Portfolio Report 3: Advanced Rcpp

Jake Spiteri

2020

Exercises on local polynomial regression

Smoothing by local polynomial regression

Question 1

We will use `RcppArmadillo` to fit a linear regression model. We want to estimate the coeffiients $\hat{\beta}$ such that $\hat{\beta} = \underset{\beta}{\text{argmin}} \|\mathbf{y} - \mathbf{X}\beta\|^2$.

A well known solution for the above minimization problem is $\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$. Computing the matrix inverse is numerically unstable, and so we will use the QR decomposition to rewrite $\mathbf{X}$.

We rewrite $\mathbf{X} = \mathbf{QR} = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1$, where $\mathbf{Q}$ is an orthogonal matrix, $\mathbf{R}$ is an upper triangular matrix $n \times p$ matrix, and $\mathbf{Q}_1$ contains the first p columns of $\mathbf{Q}$. This then allows us to rewrite $\hat{\boldsymbol{\beta}}$ as

$$\hat{\boldsymbol{\beta}} = \mathbf{R}_1^{-1}\mathbf{Q}_1^{\top}\mathbf{y}.$$

The `RcppArmadillo` package allows us to easily compute the thin QR decomposition as described above with the use of `qr_econ(Q, R, X)`.

```
sourceCpp(code = '
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export(name = "lm_arma")]]
vec lm_I(mat& X, vec& y){
  mat Q, R;
  qr_econ(Q, R, X);

  return solve(trimatu(R), Q.t() * y, solve_opts::fast);
}
')
```

To speed up the final solve returned by the RcppArmadillo function, we have told the `solve` function that R is an upper-triangular matrix by specifying `trimatu(R)`, and we have chosen to use a fast decomposition by providing the setting `solve_opts::fast`. We will test that the above code provides the same solution as R's built-in function `lm`.

```
solarAU <- read.csv("data/solarAU.csv")
solarAU$logprod <- log(solarAU$prod+0.01)
```

```r
X <- with(solarAU, cbind(1, tod, tod^2, toy, toy^2))
y <- solarAU$logprod
lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)$coefficients
```

```
## (Intercept)         tod    I(tod^2)         toy    I(toy^2)
## -6.26275685  0.86440391 -0.01757599 -5.91806924  6.14298863
```

```r
t(lm_arma(X, y))
```

```
##           [,1]      [,2]        [,3]      [,4]     [,5]
## [1,] -6.262757 0.8644039 -0.01757599 -5.918069 6.142989
```

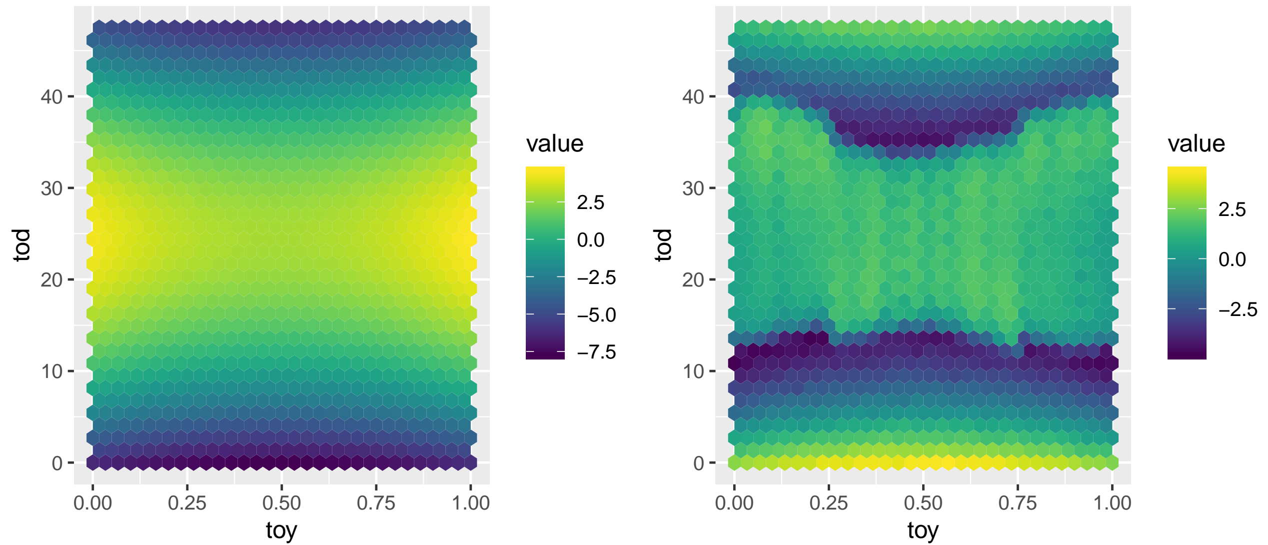
We see that we obtain the same coefficients using both methods. We will now compare their speeds. Note that we will compute the model matrix within both functions — the `lm` function will always compute the model matrix and thus in order to compare speeds we must compute the model matrix each time we run the `lm_arma` function.

```r
lm_R <- function() lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
lm_C <- function() lm_arma(with(solarAU, cbind(1, tod, tod^2, toy, toy^2)), y)
microbenchmark(R=lm_R, C=lm_C, times = 500)
```

```
## Unit: nanoseconds
##  expr min lq   mean median uq  max neval
##     R  33 36 41.438     36 37 2509   500
##     C  33 36 37.790     37 37  403   500
```

We see that the C version outperforms the built-in R version of `lm` in terms of speed. Let's plot the predictions output from our regression model, and the residuals.

```r
fit <- lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
solarAU$fitPoly <- fit$fitted.values
pl1 <- ggplot(solarAU,
              aes(x = toy, y = tod, z = fitPoly)) +
       stat_summary_hex() +
       scale_fill_gradientn(colours = viridis(50))

pl2 <- ggplot(solarAU,
              aes(x = toy, y = tod, z = logprod - fitPoly)) +
       stat_summary_hex() +
       scale_fill_gradientn(colours = viridis(50))
grid.arrange(pl1, pl2, ncol = 2)
```



There is a clear non-linear pattern in the residuals, and so we instead consider a local linear regression model. To do this we implement a locally adaptive linear model. For a fixed value $\mathbf{x}_0$ we define the local regression with coefficients obtained by minimizing the weighted loss function in which the loss of making a wrong decision is higher for predictions in the neighbourhood of $\mathbf{x}_0$. We have

$$\hat{\beta}\left(\mathbf{x}_0\right) = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} \kappa_{\mathbf{H}}\left(\mathbf{x}_0 - \mathbf{x}_i\right)\left(y_i - \tilde{\mathbf{x}}_i^T \beta\right)^2,$$

where κ_{H} is a density kernel with positive definite bandwidth matrix $\mathbf{H}$.

Below is an implementation using `Rcpp` and `RcppArmadillo`, with the Gaussian density kernel.

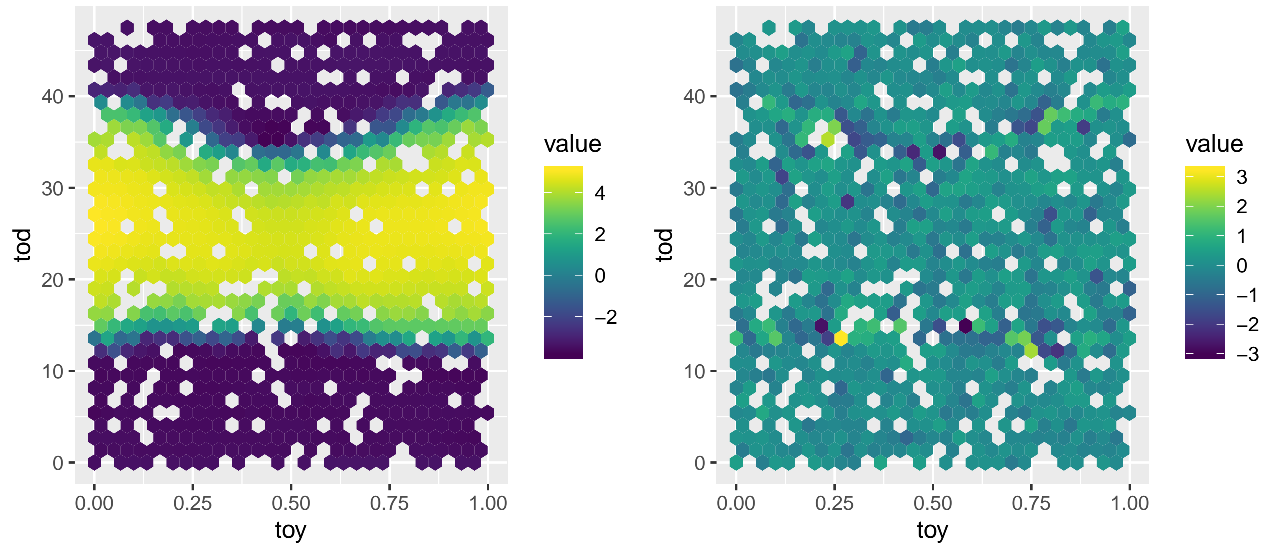
```
sourceCpp(code = '
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;

vec dmvnInt(mat & X, const rowvec & mu, mat & L)
{
  unsigned int d = X.n_cols;
  unsigned int m = X.n_rows;

  vec D = L.diag();
  vec out(m);
  vec z(d);

  double acc;
  unsigned int icol, irow, ii;
  for(icol = 0; icol < m; icol++)
  {
    for(irow = 0; irow < d; irow++)
    {
     acc = 0.0;
     for(ii = 0; ii < irow; ii++) acc += z.at(ii) * L.at(irow, ii);
     z.at(irow) = ( X.at(icol, irow) - mu.at(irow) - acc ) / D.at(irow);
    }
    out.at(icol) = sum(square(z));
  }
```

```
  out = exp( - 0.5 * out - ( (d / 2.0) * log(2.0 * M_PI) + sum(log(D)) ) );

  return out;
}

vec lm(mat X, vec y){
  vec beta;
  mat Q, R;
  qr_econ(Q, R, X);

  return solve(R, Q.t() * y);
}

// [[Rcpp::export(name = "local_lm")]]
vec local_lm(vec& y, mat& x0, mat& X0, mat& x, mat& X, mat& H){
  int nsub = x0.n_rows;
  vec out(nsub), weights;
  double fit;
  mat L = chol(H, "lower");

  for(int i=0; i<nsub; i++){
    weights = sqrt(dmvnInt(x, x0.row(i), L));
    fit = as_scalar(X0.row(i) * lm(X.each_col() % weights, y % weights));
    out(i) = fit;
  }
  return out;
}

')
```

Note that the above method requires fitting a local regression model for each $\mathbf{x}_0$ — in our case we will need to produce over 17000 regressions. To ensure that the method works we will work with a subsample of the dataset. We will use 2000 observations for $\mathbf{x}_0$.

```
n <- nrow(X)
nsub <- 2e3
sub <- sample(1:n, nsub, replace = FALSE)

y <- solarAU$logprod
solarAU_sub <- solarAU[sub, ]
x <- as.matrix(solarAU[c("tod", "toy")])
x0 <- x[sub, ]
X0 <- X[sub, ]

fit <- local_lm(y, x0, X0, x, X, diag(c(1, 0.1)^2))

solarAU_sub$fitLocal <- fit
pl1 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = fitLocal)) +
       stat_summary_hex() +
       scale_fill_gradientn(colours = viridis(50))

pl2 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = logprod - fitLocal)) +
```

```r
      stat_summary_hex() +
      scale_fill_gradientn(colours = viridis(50))

grid.arrange(pl1, pl2, ncol = 2)
```



Above we see that the method has worked. The residuals do not contain an obvious pattern. Below we ensure that the Rcpp function works by comparing it with the R version.

```r
lmLocal <- function(y, x0, X0, x, X, H){
  w <- dmvnorm(x, x0, H)
  fit <- lm(y ~ -1 + X, weights = w)
  return( t(X0) %*% coef(fit) )
}
predLocal <- sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = diag(c(1, 0.1)^2))
})
solarAU_sub$fitLocal <- predLocal

all.equal(predLocal, as.vector(fit))
```

```
## [1] TRUE
```

```r
max(abs(predLocal - as.vector(fit)))
```

```
## [1] 1.150191e-12
```

We also compare the speed of the two proposed versions.

```r
local_R <- function() {
  predLocal <- sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = diag(c(1, 0.1)^2))
})
}
local_C <- function() local_lm(y, x0, X0, x, X, diag(c(1, 0.1)^2))

microbenchmark(R=local_R(), C=local_C(), times=5)
```

```
## Unit: seconds
##   expr       min        lq      mean    median        uq       max neval
```

```
##      R 15.275059 15.366972 15.435687 15.416442 15.542873 15.577088     5
##      C  1.781468  1.782172  1.793818  1.785911  1.800508  1.819029     5
```