Bootstrapping - Practical Lesson 6

Matteo Sani
matteosan1@gmail.com

October 8, 2019

1 Bootstrapping

1.1 Recap

- basic Python;
- discount factor interpolation and forward rates: we implemented functions then classes;
- last lesson we looked at Python modules and we implemented a class to store data defining an OIS contract, and for calculating its NPV.

1.2 This lesson

Now we're going to look at how extract a discount curve from OIS market data, via a process called *bootstrapping*. This is the ABC of financial mathematics, since you almost always need a discount curve to price any contract, especially if you're interested in its NPV. We're going to concentrate on EONIA swaps in order to build an EUR discount curve.

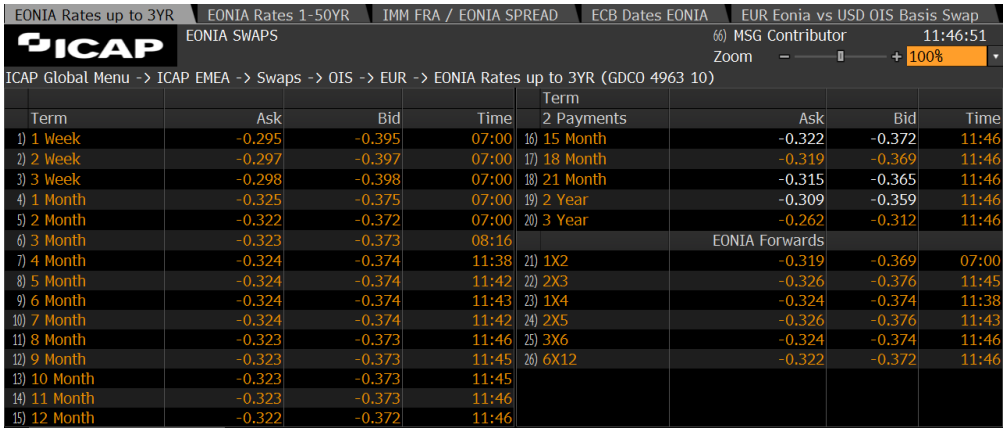
1.3 Bootstrapping

1.3.1 Getting the data

The first problem is actually getting the data from somewhere, and this is not actually as simple as it sounds.

The issue is that the EONIA swap market is Over The Counter (OTC) and it's not straightforward to access it. Unlike (some) listed futures, where anyone with a retail brokerage account can view and apply realtime prices, to trade in the EONIA swap market you have to be a financial institution or at least a large company and have an agreement with a broker which operates in the market. One of the main brokers in the OIS market is ICAP.

Though there exist some electronic platform in which market participants post bids and offers and other participants can apply them, in practice a lot of trading is still done over "voice", i.e. by phone or more commonly over chat. For convenience, however, Bloomberg provides a service which displays indicative realtime prices as provided by a selection of relevant brokers.



EONIA Rates up to 3YR | EONIA Rates 1-50YR | IMM FRA / EONIA SPREAD | ECB Dates EONIA | EUR Eonia vs USD OIS Basis Swap

ICAP — EONIA SWAPS — 66) MSG Contributor — 11:46:51 — Zoom 100%

ICAP Global Menu -> ICAP EMEA -> Swaps -> OIS -> EUR -> EONIA Rates up to 3YR (GDCO 4963 10)

Term	Ask	Bid	Time
1) 1 Week	-0.295	-0.395	07:00
2) 2 Week	-0.297	-0.397	07:00
3) 3 Week	-0.298	-0.398	07:00
4) 1 Month	-0.325	-0.375	07:00
5) 2 Month	-0.322	-0.372	07:00
6) 3 Month	-0.323	-0.373	08:16
7) 4 Month	-0.324	-0.374	11:38
8) 5 Month	-0.324	-0.374	11:42
9) 6 Month	-0.324	-0.374	11:43
10) 7 Month	-0.324	-0.374	11:42
11) 8 Month	-0.323	-0.373	11:46
12) 9 Month	-0.323	-0.373	11:45
13) 10 Month	-0.323	-0.373	11:45
14) 11 Month	-0.323	-0.373	11:46
15) 12 Month	-0.322	-0.372	11:46

Term			
2 Payments	Ask	Bid	Time
16) 15 Month	-0.322	-0.372	11:46
17) 18 Month	-0.319	-0.369	11:46
18) 21 Month	-0.315	-0.365	11:46
19) 2 Year	-0.309	-0.359	11:46
20) 3 Year	-0.262	-0.312	11:46
EONIA Forwards			
21) 1X2	-0.319	-0.369	07:00
22) 2X3	-0.326	-0.376	11:45
23) 1X4	-0.324	-0.374	11:38
24) 2X5	-0.326	-0.376	11:43
25) 3X6	-0.324	-0.374	11:46
26) 6X12	-0.322	-0.372	11:46

As part of our Quants duties we have set up an Excel spreadsheet which acquires this data from Bloomberg in realtime. From this spreadsheet, it's easy to export the data into a Python file - I have done this and saved the data in a module called `ois_data.py`.

We will use a data set extracted in this way, and derive from it the corresponding discount curve.

```
In [1]: import ois_data
        print (type(ois_data.quotes))

<class 'list'>


In [2]: ois_data.quotes[0]

Out[2]: {'months': 1, 'rate': -0.35}

In [3]: ois_data.quotes[-1]

Out[3]: {'months': 720, 'rate': 0.997}

In [4]: ois_data.observation_date

Out[4]: datetime.date(2019, 10, 30)
```

1.3.2 Building OIS instances

We can use the newly created function to build the OIS instances (`generate_swap_dates`). This function can then be used to build an OIS object based on the data contained in ois_data.

```
In [8]: # first check the 15 months rate
        ois_data.quotes[12]

Out[8]: {'months': 15, 'rate': -0.35}

In [5]: from finmarkets import OvernightIndexSwap, generate_swap_dates

        # with the new function generates all the dates from the
        # observation up to 15 months
```

```
        ois = OvernightIndexSwap(1e6,
                                 generate_swap_dates(ois_data.observation_date, 15),
                                 -0.35
                                )
        # print the last payment date (15 months after obs date)
        ois.payment_dates[-1]
```

```
Out[5]: datetime.date(2021, 1, 30)
```

```
In [7]: # remember we could use the npv method to
        # calculate the OIS's npv
        # only problem is we don't yet have a discount
        # curve with which to evaluate it!

        ois.npv(curve)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-7-f4413985072e> in <module>()
          4 # curve with which to evaluate it!
          5
    ----> 6 ois.npv(curve)


        NameError: name 'curve' is not defined
```

1.3.3 Bootstrapping

In the next we are going to somehow reverse what we did last week where we determined the OIS's NPV given a certain discount curve.

The general idea here is to find the discount curve such that it prices correctly each OIS, or at least as well as possible, by minimizing the sum of the square NPVs:

$$\min_{curve}\left\{\sum_{i=1}^{n}\text{NPV}(\text{ois}_i,\text{curve})^2\right\}$$

This description of the problem does not, in theory, specify any constraints on the pillar dates of the discount curve. However, the pillar dates determine the number of unknown variables (i.e. the dimensionality n of the optimization problem). A curve with n pillar dates has n pillar discount factors (note that the first discount factor with value date equal to the today date, is constrained to 1, so it doesn't count).

In practice, therefore, it makes sense to choose the pillar dates in such a way that there are exactly the right number of degrees of freedom in the optimization to match data. So the natural choide is to choose the pillar dates of the discount curve equal to the set of expiry dates of the swaps so that in principle we could find a vector of pillar discount factor that perfectly recover a zero NPV for every swap.

The reason for this is that each market quote will determine exactly one *free* discount factor which is not already determined by the other market quotes - this can be seen by considering the mathematical expression for calculating the fixed leg of the OIS swaps ($f_{\text{fix},\,i} = N \cdot K \cdot \frac{d_i - d_{i-1}}{360}$), and the way that the payment date schedules are constructed. Therefore, once we've fixed $\vec{d}$ to be a vector of pillar dates equal to the expiry dates of the OIS swaps, and we use the notation $\vec{x}$ to represent the vector of pillar discount factors, then the problem becomes:

$$\min_{\vec{x}} \left\{ \sum_{i=1}^{n} \text{NPV}(\text{ois}_i, \text{curve}(\tilde{\text{d}}, \tilde{\text{x}}))^2 \right\}$$

In practice this is an optmization problem: **to find the minimum of the above expression as a function of** $\vec{x}$, so we can just use one the available numerical optimization routines.

So let's start by defining a set of OIS objects to cover all the maturities defined by the market data we have collected in the `ois_data.py` file.

```
In [10]: from finmarkets import DiscountCurve, OvernightIndexSwap, generate_swap_dates
         import ois_data

         pillar_dates = [ois_data.observation_date]

         swaps = [] # container of the OIS objects

         for quote in ois_data.quotes:
             swap = OvernightIndexSwap(
                 # notional - doesn't really matter what we put here
                 1e6,

                 # payment dates
                 generate_swap_dates(
                     ois_data.observation_date,
                     quote['months']
                 ),

                 # the fixed rate (in the file is expressed in percent)
                 0.01 * quote['rate']
             )
             swaps.append(swap)
             pillar_dates.append(swap.payment_dates[-1])

         pillar_dates = sorted(pillar_dates)
         n_df_vector = len(pillar_dates)
```

```
In [11]: type(pillar_dates), len(pillar_dates), pillar_dates[0], pillar_dates[-1]

Out[11]: (list, 34, datetime.date(2019, 10, 30), datetime.date(2079, 10, 30))
```

Every optimization algorithm needs an *objective function* i.e. the function that is actually minimized to reach our goal. In our case we want to find the discount curve which minimize the sum of the squared NPVs (x will be our result aka the list of *best* discount factors*):

```
In [12]: def objective_function(x):

             curve = DiscountCurve(
                 # today date
                 ois_data.observation_date,

                 # pillar dates
                 pillar_dates,

                 # pillar discount factors
                 x
             )

             sum_sq = 0.0

             for swap in swaps:
                 sum_sq += swap.npv(curve) ** 2

             return sum_sq
```

To optimize our $\vec{x}$ we can use the `minimize` algorithm defined in `scipy.optimize`.

```
In [13]: from scipy.optimize import minimize

         # initialize to 1 the x vector (random choice)
         x0 = [1.0 for i in range(n_df_vector)]

         # set wide constraints on the discount factors
         # in the minimization problem the value of each x_i
         # will be bound between these limits
         bounds = [(0.01, 100.0) for i in range(n_df_vector)]

         # in addition we have an additional constraint:
         # we want the first pillar to be 1 (fixed)
         # (because it has pillar date = today)
         bounds[0] = (1.0, 1.0)

         # finally we run the minimization
         result = minimize(objective_function, x0, bounds=bounds)

In [14]: # print the diagnostic of the minimization problem
         result

Out[14]:       fun: 0.000737067806478276
          hess_inv: <34x34 LbfgsInvHessProduct with dtype=float64>
               jac: array([ 6.40060919e+05, -4.10141278e+01, -1.97120762e+01,  5.02186770e+00,
                 3.16953828e+01,  6.01053281e+01,  9.22468259e+01,  1.25056055e+02,
                 1.61067035e+02,  1.97872773e+02,  2.37837001e+02,  2.78325327e+02,
                -9.76591575e+02, -3.98919050e+02, -3.70947586e+02, -3.33820367e+02,
```

```
       -3.09413767e+02,  1.33740400e+02,  5.96171630e+02,  9.90164849e+02,
        1.18880634e+03,  1.09651231e+03,  7.00761702e+02,  8.12483823e+01,
       -6.55803166e+02, -1.36412700e+03, -1.91306081e+02,  1.92698529e+03,
        5.34498413e+02, -1.56729815e+02, -5.39276839e+02, -2.47024084e+03,
       -1.83758059e+02,  1.89609719e+03])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 840
      nit: 12
   status: 0
  success: True
        x: array([1.        , 1.00030147, 1.00058831, 1.00089012, 1.00118751,
       1.00147996, 1.00178743, 1.00208107, 1.00238467, 1.00267865,
       1.00298261, 1.00327737, 1.00357104, 1.00446948, 1.00531932,
       1.00615237, 1.00694015, 1.00907035, 1.00931983, 1.00710503,
       1.00189947, 0.99379259, 0.98332953, 0.9710012 , 0.95721995,
       0.94267437, 0.92772436, 0.88312793, 0.81780916, 0.7655247 ,
       0.71986347, 0.64350408, 0.5928013 , 0.5454718 ])
```

In [17]:
```
# objective function value with starting point parameters
objective_function(x0)
```

Out[17]:
```
1055914177798.2069
```

In [18]:
```
# objective function value with final values
objective_function(result.x)
```

Out[18]:
```
0.000737067806478276
```

In [21]:
```
# define the discount curve object using the
# resulting discount factors (result.x)
curve = DiscountCurve(ois_data.observation_date, pillar_dates, result.x)

from datetime import date
curve.df(date(2059, 11, 23))
```

Out[21]:
```
0.643157206728785
```

In [23]:
```
# 50 years rate
import math
-math.log(curve.df(date(2059, 11, 23))) / 50
```

Out[23]:
```
0.00882732190315717
```

In [26]:
```
list(result.x)
```

Out[26]:
```
[1.0,
 1.0003014747310168,
 1.000588313127077,
 1.0008901199536386,
```

```
           1.0011875146572378,
           1.0014799598670674,
           1.0017874342847497,
           1.0020810669287863,
           1.002384668218312,
           1.0026786511290502,
           1.0029826146845602,
           1.003277367891063,
           1.0035710351042981,
           1.0044694772149225,
           1.0053193209371638,
           1.006152366379903,
           1.0069401522566988,
           1.009070346753292,
           1.0093198286477054,
           1.0071050311457177,
           1.0018994747802303,
           0.9937925927976946,
           0.9833295330059759,
           0.9710012014584769,
           0.9572199541886813,
           0.9426743720454831,
           0.9277243626192413,
           0.8831279292517638,
           0.8178091614623915,
           0.7655246963531933,
           0.719863473051108,
           0.6435040837229044,
           0.592801300550168,
           0.545471799496346]
```

```
In [27]: pillar_dates
```

```
Out[27]: [datetime.date(2019, 10, 30),
          datetime.date(2019, 11, 30),
          datetime.date(2019, 12, 30),
          datetime.date(2020, 1, 30),
          datetime.date(2020, 2, 29),
          datetime.date(2020, 3, 30),
          datetime.date(2020, 4, 30),
          datetime.date(2020, 5, 30),
          datetime.date(2020, 6, 30),
          datetime.date(2020, 7, 30),
          datetime.date(2020, 8, 30),
          datetime.date(2020, 9, 30),
          datetime.date(2020, 10, 30),
          datetime.date(2021, 1, 30),
          datetime.date(2021, 4, 30),
```

```
datetime.date(2021, 7, 30),
datetime.date(2021, 10, 30),
datetime.date(2022, 10, 30),
datetime.date(2023, 10, 30),
datetime.date(2024, 10, 30),
datetime.date(2025, 10, 30),
datetime.date(2026, 10, 30),
datetime.date(2027, 10, 30),
datetime.date(2028, 10, 30),
datetime.date(2029, 10, 30),
datetime.date(2030, 10, 30),
datetime.date(2031, 10, 30),
datetime.date(2034, 10, 30),
datetime.date(2039, 10, 30),
datetime.date(2044, 10, 30),
datetime.date(2049, 10, 30),
datetime.date(2059, 10, 30),
datetime.date(2069, 10, 30),
datetime.date(2079, 10, 30)]
```