QR Sieve Code

John Wilson*

Version: 0.1.1†

Contents

*The title is rough and once the documentation is complete and approved the paper's authors should appear here

†If you have any questions, please contact: wilsonjr@mit.edu

1 Introduction

The QR Sieve toolkit is a set of code designed to accompany the paper *Errors in the Dependent Variable of Quantile Regression Models*, by Jerry Hausman, Haoyang Liu, Ye Luo, and Christopher Palmer. It provides a simple Matlab function to implement the method described in the paper as well as providing some examples of how to use the code. The code is still under development, however the current version is functional and this document will highlight the changes intended for the future of this toolkit.

2 Main QR Sieve function

The main workhorse of this toolkit is the function QR_sieve contained in Toolkit/QR_sieve.m and the helper functions called by this main function. This powerful function has the option to run bootstrapped sampling for inference, and can also be used to plot the results of the regresssions.

2.1 Inputs

The required inputs to this function are as follows:

1. X: An $n \times k$ double of independent variables, where n is the number of observations and k is the number of covariates.

2. y: An $n \times 1$ double of the dependent variable.

3. bootstrap: A positive integer indicating the number of bootstrap runs to perform. If set to 1, the function will perform no bootstrap inference. Rather, it will simply implement the sieve QR method on the full sample. If set to an integer greater than 1, the function will repeat the following procedure bootstrap times:

    - Redraw n observations with replacement from the data X and y. Call this new set of data X_i and y_i.
    - Recursively call the QR_sieve function, this time using as data X_i and y_i, and setting bootstrap to 1.
    - Store the results from this sieve QR regression on the sampled data, and repeat until bootstrap iterations have been achieved.

In addition to these required inputs, the function allows for several extra, optional inputs which can be used to customize the performance of the toolkit. These arguments are explained here, and their default values are shown in Table 1.

4. ntau: An integer dictating the number of knots in τ to use for the sieve QR estimation.

5. nmixtures: An integer dictating the number of components in the mixed normal distribution from which the measurement error in y is drawn.

6. n_WLS_iter: An integer dictating the number of WLS iterations to perform to get a starting point for the piecewise-constant MLE.

7. lower: A list of four doubles which define the lower boundaries on each of the variables $\begin{bmatrix}\beta & \lambda & \mu & \sigma\end{bmatrix}$ for the piecewise-constant MLE.

8. upper: A list of four doubles which define the upper boundaries on each of the variables $\begin{bmatrix}\beta & \lambda & \mu & \sigma\end{bmatrix}$ for the piecewise-constant MLE. More specifically, suppose lower $= \begin{bmatrix}\beta_l & \lambda_l & \mu_l & \sigma_l\end{bmatrix}$ and upper $= \begin{bmatrix}\beta_u & \lambda_u & \mu_u & \sigma_u\end{bmatrix}$. Then for each of the β_i coefficients estimated by the piecewise-constant MLE, we have $\beta_l \leq \beta_i \leq \beta_u$. Similarly, for each of the distributional parameters $\{(\lambda_i, \mu_i, \sigma_i)\}_{i=1}^{\text{nmixtures}}$, we have $\lambda_l \leq \lambda_i \leq \lambda_u$, $\mu_l \leq \mu_i \leq \mu_u$, and $\sigma_l \leq \sigma_i \leq \sigma_u$

9. optimizer: Parameters dictating the algorithm used in the piecewise-linear MLE. See section 2.1.1 below for more instruction about how to configure this parameter.

10. make_plot: Boolean dictating whether to plot the results or not. If bootstrap is set to 1, a single line will be plotted for each beta coefficient and for the density. If set to a number greater than 1, 3 lines will be plotted for each beta and the density: one for the results using the full dataset without sampling, and the two other lines depicting the 95% confidence interval resulting from the bootstrapped results.

2.1.1 Setting the Optimizer

We have found that attaining a global optimum satisfying the constraints of the problem is somewhat difficult. The high dimensionality of the problem complicates this optimization, and the optimizer can get stuck in a local optimum. As such, we allow the user to pass in parameters governing these optimizers. Note: in some sense this is only useful for writing the package and testing if certain algorithms will allow us to find the global optimum without needing artificial bounds placed on the beta parameters. Once we can find the right optimizing algorithm, I suspect we may want to remove this option. Currently the following optimizers are supported:

- Stochastic Gradient Descent: This is an optimizer we wrote to implement the standard stochastic gradient descent algorithm. The function accepts some parameters that can be used to customize its performance:
  - n_batches: The number of batches to use per epoch. In a given epoch, the algorithm will randomly divide the observations into n_batches different subsamples. It will then sequentially perform one iteration of gradient descent on each of these subsamples.
  - n_epochs: The number of epochs to perform. For each epoch, it will perform n_batches iterations of gradient descent on randomly sampled data, as described above. Every 2 epochs, the algorithm will compare the value of the loss function for the full dataset to the previous best value and discard the last 2 epochs if the random draws from the last 2 epochs made the function worse.

- learning_rate: A float governing the speed at which gradient descent proceeds. See the description for the decay parameter for more detail.
- decay: A float governing the decay of the learning rate. More specifically, suppose $\Delta f(X_i)$ is the gradient of the function restricted to the data in subsample i. Call the learning rate η, and the decay α. Then for each batch in each epoch the optimal x is updated following

$$x^*_{i+1} = x^*_i - \alpha^i \eta \Delta f(X_i) \tag{1}$$

- verbose: A boolean dictating whether the SGD algorithm's progress is printed every 50 epochs or not.

To use the SGD optimizing algorithm, you could set

```
n_batches     = 30;
n_epochs      = 1000;
learning_rate = .00001;
decay         = .999;
verbose       = true;
optimizer     = {'SGD', n_batches, n_epochs, ...
                 learning_rate, decay, verbose};
```

- Genetic Algorithm: This option will allow the user to employ MATLAB's built-in genetic algorithm function. To employ this function, simply define a Matlab optimoptions object with the settings you want. See the documentation here for more information. For example, you could use:

```
opts = optimoptions('ga', 'MaxGenerations', 500, ...
                    'PopulationSize', 500);
optimizer_settings = {'GA', opts};
```

WARNING: If any of your variables is fully unbounded (ie the bound is set to **Inf**) then the genetic algorithm will take several days.

- Simulated Annealing: This option will allow the user to employ MATLAB's built-in simulated annealing function. To employ this function, simply define a Matlab optimoptions object with the settings you want. See the documentation here for more information. For example, you could use:

```
opts = optimoptions(@simulannealbnd, 'AnnealingFcn', ...
                    'annealingboltz')
optimizer_settings = {'SA', opts};
```

- Matlab's fminsearch function: This option will allow the user to employ MATLAB's built-in fminsearch function. To employ this function, simply define a Matlab optimset object with the settings you want. See the documentation here for more information. For example, you could use:

```
opts = optimset('Display', 'notify')
optimizer_settings = {'FM', opts};
```

2.1.2 Default Values

The arguments (4)-(10) in the above list of inputs are optional and can be omitted. For example, if you want to run the function using all the default options you could call

```
QR_sieve(X, y, 100);
```

which will run the function on the data X and y, running 100 bootstrap runs. If you wished to change only parameter (6) above, for example, you could run

```
QR_sieve(X, y, 100, [], [], 20);
```

which will assume default values for ntau and nmixtures and set the value for n_WLS_iter to 20.

Default values for all the optional parameters are as follows (note $\bar{y}$ is the empirical mean of y, and σ_y is the empirical standard deviation of y)

Parameter name	Default value
ntau	15
nmixtures	3
n_WLS_iter	40
lower	$[-\infty, .0001, -\bar{y} - 3\sigma_y, .01\sigma_y]$
upper	$[\infty, 1, \bar{y} + 3\sigma_y, 10\sigma_y]$
optimizer	{'SGD', 30, 1000, .00001, .999, true}
make_plot	false

Table 1: Default parameter values

2.2 Outputs

Once the function has finished running, it will return four objects:

1. betas: A (number of covariates $\times$ ntau) array of the estimated coefficients for each value of tau

2. fit_hat : An array of all the estimated parameters. The first (number of covariates $\cdot$ ntau) are the same numbers as in the betas array, though flattened. The remaining numbers are the distributional parameters:

   - nmixtures-1 component weights
   - nmixtures-1 component means
   - nmixtures component standard deviations

3. betas_bootstrap: If the bootstrap parameter is set to 1, this will be the same as the betas array. If set to an integer greater than 1, it will be an (ntau $\times$ number of covariates $\times$ bootstrap) array of the coefficients for each of the bootstrap runs, for each tau and covariate.

4. fit_hat_bootstrap : If the bootstrap parameter is set to 1, this will be the same as the fit_hat array. If set to an integer greater than 1, it will be a (bootstrap × total number of parameters) array of the parameters for each of the bootstrap runs, including the coefficients and the distributional parameters.

3 Plotting Function

This function contained in Toolkit/plot_bootstrap.m takes as inputs the results of the QR Sieve function and generates plots. This is useful if the code was run on a server or make_plot was set to false. The plots that it makes are the same as if make_plot was set to true, but this function allows you to create the plots again without running the function.

3.1 Inputs

This function takes as inputs the exact output of the main sieve QR function:

1. betas
2. fit_hat
3. betas_bootstrap
4. fit_hat_bootstrap

3.2 Outputs

This function produces the same outputs as the main sieve QR function provided make_plot was set to true.

4 Example Code

In order to facilitate easy use of this toolkit, two example use cases are provided which correspond to the two examples from the paper.

4.1 Simulated Example

The set of code contained in Examples/Paper_simulation/paper_simulation_main.m corresponds to the simulated example from section 4 of the paper. Note that the example in the code is based on a much smaller sample size, since the results from the paper came from hundreds of runs of the simulation across multiple cloud computing servers which were then aggregated into one figure. As such, the results may not exactly match up, but the general shape of the betas plotted against tau will appear similar.

4.2 Educational Data

The file Examples/Angrist_et_al/Angrist_main.m contains code to replicate the educational data results from section 5 in the paper. Note that it will only run the code for one of the four years: 1980, 1990, 2000, or 2010. The user can simply comment the lines that correspond to years the user doesn't want to see.