Bayesian Principal Component Analysis

Creating One Combined DataFrame

We have 2 spreadsheets of spot yields from the Bank of England website that we will load into dataframes

Download GLC Nominal month end data_1970 to 2015.xlsx
Download GLC Nominal Month End Data (2016 to Present)

A summary of the process

- Load BoE data into one dataframe
- Truncate the data so that continous block of data available for calibration
- Interpolate the data so that continous block of data available for calibration
- Remove negative values and interpolate between remaingin values
-
- Take Logarithms
- Difference the data
- De-mean the data
- Calculate co-variance matrix
-

Basic Reasonableness Tests

We perform a couple of reasonableness checks to ensure the spreadsheet data has loaded correctly into the combined dataframe

A Check on the Number of Rows

Dataframe 1 - 1970 to 2015

the first date is 1970-01-31 and the last is 2015-12-31
one would therefore expect 12 x 46yrs = 552 entries
and indeed we see the number of rows in df is 552

Dataframe 2 - 2015 to present

the first dates is 2016-01-31 and the last is 2024-12-31
one would therefore expect 12 x 9yrs = 108 entries
and indeed we see the number of rows in df is 108

Combined DataFrame

The length of combined dataframe is 660 rows"
whereas the two separate dataframes come to 552 + 108

A Check on Sum of Values

Dataframe 1 - 1970 to 2015

manual inspection of the sum of all values in first spreadsheet is 191503.172322029
the sum of 1st dataframe is also 191503.17232202887

Dataframe 2 - 2015 to present

manual inspection of the sum of all values in second spreadsheet is 17844.9993308767
the sum of 1st dataframe is also 17844.999330876683

Combined DataFrame

the sum of combined dataframe is 209348.17165290558
and the sum of the manually observed 191503.172322029 + 17844.9993308767 = 209348.1716529057

Truncation & Interpolation of the Dataset

Principal component analysis requires same number of datapoints for each term so as to produce a rectangular matrix from which covariances can be calculated.

The dataset of spot yields contains gaps insofar that the whole set of observation dates is not consistently available for all terms. We want to choose a range of observation dates and terms that reduces the need to fill in gaps in the dataset.

We have spot yield data for terms 0.5 up to 40. The first step to identify a calibration dataset is to identify the first and last data point for each term. This gives us an initial idea of the size of the dataset available.

We make a judgement call about which terms to retain (and observation dates) to retain. If there are gaps in the data we use linear interpolation to fill them.

Matplotlib figures, subplots, axes

- Figure The whole canvas or image
- Axes One chart (with x/y axes, labels, data)
- Subplot One chart within a grid layout (i.e., an Axes)
- Grid of subplots Arrangement of multiple Axes in a Figure

Data Boundaries by Term

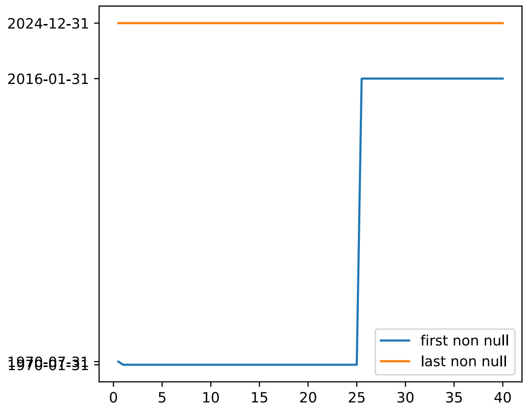
visual

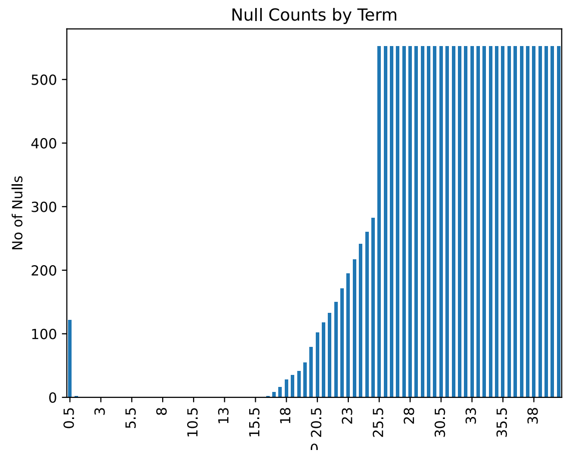
The maiximum range of observation dates for each term is found by the earliest and latest non NaN entry. We see that for beyond term 25 data is only available from 31st January 2016 and that for earlier terms available from 31st January 1970 (with an exception for term 0.5).

Null Counts

Histogram

An initial inspection of the data shows signficantly more nulls for greater terms. Beyond term 25 we see the number levels off and we later discover this is because data for term 25 onwards doesn't begin until 2016 meaning there is a significant block of NaN values from 1970 to 2016 for these terms.





tabular

start_term	end_term	earliest_date	last_date
0.5	0.5	1970-07-31	2024-12-31
1.0	25.0	1970-01-31	2024-12-31
25.5	40.0	2016-01-31	2024-12-31

Tabulated

We identify non contiguous blocks of data by determining the expected number of data points, based on first and last data point, and comparing with actual number of data points.

These are the columns which will be interpolated.

term	actual no. data-points	expected no. data-points	missing data points
0.5	538	654	116
1	658	660	2
16.5	658	660	2
17	652	660	8
17.5	644	660	16
18	632	660	28
18.5	625	660	35
19	619	660	41
19.5	605	660	55
20	581	660	79
20.5	558	660	102
21	542	660	118
21.5	527	660	133

Interpolation

Summary statistics on interpolated/truncated dataset

- term
- actual data points

Rows untouched by interpolation should have same total as before. totals for those with interpolation should could be checked for reasonableness.

Decisions

- Data for terms greater than 25 isn't available before 2016. We will therefore not model beyond term 25 in order to facilitate sufficient history of data-points.
- we ignore term 0.5 and start at term 1 due to missing datapoints for term 0.5

- we ignore terms beyond 20 since the proportion of missing datapoints is too great.
- we replace -ve values with NaN and then interpolate

term	actual no. data-points	expected no. data-points	missing data points
22	510	660	150
22.5	489	660	171
23	465	660	195
23.5	443	660	217
24	419	660	241
24.5	400	660	260
25	378	660	282

Removing Negatives

Logarithms are only defined for positive arguments. We therefore need to consider the small number of -ve values observable in the dataset:

	1	1.5	2	2.5	3	3.5	4	4.5	
563	-0.029262	-0.001047	0.043756	0.109305	0.189006	0.276475	0.367344	0.458905	0.5495
604	0.013280	-0.008485	-0.026463	-0.038360	-0.045366	-0.048285	-0.047483	-0.043166	-0.0354
605	-0.009801	-0.045821	-0.069679	-0.082474	-0.087813	-0.087702	-0.083129	-0.074636	-0.0625
606	-0.021850	-0.060837	-0.093727	-0.117590	-0.132942	-0.140738	-0.141903	-0.137297	-0.1276
607	-0.044205	-0.060842	-0.072683	-0.077088	-0.074019	-0.064180	-0.048489	-0.027846	-0.0030
608	-0.013335	-0.031917	-0.053845	-0.071520	-0.082226	-0.085494	-0.081745	-0.071701	-0.0561
609	-0.026419	-0.038957	-0.053194	-0.063327	-0.067327	-0.064971	-0.056681	-0.043049	-0.0247
610	-0.021041	-0.018181	-0.020028	-0.020669	-0.017662	-0.010179	0.001872	0.018243	0.0385
611	-0.150365	-0.143101	-0.135252	-0.128029	-0.120277	-0.110666	-0.098319	-0.082835	-0.0641
612	-0.113058	-0.102450	-0.095639	-0.087447	-0.075699	-0.059686	-0.039392	-0.015160	0.0125

For ease of analysis we set these values to NaN.

	1	1.5	2	2.5	3	3.5	4	4.5	5	5
563	NaN	NaN	0.043756	0.109305	0.189006	0.276475	0.367344	0.458905	0.549509	0.6381
604	0.01328	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
605	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
606	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
607	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0251
608	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
609	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
610	NaN	NaN	NaN	NaN	NaN	NaN	0.001872	0.018243	0.038508	0.0621
611	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
612	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.012502	0.0430

We now populate this values with interpolated values moving down the columns (terms)

	1	1.5	2	2.5	3	3.5	4	4.5	5
563	0.050886	0.090516	0.043756	0.109305	0.189006	0.276475	0.367344	0.458905	0.549509
604	0.013280	0.041635	0.040681	0.044429	0.051206	0.059899	0.039947	0.048182	0.058168
605	0.012128	0.042606	0.046637	0.054873	0.066015	0.079125	0.033601	0.043192	0.054891
606	0.010976	0.043578	0.052592	0.065317	0.080824	0.098351	0.027255	0.038202	0.051615
607	0.009824	0.044549	0.058548	0.075760	0.095634	0.117576	0.020909	0.033212	0.048338
608	0.008672	0.045521	0.064504	0.086204	0.110443	0.136802	0.014564	0.028222	0.045061
609	0.007521	0.046493	0.070459	0.096648	0.125253	0.156028	0.008218	0.023233	0.041784
610	0.006369	0.047464	0.076415	0.107092	0.140062	0.175253	0.001872	0.018243	0.038508
611	0.005217	0.048436	0.082371	0.117535	0.154871	0.194479	0.095777	0.124159	0.025505
612	0.004065	0.049407	0.088326	0.127979	0.169681	0.213705	0.189681	0.230075	0.012502

checks we can perform on the interpolated values .....

Taking Logarithmns

Purpose

We want to calculate the natural log of spot yield returs. 41508.92158837641

.apply() function

This is the second column. Same flexibility as the first.

further complications with dtype:object

sometimes pandas is treating values as generic python objects not efficient numeric types even if they look like floats

it seems to happen when slicing rows.

a fix is to use .astype(float) before applying functions like np.log

Checking the Log Calculation

given that:
$$ \sum_i \log(x_i) = \log\left(\prod_i x_i \right) $$

we can perform a check on the log calculation. however the product approach doesn't work since there are so many values we get overflow for the product side of the equation we can instead chunk up the calculation to make it more manageable we therefore calculate the product for each row then take the log the sum the log of products for each row

The Product of All Entries

the product of each row	the log of each row product	the sum of log of row products
1.500157e+36	83.298633	41508.921588
4.498612e+35	82.094247	41508.921588
1.520994e+35	81.009842	41508.921588
7.778928e+35	82.641897	41508.921588
2.583242e+35	81.539523	41508.921588

Comparing Calculations

The sum of individual 'logged values is: 41508.92158837642. The sum of the log of row product generates: 41508.92158837641. The difference between the two is 7.275957614183426e-12.

Differencing Data

We calculate differences since we are modelling changes in the yield curve:

Checks that can be made to ensure data has been differenced correctly:

- spot check a small sample of values
- total of differences = sum of first row - sum of last row

Logged Values

	1	1.500000	2	2.500000	3	3.500000	4	4.500000	5	5.
0	2.155865	2.164177	2.163407	2.159182	2.153934	2.148557	2.143398	2.138608	2.134239	2.
1	2.129794	2.127907	2.124743	2.120779	2.116447	2.112078	2.107884	2.103977	2.100422	2.
2	2.046943	2.051911	2.053485	2.053239	2.052194	2.050950	2.049814	2.048926	2.048347	2.
3	2.029005	2.062340	2.076126	2.079747	2.078543	2.075374	2.071704	2.068308	2.065603	2.
4	2.000277	2.045864	2.062064	2.064012	2.059325	2.051845	2.043562	2.035601	2.028615	2.

Differenced Values

	1	1.500000	2	2.500000	3	3.500000	4	4.500000	
0	nan	nan	nan	nan	nan	nan	nan	nan	n
1	-0.026071	-0.036270	-0.038663	-0.038403	-0.037487	-0.036478	-0.035515	-0.034631	-0.0338
2	-0.082851	-0.075995	-0.071259	-0.067540	-0.064253	-0.061128	-0.058069	-0.055051	-0.0520
3	-0.017938	0.010429	0.022642	0.026507	0.026349	0.024423	0.021889	0.019381	0.0172
4	-0.028727	-0.016476	-0.014062	-0.015735	-0.019218	-0.023528	-0.028141	-0.032706	-0.0369

```
/tmp/nix-shell-3925-0/ipykernel_221689/3217399495.py:1: PerformanceWarning: Adding/
subtracting object-dtype array to DatetimeArray not vectorized.
  df_demeaned = df - df.mean()
```

De-meaning Data

Co-variance Matrix