Value at Risk in Python – Shaping Tech in Risk Management

The aim of this article is to give a quick taste of how it is possible to build practical codes in Python for financial application using the case of Value at Risk (VaR) calculation. The following paragraph will present a brief introduction to Python, then the article will continue with a broad overview of VaR without going into the details of its mathematical properties and then it will tackle different VaR formulas, explaining the main intuition behind and presenting the relative codes. To benefit the most from this article the reader should be already at least vaguely familiar with the concept of Value at Risk, even better if also aware of the diverse mathematical properties of different VaR formulas.

Why Python?

As cited in the official documentation on www.python.org:

"*Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.*"

Historically there have often been different and diverse roles inside a financial company, specifically both the classic code developer and the employee with a finance background have been present and have been working together on finding a technological solution to perform finance related tasks. This implies that at first one would have to sketch a code in an easy to learn language and then the developer would have to rewrite the initial idea in an efficient way, using a faster and more complex programming language. Unfortunately this process has often implied that the developer, not coming from a financial background, didn't really understand the original idea in the code and possibly made some mistaking during the process of improving the code while translating it or just rewriting it. Nowadays some programming languages exist that are both fast and easy to learn so that one person can directly code the initial idea without the need of a professional developer to rewrite it. This obviously jumps over possible mistakes arising from communicational misunderstanding or poor knowledge coming from the developer or the other employees.

Python is one of those languages. Thanks to its speed and easy semantics is in fact nowadays prominent in the finance industry, and its success just keeps growing. To learn more about Python and on why it is a great tool for finance follow the links below.

- https://www.python.org/doc/essays/blurb/
- https://www.safaribooksonline.com/library/view/python-for-finance/9781491945360/ch01.html

Value at Risk, brief introduction

Value at risk (VaR) is a certified achievement in the study of quantitative risk management and even if with time its use is increasingly often being combined with other measures of risk, it is still present, in different forms, in the agenda of all market risk managers. Its general form can be written as:

$$r_\alpha = r: P(R \leq r) = \alpha$$

Where r_α is the VaR, hence that loss value for which probability of losses bigger than VaR itself is equal to α. Despite various critiques concerning the inability of VaR to measure the magnitude of losses over α, it remains a key element in financial risk management and for such reason its calculation is still very important.

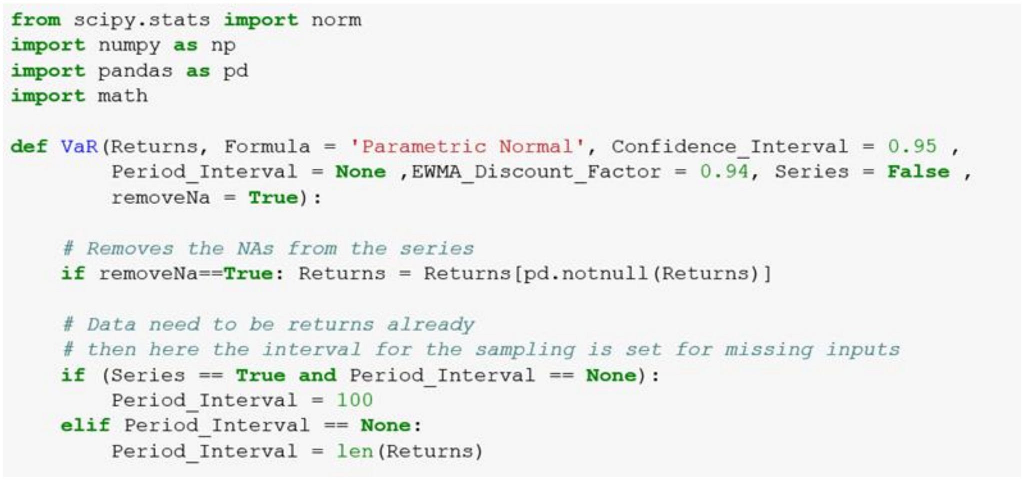
There are multiple techniques to calculate VaR with differences in formulas and sampling, strongly altering their ability to foresee future risks and to shield the asset manager from possible downsides, It is therefore of utmost importance to understand the general differences among them. In the next section a brief introduction to the main characteristic of four different VaR formulas is given and the relative Python code on how to build them is presented.

VaR – one function for four formulas

An alternative and more precise solution to download the full code is to visit this GitHub repository where the code is held:

https://github.com/BSIC/VaR

This first section contains the importing of the necessary libraries, the definition of a function that will contain all the different formulas and parameters setting, and some initial settings to handle missing inputs. In every sub-formula there will be a distinction between a call for a single VaR referring to the last period interval, and a call for a series covering the whole set of data.



```python
from scipy.stats import norm
import numpy as np
import pandas as pd
import math

def VaR(Returns, Formula = 'Parametric Normal', Confidence_Interval = 0.95 ,
        Period_Interval = None ,EWMA_Discount_Factor = 0.94, Series = False ,
        removeNa = True):

    # Removes the NAs from the series
    if removeNa==True: Returns = Returns[pd.notnull(Returns)]

    # Data need to be returns already
    # then here the interval for the sampling is set for missing inputs
    if (Series == True and Period_Interval == None):
        Period_Interval = 100
    elif Period_Interval == None:
        Period_Interval = len(Returns)
```

Parametric VaR (Variance/Covariance VAR) is the most common form used in practice due to its simple nature and the low number of parameters it requires to be computed. To build it, the only variables needed are the mean and the standard deviation of a portfolio/security. The problem that follows from such simple requirements is that it works under two restrictive assumptions, namely normal distribution and independence of returns. This leads to myopically equating all returns in terms of importance, overlooking big shocks that should be carried over and should be given more power to impact the actual VaR. It is possible to notice, in fact, that the parametric method is one of the most stable among the presented VaR formulas, highlighting its lack of capability to absorb new information and shape the capital risk appropriately.

```
#================================
#     Parametric Normal VaR
#================================
if Formula == 'Parametric Normal':

    if Series == False:
        Data = Returns[-Period_Interval:]
        stdev = np.std(Data)
        Value_at_Risk = stdev*norm.ppf(Confidence_Interval)

    if Series == True:
        Value_at_Risk = pd.Series(index=Returns.index, name = 'ParVaR')
        for i in range(0,len(Returns)-Period_Interval):
            if i == 0: Data = Returns[-(Period_Interval):]
            else: Data = Returns[-(Period_Interval+i):-i]
            stdev = np.std(Data)
            Value_at_Risk[-i-1] = stdev*norm.ppf(Confidence_Interval)
```

EWMA (Exponentially weighted moving average) is a step forward from the parametric VaR, in the sense that it tries to solve the problem of slow reaction to new information and the equal importance of returns. Using a decay factor the EWMA formula is able to weight different information as it comes in, giving more importance to recent returns and less importance to data far in the past by slowly decaying their contribution to the VaR. Through this, the measure limits the 'echo effect', occurring when a large shock of the past becomes too old to be considered and leaves the estimation, causing a big change in the VaR which is not due to a change in the markets.

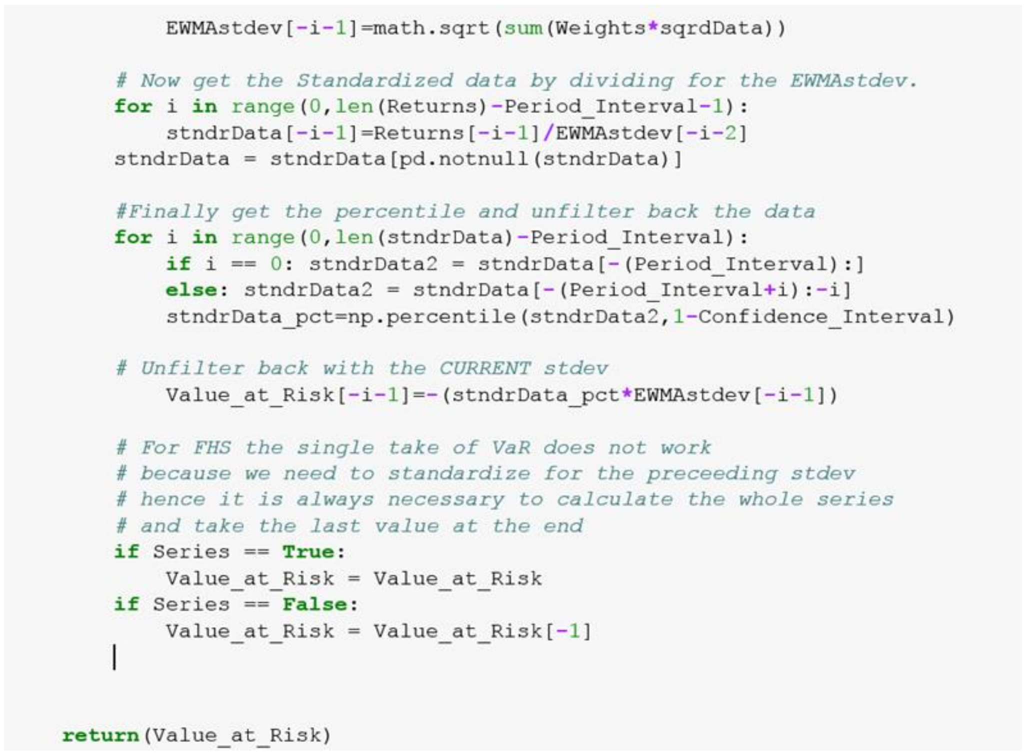
```
#============================
# EWMA Parametric VaR
#============================
if Formula == 'Parametric EWMA':

    # Defining exponentially smoothed weights components
    Degree_of_Freedom = np.empty([Period_Interval,])
    Weights = np.empty([Period_Interval,])
    Degree_of_Freedom[0]=1.0
    Degree_of_Freedom[1]=EWMA_Discount_Factor
    Range = range(Period_Interval)
    for i in range(2,Period_Interval):
        Degree_of_Freedom[i]=Degree_of_Freedom[1]**Range[i]
    for i in range(Period_Interval):
        Weights[i]=Degree_of_Freedom[i]/sum(Degree_of_Freedom)

    if Series == False:
        # Squaring returns for the formula
        sqrdData = (Returns[-Period_Interval:])**2
        EWMAstdev=math.sqrt(sum(Weights*sqrdData))
        Value_at_Risk = EWMAstdev*norm.ppf(Confidence_Interval)

    if Series == True:
        Value_at_Risk = pd.Series(index=Returns.index, name = 'EWMAVaR')
        sqrdReturns = Returns**2
        # For efficiency here we square returns first
        # so the loop does not do it repeadetly.
        # This loop repeats the VaR calculation
        # iterated for every xxx period interval
        for i in range(0,len(Returns)-Period_Interval):
            if i == 0: sqrdData = sqrdReturns[-(Period_Interval):]
            else: sqrdData = sqrdReturns[-(Period_Interval+i):-i]

            EWMAstdev=math.sqrt(sum(Weights*sqrdData))
            Value_at_Risk[-i-1] = EWMAstdev*norm.ppf(Confidence_Interval)
```

Historical Simulation (HS) VaR is instead efficient when the risk manager cannot, or doesn't intend to, make assumptions on the underlying distribution of returns as it is calculated by the simple picking of the chosen percentile loss in a given period of time. This method is even simpler than the parametric one and that is precisely its weakness. The main underlying logic is that the past is a good predictor for the future. By looking back, it is then possible to incorporate all data points into the risk calculation but this unfortunately leads to too simplistic a model.

```
#============================
#    Historical Simulation
#============================
if Formula == 'Historical Simulation':

    if Series == False:
        Data = Returns[-Period_Interval:]
        Value_at_Risk = -np.percentile(Data,1-Confidence_Interval)

    if Series == True:
        Value_at_Risk = pd.Series(index=Returns.index, name = 'HSVaR')
        for i in range(0,len(Returns)-Period_Interval):
            if i == 0: Data = Returns[-(Period_Interval):]
            else: Data = Returns[-(Period_Interval+i):-i]
            Value_at_Risk[-i-1] = -np.percentile(Data,1-Confidence_Interval)
```

Filtered Historical Simulation VaR can be described as being a mixture of the historical simulation and EWMA methods. Returns are first standardized, with volatility estimation weighted as in EWMA VaR, before a historical percentile is applied to the standardized return as in the historical model. From the graphs it is easy to spot that this model looks very much like EWMA, as returns are standardised and weighted by the same decay factor. The main difference lies in the fact that this model is generally more conservative because it looks at the worst past losses and adjust its VaR value according to it.

```
#=====================================
#    Filtered Historical Simulation
#=====================================
if Formula == 'Filtered Historical Simulation':

    # Defining exponentially smoothed weights components
    Degree_of_Freedom = np.empty([Period_Interval,])
    Weights = np.empty([Period_Interval,])
    Degree_of_Freedom[0]=1.0
    Degree_of_Freedom[1]=EWMA_Discount_Factor
    Range = range(Period_Interval)
    for i in range(2,Period_Interval):
        Degree_of_Freedom[i]=Degree_of_Freedom[1]**Range[i]
    for i in range(Period_Interval):
        Weights[i]=Degree_of_Freedom[i]/sum(Degree_of_Freedom)


    Value_at_Risk = pd.Series(index=Returns.index, name = 'FHSVaR')
    EWMAstdev = np.empty([len(Returns)-Period_Interval,])
    stndrData = pd.Series(index=Returns.index)

    # For efficiency here we square returns first avoiding looping
    sqrdReturns = Returns**2

    # Computations here happen in different times,
    # because we first need all the EWMAstdev
    # First get the stdev according to the EWMA
    for i in range(0,len(Returns)-Period_Interval):
        if i == 0: sqrdData = sqrdReturns[-(Period_Interval):]
        else: sqrdData = sqrdReturns[-(Period_Interval+i):-i]
```

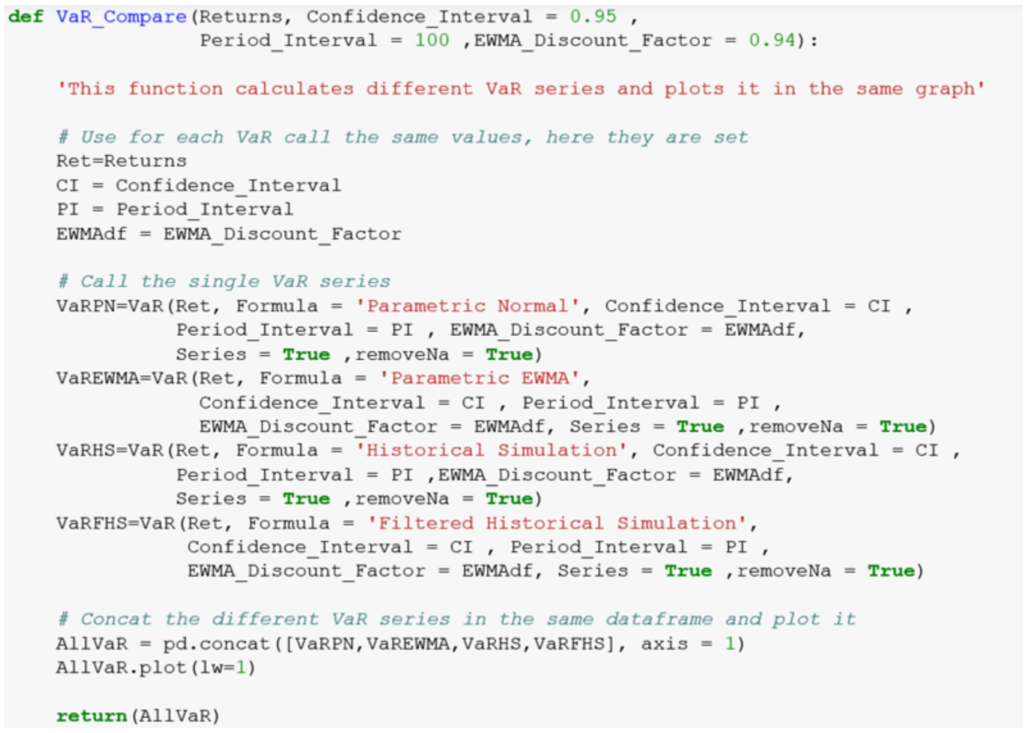


```
            EWMAstdev[-i-1]=math.sqrt(sum(Weights*sqrdData))

        # Now get the Standardized data by dividing for the EWMAstdev.
        for i in range(0,len(Returns)-Period_Interval-1):
            stndrData[-i-1]=Returns[-i-1]/EWMAstdev[-i-2]
        stndrData = stndrData[pd.notnull(stndrData)]

        #Finally get the percentile and unfilter back the data
        for i in range(0,len(stndrData)-Period_Interval):
            if i == 0: stndrData2 = stndrData[-(Period_Interval):]
            else: stndrData2 = stndrData[-(Period_Interval+i):-i]
            stndrData_pct=np.percentile(stndrData2,1-Confidence_Interval)

        # Unfilter back with the CURRENT stdev
            Value_at_Risk[-i-1]=-(stndrData_pct*EWMAstdev[-i-1])

        # For FHS the single take of VaR does not work
        # because we need to standardize for the preceeding stdev
        # hence it is always necessary to calculate the whole series
        # and take the last value at the end
        if Series == True:
            Value_at_Risk = Value_at_Risk
        if Series == False:
            Value_at_Risk = Value_at_Risk[-1]
        |


    return(Value_at_Risk)
```

The last line closes the initial 'def' statement and returns the requested VaR calculation.

VaR_Compare – let us see the differences

The next function is just a wrapper that calls the previously defined VaR function four times, each with a different formula, wraps the VaR series in one datasets and then plots it in a chart.



```
def VaR_Compare(Returns, Confidence_Interval = 0.95 ,
               Period_Interval = 100 ,EWMA_Discount_Factor = 0.94):

    'This function calculates different VaR series and plots it in the same graph'

    # Use for each VaR call the same values, here they are set
    Ret=Returns
    CI = Confidence_Interval
    PI = Period_Interval
    EWMAdf = EWMA_Discount_Factor

    # Call the single VaR series
    VaRPN=VaR(Ret, Formula = 'Parametric Normal', Confidence_Interval = CI ,
             Period_Interval = PI , EWMA_Discount_Factor = EWMAdf,
             Series = True ,removeNa = True)
    VaREWMA=VaR(Ret, Formula = 'Parametric EWMA',
               Confidence_Interval = CI , Period_Interval = PI ,
               EWMA_Discount_Factor = EWMAdf, Series = True ,removeNa = True)
    VaRHS=VaR(Ret, Formula = 'Historical Simulation', Confidence_Interval = CI ,
             Period_Interval = PI ,EWMA_Discount_Factor = EWMAdf,
             Series = True ,removeNa = True)
    VaRFHS=VaR(Ret, Formula = 'Filtered Historical Simulation',
              Confidence_Interval = CI , Period_Interval = PI ,
              EWMA_Discount_Factor = EWMAdf, Series = True ,removeNa = True)

    # Concat the different VaR series in the same dataframe and plot it
    AllVaR = pd.concat([VaRPN,VaREWMA,VaRHS,VaRFHS], axis = 1)
    AllVaR.plot(lw=1)

    return(AllVaR)
```

An example results using VaR_Compare on a series of an ETF's daily returns:

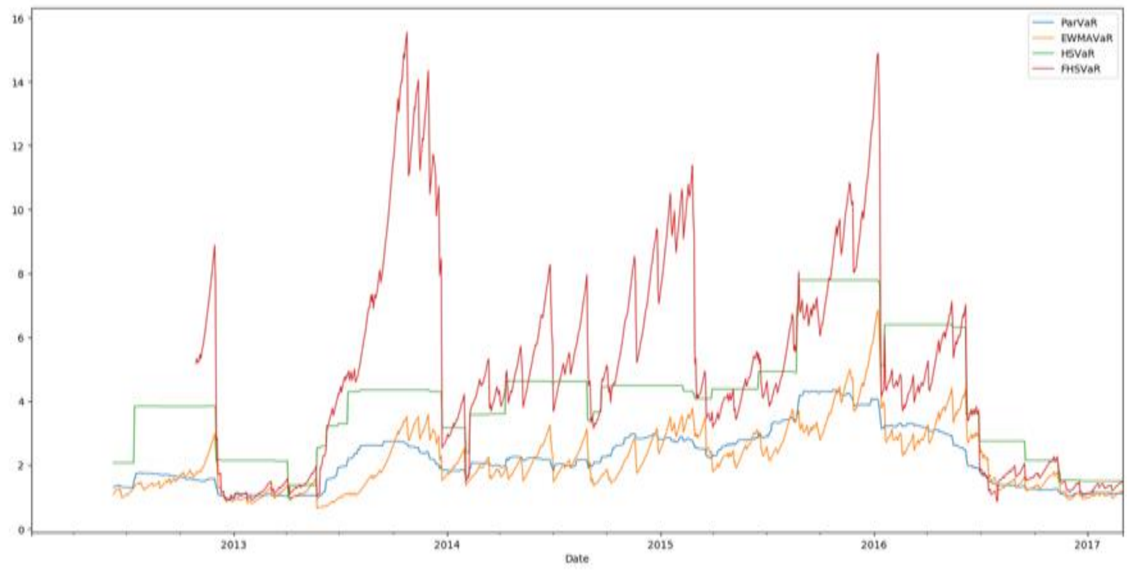


Chart 1: Plot of VaR_Compare

TAGS: Value-at-Risk, Python, parametric, EWMA, historical, filtered historical