Python for finance and optimization:

Classical tasks of Quantitative Investment Strategies.

The goal of this project is in two parts : (i) show how to deal with datasets and carry out some data visualization tasks with different libraries, and (ii) show how to build simple financial strategies, backtest them and assess their performance.

We shall use classical Python libraries: **pandas**, **numpy**, **matplotlib** and **seaborn**.

Part I

From a raw dataset to a meaningful one

- Download the Excel file 'sbf120_as_of_end_2018.xlsx' that contains prices of SBF 120 components (as of end 2018) over the period 2011-Sept 2021.
- Discover what is in the file using **pd.read_excel**. We can use **type()** to know if there are several sheets or only one. In our case **type()** returns **dict** which means there are several sheets and we use **len()** to know the number of sheets.

(It's always important to open the file in Excel to understand how the data is organized. For example, this file contains three columns for each asset: one for the asset value, one for its market cap, and one for the date of quotation. However, the file could have been structured differently—for instance, with just three general columns where asset values are listed sequentially along with their corresponding market caps and dates.)

- Use the **pd.to_datetime** function of pandas to obtain a dataframe indexed by dates, with tickers as columns and prices as values.
- Before dealing with visualization, we have to deal with the missing values. We choose not to replace the initial Nans and final ones. However, those appearing after a first value is known or before the last value, we want to replace them with the next available value.

Let us deal with visualization now:

- Using the dataset, we draw (using matplotlib) the following graph (Figure 1) that represents the prices of BNP and Société Générale from January 2019 (with two axes). We tried to annotate on a proper way the following graph.

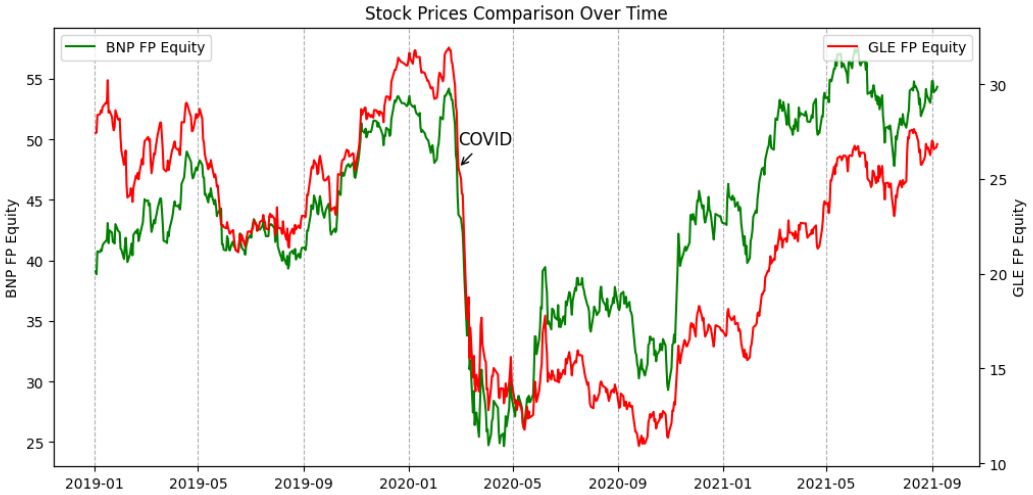

Figure 1: Prices of BNP FP & GLE FP

- Now we try to introduce two new analysis graphs thanks to Seaborn. We want a One-dimensional analysis of daily price returns and a two dimensional analysis of daily price returns.

  (Librairies used for this part are not detailed in the appendix because it is too specific. The librairy of Seaborn you use really depends on what type of analysis you want to do, and what type of data you are working on. Further info on : https://seaborn.pydata.org/).

- FYI : We used **sns.violinplot()** for this graph.

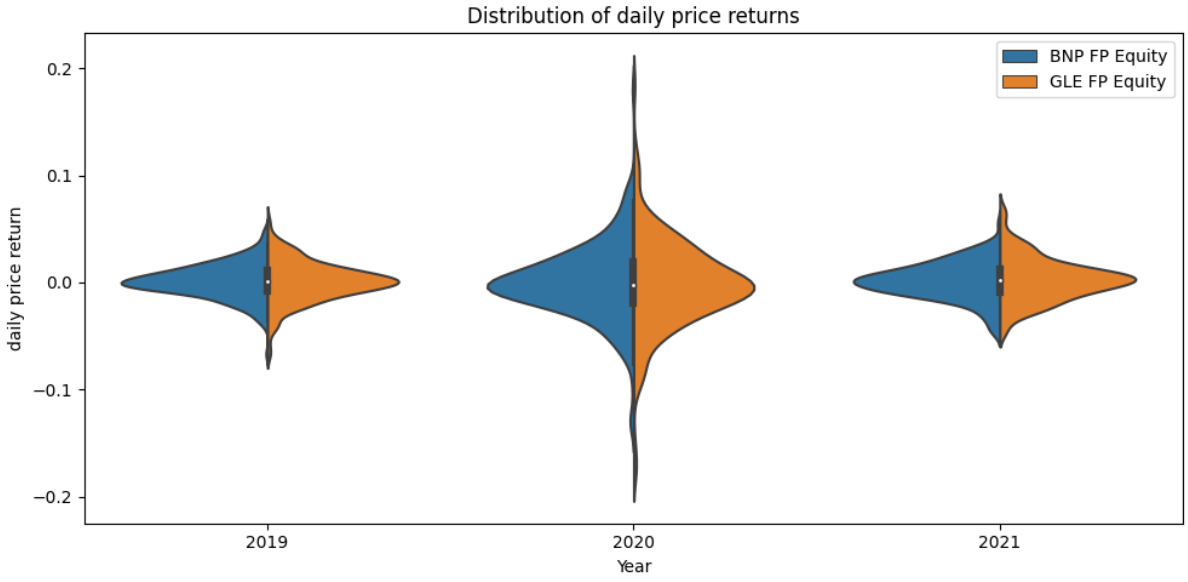

Figure 2: One-dimensional analysis

- This graph illustrates the fact that during a stress period the distribution of returns is clearly modified. Supposed that daily returns follow a normal distribution we can observe a modification of the Kurtosis of the distribution. The year 2020 was shocked by the pandemic and led to significant slumps and rises that can be observe on this graph and be confirmed by the following one.
- FYI : We used **sns.joinplot()** for this graph.

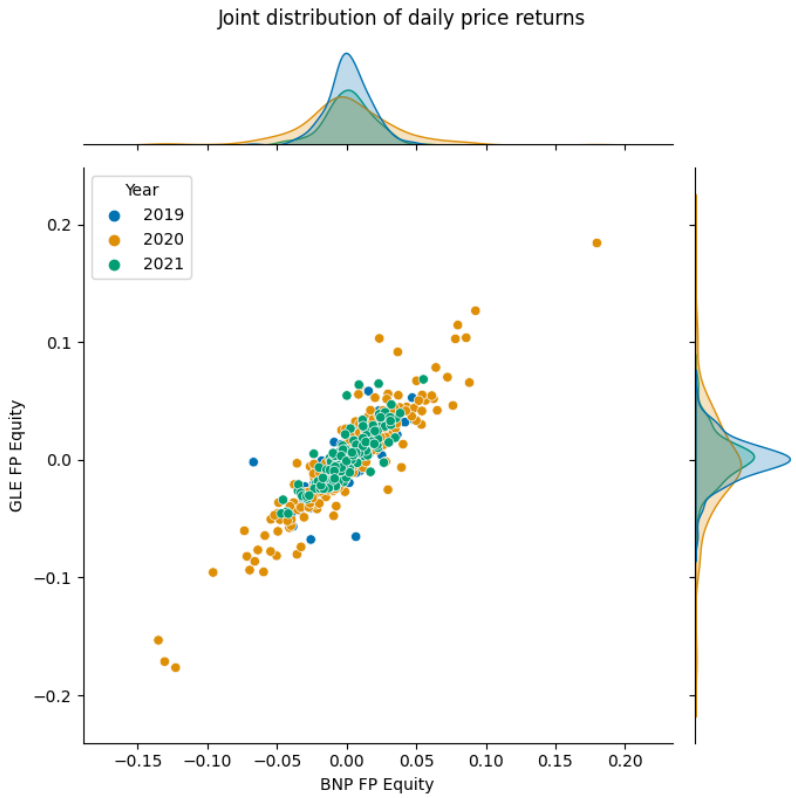

Figure 3: Two-dimensional analysis

****

Part II

A class for analysing strategies

- Firstly, what we do is the same as in the first part but without explanation. We clean the dataframe in order to have the daily returns of the 120 stocks over the period 2019-2020.
- We built a procedure to obtain a dataframe indexed by dates, with tickers column's headers, market capitalization and daily returns as values.
- Now we build a Class in order to analyse strategies; this class Strategy is built with these following characteristics:

– a constructor taking a string (for the name of the strategy) and a Series of returns as inputs. *(Be careful, returns has to be a 1-Dimension vector and not a dataframe with multiple strategy returns)*

– methods computing annual volatility, Sharpe ratio and maximum drawdown (MDD).

– a method to illustrate graphically the strategy (the way you like).

Reminder : If $(S_n)_{0\leq n\leq N}$ *is the sequence of values of a strategy over a given period, the maximum drawdown is defined as :*

$$MDD = -\min_{0\leq n\leq N}\min_{m\geq n}\frac{S_m - S_n}{S_n}$$

In order to report a clean plot, method Max_Drawdown returns a dictionary. This one is composed of three values: the value of the MDD and the index of the start date and end date of the MDD. This was necessary to plot where it begins and finish on the graph.

We test the Strategy class with a long only strategy on the 'GLE FP EQUITY' stock.

Figure 4: Long only strategy on GLE Equity stock.

****

Part III

How to build a classical strategy

Once these tools are built, we are fully ready to design strategies and try to backtest them in order to see their efficiency.

- The first strategy we can build is capitalization-weighted strategy, also known as a market-cap-weighted strategy. This classical investment strategy is an approach where each stock in a portfolio or index is weighted according to its market capitalization.

$$Market\ Cap = Stock\ Price\ \times Number\ of\ Outstanding\ Share$$

  It represents the total market value of a company's equity. In a capitalization-weighted index, larger companies have more influence on the overall performance because they have higher market caps, smaller companies have less impact on the portfolio.

  Knowing that we have the market price and the capitalization of every stocks of SBF120 over more than ten years. We will build a Class CapiWeighted that takes as input SBF120's dataframe and returns the performance of the portfolio balanced with the different market caps.

  In order to perform this, Class will daily compute the Total Market Cap:

$$TotalMarketCap = \sum_{j=1}^{n} P_j \cdot N_j$$

  Where:

- $P_i = \ price\ of\ asset\ i$
- $N_i = \ number\ of\ outstanding\ shares\ of\ asset\ i$ *(In our case, we directly have the Market Cap of each asset)*

And we have to compute the allocation in each and every asset:

$$\alpha_i = \frac{P_i * N_i}{\sum_{j=1}^{n} P_j * N_j}$$

Where:

- $\alpha_i \epsilon\ [0,1] is\ the\ proportion\ of\ total\ portfolio\ invested\ in\ asset\ i$
- $\sum_{i=1}^{n} \alpha_i = 1$

Then, assuming that each asset i has a return r_i, the total portfolio return R_p is :

$$R_p = \sum_{i=1}^{n} \alpha_i * r_i$$

(#Coding trick#: In our coding illustration, this formula above is computed with one specific line where we convert DataFrame into numpy arrays. This is because pandas is not able to multiply pairwise two DataFrame of the same size if they do not have same index or columns names).

And if the initial portfolio value is V_0, the value at time t, V_t is:

$$V_t = V_0 * (1 + R_p)$$

Hence, the objective of the following Class is to return V_t at each time t. For our example, we will perform the calculus over 2019-2020.

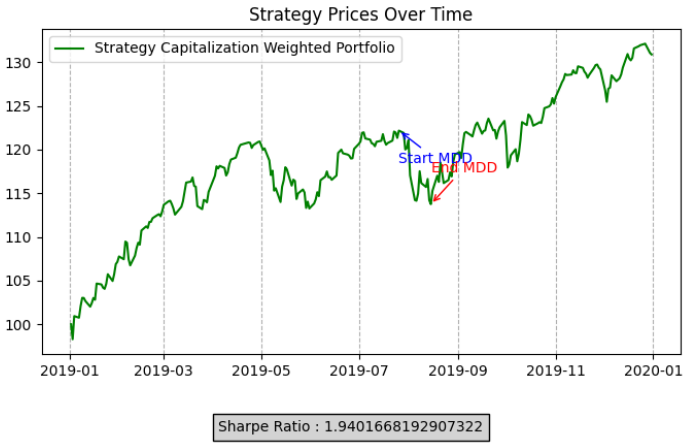

Figure 5: Capitalization Weighted strategy on SBF120.

****

Conclusion:

The goal of this project was to show how to deal with datasets and carry out some data visualization tasks with different libraries, and show how to build simple financial strategies, backtest them and assess their performance. These are one of the most common tasks required in Quantitative Investment

Strategies. We have to pinpoint the fact that a strategy could never be used if it has not been deeply backtested and challenged. Sharpe Ratio and Maximal Drawdown are simple risk indicators for a strategy but they are unmissable. The coding techniques employed in this project are major to get into market finance, Python functions are not so numerous but it is crucial to master them. I was intentionally evasive about plotting techniques because they really depend on what type of data you are dealing with. Furthermore, Matplotlib and Seaborn are so wide that it would require an entire class, for these reasons I did not want to get into Python technical explanations and go out of the finance main line.

In the next project, I will develop more quantitative strategies. It implies that I will go deeper in the math and will not explain the coding tricks anymore. If there are some Machine Learning stuff I could make a quick reminder on the topic.

Appendix:

Additional information on several functions used in this project that I felt relevant to explain.

DataFrame.iloc[,] : Accesses rows/columns by position (i.e., by integer index). It is like using numerical indices: df.iloc[0] returns the first row. This function returns: a Series (for single row/column) or a DataFrame (for multiple rows/columns).

DataFrame.loc[,] : Accesses rows/columns by label (i.e., by index/column name). We use it when we are selecting data by names rather than numbers; df.loc['2024-01-01'] returns the row with the index '2024-01-01'. This function returns: a Series (for single row/column) or a DataFrame (for multiple rows/columns).

DF1.join(DF2) : Combines two DataFrames using their index (like a SQL join). You can specify how='left', 'right', 'outer', or 'inner'.

Join Type	What it Does	Keeps Rows From	Fills Missing With
`inner`	Keeps only matching indexes	Both DataFrames	Drops non-matches
`outer`	Combines all indexes	Both DataFrames	NaN for missing data
`left`	All rows from left (caller)	Left DataFrame	NaN for missing in right
`right`	All rows from right	Right DataFrame	NaN for missing in left

pd.concat([DF1, DF2, …], axis= 0 or 1) : We use this for combining many DataFrames or Series along a given axis. We choose the axis: axis=0 (stack rows), axis=1 (stack columns). It aligns on index by default.

DataFrame.first_valid_index() : returns the label of the first row in a DataFrame (or Series) that has at least one non-NaN value.

DataFrame.last_valid_index() : returns the label of the last row in a DataFrame (or Series) that has at least one non-NaN value.

DataFrame.bfill() : (bfill for Backward fill) Fills missing (NaN) values by copying the next non-NaN value downward.

Ax.annotate() : in Matplotlib allows you to add text (and optionally an arrow) to a figure at a specific position. It's commonly used to highlight important data points on a plot.

Parameter	Description
`text`	The text you want to display
`xy=(x, y)`	The **coordinates of the point to annotate** (the tip of the arrow)
`xytext=(x_text, y_text)`	The **location where the text should appear**
`arrowprops`	*(Optional)* A dictionary that **defines the arrow** style pointing from the text to the point