LA with RcppArmadillo

Rachel Wood

2023-03-28

This portfolio aims to perform smoothing with RcppArmadillo, using local polynomial regression.

We load the relevant data on solar power production in Australia:

```
library(dplyr)
load("solarAU.RData")
head(solarAU)
```

```
##       prod          toy tod
## 8832 0.019 0.000000e+00   0
## 8833 0.032 5.708088e-05   1
## 8834 0.020 1.141618e-04   2
## 8835 0.038 1.712427e-04   3
## 8836 0.036 2.283235e-04   4
## 8837 0.012 2.854044e-04   5
```

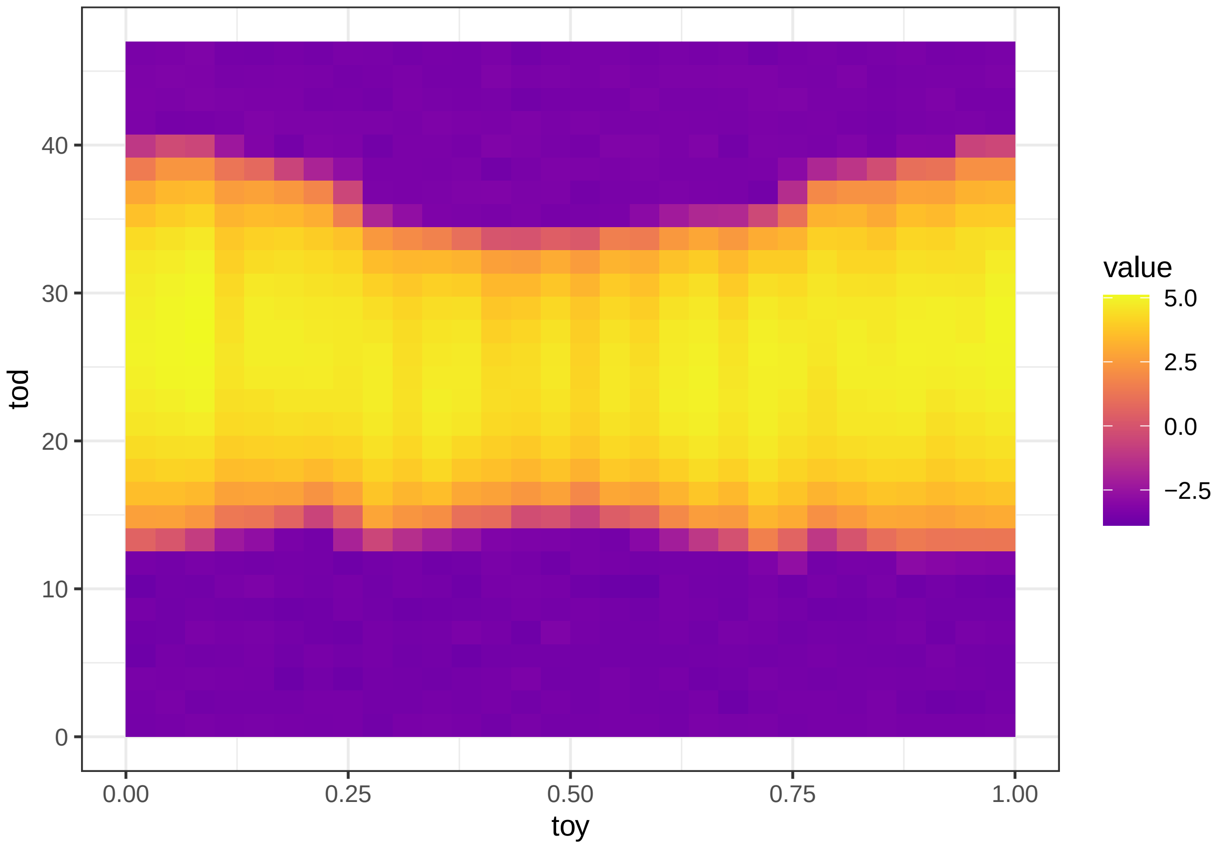
```
solarAU <- as_tibble(solarAU)
```

We work with the log of the production as it is less skewed (adding 0.01 to all entries to avoid numerical errors)

```
solarAU <- solarAU %>%
  mutate(logProduction = log(prod + 0.01))
```

We can now plot the production in a 2d summary:

```
library(ggplot2)
library(viridis)
ggplot(solarAU,
       aes(x = toy, y = tod, z = logProduction)) +
       stat_summary_2d()
```



We can see there is greater solar production in the middle of the day and in the winter, which is to be expected.

Linear Model

We consider the following model:

$$\mathbb{E}(y|x) = \beta_0 + \beta_1 \text{tod} + \beta_2 \text{tod}^2 + \beta_3 \text{toy} + \beta_4 \text{toy}^2$$

We can use R to solve this easily:

```
fit <- lm(logProduction ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
R_beta <- fit$coefficients
```

Using RcppArmadillo

We now want to use RcppArmadillo to fit the same model, taking as input

```
X <- with(solarAU, cbind(1, tod, tod^2, toy, toy^2))
y <- solarAU$logProduction
```

QR decomposition

We first try to perform the computation using a QR decomposition

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export]]
vec arma_QR_lm(mat& X, vec& y){
  mat Q;
  mat R;

  qr(Q, R, X);
  vec Qty = trans(Q) * y;

  vec beta = solve(R, Qty);
  return beta;
}
```

We now check that the two vectors are the same:

```
arma_QR_beta <- arma_QR_lm(X,y)
max(abs(R_beta - arma_QR_beta))
```

```
## [1] 7.371881e-14
```

We can now use the microbenchmark function to compare the running times

```
library(microbenchmark)
microbenchmark(R = beta <- lm(logProduction ~ tod + I(tod^2) + toy + I(toy^2), data =
↪  solarAU),
               "arma QR"= arma_QR_beta <- arma_QR_lm(X,y), times = 10)
```

```
## Unit: milliseconds
##     expr         min         lq        mean     median          uq        max
##        R    2.749813    4.00576    5.650517    4.57964    6.294626   13.24599
##  arma QR 1875.305545 1887.65844 1921.956164 1920.25532 1927.943926 1996.93768
##  neval
##     10
##     10
```

We can see here however that the `arma` function performed much worse than the base `R` one. This might be because we have used `qr` instead of `qr_econ` in our Rcpp code. We can rewrite the Rcpp function

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export]]
vec arma_QR_lm_fast(mat& X, vec& y){
  mat Q;
```

```
  mat R;

  qr_econ(Q, R, X);
  vec Qty = trans(Q) * y;

  vec beta = solve(R, Qty);
  return beta;
}
```

```
library(microbenchmark)
microbenchmark(R = R_beta <- lm(logProduction ~ tod + I(tod^2) + toy + I(toy^2), data =
↪  solarAU)$coefficients,
               "arma QR"= arma_QR_beta <- arma_QR_lm(X,y),
               "arma QR fast" = arma_QR_beta_fast <- arma_QR_lm_fast(X,y),
               times = 10)
```

```
## Unit: microseconds
##          expr         min          lq         mean      median          uq
##             R    2863.699    3027.190    3804.5687    3390.668    4512.282
##       arma QR 1914615.910 1931560.619 1948499.5215 1937837.350 1981047.925
##  arma QR fast     317.314     331.937     512.6051     441.380     468.412
##          max neval
##     6015.858    10
##  1995284.127    10
##     1363.821    10
```

We can now see, the RcppArmadillo version outperforms the R version and produces the same results:

```
arma_QR_beta_fast <- arma_QR_lm_fast(X,y)
max(abs(arma_QR_beta_fast - R_beta))
```

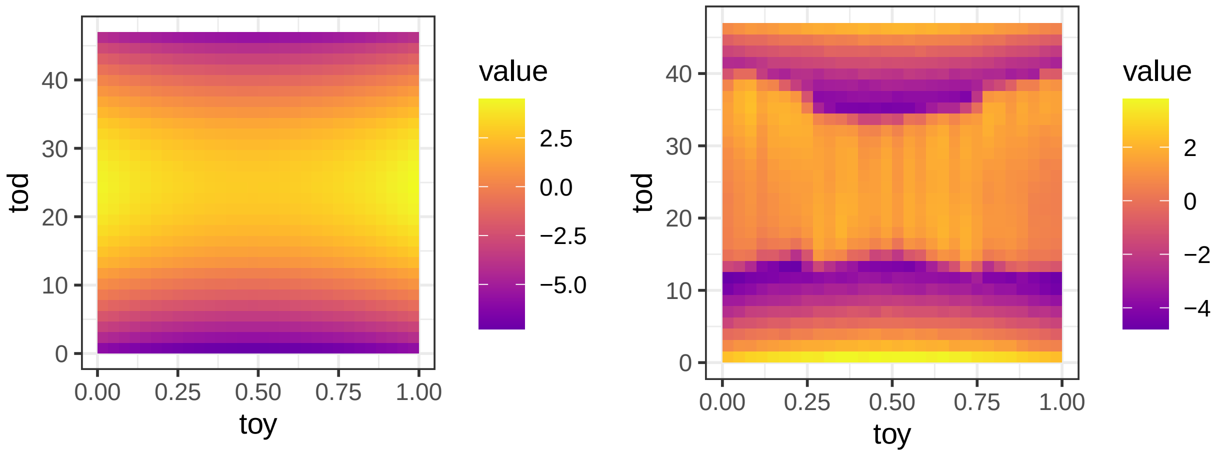
```
## [1] 8.792966e-14
```

Model Checking

We can now plot fitted values and the residuals to check if the model seems valid:

```
library(gridExtra)

solarAU <- solarAU %>%
  mutate(fitted = fit$fitted.values,
         residuals = fit$residuals)

fitted_plot <- ggplot(data = solarAU,
                      aes(x = toy, y = tod, z = fitted)) +
  stat_summary_2d()+
  theme(aspect.ratio = 1)

res_plot <- ggplot(data = solarAU,
                      aes(x = toy, y = tod, z = residuals)) +
```

```
  stat_summary_2d()+
  theme(aspect.ratio = 1)

grid.arrange(fitted_plot, res_plot, ncol = 2)
```



We can see from these plots there is a non linear pattern, particularly in the residuals, so we adapt the model.

Kernel Local Least Regression

In this model we take the coefficients to be dependent on the covariates $\hat{\beta} = \hat{\beta}(x)$. For a fixed x_0, we take

$$\hat{\beta}(x_0) = \arg\min_{\beta} \sum_{i=1}^{n} \kappa_H(x_0 - x_i)(y_i - \tilde{x}_i^T \beta)^2$$

for a kernel κ_H with bandwidth H. We create this function in R:

```
library(mvtnorm)

lm_local <- function(y, x0, X0, x, X, H){
  w <- dmvnorm(x, x0, H)
  fit <- lm(y ~ -1 + X, weights = w)
  return( t(X0) %*% coef(fit) )
}
```

To reduce the computational cost, we will only fit the model on 2000 data points:

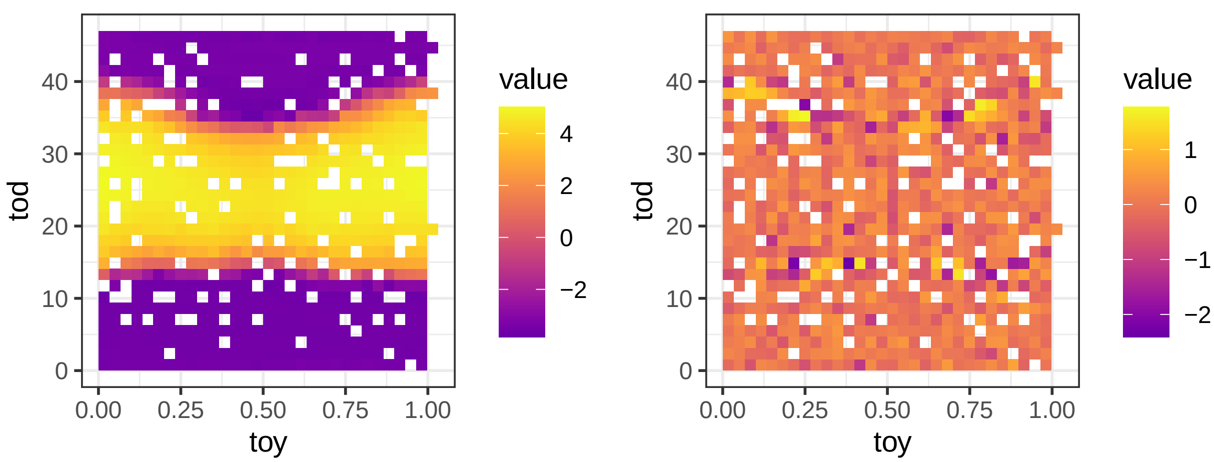
```
sample_ind <- sample(1:nrow(solarAU), 2000)

x <- solarAU %>%
  select(tod, toy) %>%
  as.matrix()
x_sample <- x[sample_ind,]
X_sample <- X[sample_ind,]

y_sample <- solarAU$logProduction
```

```
solar_sample <- solarAU[sample_ind,]

X_sample <- X[sample_ind,]
```

We can now fit $\hat{\beta}(x_0)$ for each sample data point:

```
pred_local <- sapply(1:2000, function(ii){
  lm_local(y = y, x0 = x_sample[ii, ], X0 = X_sample[ii, ], x = x, X = X, H = diag(c(1,
  ↪  0.1)^2))
})
```

```
solar_sample <- solar_sample %>%
  mutate("local fitted" = pred_local,
         residuals = logProduction - `local fitted`)

local_fitted_plot <- ggplot(solar_sample,
       aes(x = toy, y = tod, z = `local fitted`)) +
       stat_summary_2d() +
       theme(aspect.ratio = 1)

local_res_plot <- ggplot(solar_sample,
       aes(x = toy, y = tod, z = residuals)) +
       stat_summary_2d() +
       theme(aspect.ratio = 1)

grid.arrange(local_fitted_plot, local_res_plot, ncol = 2)
```



Using RcppArmadillo

We now want to implement the above with RcppArmadillo in the hopes of speeding this up, using our previously defined `arma_QR_lm_fast()` function and a multivariate `dmvnInt()` function analagous to `dmvnorm()` R function

```
// [[Rcpp::export]]
vec arma_lm_local(vec& y, mat& x0,  mat& X0, mat& x, mat& X, mat& H){

  mat L = chol(H, "lower");
  int row = x0.n_rows;

  vec fitted(row), w;
  double fit;


  for (int i=0; i<row; i++) {

    w = sqrt(dmvnInt(x, x0.row(i), L));
    fit = as_scalar(X0.row(i) * arma_QR_lm_fast(X.each_col() % w, y % w));
    fitted(i) = fit;
  }

  return fitted;

}
```

We can now run this for our sample as before and check our results are consistent:

```
arma_pred_local <- arma_lm_local(y = y, x0 = x_sample, X0 = X_sample, x = x, X = X, H =
↪  diag(c(1, 0.1)^2))
max(abs(arma_pred_local - pred_local))
```

```
## [1] 4.754863e-12
```

We can now compare the running times of our functions:

```
microbenchmark(R = pred_local <- sapply(1:1000, function(ii){
                     lm_local(y = y, x0 = x_sample[ii, ], X0 = X_sample[ii, ], x = x, X =
                     ↪  X, H = diag(c(1, 0.1)^2))
                    }),
               "arma"= arma_lm_local(y = y, x0 = x_sample, X0 = X_sample, x = x, X = X, H
               ↪  = diag(c(1, 0.1)^2)))
```

```
## Unit: seconds
##  expr      min       lq     mean   median       uq      max neval
##     R 5.989946 7.406814 8.371587 8.729386 9.011346 9.973653   100
##  arma 1.146207 1.188016 2.253702 2.395459 3.217466 3.404163   100
```