The Bootstrap

SEP
2018

Our goal is to do inference on a parameter of interest θ, like the integrated volatility. That is, we want to build a confidence interval for θ, by using an estimator $\widehat{\theta}$, like the realized volatility. There are three ways to do it: finite sample theory, asymptotic distribution theory and bootstrap theory.

In finite sample theory we assume that we know the true distribution of the data, and then we can find the exact finite sample distribution of $\widehat{\theta}$, say:

$$\widehat{\theta} \sim \mathrm{N}\left(\theta, \frac{\sigma^2}{n}\right)$$

The problem is that this is not always possible to do, the theory can get very complicated quickly, and it usually requires very strong assumptions.

In asymptotic distribution theory we use asymptotic arguments ($n \to \infty$) to get an **approximation** to the exact finite sample distribution of $\widehat{\theta}$. Specifically, from the central limit theorem (CLT), we get:

$$\sqrt{n}(\widehat{\theta} - \theta) \xrightarrow{d} \mathrm{N}(0, \sigma^2)$$

And if n is big enough, we have that $\widehat{\theta}$ is approximately distributed as $\mathrm{N}(\theta, \frac{\sigma^2}{n})$.

In bootstrap theory we also want to obtain the finite sample distribution of $\widehat{\theta}$. Imagine we could somehow obtain new samples of the data. Then, for each new sample, we could compute a new $\widehat{\theta}$, and after repeating this process many times, we would have an idea of the exact finite sample distribution of $\widehat{\theta}$.

Question: What is the problem with this procedure?

We cannot get new samples from the data. We will always only have one sample of the data. In other words, we cannot go to the stock exchange, and resample new prices for a day in the past. We have what we have, a single sample from the price process.

The insight in bootstrap is that, even though we only have one sample from the data, we can use it to generate new samples. That is, from this one sample, we can apply the procedure suggested above. We can do so by resampling the data we have with replacement, effectively generating a new data sample.

If the data is $(X_1, X_2, \cdots, X_n)$, we can resample with replacement from it a new data set of size n. Using this new data set, we can compute $\widehat{\theta}^*$. Then we resample with replacement again, and compute a new $\widehat{\theta}^*$, and so on. After computing a large number of $\widehat{\theta}^*$'s, we get an approximation of the exact finite sample distribution of $\widehat{\theta}$, and can compute the confidence intervals for θ.

Implementing the Bootstrap in Matlab

Now, we will go over how to build the basis of the bootstrap in Matlab. Bootstrapping has two main steps: resampling the data, and computing $\widehat{\theta}^*$. The second part, computing $\widehat{\theta}^*$, is what changes from one bootstrap to another, say from bootstrapping the realized volatility to bootstrapping the truncated volatility. However, the first part (resampling the data) does not change from one problem to the other. We always need to resample the data with replacement. The only thing that changes is the data, but the resampling procedure is always the same. The idea is to break the bootstrap in two steps: the first is resampling the data, and the second is computing $\widehat{\theta}^*$.

At the end of these lecture notes, we will have a function that will do the first step, resample the data with replacement. This greatly simplifies the bootstrapping procedure, and all that will be left to do is compute $\widehat{\theta}^*$.

Implementing the Base Case

What is easier than obtaining the bootstrap samples for 250 days? Obtaining the bootstrap samples for 1 day. What is easier than obtaining J bootstrap samples? Obtaining 1 bootstrap sample. So, we will start with a simple case. Assume we have the continuous returns for a single day, and that we want to obtain one bootstramp sample by resampling with replacement from the continuous returns.

Let the continuous return be stored in vector form:

$$r^c = \begin{pmatrix} r_1^c \\ r_2^c \\ \vdots \\ r_n^c \end{pmatrix}$$

We want to bootstrap the sample stored in r^c. Let's create a function for that:

```
function bsample = getBSample(...)
```

Question: What do we need to feed this function?

It needs the data that we want to bootstrap, and since we are bootstrapping returns, it also needs how many subintervals to break the returns into. It might also be useful to feed how big the data is.

```
function bsample = getBSample(rc,n,kn)
```

Here rc represents the vector of continuous returns, n how may returns we have in a day, and kn is the size of the subintervals for the returns. Now, we need to break the vector r^c into $M \equiv \left\lfloor \frac{n}{k_n} \right\rfloor$ subintervals:

$$r^c = \begin{pmatrix} r_1^c \\ \vdots \\ r_{k_n}^c \\ r_{k_n+1}^c \\ \vdots \\ r_{2k_n}^c \\ \vdots \\ r_{(n-1)k_n+1}^c \\ \vdots \\ r_{nk_n}^c \end{pmatrix}$$

From the 1st subinterval, we redraw with replacement k_n returns. From the 2nd subinterval, we redraw with replacement k_n returns. And so on. Let's do that in Matlab:

```
function bsample = getBSample(rc,n,kn)
    bsample = zeros(n,1); % to store the bootstrap sample
    M = floor(n/kn); % number of subintervals
    for s = 1:M
        id_sub = ((s-1)*kn+1:s*kn)';
        bsample(id_sub) = % redraw with replacement from rc
    end
end
```

In the code above, we declare `bsample` to store the bootstrap sample, we then compute the number `M` of subintervals. Then, for each subinterval, we redraw with replacement from `rc` and store the new sample. We are missing this last piece.

To redraw with replacement, we need to draw with replacement k_n random numbers from the uniform (discrete) distribution. If we are in the first subinterval, then we need k_n random numbers from $1, 2, \cdots, k_n$. To do so, we can use the matlab function `unidrnd`. This function returns random numbers from 1 to some number K. K is the first parameter of `unidrnd`. The next two parameters are how many numbers you need to be returned. For example, if we call `unidrnd(11,5,2)`, we will get back a 5×2 matrix of random numbers between 1 and 11.

```
function bsample = getBSample(rc,n,kn)
    bsample = zeros(n,1); % to store the bootstrap sample
    M = floor(n/kn); % number of subintervals
    for s = 1:M
        id_sub = ((s-1)*kn+1:s*kn)';
        id_rc = unidrnd(kn,kn,1); % resample with replacement
        bsample(id_sub) = rc(id_rc);
    end
end
```

We save the random draw from the discrete uniform in `id_rc`, and then obtain the random sample by call-

ing `rc(id_rc)`. This random sample for the first subinterval is stored in `bsample` at the correct location.

Question: We are almost done, but there is a logical mistake in the code above. What is it?

When we move to the 2nd subinterval, the next `id_rc` will give us a new sample from the 1st subinterval, not from the 2nd subinterval. Fortunatelly, that is easy to fix. We just need to move the random numbers from somewhere between $\{1, \cdots, k_n\}$ to somewhere beteween $\{1, \cdots, k_n\} + k_n$. So that we get the numbers for the 2nd subinterval. To get the numbers for the 3rd subinterval, we need to add $2k_n$, and so on. Making this little adjustment:

```
function bsample = getBSample(rc,n,kn)
    bsample = zeros(n,1); % to store the bootstrap sample
    M = floor(n/kn); % number of subintervals
    for s = 1:M
        id_sub = ((s-1)*kn+1:s*kn)';
        id_rc = unidrnd(kn,kn,1) + (s-1)*kn; % resample with replacement
        bsample(id_sub) = rc(id_rc);
    end
end
```

Now, if we give this function our vector of diffusive returns for 1 day, we will get back a new vector of diffusive returns for 1 day, resampled with replacement from M subintervals.

Extending for Multiple Assets

Now we can obtain a bootstrap sample for a single day, for a single continuous return. What if we have more than one continuous return? Assume we have two continnuous returns for a single day: r_1^c and r_2^c. Also, let's assume they are stored in vector form as before, and let's define r^c as the horizontal-join of r_1^c and r_2^c:

$$r^c = \begin{pmatrix} r_1^c & r_2^c \end{pmatrix} = \begin{pmatrix} r_{1,1}^c & r_{2,1}^c \\ r_{1,2}^c & r_{2,2}^c \\ \vdots & \vdots \\ r_{1,n}^c & r_{2,n}^c \end{pmatrix}$$

Let's extend the function `getBSample` to deal with an `rc` as above (a $n \times 2$ matrix). In this case, it is important to resample with replacement both stocks at the same time. If we resample them separately, we will destroy the correlation between the assets.

We need to add a new input to the function: how many assets we have in `rc`. Denote this number by `a`. Additionaly, we then need to adjust `bsample` so that it can hold a matrix instead of a vector. Lastly, we need to add `:` when calling `rc`:

```
function bsample = getBSample(rc,n,a,kn)
    bsample = zeros(n,a); % to store the bootstrap sample
    M = floor(n/kn); % number of subintervals
```

```
    for s = 1:M
        id_sub = ((s-1)*kn+1:s*kn)';
        id_rc = unidrnd(kn,kn,1) + (s-1)*kn; % resample with replacement
        bsample(id_sub,:) = rc(id_rc,:);
    end
end
```

We can now deal with the case where we need a bootstrap sample for a single day for multiple assets at the same time.

Putting the getBSample Function In Use

Next, we are going to actually use this function to solve the case where we have multiple assets, multiple days, and need multiple bootstrap samples.

Assume we have two different assets, and a market index, and that we have computed their continuous returns. Assume these continuous returns are stored in three different vectors: r_1^c, r_2^c and r_m^c. These vectors are of the size $n \times T$, since we now have T days of data. We can then build r^c as before:

$$r^c \equiv \begin{pmatrix} r_1^c & r_2^c & r_m^c \end{pmatrix}$$

Assume we need J bootstrap repetitions, then we can write our main script as:

```
theta_hat = zeros(J); % save the estimated values for each boostrap repetition

% for each bootstrap repetition
for j = 1:J
    bsample = zeros(n*T,a); % to store the bootstrap sample
                            % n*T returns for a assets
    % for each day
    for t = 1:T
        id_bsample = (1:n)' + (t-1)*n;
        % get the bsample for this day
        bsample(id_bsample) = getBSample(rc(id_bsample),n,a,kn);
    end

    % compute theta hat
    theta_hat(j) = % whatever you need to compute, like RV or TV
end

% compute the quantiles of theta_hat to find the confidence interval for theta
CI_low = quantile(theta_hat, 0.025);
CI_up = qunatile(theta_hat, 0.975);
```

The script above uses the function we created to simplify the bootstrap procedure. Because we built `getBSample` with care, the only thing that is left to do in the bootstrap is to compute $\widehat{\theta}^*$. It is usually a good idea to also have a function for that, simplifying even further the bootstrap.

Extending for Multiple Days

The sampling part of the bootstrap can be simplified even further, if we extend `getBSample` to work even when `rc` contains data on more than a single day. To do so, we need to add a new input: the number of

days of data stored in `rc`. Denote this number by T.

To extend `getBSample` we need to adjust `bsample`, and include a new for-loop for each day. To do so, let's create a new function called `getTBSample`, and use `getBSample` to build it!

```
function bsample = getTBSample(rc,n,a,T,kn)
    bsample = zeros(n*T,a); % to store the bootstrap sample
    M = floor(n/kn); % number of subintervals
    for t = 1:T
        % get returns matrix for day t
        id_day = (1:n)' + (t-1)*n;
        bsample(id_day,:) = getBSample(rc(id_day,:),n,a,kn);
    end
end
```

We increased the size of `bsample` to hold the data for multiple days, and a for-loop to resample at each day. At each day, we obtain the return data for that day via `id_day`, and call `getBSample` to get a bootstrap sample for that single day. We then save this new sample in `bsample` and repeat the procedure for the next day.

Using this new function, we can revisit the script from the previous section:

```
theta_hat = zeros(J); % save the estimated values for each boostrap repetition
% for each bootstrap repetition
for j = 1:J
    % obtain a bootstrap sample
    bsample = getTBSample(rc,n,a,T,kn);
    % compute theta hat
    theta_hat(j) = % whatever you need to compute, like RV or RB or Rrho
end
% compute the quantiles of theta_hat to find the confidence interval for theta
CI_low = quantile(theta_hat, 0.025);
CI_up = qunatile(theta_hat, 0.975);
```

By using functions to build smaller blocks of computation, we can create powerful and readable scripts, that are much easier to understand and debug. And when you come back to them a couple of months from now, or even years, you will be able to understand what is happening much quicker.

Challenge: 3-lines Bootstrap Sampling Function

If you made it this far, congratulations! You now know how to create a neat bootstrap script!

Here is a challenge for you. The bootstrap script we created above is not the fastest script out there. And there are multiple ways to improve it. If you spend some time thinking about how to get bootstrap samples efficiently, you will notice that finding the random indices does not depend on the number of assets we have. That is, even if we have more than one asset (`rc` is a matrix), it makes no difference when finding `id_rc`. You will also notice that you can get `id_rc` for all subintervals M at the same time. It is also possible to get `id_rc` for all days T at the same time. And believe it or not, it is also possible to get `id_rc` for all bootstrap repetitions J at the same time.

The 3-Lines Bootstrap Sampling Function Challenge: create the 3-lines function `getJTBSample` (as be-

low) that computes the indices `id_rc` that recover the bootstrap samples from `rc`.

```
function id_rc = getJTBSample(n,kn,T,J)
    M = floor(n/kn);    % this is the first line, so you only need 2 more! :D
    offset = ?;         % hint: this is a M*kn*T x 1 vector
    id_rc = ? + offset; % hint: this is a M*kn*T x 1 x J matrix (3-dimensional)
end
```

To help you solve this very hard challenge, here are some step-by-step suggestions:

1. Extend the `getTBSample` to return multiple bootstrap samples at the same time.
    - Hint: Initialize `bsample` as `zeros(n*T,a,J)` (a 3 dimensional matrix).
2. Substitute the for-loop for the subintervals for a Kronecker product.
    - Hint: first find the indices for all subintervals using a big `unidrnd`, notice that for the 2nd subinterval forward the indices will not be right, you need to make an adjustment as we did in the **Base Case**. Call this adjustment an `offset` and use a kronecker product to find the correct offset so that the `unidrnd` values are right for each subinterval.
3. Substitute the for-loop for the days for a Kronecker product.
    - Hint: now `unidrnd` also has to generate numbers for all subintervals and all days; make sure to adjust the `offset` so that you correct the indices for days and subinterval.
4. Substitute the for-loop for the boostrap samples for a Kronecker product.
    - Hint: now `unidrnd` has to generate numbers for all subintervals, all days and all bootstrap samples, so make it a multidimensional matrix, like `unidrnd(kn,x,y,z)`.
5. Create the function `getJTBSample` that returns a 3-dimensional matrix `id_rc` containing the indices that give us J bootstrap samples for all days, resampling with replacement from M subinterval for each day.
