Lesson Notes

MODULE 2 | LESSON 4

ORDER STATISTICS VOLATILITY ESTIMATOR IN ACTION

Reading Time	30 minutes
Prior Knowledge	Probability density function (pdf), Cumulative density function (cdf), jumps, continuous volatility
Keywords	incomplete beta function

In the last lesson, we walked through the research by Spadafora et al. ("Volatility Estimator") wherein they introduced their order statistics volatility estimator, which allows for the occurrence of jumps in asset prices (and therefore in asset price return volatility). In this lesson, we have the luxury of seeing this model in action: We walk through the code provided by the authors, which instantiates their model, and we experiment with the results by modifying our simulated data.

Lucky for us, we have already spent some time understanding the theory and mechanics behind this model for estimating volatility in the presence of jumps. We recall that the model assumes a Levy process, which includes three terms: a deterministic drift term, a continuous Brownian motion term (scaled by the volatility we seek here to estimate), and a jump term. Now, we get to examine the code itself to see how it implements these ideas and then experiment with the code, comparing its output to our intuitive expectations to improve our understanding.

1. Local Volatility, `local_volatility`

As seen in Spadafora et al. ("Volatility Estimator", there are seven functions in the VolatilityEstimator repo. Two of them are simply plotting functions, to which we will return. The function `local_volatility` simply calculates the standard deviation for each window (of length `bandwidth`) in the entire return series. It is completely naïve and agnostic as to whether the window of returns includes jumps or not: It uses all returns in the standard deviation calculation regardless of how they might be identified in other parts of the code and whether they are continuous or jumps.

- The function initializes an array of zeros with the same length as the returns array (line 16).
- It populates the latter part of the array with the rolling standard deviation for the period of length `bandwidth` preceding it (line 17–18).
- Since the first few return observations (up to index 'bandwidth') are not sufficient to first properly calculate standard deviation, the function simply backfills this part with the first properly calculated standard deviation – the one based on the first array of length `bandwidth`.

2. Theoretical Volatility, `_thrLocalVol`

This function basically corresponds to Algorithm 1 from the paper. It is very similar to the previous local volatility function, except that it requires a binary array ('ju' short for jump) as an input to indicate where the jumps are: A zero value in a given index `i` indicates that the return associated with that index should be included in the calculation of the continuous "theoretical volatility," while a value of one in the `ju` array indicates a jump.

This logic is reversed in line 84, which creates the no-jump `no_ju` array: by subtracting each `ju` array values from the value one, we now have an array indicating with the value one which returns should be included in the continuous theoretical volatility and zeros for the jump returns.

In line 87, the jump returns are multiplied by zero and the non-jump returns are multiplied by one. Then, the theoretical volatility can be calculated by squaring the returns: The jump returns have been converted to zeros and effectively filtered out from the calculation. The sum of these squared returns can be divided by the sum of the ones in the rolling window, which is equal to the sum of the ones in the no-jumps array (line 88). We take the square root to get the volatility from the variance (line 89). As before, the early volatilities are backfilled (line 90).

3. Local Volatility with Order Statistics, `local_vol_order_stats`

This function takes us a step further than the previous function and corresponds to Algorithm 2.

Line 37 in Spadafora et al. ("Volatility Estimator") calculates the theoretical (continuous) volatility using the function above `_thrLocalVol` assuming at first that there are no jumps (since `ju` has been initialized to all zeros in line 33). Notice we enter a loop in line 35, which corresponds to the pseudocode from Algorithm 2 in the paper, "for t=1,. . . ,n." In the code version of the pseudocode, `n = maxiter` ("maximum iterations"), which as you can see is set by default to 100 (line 32).

Now that these theoretical volatilities have been calculated (albeit in the first instance with the assumption there are no jumps), this function renormalizes the returns by dividing them by their respective volatilities. Next, it calls the function `_getkJumpProb` (which we will discuss in the next section) to update the Boolean jump indicator array (line 39). As explained in the comment above it, line 42 sets any returns to continuous (not jumps) if their value is less than its respective theoretical volatility just calculated in line 37.

Line 43 tests whether any of the jump determinations have changed since the last iteration: The syntax is actually written such that if the length of the jumps array minus the number of unchanged jump identifications is less than or equal to zero, then the loop breaks. When you think about it, the number of changed jump identifications cannot be greater than the total number of returns, so this is really a test to ensure that the number of unchanged returns (`np.sum(ju_old == ju)`) equals the number of returns (the length of the array). In other words, we consider that since the jump identifications have not changed, we have reached convergence and additional iterations through the loop will not change the jump identification. If you are following along with the pseudocode in the paper, we have reached the last line: "until the identified jumps before and after the application of the estimator are the same" (Spadafora et al. "Jumping VaR" 11).

4. Get the Probability that the k-th Order Statistic is a Jump, `_getkJumpProb`

The NumPy argsort function returns the array of the indices of the values in the input array, if they were sorted (in ascending order). As shown in Spadafora et al. ("Volatility Estimator"), line 51 therefore returns

the ordered index of the returns. Line 52, on the other hand, actually orders the renormalized (meaning, divided by their current local volatility) returns.

Line 58 set up the loop (Spadafora et al. "Volatility Estimator"). You might expect that the upper limit of the range of the loop would simply be the number of returns, but it is not. It is only half the number of returns: Since we have assumed a Gaussian distribution, which we know to be symmetric, we only need to perform this loop half as many times as there are returns—but we have to test the kth largest positive and the kth largest negative return in each iteration of the loop in order to test all the returns. In this way, each return, positive and negative, will be tested for the probability that it is a jump.

Line 59 checks whether the kth order statistic (i.e., the smallest return) is already a jump, and if it is, then the number of non-jumps is decremented by one. If it is not already considered a jump, then another function is called in line 62, `_kSmallestCDF`, which we discuss next, to calculate the probability that this return is not a jump, given the relevant order statistics information (Spadafora et al. "Volatility Estimator"). If the probability that the returns is not a jump is less than the `prob_cut_off` (i.e., “the p tolerance level,”) (Spadafora et al. "Jumping VaR" 11), then the Boolean jumps array is updated to reflect that this return is considered a jump (and the number of non-jump returns is again decremented by one).

Then, this procedure is repeated for the other side of the distribution, namely, the negative returns.

When all the returns have been tested for their probability of being a jump, the fully updated Boolean jump array is returned by this function.

5. The Cumulative Density Function for the k-th order statistic, `_kSmallestCDF`

This is where the magic of order statistics happens. Really, the scipy.special function `betainc` does the heavy lifting here by calculating the incomplete beta function. Feel free, of course, to read in the documentation for this function that it can be expressed in terms of the Gamma function. As you see in Spadafora et al., it can also be used to find the cdf “of the k-th order statistic in dataset of n Standard Normal random variables” ("Jumping VaR" 27). Now, the arguments for the function `_kSmallestCDF` are x, k, and n, while the arguments for the `betainc` function are a, b, and x (Spadafora et al. "Volatility Estimator"), so this requires a little disentangling:

`betainc` find the cumulative probability for x assuming the distribution is bounded by a and b.

When this function is put in service of the `_kSmallestCDF` function defined by the authors in the repo, it is returning the cumulative probability of x* assuming the distribution is bounded by k and $n-k+1$. The asterisk on x* is to distinguish this x value from the betainc argument as well as the `_kSmallestCDF` argument because, as you can see in line 95 (Spadafora et al. "Volatility Estimator"), the normalized return x is itself converted to a cdf before it is passed as the argument to `betainc`. With this transformation, and the special property of the incomplete beta function, we find the probability that the kth order statistic is not drawn from the reference distribution—in which case, it is determined to be a jump.

When the `local_vol_order_stats` has identified all the returns it considers jumps, then the new continuous volatility can be calculated.

You are encouraged to experiment with the parameter values, especially the following, to develop an intuition about the relationship between jumps and continuous volatility:

- th_gauss_vol
- jumps_percentage

- prob_cut_off
- jumps_size
- array_size
- bandwidth

In addition to deepening your understanding in this way, please try to find parameter values when the model breaks down, that is, where it clearly misidentifies jumps, identifies too many jumps, and/or fails to identify jumps. Post your results to the forum and give your best explanation for what is going on or going wrong. Why is the model failing under the circumstance of the input values you have provided?

As an example, the parameters might be set as follows, such that there are no jumps in the simulated date, but there is some tolerance for a return to be misidentified as a jump (10%):

th_gauss_vol = 0.2

jumps_percentage = 0.0

prob_cut_off = .1

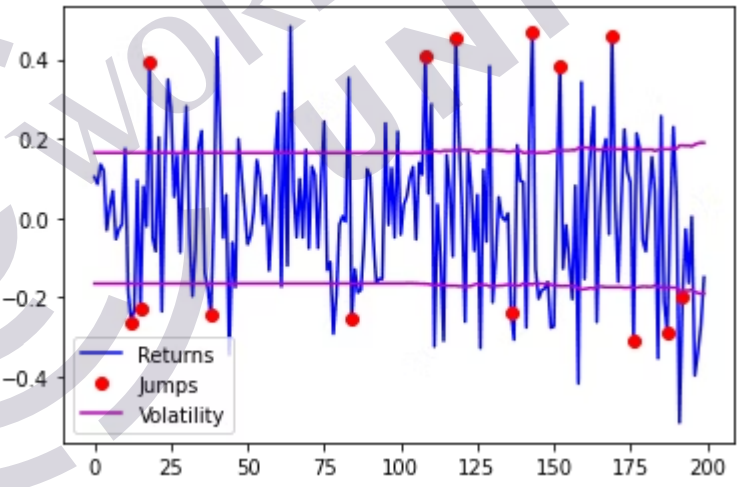
jumps_size = 0.0

array_size = 200

bandwidth = 100

The following results show that the model takes advantage of that tolerance, identifying some returns as jumps even though they are just realizations from the Gaussian reference distribution.

Figure 1: Jump Identification Despite No Modeled Jumps



Similarly, with the following parameter values, we see the model has not identified any jumps although the simulated data clearly includes some:

th_gauss_vol = 0.2

jumps_percentage = 0.05

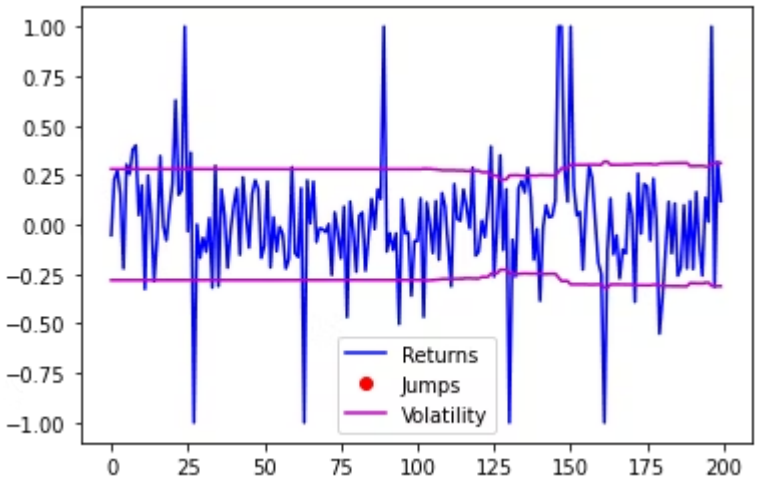
prob_cut_off = .0

jumps_size = 1.0

array_size = 200

bandwidth = 100

Figure 2: No Jump Identification Due to Zero Tolerance



And as a result, the volatility calculated by the model is well overestimated compared to the theoretical value supplied as an input:

Theoretical Gaussian Volatility: 0.2

Standard Volatility: 0.31014746420367123

Notice too in the above figure that when the array size is larger than the `bandwidth` window, you can see how the volatility evolves over time.

6. Conclusion

In this lesson, we saw an example of implementing a volatility estimation model in Python. We matched the code to the pseudocode with reference to the theory behind it.

In the next module, we look at another innovative model that uses state-of-the-art transformer technology to better estimate Value at Risk, as well as the Python code that implements it.

References

- Spadafora, Luca, et al. "Jumping VaR: Order Statistics Volatility Estimator for Jumps Classification and Market Risk Modeling." *arXiv*, 22 Mar 2018, https://arxiv.org/abs/1803.07021.
- Spadafora, Luca, et al. "Volatility Estimator." *GitHub*. https://github.com/sigmaquadro/VolatilityEstimator