights jointly unpenalized or penalized.

gam\_20 %>%

gratia::penalty() %>%

gratia::draw()

The stiffest penalties are meted out to the 5th and 6th basis functions, I think because these two basis functions cover the most rows in the dataset, but I’m not 100% confident in that explanation.

To see the overfitting in action, we can disable the smoothing penalty in the above spline-comparison plot by used fixed (fx = TRUE) regression splines. Now, the model’s main goal is to minimize the error, and the 50-dimension spline basis gives the model many, many degrees of freedom.

gam\_20\_fx <- gam(

accel ~ s(times, bs = "cr", k = 20, fx = TRUE),

data = mcycle,

method = "REML"

)

gam\_50\_fx <- gam(

accel ~ s(times, bs = "cr", k = 50, fx = TRUE),

data = mcycle,

method = "REML"

)

mcycle$.fitted20\_fx <- fitted.values(gam\_20\_fx)

mcycle$.fitted50\_fx <- fitted.values(gam\_50\_fx)

ggplot(mcycle) +

aes(x = times, y = accel) +

geom\_point(alpha = .5) +

geom\_line(aes(y = .fitted20\_fx, color = "20"), size = 1) +

geom\_line(aes(y = .fitted50\_fx, color = "50"), size = 1) +

labs(

x = "time after impact [ms]",

y = "acceleration [g]",

color = "Dimension"

)

Thus, when we disable the penalty, the 50-dimension splines is free to wiggle all over the place.

**How much smoothing happened: Effective degrees of freedom**

The smoothing parameter, *λ*, is a hyperparameter. It controls the spline coefficients (basis function weights), and it is estimated from the data. We can set the *λ* manually and crank it up way, way up. In this case, the model tries to find the least wiggly curve that decreases modeling error: A straight line.

gam\_20\_sp <- gam(

accel ~ s(times, bs = "cr", k = 20, sp = 10000000),

data = mcycle,

method = "REML"

)

mcycle$.fitted20\_sp <- fitted.values(gam\_20\_sp)

ggplot(mcycle) +

aes(x = times, y = accel) +

geom\_point() +

geom\_line(aes(y = .fitted, color = "estimated"), size = 1) +

geom\_line(aes(y = .fitted20\_fx, color = "no smoothing"), size = 1) +

geom\_line(aes(y = .fitted20\_sp, color = "10,000,000"), size = 1) +

labs(

x = "time after impact [ms]",

y = "acceleration [g]",

color = "lambda"

)

We need some way to talk about how much smoothing took place. On the one hand, we might treat each basis function as an independent predictor that uses up a full degree of freedom in fitting the curve. On the other hand, we might penalize the basis function weights so much that they produce a straight line, and thus, the batch of predictors effectively acts just like a single predictor variable would. That is, they are effectively estimating a curve that has just 1-degree-of-freedom’s worth of action in it. And indeed this is how mgcv describes the smoothness of the models: It reports the *effective (or estimated) degrees of freedom* (EDFs) behind each smooth.

If we look at the model summary, we see that our 20-dimension basis smooth has an EDF of 11.78 (see edf under Approximate significance of smooth terms).

summary(gam\_20)

#>

#> Family: gaussian

#> Link function: identity

#>

#> Formula:

#> accel ~ 1 + s(times, bs = "cr", k = 20)

#>

#> Parametric coefficients:

#> Estimate Std. Error t value Pr(>|t|)

#> (Intercept) -25.546 1.956 -13.06 <2e-16 \*\*\*

#> ---

#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

#>

#> Approximate significance of smooth terms:

#> edf Ref.df F p-value

#> s(times) 11.78 13.97 33.93 <2e-16 \*\*\*

#> ---

#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

#>

#> R-sq.(adj) = 0.782 Deviance explained = 80.1%

#> -REML = 616.01 Scale est. = 509.01 n = 133

Similarly, our 50-dimension basis smooth has only 12.81 effective degrees of freedom, but the unpenalized version uses all 49 of its basis function curves and uses 49 degrees of freedom.

gam\_50

#>

#> Family: gaussian

#> Link function: identity

#>

#> Formula:

#> accel ~ s(times, bs = "cr", k = 50)

#>

#> Estimated degrees of freedom:

#> 12.8 total = 13.81

#>

#> REML score: 616.068

gam\_50\_fx

#>

#> Family: gaussian

#> Link function: identity

#>

#> Formula:

#> accel ~ s(times, bs = "cr", k = 50, fx = TRUE)

#>

#> Estimated degrees of freedom:

#> 49 total = 50

#>

#> REML score: 403.4616

**The big trick: Turn λ into a random effect variance in a mixed model**

Okay, so far, here’s what we have:

* A spline decomposes a predictor into a number of wiggly basis functions.
* A penalized spline adds a penalty term to the model to reduce wiggliness.
* This penalty shrinks model coefficients so that they use a smaller number of degrees of freedom used by the model.
* The amount of smoothing is controlled by a hyperparameter *λ*.

We saw another hyperparameter earlier on in this post whose job was to pull individual parameter estimates closer to 0: *σ*b. Both of these hyperparameters are estimated from the data and perform shrinkage on a batch of related coefficients (random effects or basis function weights).

\[\begin{align\*} \mathbf{y} &= \mathbf{X}\mathbf{\beta} + \mathbf{Zb} + \mathbf{\epsilon} \\ \mathbf{b} &\sim \textsf{Normal}(0, \sigma/\lambda) \\ \mathbf{\epsilon} &\sim \textsf{Normal}(0,\sigma) \\ \mathbf{X}, \mathbf{Z} &: \textrm{matrices from transforming spline and penalty matrices} \\ \mathbf{X} &: \textrm{unpenalized (fixed) effect model matrix} \\ \mathbf{Z} &: \textrm{penalized (random) effects model matrix} \\ \lambda &: \textrm{smoothness parameter} \\ \sigma &: \textrm{residual variance component} \\ \end{align\*}\]

And right there, on the second line, we see the mixed effects magic again: *σ*/*λ* = *σb*: Model coefficients are related under a common distribution so that they can share information with each other. We can smuggle penalized smooths into the mixed effects framework.

**So… you can turn splines into mixed models?**

Yes. What this means is that we can use nlme or lme4 to estimate a smooth as mixed effects model. mgcv provides this feature in its aptly named smooth2random() function.

Below you can see the extent of the internal transformation needed to convert our nice wiggly cubic regression basis into matrices for the mixed effects framework.

# Construct a smoothing basis outside of a model

sm\_raw <- smoothCon(

s(times, k = 20, bs = "cr"),

data = mcycle,

absorb.cons = TRUE,

diagonal.penalty = TRUE

)

re <- smooth2random(sm\_raw[[1]], "", type = 2)

# 1 fixed effect and 18 random effect columns

mixed\_matrix <- cbind(re$Xf, re$rand$Xr)

ggmatplot(mixed\_matrix) +

labs(title = NULL)

I don’t understand it very well myself, and I probably won’t be converting smoothing bases into mixed effects model matrices and fitting them with lme4 anytime soon, but it’s useful to know about this idea because of the following fact.

**This point seems obscure, but it is what brms uses!**

What happens when we fit a smooth in brms? We are fitting a mixed effects model.

b\_gam\_20 <- brm(

accel ~ s(times, bs = "cr", k = 20),

data = mcycle,

family = gaussian,

file = "b-gam-20"

)

summary(b\_gam\_20)

#> Family: gaussian

#> Links: mu = identity; sigma = identity

#> Formula: accel ~ s(times, bs = "cr", k = 20)

#> Data: mcycle (Number of observations: 133)

#> Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;

#> total post-warmup samples = 4000

#>

#> Smooth Terms:

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> sds(stimes\_1) 5.00 1.14 3.21 7.66 1.00 1095 1579

#>

#> Population-Level Effects:

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> Intercept -25.50 2.02 -29.39 -21.45 1.00 5454 2867

#> stimes\_1 1.65 0.34 0.98 2.30 1.00 5156 3033

#>

#> Family Specific Parameters:

#> Estimate Est.Error l-95% CI u-95% CI Rhat Bulk\_ESS Tail\_ESS

#> sigma 22.77 1.45 20.20 25.81 1.00 4851 3277

#>

#> Samples were drawn using sampling(NUTS). For each parameter, Bulk\_ESS

#> and Tail\_ESS are effective sample size measures, and Rhat is the potential

#> scale reduction factor on split chains (at convergence, Rhat = 1).

You see sds(times\_1)? That’s the variance of smooth weights. You see stimes\_1? That the singe fixed effects term (re$Xf) in the code above. I’m pretty sure about this because I learned smooth2random() from studying the brms source code.

Indeed, if we look at the actual Stan code used to fit the model, we see a mixed effects model. **I suggest you only skim this code.**

b\_gam\_20$model

#> // generated with brms 2.14.4

#> functions {

#> }

#> data {

#> int N; // total number of observations

#> vector[N] Y; // response variable

#> // data for splines

#> int Ks; // number of linear effects

#> matrix[N, Ks] Xs; // design matrix for the linear effects

#> // data for spline s(times,bs="cr",k=20)

#> int nb\_1; // number of bases

#> int knots\_1[nb\_1]; // number of knots

#> // basis function matrices

#> matrix[N, knots\_1[1]] Zs\_1\_1;

#> int prior\_only; // should the likelihood be ignored?

#> }

#> transformed data {

#> }

#> parameters {

#> real Intercept; // temporary intercept for centered predictors

#> vector[Ks] bs; // spline coefficients

#> // parameters for spline s(times,bs="cr",k=20)

#> // standarized spline coefficients

#> vector[knots\_1[1]] zs\_1\_1;

#> real sds\_1\_1; // standard deviations of spline coefficients

#> real sigma; // residual SD

#> }

#> transformed parameters {

#> // actual spline coefficients

#> vector[knots\_1[1]] s\_1\_1;

#> // compute actual spline coefficients

#> s\_1\_1 = sds\_1\_1 \* zs\_1\_1;

#> }

#> model {

#> // likelihood including all constants

#> if (!prior\_only) {

#> // initialize linear predictor term

#> vector[N] mu = Intercept + rep\_vector(0.0, N) + Xs \* bs + Zs\_1\_1 \* s\_1\_1;

#> target += normal\_lpdf(Y | mu, sigma);

#> }

#> // priors including all constants

#> target += student\_t\_lpdf(Intercept | 3, -13.3, 35.6);

#> target += student\_t\_lpdf(sds\_1\_1 | 3, 0, 35.6)

#> - 1 \* student\_t\_lccdf(0 | 3, 0, 35.6);

#> target += std\_normal\_lpdf(zs\_1\_1);

#> target += student\_t\_lpdf(sigma | 3, 0, 35.6)

#> - 1 \* student\_t\_lccdf(0 | 3, 0, 35.6);

#> }

#> generated quantities {

#> // actual population-level intercept

#> real b\_Intercept = Intercept;

#> }

Okay, that’s a lot. But let me highlight and translate the key part of it.

\[\begin{align\*} \mathbf{S} &: \texttt{s\_1\_1} & \texttt{// actual spline coefficients}\\ \mathbf{\sigma} &: \texttt{sds\_1\_1} & \texttt{// standard deviations of spline coefficients}\\ \mathbf{Z} &: \texttt{zs\_1\_1} & \texttt{// standarized spline coefficients}\\ \mathbf{S} &= \sigma \cdot \mathbf{Z} & \texttt{s\_1\_1 = sds\_1\_1 \* zs\_1\_1}; \\ \mathbf{Z} &\sim \textsf{Normal}(0, 1) & \texttt{target += std\_normal\_lpdf(zs\_1\_1);}\\ \mathbf{\sigma} &\sim \textsf{StudentT}(3, 0, 35.6) & \texttt{target += student\_t\_lpdf(sds\_1\_1 | 3, 0, 35.6) ...}\\ \end{align\*}\]

The spline coefficients start as *z* scores, drawn from normal distribution with mean 0 and a standard deviation of 1. They are then scaled by *σ*, a scaling factor. This scaling factor acts as the hyperparameter that is learned from the data. This formulation of a mixed model is call the *noncentered parameterization*.

We can see completely analogous code in the Bayesian radon model above. I’ll just show the relevant part:

b\_radon$model

#> ...

#> parameters {

#> ...

#> vector[M\_1] sd\_1; // group-level standard deviations

#> vector[N\_1] z\_1[M\_1]; // standardized group-level effects

#> }

#> transformed parameters {

#> vector[N\_1] r\_1\_1; // actual group-level effects

#> r\_1\_1 = (sd\_1[1] \* (z\_1[1]));

#> }

#> model {

#> ...

#> target += student\_t\_lpdf(sd\_1 | 3, 0, 2.5)

#> - 1 \* student\_t\_lccdf(0 | 3, 0, 2.5);

#> target += std\_normal\_lpdf(z\_1[1]);

#> }

**Simple random effects are category smoothers**

One consequence of this relationship is that you can walk this relation backwards: You can fit a simple random effects using a basis matrix and penalty matrix. Indeed, mgcv provides a random effects ("re") smoother basis so we can estimate our mixed model from above using a smooth.

gam\_radon <- gam(

log\_radon ~ 1 + s(county, bs = "re"),

data = radon,

method = "REML"

)

In this case, the basis matrix is just a single indicator variable for county.

ggmatplot(model.matrix(gam\_radon)[, -1])

(Yikes, that one stings the eyes!)

The penalty matrix is a diagonal as each county effect is equally penalized.

gam\_radon %>%

gratia::penalty() %>%

gratia::draw() +

theme(axis.text = element\_blank())

mgcv knows about how *σ*/*λ* = *σ*b and it will provide the random effect variance estimate for us with [gam.vcomp()](https://rdrr.io/pkg/mgcv/man/gam.vcomp.html) (variance components):

gam.vcomp(gam\_radon)

#>

#> Standard deviations and 0.95 confidence intervals:

#>

#> [std.dev](http://std.dev) lower upper

#> s(county) 0.2976671 0.2202676 0.4022639

#> scale 0.7660696 0.7304109 0.8034692

#>

#> Rank: 2/2

Which closely matches the estimates from the Bayesian model above:

b\_radon %>%

posterior\_summary(pars = c("sd\_county\_\_Intercept", "sigma")) %>%

round(3)

#> Estimate Est.Error Q2.5 Q97.5

#> sd\_county\_\_Intercept 0.305 0.046 0.220 0.400

#> sigma 0.767 0.019 0.731 0.805

What I think is the coolest feature of random intercepts as smooths is what the effective degrees of freedom tells us:

nlevels(radon$county)

#> [1] 85

summary(gam\_radon)

#>

#> Family: gaussian

#> Link function: identity

#>

#> Formula:

#> log\_radon ~ 1 + s(county, bs = "re")

#>

#> Parametric coefficients:

#> Estimate Std. Error t value Pr(>|t|)

#> (Intercept) 1.350 0.047 28.72 <2e-16 \*\*\*

#> ---

#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

#>

#> Approximate significance of smooth terms:

#> edf Ref.df F p-value

#> s(county) 39.57 84 1.573 <2e-16 \*\*\*

#> ---

#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

#>

#> R-sq.(adj) = 0.126 Deviance explained = 16.4%

#> -REML = 1092.4 Scale est. = 0.58686 n = 919

The data have 85 counties, but there is effectively 39.5 counties worth of parameters here. For what it’s worth, we find a similar number when we ask brms to use the LOO method to estimate the effective number of parameters from its model.

loo(b\_radon)

#>

#> Computed from 4000 by 919 log-likelihood matrix

#>

#> Estimate SE

#> elpd\_loo -1083.8 28.7

#> p\_loo 44.9 4.0

#> looic 2167.7 57.5

#> ------

#> Monte Carlo SE of elpd\_loo is 0.1.

#>

#> Pareto k diagnostic values:

#> Count Pct. Min. n\_eff

#> (-Inf, 0.5] (good) 917 99.8% 398

#> (0.5, 0.7] (ok) 2 0.2% 227

#> (0.7, 1] (bad) 0 0.0%

#> (1, Inf) (very bad) 0 0.0%

#>

#> All Pareto k estimates are ok (k < 0.7).

#> See help('pareto-k-diagnostic') for details.

Here p\_loo is the effective number of parameters from a leave-one-out cross-validation method. It’s around 45. Our mgcv model has around 41.6 parameters (39.6 from the smooth plus intercept and sigma). These two parameter estimating methods are unrelated (as far as I know), but they both seem to telling us something similar about how much information in county variable we have after partially pooling down the model parameters.

For a deeper dive on random effects, Gavin Simpson recently wrote about [how to use mgcv for random effects](https://fromthebottomoftheheap.net/2021/02/02/random-effects-in-gams/), so I encourage readers to look at that post.

**Okay, why are you telling me this?**

The purpose of this post was to demystify the connection between penalized smooths and mixed effects models. Formally, it all boils down to a hyperparameter that penalizes model coefficients (or pools information) so that the model uses a smaller number of effective parameters. This connection enriches my understanding of the two kinds of models. A by-county random intercept is a category smoother. We can estimate a smooth by putting a prior on the variance of the basis function weights (provided that we incorporate the penalty matrix somehow). It’s wonderful.

Random effects are funny things. We are first taught them in a repeated measures framework as a way to handle a problem in the data (non-independence in clustered units). But they are so much more. To use a Richard McElreath phrase, they provide an “adaptive regularizing prior”: Something that constrains our estimates to reduce overfitting but is actively learned from the data.

1. sessioninfo::session\_info()
2. #> - Session info ---------------------------------------------------------------
3. #> setting value
4. #> version R version 4.0.4 (2021-02-15)
5. #> os Windows 10 x64
6. #> system x86\_64, mingw32
7. #> ui RTerm
8. #> language (EN)
9. #> collate English\_United States.1252
10. #> ctype English\_United States.1252
11. #> tz America/Chicago
12. #> date 2021-02-26
13. #>
14. #> - Packages -------------------------------------------------------------------
15. #> ! package \* version date lib
16. #> abind 1.4-5 2016-07-21 [1]
17. #> arrayhelpers 1.1-0 2020-02-04 [1]
18. #> assertthat 0.2.1 2019-03-21 [1]
19. #> backports 1.2.1 2020-12-09 [1]
20. #> base64enc 0.1-3 2015-07-28 [1]
21. #> bayesplot 1.8.0.9000 2021-02-01 [1]
22. #> boot 1.3-27 2021-02-12 [1]
23. #> bridgesampling 1.0-0 2020-02-26 [1]
24. #> brms \* 2.14.4 2020-11-03 [1]
25. #> Brobdingnag 1.2-6 2018-08-13 [1]
26. #> broom 0.7.5 2021-02-19 [1]
27. #> callr 3.5.1 2020-10-13 [1]
28. #> cellranger 1.1.0 2016-07-27 [1]
29. #> cli 2.3.1 2021-02-23 [1]
30. #> coda 0.19-4 2020-09-30 [1]
31. #> codetools 0.2-18 2020-11-04 [1]
32. #> colorspace 2.0-0 2020-11-11 [1]
33. #> colourpicker 1.1.0 2020-09-14 [1]
34. #> crayon 1.4.1 2021-02-08 [1]
35. #> crosstalk 1.1.1 2021-01-12 [1]
36. #> curl 4.3 2019-12-02 [1]
37. #> DBI 1.1.1 2021-01-15 [1]
38. #> dbplyr 2.1.0 2021-02-03 [1]
39. #> digest 0.6.27 2020-10-24 [1]
40. #> distributional 0.2.2 2021-02-02 [1]
41. #> dplyr \* 1.0.4 2021-02-02 [1]
42. #> DT 0.17 2021-01-06 [1]
43. #> dygraphs 1.1.1.6 2018-07-11 [1]
44. #> ellipsis 0.3.1 2020-05-15 [1]
45. #> emmeans 1.5.4 2021-02-03 [1]
46. #> estimability 1.3 2018-02-11 [1]
47. #> evaluate 0.14 2019-05-28 [1]
48. #> fansi 0.4.2 2021-01-15 [1]
49. #> farver 2.0.3 2020-01-16 [1]
50. #> fastmap 1.1.0 2021-01-25 [1]
51. #> forcats \* 0.5.1 2021-01-27 [1]
52. #> fs 1.5.0 2020-07-31 [1]
53. #> gamm4 0.2-6 2020-04-03 [1]
54. #> generics 0.1.0 2020-10-31 [1]
55. #> ggdist 2.4.0 2021-01-04 [1]
56. #> ggplot2 \* 3.3.3 2020-12-30 [1]
57. #> ggridges 0.5.3 2021-01-08 [1]
58. #> git2r 0.28.0 2021-01-10 [1]
59. #> glue 1.4.2 2020-08-27 [1]
60. #> gratia 0.5.1.9002 2021-02-16 [1]
61. #> gridExtra 2.3 2017-09-09 [1]
62. #> gtable 0.3.0 2019-03-25 [1]
63. #> gtools 3.8.2 2020-03-31 [1]
64. #> haven 2.3.1 2020-06-01 [1]
65. #> here 1.0.1 2020-12-13 [1]
66. #> highr 0.8 2019-03-20 [1]
67. #> hms 1.0.0 2021-01-13 [1]
68. #> htmltools 0.5.1.1 2021-01-22 [1]
69. #> htmlwidgets 1.5.3 2020-12-10 [1]
70. #> httpuv 1.5.5 2021-01-13 [1]
71. #> httr 1.4.2 2020-07-20 [1]
72. #> igraph 1.2.6 2020-10-06 [1]
73. #> inline 0.3.17 2020-12-01 [1]
74. #> jsonlite 1.7.2 2020-12-09 [1]
75. #> knitr \* 1.31 2021-01-27 [1]
76. #> labeling 0.4.2 2020-10-20 [1]
77. #> later 1.1.0.1 2020-06-05 [1]
78. #> lattice 0.20-41 2020-04-02 [1]
79. #> lifecycle 1.0.0 2021-02-15 [1]
80. #> lme4 1.1-26 2020-12-01 [1]
81. #> loo 2.4.1 2020-12-09 [1]
82. #> lubridate 1.7.9.2 2020-11-13 [1]
83. #> magrittr 2.0.1 2020-11-17 [1]
84. #> markdown 1.1 2019-08-07 [1]
85. #> MASS 7.3-53.1 2021-02-12 [1]
86. #> Matrix 1.2-18 2019-11-27 [1]
87. #> matrixStats 0.58.0 2021-01-29 [1]
88. #> mgcv \* 1.8-33 2020-08-27 [1]
89. #> mime 0.10 2021-02-13 [1]
90. #> miniUI 0.1.1.1 2018-05-18 [1]
91. #> minqa 1.2.4 2014-10-09 [1]
92. #> modelr 0.1.8 2020-05-19 [1]
93. #> multcomp 1.4-16 2021-02-08 [1]
94. #> munsell 0.5.0 2018-06-12 [1]
95. #> mvnfast 0.2.5.1 2020-10-14 [1]
96. #> mvtnorm 1.1-1 2020-06-09 [1]
97. #> nlme \* 3.1-152 2021-02-04 [1]
98. #> nloptr 1.2.2.2 2020-07-02 [1]
99. #> patchwork 1.1.1 2020-12-17 [1]
100. #> pillar 1.5.0 2021-02-22 [1]
101. #> pkgbuild 1.2.0 2020-12-15 [1]
102. #> pkgconfig 2.0.3 2019-09-22 [1]
103. #> plyr 1.8.6 2020-03-03 [1]
104. #> prettyunits 1.1.1 2020-01-24 [1]
105. #> processx 3.4.5 2020-11-30 [1]
106. #> projpred 2.0.2 2020-10-28 [1]
107. #> promises 1.2.0.1 2021-02-11 [1]
108. #> ps 1.5.0 2020-12-05 [1]
109. #> purrr \* 0.3.4 2020-04-17 [1]
110. #> R6 2.5.0 2020-10-28 [1]
111. #> ragg 1.1.0 2021-02-15 [1]
112. #> Rcpp \* 1.0.6 2021-01-15 [1]
113. #> D RcppParallel 5.0.2 2020-06-24 [1]
114. #> readr \* 1.4.0 2020-10-05 [1]
115. #> readxl 1.3.1 2019-03-13 [1]
116. #> reprex 1.0.0 2021-01-27 [1]
117. #> reshape2 1.4.4 2020-04-09 [1]
118. #> rlang 0.4.10 2020-12-30 [1]
119. #> rprojroot 2.0.2 2020-11-15 [1]
120. #> rsconnect 0.8.16 2019-12-13 [1]
121. #> rstan 2.21.2 2020-07-27 [1]
122. #> rstanarm 2.21.1 2020-07-20 [1]
123. #> rstantools 2.1.1 2020-07-06 [1]
124. #> rstudioapi 0.13 2020-11-12 [1]
125. #> rvest 0.3.6 2020-07-25 [1]
126. #> sandwich 3.0-0 2020-10-02 [1]
127. #> scales 1.1.1 2020-05-11 [1]
128. #> sessioninfo 1.1.1 2018-11-05 [1]
129. #> shiny 1.6.0 2021-01-25 [1]
130. #> shinyjs 2.0.0 2020-09-09 [1]
131. #> shinystan 2.5.0 2018-05-01 [1]
132. #> shinythemes 1.2.0 2021-01-25 [1]
133. #> StanHeaders 2.21.0-7 2020-12-17 [1]
134. #> statmod 1.4.35 2020-10-19 [1]
135. #> stringi 1.5.3 2020-09-09 [1]
136. #> stringr \* 1.4.0 2019-02-10 [1]
137. #> survival 3.2-7 2020-09-28 [1]
138. #> svUnit 1.0.3 2020-04-20 [1]
139. #> systemfonts 1.0.1 2021-02-09 [1]
140. #> textshaping 0.3.0 2021-02-10 [1]
141. #> TH.data 1.0-10 2019-01-21 [1]
142. #> threejs 0.3.3 2020-01-21 [1]
143. #> tibble \* 3.0.6 2021-01-29 [1]
144. #> tidybayes 2.3.1 2020-11-02 [1]
145. #> tidyr \* 1.1.2 2020-08-27 [1]
146. #> tidyselect 1.1.0 2020-05-11 [1]
147. #> tidyverse \* 1.3.0 2019-11-21 [1]
148. #> tjmisc 0.0.0.9000 2021-02-25 [1]
149. #> utf8 1.1.4 2018-05-24 [1]
150. #> V8 3.4.0 2020-11-04 [1]
151. #> vctrs 0.3.6 2020-12-17 [1]
152. #> withr 2.4.1 2021-01-26 [1]
153. #> xfun 0.21 2021-02-10 [1]
154. #> xml2 1.3.2 2020-04-23 [1]
155. #> xtable 1.8-4 2019-04-21 [1]
156. #> xts 0.12.1 2020-09-09 [1]
157. #> zoo 1.8-8 2020-05-02 [1]
158. #> source
159. #> CRAN (R 4.0.0)
160. #> CRAN (R 4.0.2)
161. #> CRAN (R 4.0.2)
162. #> CRAN (R 4.0.3)
163. #> CRAN (R 4.0.0)
164. #> local
165. #> CRAN (R 4.0.3)
166. #> CRAN (R 4.0.2)
167. #> CRAN (R 4.0.2)
168. #> CRAN (R 4.0.0)
169. #> CRAN (R 4.0.4)
170. #> CRAN (R 4.0.3)
171. #> CRAN (R 4.0.2)
172. #> CRAN (R 4.0.4)
173. #> CRAN (R 4.0.2)
174. #> CRAN (R 4.0.2)
175. #> CRAN (R 4.0.3)
176. #> CRAN (R 4.0.2)
177. #> CRAN (R 4.0.3)
178. #> CRAN (R 4.0.3)
179. #> CRAN (R 4.0.2)
180. #> CRAN (R 4.0.3)
181. #> CRAN (R 4.0.3)
182. #> CRAN (R 4.0.3)
183. #> CRAN (R 4.0.3)
184. #> CRAN (R 4.0.3)
185. #> CRAN (R 4.0.3)
186. #> CRAN (R 4.0.2)
187. #> CRAN (R 4.0.2)
188. #> CRAN (R 4.0.3)
189. #> CRAN (R 4.0.0)
190. #> CRAN (R 4.0.2)
191. #> CRAN (R 4.0.3)
192. #> CRAN (R 4.0.2)
193. #> CRAN (R 4.0.3)
194. #> CRAN (R 4.0.3)
195. #> CRAN (R 4.0.2)
196. #> CRAN (R 4.0.2)
197. #> CRAN (R 4.0.3)
198. #> CRAN (R 4.0.3)
199. #> CRAN (R 4.0.3)
200. #> CRAN (R 4.0.3)
201. #> CRAN (R 4.0.3)
202. #> CRAN (R 4.0.2)
203. #> gavinsimpson/gratia@32daf2f)
204. #> CRAN (R 4.0.2)
205. #> CRAN (R 4.0.2)
206. #> CRAN (R 4.0.0)
207. #> CRAN (R 4.0.2)
208. #> CRAN (R 4.0.3)
209. #> CRAN (R 4.0.2)
210. #> CRAN (R 4.0.3)
211. #> CRAN (R 4.0.3)
212. #> CRAN (R 4.0.3)
213. #> CRAN (R 4.0.3)
214. #> CRAN (R 4.0.2)
215. #> CRAN (R 4.0.2)
216. #> CRAN (R 4.0.3)
217. #> CRAN (R 4.0.3)
218. #> CRAN (R 4.0.3)
219. #> CRAN (R 4.0.2)
220. #> CRAN (R 4.0.2)
221. #> CRAN (R 4.0.2)
222. #> CRAN (R 4.0.3)
223. #> CRAN (R 4.0.3)
224. #> CRAN (R 4.0.3)
225. #> CRAN (R 4.0.3)
226. #> CRAN (R 4.0.3)
227. #> CRAN (R 4.0.2)
228. #> CRAN (R 4.0.4)
229. #> CRAN (R 4.0.3)
230. #> CRAN (R 4.0.3)
231. #> CRAN (R 4.0.3)
232. #> CRAN (R 4.0.4)
233. #> CRAN (R 4.0.2)
234. #> CRAN (R 4.0.2)
235. #> CRAN (R 4.0.2)
236. #> CRAN (R 4.0.3)
237. #> CRAN (R 4.0.2)
238. #> CRAN (R 4.0.3)
239. #> CRAN (R 4.0.0)
240. #> CRAN (R 4.0.3)
241. #> CRAN (R 4.0.2)
242. #> CRAN (R 4.0.3)
243. #> CRAN (R 4.0.4)
244. #> CRAN (R 4.0.3)
245. #> CRAN (R 4.0.2)
246. #> CRAN (R 4.0.2)
247. #> CRAN (R 4.0.2)
248. #> CRAN (R 4.0.3)
249. #> CRAN (R 4.0.3)
250. #> CRAN (R 4.0.3)
251. #> CRAN (R 4.0.3)
252. #> CRAN (R 4.0.2)
253. #> CRAN (R 4.0.2)
254. #> CRAN (R 4.0.4)
255. #> CRAN (R 4.0.3)
256. #> CRAN (R 4.0.3)
257. #> CRAN (R 4.0.2)
258. #> CRAN (R 4.0.2)
259. #> CRAN (R 4.0.3)
260. #> CRAN (R 4.0.2)
261. #> CRAN (R 4.0.3)
262. #> CRAN (R 4.0.3)
263. #> CRAN (R 4.0.2)
264. #> CRAN (R 4.0.3)
265. #> CRAN (R 4.0.2)
266. #> CRAN (R 4.0.2)
267. #> CRAN (R 4.0.3)
268. #> CRAN (R 4.0.2)
269. #> CRAN (R 4.0.2)
270. #> CRAN (R 4.0.2)
271. #> CRAN (R 4.0.2)
272. #> CRAN (R 4.0.3)
273. #> CRAN (R 4.0.2)
274. #> CRAN (R 4.0.2)
275. #> CRAN (R 4.0.3)
276. #> CRAN (R 4.0.3)
277. #> CRAN (R 4.0.3)
278. #> CRAN (R 4.0.2)
279. #> CRAN (R 4.0.2)
280. #> CRAN (R 4.0.2)
281. #> CRAN (R 4.0.2)
282. #> CRAN (R 4.0.3)
283. #> CRAN (R 4.0.3)
284. #> CRAN (R 4.0.2)
285. #> CRAN (R 4.0.2)
286. #> CRAN (R 4.0.3)
287. #> CRAN (R 4.0.2)
288. #> CRAN (R 4.0.2)
289. #> CRAN (R 4.0.2)
290. #> CRAN (R 4.0.2)
291. #> local
292. #> CRAN (R 4.0.2)
293. #> CRAN (R 4.0.3)
294. #> CRAN (R 4.0.3)
295. #> CRAN (R 4.0.3)
296. #> CRAN (R 4.0.3)
297. #> CRAN (R 4.0.2)
298. #> CRAN (R 4.0.2)
299. #> CRAN (R 4.0.2)
300. #> CRAN (R 4.0.2)
301. #>
302. #> [1] C:/Users/Tristan/Documents/R/win-library/4.0
303. #> [2] C:/Program Files/R/R-4.0.4/library
304. #>
305. #> D -- DLL MD5 mismatch, broken installation.