Stat 610 Lab 3: Testing and top-down design

October 2, 2020

Linear regression is often not flexible enough, and several alternatives have been proposed. One such example is *locally weighted regression*. Suppose we have a single predictor, x, and that our model is $y = f(x) + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$. We don't want to assume that f is linear, but we can estimate it approximately at z as $f(z) = a_0 + b_0 x_0$, where a_0 and b_0 are chosen to minimize

$$\sum_{i=1}^{n} W\left(|x_i - z|/\omega\right)\left(y_i - a_0 - b_0 x_i\right)^2$$

$W(r)$ is a positive, even weight function defined as

$$W(r) = \begin{cases} (1 - |r|^3)^3 & |r| < 1 \\ 0 & \text{o.w.} \end{cases}$$

ω is the window size or bandwidth, and it controls how many neighboring values contribute to the regression.

The minimization problem is a weighted least squares problem (bonus: show this), and the solution is

$$\hat{f}(z) = \begin{pmatrix} 1 & z \end{pmatrix} (X^T W_z X)^{-1} X^T W_z y$$

where $X \in \mathbb{R}^{n \times 2}$ is a matrix whose first column contains all 1's and whose second column contains the values of the predictor (so $X_{i2} = x_i$) and W_z is the diagonal matrix whose ith element is $W(|x_i - z|/\omega)$.

We would like to make a function that fits a local regression. That is, if we are given an n-vector of predictors $x = (x_1, \ldots, x_n)$, an n-vector of response variables $y = (y_1, \ldots, y_n)$, and an m-vector $z = (z_1, \ldots, z_m)$ of points for which we want fits, our function should return an m-vector containing $(\hat{f}(z_1), \ldots, \hat{f}(z_m))$.

Top-level function

If we are designing this function from the top down, our first task is to write a function that takes as input x, y, z, and ω and returns a vector $(\hat{f}(z_1), \ldots, \hat{f}(z_m))$.

That is, the function definition should look like:

```
llr = function(x, y, z, omega) {

}
```

Inside, we will need to compute $\hat{f}(z_i)$ for each z_i. To do so, we can promise to make a function that computes the fits at a point z_i and apply it to each element of z. If we had such a function, our local regression function would look like this:

```
llr = function(x, y, z, omega) {
    fits = sapply(z, compute_f_hat, x, y, omega)
    return(fits)
}
```

Easy, right?

We should also be writing tests for the functions we create. At this point, one of the few things we know about the output of `llr` is that it should be the same length as `z`. In the test file, there is a test for that situation.

Second-level function

In our definition of `llr`, we have used a function called `compute_f_hat`, that we need to define now. Its arguments are `z`, `x`, `y`, and `omega`, in that order (why? think about how `sapply` works), and so the function definition will look like:

```
compute_f_hat = function(z, x, y, omega) {
    Wz = make_weight_matrix(z, x, omega)
    X = make_predictor_matrix(x)
    f_hat = c(1, z) %*% solve(t(X) %*% Wz %*% X) %*% t(X) %*%  Wz %*% y
    return(f_hat)
}
```

The third line above is simply the formula given above for the fit from the weighted regression (`solve` means matrix inverse). We're still not quite done: we need to create the functions `make_weight_matrix` and `make_predictor_matrix`.

Third-level functions

Make a file called `llr_functions.R`, and copy over our definitions of `llr` and `compute_f_hat`.

Add tests to the `test_lab_4.R` file for the `make_weight_matrix` and `make_predictor_matrix` functions.

Add your own implementations of `make_weight_matrix` and `make_predictor_matrix`, and then run `testthat::test_dir(".")` to see if they work correctly.

Try it out

See if it works on some data.

One possibility is the `french_fries` data: you could try

```
data(french_fries)
french_fries = french_fries[complete.cases(french_fries),]
z = seq(0, 15, length.out = 100)
fits = llr(z = z, x = french_fries$potato, y = french_fries$buttery, omega = 2)
plot(z, fits)
```

Try different values of omega. What happens to the fits?