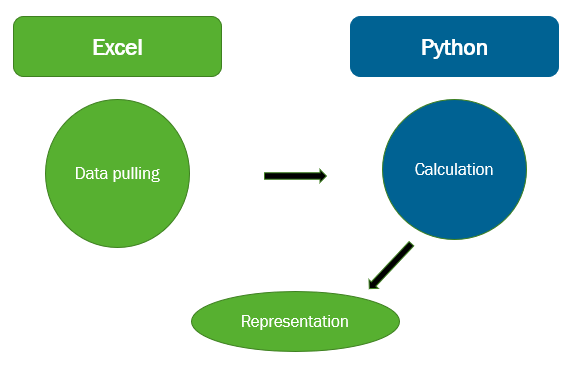
**“Peer” group analysis – Function description**

Summary:

This is the outcome on the real-time trading analysis on the European and American stocks. We started with business peers and hoped to generate a synthetic index from that. The main finding of this project is that a good index seems to be constructed out of stocks, which aren’t necessary the direct peers. The method used here looks for a basket of stocks, which are mostly suitable to describe the price of a stock of interest over the course of a day. (Stocks with similar features).

The analysis is done in python. To reach a larger audience, we combine it with Excel in the following manner.



The data will be pulled from excel using VBA (DBH-functions) and passed on to Python for simulation and calculation. The results will be passed back to Excel for representation.

**Installation:**

One needs to have the following programmes installed:

* Anaconda with python 3.7+  
  A distribution of softwares to support python and R users.
* Any python editor (Spyder incl. in Anaconda)
* Pip (should be incl. in Anaconda)  
  Operates in Anaconda prompt. It is useful to install all the libraries.
* Excel Macro (“xlwings”)  
  The file can be downloaded here: <https://docs.xlwings.org/en/stable/addin.html>

(There is a description of how to install it on Excel)

Then you will need to specify your interpreter in Excel. Go to xlwings-tab, the first empty space to the left with “Interpreter”: C:\Users\x.cheng\AppData\Local\Continuum\anaconda3\pythonw.exe

(This is the python execution file-path in your main anaconda folder)

Normally, one needs all the common libraries, including

* *Numpy*: numerical math library
* *Pandas*: dataframe library
* *Sklearn*: machine learning library
* *Scipy*: scientific statistical library
* *Xlwings*: Excel and Python linking library

To install all the libraries, I would just run the excel sheet. It will pop up an error message saying something is missing or not defined. Install that particular one by one.

**IMPORTANT**: when installing these, always use pip to install in the command line.   
For example:   
*pip install numpy*

If an error pops up, do: *pip install numpy –user*

Upgrade it straight afterwards  
*pip install --upgrade numpy*

If error: *pip install --upgrade numpy --user*

**NEVER** *conda*, it confuses anaconda. **This almost broke my laptop.**

**Schematic of the LassoExcel sheet:**



Green is Excel/VBA. Blue is python. <…> are all vba codes. Some of them triggers the python script. The codes are all global functions. Object orientated programming could be more efficient though.

**VBA:**

VBA triggers python by running “RunPython”-function of xlwings library (see VBA environment.)  
It is an one-liner. As the input is a string, it is quite hard to call a function in python, which takes a argument. This will be useful if we get it working.

According to xlwings’ helpdesk, “RunPython” can be made such way that it takes an argument but not convenient. One needs to combine strings including the “ ‘ ”.  
Also, RunPython doesn’t return values to pass on. To do this it is suggested to use User Defined Functions (UDFs).

**Data consumption issue:**

Bloomberg restricts a user’s daily data usage to be 500 000 hits and monthly cap will be calculated using an algorithmic weighting system, which depends on the fields. There are no further information on how they calculate that. According to the agent, pulling past history data only counts as 1 hit, if it is applied to one field. (see Bloomberg chat reference: H#1066714325)

My understanding:   
This seems to make sense. As Bloomberg doesn’t own any raw data, they are not allowed to charge us for that, but only for the service. Hence, they charge for the number of times of entering the function “DBH” or so on, which is essentially their service. This is also why they introduced the daily/monthly cap to limit the maximal number of times for ppl to access their server. Any processed data by Bloomberg seems to require the usage of their server, therefore the weird weighting system.

This means we are good with the data consumption. A rough estimate for our daily data usage is given below with the following assumptions:

* three markets
* 500 universe peers in each market (more than we actually need)
* One security type per peer
* DBH function for each peer will be executed twice (once for the main training data and once for the extra minutes)
* 20 equities to be traded per market per day
* 3 Lasso parameters for each trade equity
* 50 selected stocks for each lasso setting
* Refresh the live data of the each selected stock 30 times per day.

This makes 3 x (500 x 1 x 2 + 20 x 3 x 50 x 30) = 273 000 hits.  
This turns out to be quite large, but we are under daily cap. I believe the grey marked items are all over-estimated.

**Below: all VBA function descriptions:**

<*UpdateUniverse*>:

It combines “Company universe” and the “additional tickers” and shows them in a new list.  
This will be used as the overall list for the universe.

<*LoadData*>:

It pulls all the Data from the “All ticker” list using the parameters specified in the grey box in the “Main”-tab (BDH-function). The raw data is saved in “RawData\_Train” – tab.

Only one data type (Open) is pulled for now and saved in the following format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| BarTp | Trade |  | BarTp | Trade |  | BarTp | … |
|  |  |  |  |  |  |  |  |
| Equity 1 |  |  | Equity 2 |  |  | Equity 3 | … |
| Dates | Open |  | Dates | Open |  | Dates | … |
| Xx/xx/20xx | yy.yyyy |  | Xx/xx/20xx | yy.yyyy |  | Xx/xx/20xx | … |
| … |  |  | … |  |  | … | … |

As the data cleaning is excuded in python, to include more data type, we need to change python codes.

<*LoadMinData*>:

It is the exact function of <*LoadData*>, just with different parameter in the green box in “Main”-tab.

It pulls a second data set (the first few min of data of the day of interest) and save in “RawData\_Train2”-Tab.

<*ShowPeers*>:

Calls a particular function (“*GetPeerParameters*”) in the python file (“*mymodule*.py”).   
It out puts the Lasso-selected peers and the parameters and the normalized contributions. These are then shown in the “main”-tab.

The training results, the output of the “*GetPeerParameters*” in python, will be saved in “Result\_Train”—tab in the following format.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dates | Data | Basket | vsBasket | MSE |
| Xx/xx/20xx | x | y | y-x | (y-x)^2 |
| … | … | … | … | … |

Data: Equity data of interest  
Basket: Lasso output, which is optimised against Data  
vsBasket: Error of the model  
MSE (mean squared error): vsBasket squared.

The Averaged model error in the plot is calculated using the averaged MSE over the entire list and then square rooted.

The data range for the training plot is set to be 2 days. If this is setup dynamically, that would be better.

<*LoadLiveData*>:

Pulls ONLY today’s data of the peers, which are selected by < *ShowPeers* >. As we need to match the dimensions of the predicting data set with the traning data set, (due to LASSO function in python (SciKit-learn)), I set the values of all other unselected peer to 0. Doing so, we avoid downloading unessesary data from Bloomberg. The output will be saved in the similar matter as <*LoadData*> in the “DataValid”-tab.

In addition:

On the “Main”-tab, there is a “History?:” option. This can be used to look at the past history data.  
By typing:

* Yes: it uses only the date given in the green area in the “main”-tab and pull the data over the entire day. The time range is set to be the start and end times in the grey box in the “main”-tab.
* No: It pulls up-to-date data (live) starting from the “Time to be included” in the green box in “main”-tab.

<*ShowPred*>:

Calls the “*showLivePrediction*” function in the python file and saves everything in the “Result”-tab in the same manner as the <*showPeers*>.

**Python:**

VBA mainly calls two python functions: *GetPeerParameters* and *showLivePrediction.* In the following, I will walk through them. Using objected programming, one could redo this function to have easier access to other quantities.

*GetPeerParameters:*

The function acquires parameters from the “Main”-tab and cleans the data in the “RawData\_Train” and “RawData\_Train2”-tabs into a single data frame. These are fed into the lasso training.

It takes 2 inputs: 1: Ticker name in string, 2: alpha tuning parameter for Lasso.

The function outputs a list of 2.

1. Dataframe of peers, including the coefficients from the lasso and the contributions.  
   Listed in the “Main”-tab (Columns: L-N)
2. Dataframe of the training results, including Target Data, Basket value, vsBasket values and mean squred error.

Listed in the “Result\_Train”-tab.

The following graph shows the relationships between all used subfunction.



* *prepDataSet:*The function mainly loads the main set of the history data from “RawData\_Train”-tab and the smaller section of the to-be-included-min data in “RawData\_Train2”-tab. It cleans and fills the data and outputs a single dataframe. This is the overall training dataset.  
  The output has the form:

|  |  |  |  |
| --- | --- | --- | --- |
| Dates | Equity 1 | Equity 2 | … |
| Xx/xx/20xx | x | y | … |
| … | … | … | … |

This function collects the parameters from the “Main”-tabs and feeds these into the sub functions:  
*GetDataFrame, filterData*

* *GetDataFrame(sheetName, N, M):*This function converts a particular Excel sheet into a dataframe and condenses the dataframe by deleting the empty columns and multiple double time series.  
  Inputs:
  + *sheetName* is a string. The name of the sheet in Excel.
  + *N, M* are numbers. These defines the dimension of the sheet, which we want to convert into dataframe.

Output is a single dataframe as shown in *prepDataSet.* It also includes the times when the market was closed.

* *filterData* *(dataDF, startDate, endDate, startTime, endTime):*

This function selects only the data when the market is open. A If loop in the function considers the case if multiple days of data are pulled from Bloomberg. Data of each day are saved into individual dataframes, which are then combined together into a single one.

Inputs:

* + *dataDF* is a single raw dataframe without any selection on the times
  + all the rest are parameters acquired from the “Main”-tab and are all strings. (Format: “yyyy-mm-dd” and “hh:mm:ss”)

Output is also a single dataframe, running only within the market hours.

* *getCoefDF(data, avg, nrTrain, stockname, alph)*:  
  This function prepares everything in returns and splits the data into training variables and the target data. After applying Lasso, the function creates 2 dataframes as outputs:

1. coefDF: is the dataframe for the lasso coefficients and contributions to the target stock.   
   To calculate the contributions, I assume the summed squared of the coefficients is normalized to unity. This is performed in *getContr(peerDFdata), where* peerDFdata is the coefDF.
2. trainResDF is the dataframe for the target stock prices, basket trained prices, vsbasket and MSE.

Those are the training outcome. We use the target as a reference, which determines the coefficients.   
Input:

* + data: cleaned dataframe within the market opening hours.
  + Avg is a number. This gives the option of average the training dataset. (applicable only in Jupyter notebooks.)
  + nrTrain is a number. This gives the length of dataframe, which will be used for model training.
  + Stockname is a string. The ticker name of the company of interest.
  + Alph is a number, which set the tuning of Lasso.

Reason for converting into returns:  
Generally speaking, a fit represents the data the best, if the range of the variables in the prediction is the same range during the training. When the variable range in the prediction exceeds the training range, the prediction will not be good, because it is not defined for this.   
In our case, if we use absolute prices, the prediction will be worse if the price values exceed the history data range, which is mostly the case, as the price increase. By converting everything into returns, we rescale the data back into the defined range. Hence, I think it is important to convert into returns at the first and then apply the Lasso.

* *getLassoFit(data, nrTrain, stockname, alph):*

This function converts the data into returns. It sets the condition for Lasso and applies the fit and returns the fit function.  
The inputs are as described above.

* *prepTrainData(data, nrTrain, stockname):*

This separates the data set into training variables and targetset and outputs in a list of 2.

*showLivePrediction*

This function is very similar to the getPeerParameter. Hence when executing this, we run a lot code twice. Quite a lot of improvement can be done here. The function combine the predition dataframe and the training dataframe, into a single one. It recalculates the coefficience and applies it predict the stock price of interest. The output of the function is only one Dataframe saved in the “Results”-Tab.

The structure is shown below.



* createNewDF(trainDF):

This function combines the single cleaned training dataframe with the “DataValid”-tab, which is processed the same way. It outputs also a single DF.

* getLassoValidDF(data, nrTrain, stockname, alph):

This converts the data into returns and recalculates the coefficients and applies it to the prediction data.

Potential improvement:

1. <*LoadData*> and <*LoadMinData*> could be combined together and saved externally for future usage.
2. Data acquired by <*LoadLiveData*> could be saved externally for reusage in the future. When one wants to update the prediction, we don’t need to download the same data again. At the end of the day, this complete set can be appended into the main data (along with <*LoadData*> and <*LoadMinData*>).
3. <*ShowPred*> and <*ShowPeers*> are the most time consuming, as they actually have done the same calculations twice.
   1. One way to avoid this, is to use Excel and the coefficients, which are output by <show peers>. We can then only plot the prediction.
   2. Other way (I prefer this), to lower the tolerance of the simulation for <*showPeers*> and speed up the calculations. If one is happy with the peers, we can than increase the tolerance/ precision in <*ShowPred*>.
4. Due to the conversion into return, the prediction always starts of at 0 bps. This means that the start of the history data will contribute more to the prediction, especially around the start of the prediction, than anywhere else. This means that we use morning to predict mornings. It would be interesting to see if this is actually a good measure. In particular, when we use multiple days of data to predict, we effectively overweight the start of the history data. We need to look more into this!