Financial Big Data Project:
Implementation of volatility-based strategies

Caterina La Cava (358741), Léo Lagast (290246)

January 29, 2024

Abstract

This project concerns the implementation of different volatility strategies, namely a risk parity strategy, an inverse volatility strategy, and a minimum variance strategy. The data consists of 4 years of tick-by-tick transaction prices on 87 US equity large stocks. Clustering algorithms are used to classify assets into different sub-portfolios, with the objective to apply the trading strategies of above only on these sub-portfolios. The cumulative return of the final portfolio changes as time passes depending on the strategy, the clustering algorithm, and the estimated covariance matrix used in the implementation of the strategies. Louvain clustering and Bahc covariance matrix show the best performance for risk parity and inverse volatility strategies in terms of cumulative return, even outperforming the S&P500.

Introduction

Our objective is to implement known volatility strategies on some obtained sub-portfolio returns: the idea is to extract the data structure from the available data through clustering algorithms to classify assets into groups; weight cluster components with a simple strategy to obtain what we call sub-portfolios; aggregate sub-portfolios in a unique portfolio assigning optimal sub-portfolios weights with known volatility strategies. In the implementation of this idea, there are different decisions to make and each of them could affect the performance of the portfolio's cumulative return over time. Think for example to the choice of the clustering algorithm to use, the way of estimating returns covariance matrices, or to which strategy follow to decide allocation weights in each sub-portfolio. Even the first steps of data pre-processing are essential and sensitive in the pipeline: in fact and most importantly, one of the challenges of the project is to develop our idea above managing a very large quantity of data; thus, we must take decisions on which part of the data available to use, how to clean and treat them and how to solve data specific challenges that we may encounter.
The aim, hence, is to explore as much as we can how different algorithms, assumptions, and decisions could influence the performance of well-known trading strategies when adapting them for day-by-day trading.
We find the topic particularly interesting because it gives us the possibility to approach an allocation problem in different ways, trying combinations of decisions without knowing the "absolute correct" road to follow.

Data set

We obtained a total of 24.5 megabytes of financial data related to US equity transactions regarding 87 different assets. In particular, for each US-listed company available, we work with two kinds of files: bbo and trade files. For both types of files, we have at our disposal 5 years of data, from the beginning of 2004 to the end of 2008, divided into daily files. Daily files correspond to the trading days in each year and, in each of them, are collected intraday data about trade prices, trade volumes (in trade files) and ask prices, bid prices, ask volumes, and bid volumes (in bbo files).
Figure 1 illustrates the organization of the data folder before the pre-processing of the data.

```
/data
└── /raw
    └── /sp100_2004-8
        ├── /bbo
        │   ├── /companyA_bbo
        │   │   ├── day1.csv File 1
        │   │   ├── day2.csv File 2
        │   │   └── ...
        │   ├── /companyB_bbo
        │   │   ├── day1.csv File 1
        │   │   ├── day2.csv File 2
        │   │   └── ...
        │   └── ...
        └── /trade
            ├── /companyA_trade
            │   ├── day1.csv File 1
            │   ├── day2.csv File 2
            │   └── ...
            ├── /companyB_trade
            │   ├── day1.csv File 1
            │   ├── day2.csv File 2
            │   └── ...
            └── ...
```

Figure 1: Initial structure of the data folder.

Thus, inside the bbo and trade folders, we find in both 87 folders (one for each asset), that we will call asset folders.
To better work with all the available files, for each asset folder, we proceed as follows.

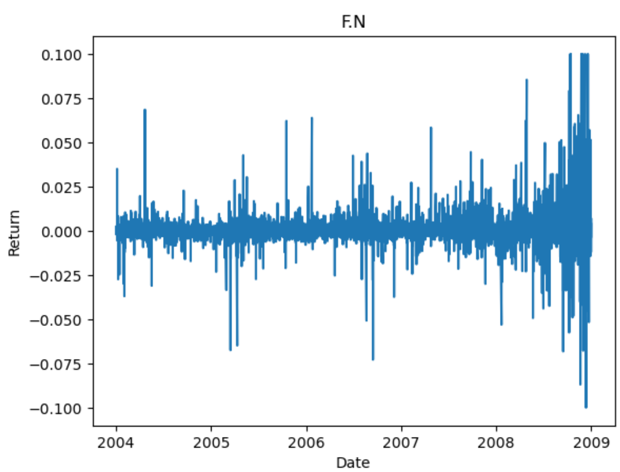
- Concatenate the daily trade files to obtain a unique data frame whose rows correspond to intraday transactions starting from 2004 and ending in 2008.
- Concatenate the daily bbo files to obtain a unique data frame whose rows correspond to intraday transactions starting from 2004 and ending in 2008.
- Merge the two data frames listed above by datetime.
- Replace the items "()" found in some observations with `NaN` values.
- Fill `NaN` values in each column with the previous observation in the same column.
- If present, in each column fill `NaN` values at the beginning of the data frame with the value of the subsequent observation in the same column.
- Save the final asset data frame in a parquet file and store it in a clean folder within the data folder.

During the cleaning, we noticed that Microsoft (ticker 'MSFT') and Oracle (ticker 'ORCL') have a very high percentage of `NaN` values; for this reason, we decided to not consider them in our analysis as replacing in any way all these values would have brought to unrealistic prices.
Figure 2 shows how the cleaned data folder looks.

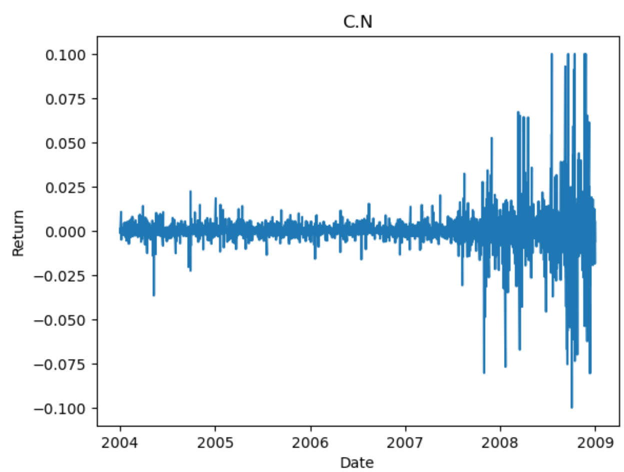
```
/data
├── /raw
│   └── /sp100_2004-8
│       ├── /bbo
│       │   └── ...
│       └── /trade
│           └── ...
└── /clean
    └── /sp100_2004-8
        ├── companyA.parquet File 1
        ├── companyB.parquet File 2
        └── ...
```

Figure 2: Structure of the data folder with both raw and clean data.

Later, we open the saved parquet files and, for each of them, we replace the observations collected in the same minute by their average to then compute simple returns of the trade price time series. Moreover, we collect the trade price returns of all the assets in one unique dataframe merging them by minute and filling `NaN` values (deriving from the merging) in each column with the value of the previous observation in the same column. We save the resulting data frame in a parquet file to access it easily (data $\rightarrow$ clean $\rightarrow$ returns.parquet).
We explore the returns time series qualitatively by plotting them. As expected, most of the series present high volatility and pronounced peaks starting from the end of 2007 due to the financial crisis; an example is shown in Figure 3.



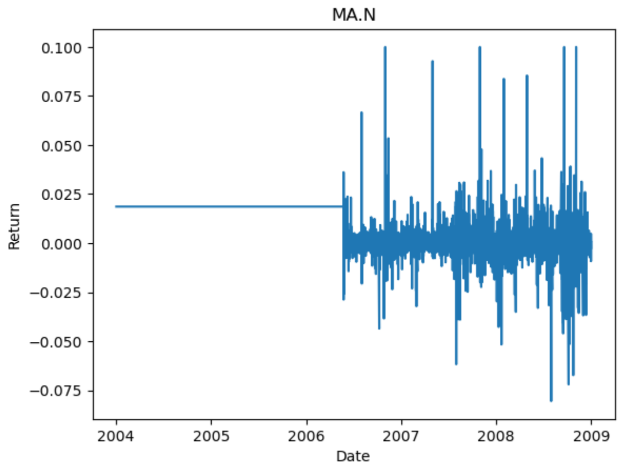
Ford Motor Company.



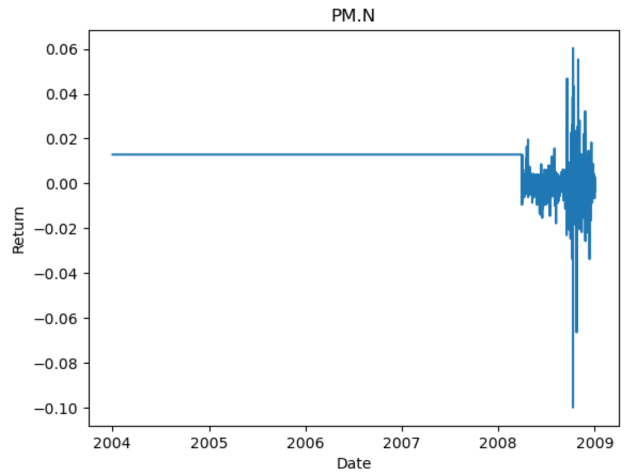
Citigroup Inc.

Figure 3: Effects of the financial crisis.

Furthermore, we also notice that some assets have constant returns for a long period, sometimes years; we find in total six assets whose names are available in the Appendix 0.1.
Figure 4 shows an example of the behavior of those time series.



Mastercard Inc.



Philip Morris International.

Figure 4: Bad behavior of some assets.

We then decide to remove bad behavior assets because constant returns cause us problems in computing correct correlation and covariance matrices. We think that the total number of assets, that now amount to 79, is anyway enough for a complete analysis.

Methods

Our idea is to use all the returns collected in the available 5 years time to cluster assets into groups; then construct sub-portfolios whose components are given by assets in the same cluster, build different strategies to weight these sub-portfolios and compare the cumulative returns of the strategies.
We try two different algorithms to cluster assets. First, we use the Louvain clustering algorithm applied to linear asset returns to cluster assets into different groups. We find 6 clusters whose components are shown in the Appendix 0.2. Looking carefully at the companies classified in the same cluster, we notice that the algorithm can divide companies by industry sectors, hence it seems to perform well.
Table 1 shows the main industry sectors represented by the clusters.

Sub-portfolio	Industry
1	Chemicals, manufacturing, technology
2	Healthcare, consumer goods, telecommunications
3	Retail services
4	Financial services
5	Oil producers
6	Energy producers

Table 1: Main industries sectors after performing Louvain clustering.

Then, we use the Marsili Giada clustering algorithm applied to the correlation matrix of asset returns; to estimate the latter, we utilize the clipped correlation matrix which is constructed clipping to a certain threshold the eigenvalues outside of the bulk of the linear correlation matrix. In total, We find 15 clusters which are listed in Appendix **??**. We notice immediately the difference in the two clustering algorithms: Marsili Giada groups assets in 15 different clusters; in the first two, there are the majority number of assets, while some of them are composed only of two assets. In the case of Marsili Giada, however, we don't find a clear pattern in the given clusters as in Louvain. Independently on the cluster algorithm used, in the following, we explain how we decide the allocation within each cluster.

The allocation in each cluster is made dynamically and is given by the minimum variance weights: for a given cluster and at the end of day t, a covariance matrix Σ_t is estimated with the day t returns of the assets in the cluster. New weights are then computed for the sub-portfolio allocation at day $t+1$. At the end of day $t+1$, a new covariance matrix Σ_{t+1} is estimated with day $t+1$ returns and gives the sub-portfolio allocation for day $t+2$, etc. To estimate Σ_t from returns, we try both the bahc [1] and the clipped covariance matrix; this, then, makes us able to compare results for different combinations of clustering algorithms and covariance matrices.

In the end, we obtain N sub-portfolios (where N is the total number of cluster output of the clustering algorithm) and we develop trading strategies imagining we could trade these sub-portfolios in the market. Moreover, our allocation objective is to decide how to weight these sub-portfolios; the list of the weights associated with the sub-portfolios creates our portfolio.

A more intuitive illustration of how we create our final portfolio is shown in Figure 5.

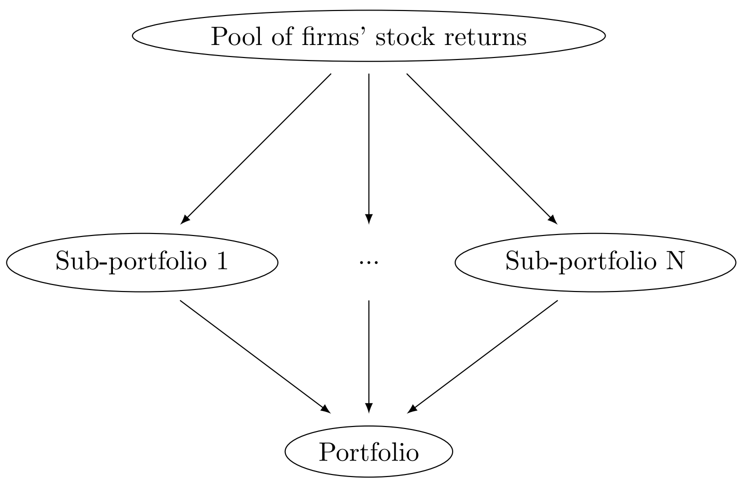


Figure 5: Diagram summarizing the implementation of the portfolio. The components of the N sub-portfolios are obtained by performing a clustering algorithm. The allocation for each sub-portfolio is computed with minimum variance weights. The allocation of every sub-portfolio can vary between risk-parity, inverse volatility, or minimum variance.

Implementation of strategies

Modern portfolio theory, developed by Harry Markowitz in the 1950s, created the foundations for these strategies by introducing the concept of diversification to minimize risk. The starting point for calculating these weights is, in all three cases, an estimate of the covariance matrix of the sub-portfolios, which enables the different strategies determined dynamically to be implemented. The minimum variance portfolio aims to minimize volatility by selecting a fraction of each asset, such that their resulting volatility is minimized while the risk-parity portfolio seeks to balance the contributions of each asset to the portfolio's overall risk, rather than focusing solely on weightings according to their expected return. Risk-parity strategies became popular with Bridgewater's first All-weather funds in the 1990s. This approach is based on the idea that each asset contributes differently to the overall risk of the portfolio, and therefore an equal distribution of risk between assets can lead to better diversification and a reduction in overall portfolio risk. Historically, they seem to handle turbulent phases better than traditional portfolios like 60/40 portfolios. The inverse volatility strategy involves investing more in less volatile assets and less in more volatile assets, to reduce the overall volatility of the portfolio.

Minimum variance weights

The minimum variance weights are computed as follows.
For an n-dimensional time series $\{(r_{1,t}, \dots, r_{n,t})\}_{t=1}^T$ and an estimated covariance matrix Σ_t, the minimum variance weights are given by: the formula

$$w_{mv} = \frac{\Sigma^{-1}\mathbb{1}}{\mathbb{1}^T\Sigma^{-1}\mathbb{1}}.$$

Risk parity weights

It is known that the risk parity strategy tries to give to each asset the allocation that makes it contribute equally to the portfolio aggregate risk. The theory of risk parity can be found in [3] in greater detail, together with examples providing intuition.
Given a portfolio of n assets with weights w, covariance matrix Σ and volatility $\sigma(w) = \sqrt{w^T\Sigma w}$, the Euler Theorem for homogeneous functions implies that $\sigma(w)$ can be decomposed as:

$$\sigma(w) = \sum_{i=1}^n \sigma_i(w),$$

where

$$\sigma_i(w) = \frac{w_i(\Sigma w)_i}{\sqrt{w^T\Sigma w}}.$$

The quantity $\sigma_i(w)$ corresponds to the contribution of asset i to the overall portfolio risk.
Therefore, the risk-parity strategy computes weights such that $\sigma_i(w) = \sigma_j(w)$ for every $i,\ j \in \{1, \dots, n\}$.
The problem hence consists in finding w_{rp} that solves the minimization problem:

$$\arg\min_w \left\{ \sum_{i=1}^n \left(w_i - \frac{\sigma(w)^2}{n(\Sigma w)_i} \right)^2 : \mathbb{1}^T w = 1 \right\}.$$

The implementation of the functions that compute risk-parity weights comes from [2] and has been adapted to our needs.

Inverse volatility weights

The inverse volatility strategy is somehow similar to the risk parity strategy but does not take into account the correlations between assets. If $\Sigma = (\sigma_{ij})_{i,j=1}^n$ is the covariance matrix of n assets, the inverse volatility creates a portfolio with weights w_{iv}, defined as:

$$w_{iv} = \left(\frac{\sigma_{11}}{\mathrm{tr}(\Sigma)}, \dots, \frac{\sigma_{nn}}{\mathrm{tr}(\Sigma)} \right).$$

Results

Figure 6 shows the performance of the 3 considered strategies when applying Louvain clustering and using the Bahc covariance matrix (with K=1, Nboot=50).

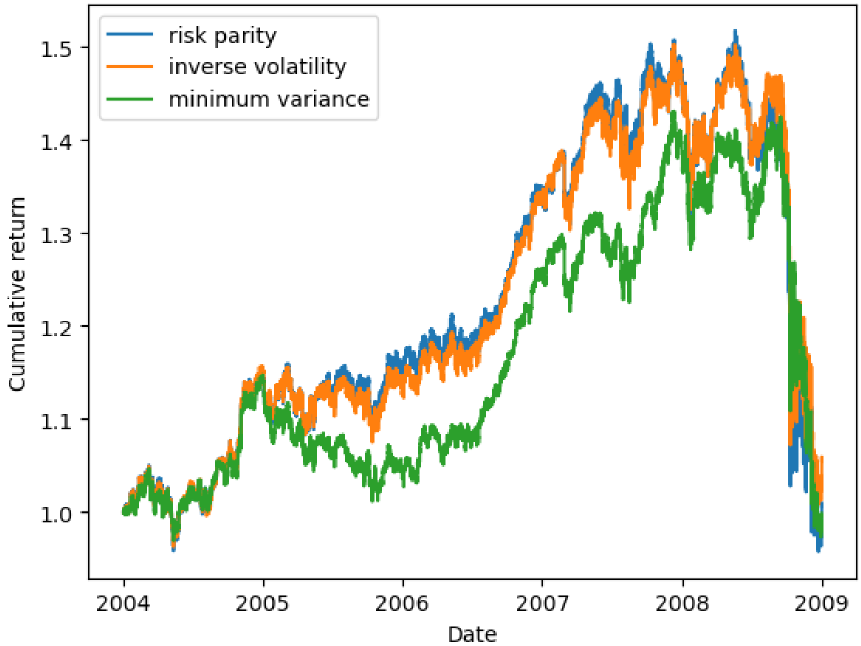


Figure 6: Louvain clustering and bahc covariance matrix.

During the first year all the strategies seem to perform in the same way; then, starting from 2005, risk parity and inverse volatility cumulative returns are similar and both above the minimum variance ones.
Figure 7 shows a comparison between the cumulative returns of the 3 strategies explained above obtained by applying Louvain clustering to classify assets and using the bahc covariance matrix (with K=1, Nboot=50) and the cumulative return of the S&P500.

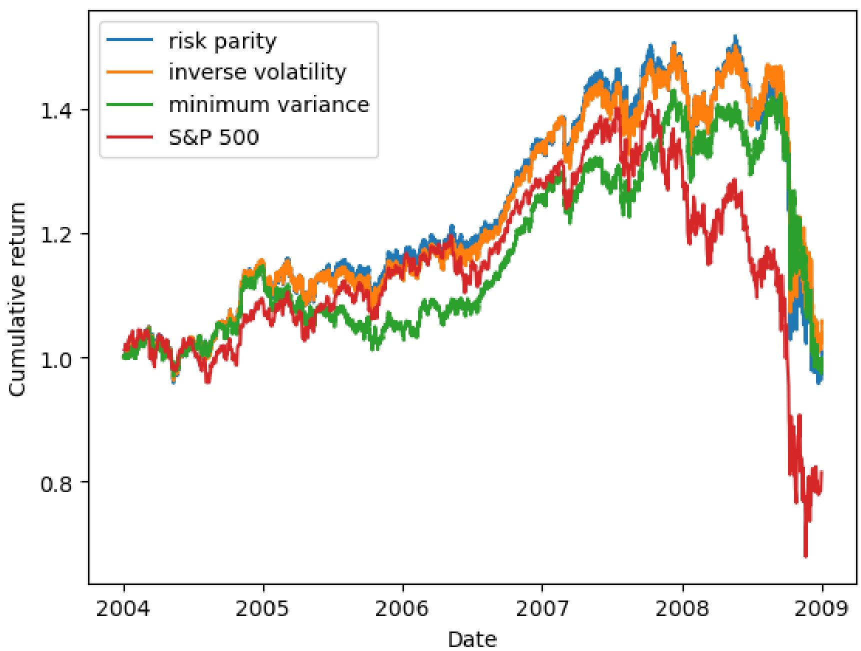


Figure 7: Louvain clustering, Bahc covariance matrix, S&P500.

We notice that the risk parity and inverse volatility strategies outperform the S&P500 almost always, while the minimum variance cumulative return is below the S&P500, except in the last year, during the financial crisis. However, until now, we have not taken into account transaction costs: we are supposed to be able to change our position and trade every day without considering the cost of these trades. This assumption is unrealistic, especially when positions are changed every day. When it comes to considering trading costs, our strategy might not be better than S&P500; it would be interesting to dig into this topic, for example implementing strategies that take into account trading costs.
Figure 8 shows the cumulative returns of the strategies when applying the Marsili Giada clustering algorithm and using the clipped covariance matrix. The S&P500 cumulative return, even if not displayed, is positioned between the minimum variance and the inverse volatility, a bit farther from the inverse volatility than in Figure 7.

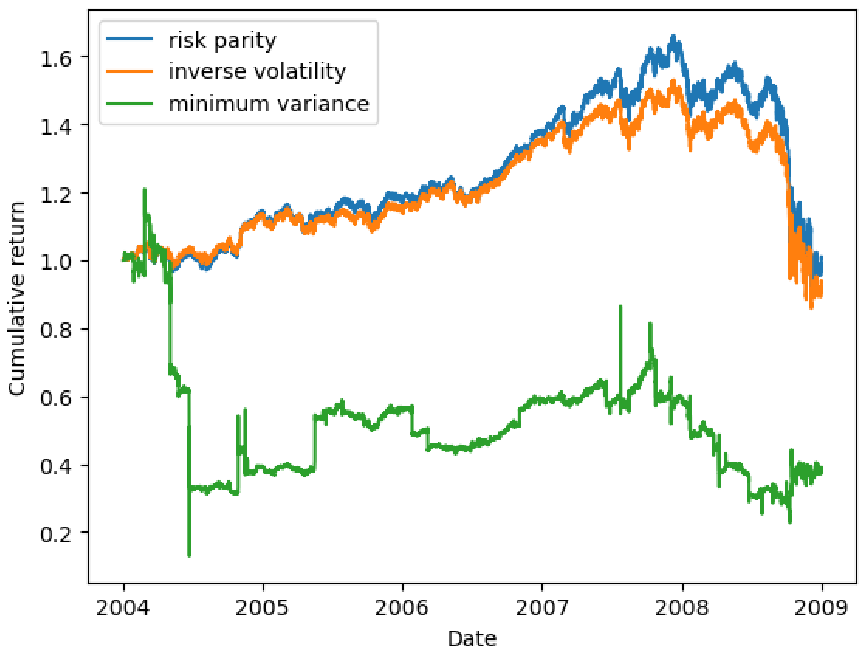


Figure 8: Marsili Giada clustering and clipped covariance matrix.

Here, compared to Figure 6, the minimum variance strategy cumulative return is way below the other strategies; furthermore, it seems less stable than the others, as we can see for example in the middle of 2004 or in the middle of 2007. Moreover, risk parity cumulative return is always above inverse volatility's, reaching also 1.6, something that does not happen in Figure 6.
The comparison between Figure 6 and 8 is an example of how changing the clustering algorithm and how we compute the covariance matrix affect results. We could expect a difference in results especially due to the different output of the two clustering algorithms: Louvain clustering classifies assets into 6 sub-portfolios, while Marsili Giada in 15 sub-portfolios; is quite different.
we try also to obtain the same plot for the combination of Louvain clustering and clipped covariance matrix and, even if the only different input concerning the first case is the covariance matrix, we find a very different plot than Figure 6. We notice that the clipped covariance matrix somehow is less stable than the Bahc one; it gives us different results every time that we run the algorithm and the resulting strategies' cumulative returns fluctuate a lot. For example, in one of the attempts, we find Figure 9.

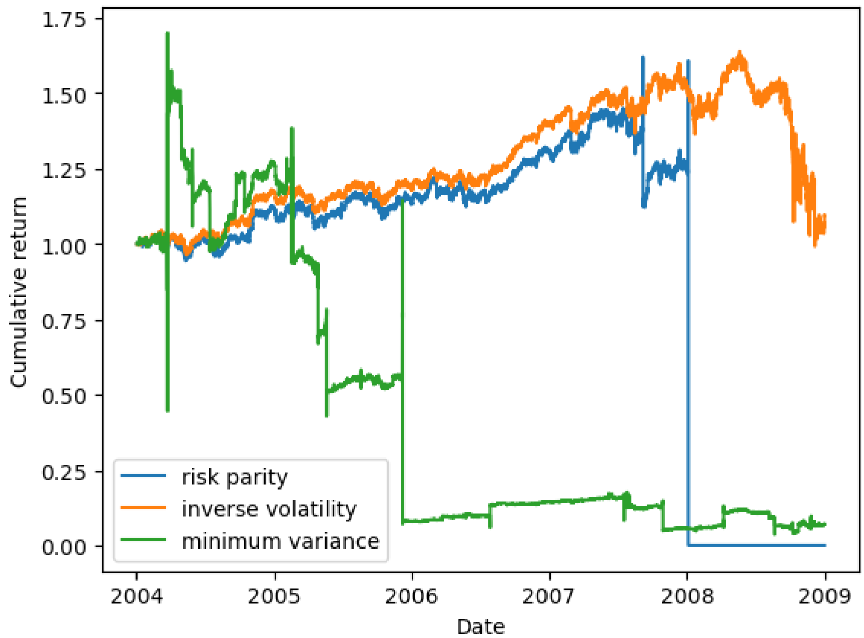


Figure 9: Louvain clustering and clipped covariance matrix.

With the code provided it is possible to run every combination of clustering algorithms and covariance/correlation matrices.

Discussion

The first challenge is to get clean data. Our goal was first to obtain a single data frame containing the linear returns of every asset for every minute in the considered years. The idea behind resampling minute-by-minute is to get rid of the noise and the oscillations of what is inherently contained in tick-by-tick data. This operation lasts several dozens of minutes, even with the use of parallelization and delayed functions.
The second challenge was to decide the destiny of `NaN` values. We decided to apply the radical method of filling a `NaN` with its former value, to replicate the idea that at time t, the best information we have is the last one. As a last resort and especially for the oldest values of the data frame, we applied a backward filling method, i.e. a `NaN` is filled with the first next valid value. There are only a few of them and we estimate that this is not a big issue given the fact that we deal with very small numbers.
These practices may transform drastically the time series of some assets since they can contain large parts of zeros or constants values, but our goal was mostly to have as much possible a data frame with columns of the same length. The main drawback of these fillings appears when we computed the covariance matrix: indeed, if an asset originally had `NaN` values for a whole day, the resulting values of the asset trade price for that day would be either zero or constant and so do its returns. This is problematic as the covariance matrix of returns would then be equal to 0 on the row and the column corresponding to this asset, and therefore would not be invertible. When this phenomenon happened for years since the beginning of the time series, we decided to remove the asset from the analysis (removed assets in 0.1); when it happens for only a small period in the middle of the series (as in the case of Sprint Corporation), we decided to make the asset non-tradable during that period, non considering it as part of the sub-portfolios.
Another problem we encountered is related to the combination of Marsili Giada clustering and Bahc matrices. Above we discussed the case of Marsili Giada clustering and clipped covariance matrix; so, we give the clipped correlation matrix as input of the Marsili Giada algorithm, and in the portfolio construction we

use the clipped covariance matrix. Instead, we tried to put the bahc correlation matrix as input of Marsili Giada and use in the portfolio construction the bahc covariance matrix. In this case, the number of clusters decreased becoming only 3, but then the portfolio construction function did not work, causing different errors. Unfortunately, we had no time to investigate more on the reasons why this happened. However, we were impressed by how different correlation matrices in input could influence the output of Marsili Giada clustering.

There could be a lot of interesting aspects to dig into more in the field of this project's topic, for example, the behavior of the clipped covariance matrix for different types of return time series that maybe could highlight the reasons for the dissimilarities between Figure 6 and Figure 9. We are glad to have the opportunity to work with a very high amount of data and to solve some of the encountered challenges. In conclusion, this project made us able to think about the importance of the decision-making process, to use nonstandard but more suitable methods to treat financial data, and to work with a high volume of information.

Appendix

0.1 List of assets with years of constant returns

Ticker	Company Name
DVN	Devon Energy Corporation
MA	Mastercard Inc.
MS	Morgan Stanley
NOV	Nov Inc.
PM	Philip Morris International Inc.
V	Visa Inc.

Table 2: List of removed assets.

0.2 List of components of each sub-portfolios Louvain clustering

Ticker	Company Name
CAT	Caterpillar Inc.
DD	DuPont de Nemours, Inc.
DOW	Dow Inc.
EMR	Emerson Electric Co.
FDX	FedEx Corporation
GD	General Dynamics Corporation
HON	Honeywell International Inc.
LMT	Lockheed Martin Corporation
MON	Monsanto Company
NKE	NIKE, Inc.
NSC	Norfolk Southern Corporation
RTN	Raytheon Technologies Corporation
UNP	Union Pacific Corporation
UPS	United Parcel Service, Inc.
WY	Weyerhaeuser Company
XRX	Xerox Holdings Corporation

Table 3: Components of sub-portfolio 1.

Ticker	Company Name
ABT	Abbott Laboratories
AVP	Avon Products, Inc.
BAX	Baxter International Inc.
BMY	Bristol-Myers Squibb Company
CL	Colgate-Palmolive Company
HNZ	H.J. Heinz Company
JNJ	Johnson & Johnson
KFT	Kraft Foods Inc.
KO	The Coca-Cola Company
MDT	Medtronic plc
MO	Altria Group, Inc.
MRK	Merck & Co., Inc.
PEP	PepsiCo, Inc.
PFE	Pfizer Inc.
PG	Procter & Gamble Company
S	Sprint Corporation
T	AT&T Inc.
TWX	Time Warner Inc.
VZ	Verizon Communications Inc.

Table 4: Components of sub-portfolio 2.

Ticker	Company Name
BA	Boeing Co.
CVS	CVS Health Corporation
DIS	The Walt Disney Company
HD	The Home Depot, Inc.
HPQ	HP Inc.
IBM	International Business Machines Corporation
LOW	Lowe's Companies, Inc.
MCD	McDonald's Corporation
MMM	3M Company
TGT	Target Corporation
TXN	Texas Instruments Incorporated
UNH	UnitedHealth Group Incorporated
UTX	Raytheon Technologies Corporation
WAG	Walgreen Co.
WMT	Walmart Inc.

Table 5: Components of sub-portfolio 3.

Ticker	Company Name
ALL	Allstate Corporation
AXP	American Express Company
BAC	Bank of America Corporation
BK	The Bank of New York Mellon Corporation
C	Citigroup Inc.
COF	Capital One Financial Corporation
EMC	Emerson Electric Co.
F	Ford Motor Company
GE	General Electric Company
GS	The Goldman Sachs Group, Inc.
JPM	JPMorgan Chase & Co.
MET	MetLife, Inc.
USB	U.S. Bancorp
WFC	Wells Fargo & Co.

Table 6: Components of sub-portfolio 4.

Ticker	Company Name
AA	Alcoa Corporation
APA	Apache Corporation
BHI	Baker Hughes Company
COP	ConocoPhillips
CVX	Chevron Corporation
FCX	Freeport-McMoRan Inc.
HAL	Halliburton Company
OXY	Occidental Petroleum Corporation
SLB	Schlumberger Limited
WMB	The Williams Companies, Inc.
XOM	Exxon Mobil Corporation

Table 7: Components of sub-portfolio 5.

Ticker	Company Name
AEP	American Electric Power Company, Inc.
ETR	Entergy Corporation
EXC	Exelon Corporation
SO	The Southern Company

Table 8: Components of sub-portfolio 6.

0.3 List of components of each sub-portfolios Marsili Giada clustering

Ticker	**Company Name**
AA	Alcoa Corporation
ABT	Abbott Laboratories
AEP	American Electric Power Company, Inc.
APA	Apache Corporation
BA	Boeing Co.
BAC	Bank of America Corporation
CVS	CVS Health Corporation
EMC	Emerson Electric Co.
EMR	Emerson Electric Co.
ETR	Entergy Corporation
FDX	FedEx Corporation
GE	General Electric Company
HD	The Home Depot Inc.
HPQ	HP Inc.
KFT	Kraft Foods Inc.
LMT	Lockheed Martin Corporation
MDT	Medtronic plc
MMM	3M Company
MO	Altria Group, Inc.
MON	Monsanto Company
MRK	Merck & Co., Inc.
NSC	Norfolk Southern Corporation
SO	Southern Company
VZ	Verizon Communications Inc.
WFC	Wells Fargo & Co.
XOM	Exxon Mobil Corporation

Table 9: Components of sub-portfolio 1.

Ticker	Company Name
ALL	Allstate Corporation
AXP	American Express Company
BHI	Baker Hughes Company
C	Citigroup Inc.
CAT	Caterpillar Inc.
COP	ConocoPhillips
DIS	The Walt Disney Company
F	Ford Motor Company
HAL	Halliburton Company
JPM	JPMorgan Chase & Co.
KO	The Coca-Cola Company
LOW	Lowe's Companies, Inc.
PEP	PepsiCo, Inc.
PFE	Pfizer Inc.
T	AT&T Inc.
TGT	Target Corporation
UNP	Union Pacific Corporation
UPS	United Parcel Service, Inc.
UTX	Raytheon Technologies Corporation
WMT	Walmart Inc.

Table 10: Components of sub-portfolio 2.

Ticker	Company Name
AVP	Avon Products, Inc.
DOW	Dow Inc.
TXN	Texas Instruments Incorporated

Table 11: Components of sub-portfolio 3.

Ticker	Company Name
BAX	Baxter International Inc.
BK	The Bank of New York Mellon Corporation
SLB	Schlumberger Limited
TWX	Time Warner Inc.
WMB	The Williams Companies, Inc.

Table 12: Components of sub-portfolio 4.

Ticker	Company Name
BMY	Bristol-Myers Squibb Company
JNJ	Johnson & Johnson
S	Sprint Corporation

Table 13: Components of sub-portfolio 5.

Ticker	Company Name
CL	Colgate-Palmolive Company
DD	Du Pont de Nemours, Inc.

Table 14: Components of sub-portfolio 6.

Ticker	Company Name
COF	Capital One Financial Corporation
RTN	Raytheon Technologies Corporation
XRX	Xerox Holdings Corporation

Table 15: Components of sub-portfolio 7.

Ticker	Company Name
CVX	Chevron Corporation
MET	MetLife, Inc.

Table 16: Components of sub-portfolio 8.

Ticker	Company Name
EXC	Exelon Corporation
FCX	Freeport-McMoRan Inc.
GD	General Dynamics Corporation

Table 17: Components of sub-portfolio 9.

Ticker	Company Name
GS	The Goldman Sachs Group, Inc.
IBM	International Business Machines Corporation

Table 18: Components of sub-portfolio 10.

Ticker	Company Name
HNZ	H.J. Heinz Company
OXY	Occidental Petroleum Corporation

Table 19: Components of sub-portfolio 11.

Ticker	Company Name
HON	Honeywell International Inc.
PG	Procter & Gamble Company
USB	U.S. Bancorp

Table 20: Components of sub-portfolio 12.

Ticker	Company Name
MCD	McDonald's Corporation
UNH	UnitedHealth Group Incorporated

Table 21: Components of sub-portfolio 13.

Ticker	Company Name
NKE	NIKE, Inc.

Table 22: Components of sub-portfolio 14.

Ticker	Company Name
WAG	Walgreen Co.
WY	Weyerhaeuser Company

Table 23: Components of sub-portfolio 15.

References

[1] Bongiorno and Challet (2020). *Package BAHC*. [Online; accessed 28-January-2024]. URL: `https://pypi.org/project/bahc/`.

[2] fjrodriguez2. *Risk Parity in Python*. [Online; accessed 21-January-2024]. URL: `https://quantdare.com/risk-parity-in-python/`.

[3] Risk Parity. *Risk Parity — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-January-2024]. URL: `https://en.wikipedia.org/wiki/Risk_parity`.

Code

```
import numpy as np
import pandas as pd
import os
import gzip
import tarfile
import xlrd
import datetime
import re
import dask
import vaex
import glob
import bahc
import community
import networkx as nx
from scipy.optimize import minimize
from numpy import linalg as LA
import matplotlib.pyplot as plt
import datetime as datetime
```

```python
# Load and store data

def get_file_names(folder):
    """
    Function that returns an array of strings, where each string is the name of a file in the folder passed as argument
    Args:
    - folder: path
    Returns:
    - file_names: array of strings
    """
    file_names = []
    for file in os.listdir(folder):
        file_names.append(file)

    return file_names


def create_file(file_name, content):
    """
    Creates a file in the data folder, it takes as argument the name of the file and the content of the file
    Args:
    - file_name: name of a file
    - content: content of the file
    """
    with open(file_name, 'w') as file:
        file.write(content)


def extract_tar(file_path, output_path):
    """
    Extracts the contents of a .tar file to the specified output path.
    Args:
    - file_path: name of a file
    - output_path: path where the content of the file will be stored
    """
    try:
        with tarfile.open(file_path, 'r') as tar:
            tar.extractall(output_path)
        print(f"Extraction of {file_path} successful.")
    except tarfile.TarError as e:
        print(f"Error extracting {file_path}: {e}")


def extract_csv_gz_file(source_file, destination_directory):
    """
    Extracts a .csv.gz file to a specified directory.
    Args:
    - source_file: The path to the .csv.gz file to be extracted.
```

```
    - destination_directory: The directory where the extracted file will be saved.
    """
    if not os.path.exists(destination_directory):
        os.makedirs(destination_directory)
    try:
        file_name = os.path.basename(source_file)
        output_file = os.path.join(destination_directory, os.path.splitext(file_name)[0])
        with gzip.open(source_file, 'rb') as f_in, open(output_file, 'wb') as f_out:
            f_out.write(f_in.read())
        print(f"File extracted to: {output_file}")
    except Exception as e:
        print(f"Error extracting file: {e}")


def xl_to_datetime(xltime):
    """
    Transforms xltime into an object datetime
    Args:
    - xltime: float
    Returns:
    - date_time_obj: datetime object
    """
    date_value = int(xltime)
    time_value = (xltime - date_value) * 24 * 60 * 60  # Convert fraction of a day to seconds
    date_tuple = xlrd.xldate_as_tuple(date_value, 0)  # 0 for 1900-based date system
    year, month, day, hour, minute, second = date_tuple
    date_time_obj = datetime.datetime(year, month, day, hour, minute, second) + datetime.timedelta(seconds=time_value)

    return date_time_obj


def convert_to_float(value):
    """
    Converts the value to float if it is possible, otherwise it returns nan
    Args:
    - value: float
    Returns:
    - float_value: or a float or nan
    """
    try:
        float_value = float(value)
        return float_value if np.isfinite(float_value) else np.nan
    except (ValueError, TypeError):
        return np.nan

```

```python

def resample_df(df):
    """
    Resamples the dataframe df to 1 minute frequency; one apply the function xl_to_datetime to the column xltime of merged_df
    Args:
    - df: dataframe
    Returns:
    -  df: resampled dataframe
    """
    df['datetime'] = df['xltime'].apply(xl_to_datetime)
    df['bid-price'] = df['bid-price'].astype(float)
    df['ask-price'] = df['ask-price'].astype(float)
    df['bid-volume'] = df['bid-volume'].astype(float)
    df['ask-volume'] = df['ask-volume'].astype(float)

    #drop the column xltime
    df = df.drop(columns=['xltime'])

    #set the column datetime as index
    df = df.set_index('datetime')
    df = df.resample('1T').agg({
        'bid-price': 'mean',
        'ask-price': 'mean',
        'bid-volume': 'sum',
        'ask-volume': 'sum'
    })

    return df


def create_folder(directory_path, folder_name):
    """
    Combines directory path and folder name to create the full path for the new folder
    Args:
    - directory_path: path
    - folder_name: name of the folder
    """
    new_folder_path = os.path.join(directory_path, folder_name)

    # Create the new folder if it doesn't already exist
    if not os.path.exists(new_folder_path):
        os.makedirs(new_folder_path)


def clean_dataframe(df):
    """
    Cleans the dataframe in input by replacing the non-float values with the previous value
```

```
    Args:
    - df: dataframe
    Returns:
    - df: cleaned dataframe
    """
    for column_name in df.columns[1:]:
        # Convert the column to numeric, coercing non-numeric values to NaN
        numeric_column = pd.to_numeric(df[column_name], errors='coerce')
        mean_value = numeric_column.mean()
        if isinstance(df[column_name][0], float)==False:
            df[column_name][0] = mean_value
        for row in range(1, len(df[column_name])):
            if isinstance(df[column_name][row], float)==False:
                df[column_name][row] = df[column_name][row-1]

    return df


def clean_dataframe_faster(df):
    """
    Cleans the dataframe in input by replacing the non-float values with the previous value
    Args:
    - df: dataframe
    Returns:
    - df: cleaned dataframe
    """
    for column_name in df.columns[1:]:
        # Convert the column to numeric, coercing non-numeric values to NaN
        numeric_column = pd.to_numeric(df[column_name], errors='coerce')
        mean_value = numeric_column.mean()

        # Use vectorized operations to replace non-float values
        non_float_mask = ~pd.api.types.is_float_dtype(df[column_name])

        df.loc[non_float_mask, column_name] = df[column_name].shift(1)
        df.loc[0, column_name] = mean_value

    return df


dask.config.set(scheduler="processes")
@dask.delayed
def load_trade(filename,
               tz_exchange="America/New_York",
               only_non_special_trades=True,
               only_regular_trading_hours=True,
               open_time="09:30:00",
               close_time="16:00:00",
               merge_sub_trades=True):
```

```python
    """
    Loads a trade files
    Args:
    - filename: name of the file
    - tz_exchange: timezone
    - only_non_special_trades: boolean
    - only_regular_trading_hours: boolean
    - open_time: string
    - close_time: string
    - merge_sub_trades: boolean
    Returns:
    - DF: dataframe
    """
    try:
        if re.search('(csv|csv\\.gz)$',filename):
            DF = pd.read_csv(filename, engine = "pyarrow")
        if re.search(r'arrow$',filename):
            DF = pd.read_arrow(filename)
        if re.search('parquet$',filename):
            DF = pd.read_parquet(filename)
    except Exception as e:
        return None
    try:
        DF.shape
    except Exception as e:
        print("DF does not exist")
        print(e)
        return None
    if DF.shape[0]==0:
        return None
    if only_non_special_trades:
        DF = DF[DF["trade-stringflag"]=="uncategorized"]
    DF.drop(columns=["trade-rawflag","trade-stringflag"],axis=1,inplace=True)
    DF.index = pd.to_datetime(DF["xltime"],unit="d",origin="1899-12-30",utc=True)
    DF.index = DF.index.tz_convert(tz_exchange)  # .P stands for Arca, which is based at New York
    DF.drop(columns="xltime",inplace=True)
    if only_regular_trading_hours:
        DF=DF.between_time(open_time,close_time)    # warning: ever heard e.g. about Thanksgivings?
    if merge_sub_trades:
           DF=DF.groupby(DF.index).agg(trade_price=pd.NamedAgg(column='trade-price', aggfunc='mean'),
                                        trade_volume=pd.NamedAgg(column='trade-volume', aggfunc='sum'))

    return DF


@dask.delayed
```

```python
def load_bbo(filename,
             only_regular_trading_hours=True,
             merge_sub_trades=True):
    """
    Loads a bbo files
    Args:
    - filename: name of the file
    - only_regular_trading_hours: boolean
    - merge_sub_trades: boolean
    Returns:
    - DF: dataframe
    """

    try:
        if re.search(r'(csv|csv\.gz)$',filename):
            DF = pd.read_csv(filename)
        if re.search(r'arrow$',filename):
            DF = pd.read_arrow(filename)
        if re.search(r'parquet$',filename):
            DF = pd.read_parquet(filename)
    except Exception as e:
        return None
    try:
        DF.shape
    except Exception as e: # DF does not exist
        print("DF does not exist")
        print(e)
        return None
    if DF.shape[0]==0:
        return None
    DF.index = pd.to_datetime(DF["xltime"],unit="d",origin="1899-12-30",utc=True)
    DF.index = DF.index.tz_convert(tz_exchange)  # .P stands for Arca, which is based at New York
    DF.drop(columns="xltime",inplace=True)
    if only_regular_trading_hours:
        DF=DF.between_time("09:30:00","16:00:00")    # ever heard about Thanksgivings?
    if merge_sub_trades:
        DF=DF.groupby(DF.index).last()

    return DF


@dask.delayed
def load_merge_trade_bbo(ticker,date,
                         dirBase="data/raw/sp100_2004-8/",
                         suffix="csv.gz",
                         suffix_save=None,
                         dirSaveBase="data/clean/sp100_2004-8/events",
                         saveOnly=False,
```

```
                             doSave=False
                            ):
    """
    Loads and merges the trade and bbo files
    Args:
    - ticker: name of the ticker
    - date: date
    - dirBase: directory
    - suffix: suffix
    - suffix_save: suffix
    - dirSaveBase: directory
    - saveOnly: boolean
    - doSave: boolean
    Returns:
    - events: dataframe
    """
    file_trade=dirBase+"/"+"/trade/"+ticker+"/"+str(date.date())+"-"+ticker+"-trade."+suffix
    file_bbo=file_trade.replace("trade","bbo")
    trades=load_trade(file_trade)
    bbos  =load_bbo(file_bbo)
    try:
        trades.shape + bbos.shape
    except:
        return None

    events=trades.join(bbos,how="outer")

    if doSave:
        dirSave=dirSaveBase+"/"+"/events/"+ticker
        if not os.path.isdir(dirSave):
            os.makedirs(dirSave)

        if suffix_save:
            suffix=suffix_save

        file_events=dirSave+"/"+str(date.date())+"-"+ticker+"-events"+"."+suffix

        saved=False
        if suffix=="arrow":
            events=vaex.from_pandas(events,copy_index=True)
            events.export_arrow(file_events)
            saved=True
        if suffix=="parquet":
         #   pdb.set_trace()
            events.to_parquet(file_events,use_deprecated_int96_timestamps=True)
            saved=True

        if not saved:
            print("suffix "+suffix+" : format not recognized")
```

```python

        if saveOnly:
            return saved

    return events


def data_to_parquet(ticker):
    """
    Loads the data of the ticker in input from the raw folder, cleans it and stores it in the clean folder
    Args:
    - ticker: name of the ticker
    """
    if ~os.path.exists(f"data/clean/sp100_2004-8/{ticker}.parquet"):
        trade_files=glob.glob(f"data/raw/sp100_2004-8/trade/{ticker}/*.csv.gz")
        # we have a name for each trading file that we find in the directory; each trading file correspond to one
        # trading day
        trade_files.sort()
        allpromises=[load_trade(fn) for fn in trade_files]
        trades=dask.compute(allpromises)[0]
        trades=pd.concat(trades)

        bbo_files=glob.glob(f"data/raw/sp100_2004-8/bbo/{ticker}/*.csv.gz")
        bbo_files.sort()
        allpromises=[load_bbo(fn) for fn in bbo_files]
        bbos=dask.compute(allpromises)[0]
        bbos=pd.concat(bbos)

        events=trades.join(bbos,how="outer")

        # Filling NaNs in 'ask_price' column with the last known value from 'ask_price' column
        events = events.replace('()', np.nan)
        events['ask-price'] = events['ask-price'].bfill()
        events['bid-price'] = events['bid-price'].bfill()
        events['ask-volume'] = events['ask-volume'].bfill()
        events['bid-volume'] = events['bid-volume'].bfill()
        events['ask-price'] = events['ask-price'].ffill()
        events['bid-price'] = events['bid-price'].ffill()
        events['ask-volume'] = events['ask-volume'].ffill()
        events['bid-volume'] = events['bid-volume'].ffill()

        events = events.dropna(subset=['trade_price'])
        events["bid-price"] = events["bid-price"].values.astype("float")
        events["bid-volume"]=events["bid-volume"].values.astype("float")
        events["ask-price"]=events["ask-price"].values.astype("float")
        events["ask-volume"]=events["ask-volume"].values.astype("float")

```

```python
        events.to_parquet(f"data/clean/sp100_2004-8/{ticker}.parquet")


def process_parquet_files(tickers, trading_returns):
    """
    Loads the data of the tickers in input from the clean folder, resamples it and stores it in the resampled folder
    Args:
    - tickers: array of tickers
    - trading_returns: dataframe
    Returns:
    - trading_returns: resampled, cleaned dataframe of trading returns
    """
    for ticker in tickers:
        df = pd.read_parquet(f"data/clean/sp100_2004-8/{ticker}")
        df = df.resample('1T').mean().dropna()
        df_prices = df.drop(columns=['trade_volume', 'bid-volume', 'ask-volume'])
        df_returns = (df_prices / df_prices.shift(1) - 1).dropna()
        trading_returns = pd.concat([trading_returns, df_returns['trade_price'].to_frame(name= ticker[:-8])], axis=1, join='outer')
        trading_returns.ffill(inplace=True)
        trading_returns.bfill(inplace=True)

    for column in trading_returns.columns:
        trading_returns[column] = np.where(np.abs(trading_returns[column]) > 0.1, 0.1 * np.sign(trading_returns[column]), trading_returns[column])

    return trading_returns


# Covariance matrix

def covariance_matrix(trading_returns_df, if_bahc = False, corr_with_bahc = False):
    """
    Computes the covariance matrix of the trading returns dataframe
    Args:
    - trading_returns_df: dataframe
    - bahc: boolean
    Returns:
    - cov_matrix: covariance matrix
    """
    if if_bahc:
        if corr_with_bahc:
            corr = bahc.filterCovariance(trading_returns_df.T.to_numpy(), K = 1, Nboot=50, is_correlation=True)
        else:
            corr = trading_returns_df.corr()
        cov_matrix = bahc.filterCovariance(trading_returns_df.T.to_numpy(), K = 1, Nboot=50)

```

```
        return corr, cov_matrix, 0
    else:
        # number of timesteps
        T = trading_returns_df.shape[0]
        initial_corr = trading_returns_df.corr()
        # number of assets
        N = initial_corr.shape[0]
        if N>T:
            print("N is bigger than T and risk estimation error may diverge")
        q = N/T
        eigenvalues_e, eigenvectors_e = LA.eig(initial_corr)
        lambda_plus = (1+np.sqrt(q))**2
        # number of eigenvalues outside of the random bulk
        outside_bulk_eigenvalues = np.sum(eigenvalues_e>lambda_plus)
        corr_clipped = eigenvalue_clipping(eigenvalues_e,eigenvectors_e,lambda_plus)
        # from correlation matrix to covariance matrix
        trading_std = np.array(trading_returns_df.std())
        cov_matrix = np.outer(trading_std, trading_std) * np.array(corr_clipped)

        return corr_clipped, cov_matrix, outside_bulk_eigenvalues


def eigenvalue_clipping(lambdas,v,lambda_plus):
    """
    Clips the eigenvalues of the correlation matrix to lambda_plus
    Args:
    - lambdas: eigenvalues
    - v: eigenvectors
    - lambda_plus: threshold
    Returns:
    - C_clean: clipped correlation matrix
    """
    N=len(lambdas)
    sel_bulk=lambdas<=lambda_plus
    N_bulk=np.sum(sel_bulk)
    sum_lambda_bulk=np.sum(lambdas[sel_bulk])
    delta=sum_lambda_bulk/N_bulk
    lambdas_clean=lambdas
    lambdas_clean[lambdas_clean<=lambda_plus]=delta
    C_clean=np.zeros((N, N))
    v_m=np.matrix(v)
    for i in range(N-1):
        C_clean=C_clean+lambdas_clean[i] * np.dot(v_m[i,].T,v_m[i,])

    np.fill_diagonal(C_clean,1)

    return C_clean
```

```python
# Clustering function to decide which clustering algorithm to use

def clustering(returns_df, if_marsili_giada = False, if_bahc = False):
    """
    Computes the clusters of the returns dataframe
    Args:
    - returns_df: dataframe
    - if_marsili_giada: boolean
    - if_bahc: boolean
    Returns:
    - clusters_constituents: dictionary
    - sub_portfolios: array of strings
    - sub_portfolios_returns: dataframe
    """
    if if_marsili_giada:
        corr_matrix, cov_matrix, eigenvalue_clipping = covariance_matrix(returns_df
            , if_bahc, True )
        last_cluster = aggregate_clusters(corr_matrix)
        s_i = last_cluster['s_i']
        marsili_clusters = pd.DataFrame()
        marsili_clusters.index = returns_df.columns
        marsili_clusters[0] = s_i
        array_of_numbers = marsili_clusters[0].unique()
        for i in range(0, len(marsili_clusters[0].unique())):
            marsili_clusters[0] = np.where(marsili_clusters[0] == array_of_numbers[
                i], i, marsili_clusters[0])
        clusters = marsili_clusters
        clusters_dict =  marsili_clusters[0].to_dict()

    else:
        louvain_clusters = LouvainCorrelationClustering(returns_df)
        louvain_clusters.index = returns_df.columns
        clusters = louvain_clusters
        clusters_dict = louvain_clusters[0].to_dict()

    sub_portfolios_returns = pd.DataFrame(0.0, index = returns_df.index, columns =
        ['sub_pf_{}'.format(i) for i in range(clusters.max().max() + 1)])
    sub_portfolios = ["sub_pf_{}".format(i+1) for i in range(clusters.max().max() +
        1)]
    clusters_constituents = {}
    for letter, value in clusters_dict.items():
        clusters_constituents.setdefault(value, []).append(letter)

    return clusters_constituents, sub_portfolios, sub_portfolios_returns


# Marsili Giada clustering algorithm

def expand_grid_unique(x, y, include_equals=False):
    """
```

```
    Expands the grid of unique values
    Args:
    - x: array
    - y: array
    - include_equals: boolean
    Returns:
    - combinations: array
    """
    x = list(set(x))
    y = list(set(y))

    def g(i):
        z = [val for val in y if val not in x[:i - include_equals]]
        if z:
            return [x[i - 1]] + z

    combinations = [g(i) for i in range(1, len(x) + 1)]

    return [combo for combo in combinations if combo]


def max_likelihood(c, n):
    """
    Computes the maximum likelihood
    Args:
    - c: float
    - n: float
    Returns:
    - max_likelihood: float
    """
    if n > 1:
        return np.log(n / c) + (n - 1) * np.log((n * n - n) / (n * n - c))
    else:
        return 0

def max_likelihood_list(cs, ns):
    """
    Computes the maximum likelihood of a list
    Args:
    - cs: dictionary
    - ns: dictionary
    Returns:
    - Lc: dictionary
    """
    Lc = {}
    for x in cs.keys():
        if ns[x] > 1:
            Lc[x] = np.log(ns[x] / cs[x]) + (ns[x] - 1) * np.log((ns[x] * ns[x] -
                ↪ ns[x]) / (ns[x] * ns[x] - cs[x]))
        else:
```

```
            Lc[x] = 0

    return Lc

def find_max_improving_pair(C, cs, ns, i_s):
    """
    Finds the maximum improving pair
    Args:
    - C: matrix
    - cs: dictionary
    - ns: dictionary
    - i_s: dictionary
    Returns:
    - pair_max_improv: array
    - Lc_max_impr: float
    - Lc_old: array
    """
    Lc_old = max_likelihood_list(cs, ns)
    names_cs = list(cs.keys())
    max_impr = -1e10
    pair_max_improv = []

    for i in names_cs[:-1]:
        names_cs_j = names_cs[names_cs.index(i) + 1:]
        for j in names_cs_j:
            ns_new = ns[i] + ns[j]
            i_s_new = i_s[i] + i_s[j]
            cs_new = np.sum(C[np.ix_(i_s_new, i_s_new)])
            max_likelihood_new = max_likelihood(cs_new, ns_new)
            improvement = max_likelihood_new - Lc_old[i] - Lc_old[j]

            if improvement > max_impr:
                max_impr = improvement
                pair_max_improv = [i, j]
                Lc_max_impr = max_likelihood_new

    return {"pair": pair_max_improv, "Lc_new": Lc_max_impr, "Lc_old": [Lc_old[x] for x in pair_max_improv]}

def aggregate_clusters(C):
    """
    Aggregates the clusters
    Args:
    - C: matrix
    Returns:
    - last_clusters: dictionary
    """
    N = C.shape[0]
    cs = {i: 1 for i in range(N)}
    s_i = {i: [i] for i in range(N)}
```

```
    ns = {i: 1 for i in range(N)}
    i_s = {i: [i] for i in range(N)}
    clusters = []

    for i in range(1, N):  # hierarchical merging
        improvement = find_max_improving_pair(C, cs, ns, i_s)
        Lc_old = improvement['Lc_old']
        Lc_new = improvement['Lc_new']

        if Lc_new < sum(Lc_old):
            print(" HALF CLUSTER  Lc.new > max(Lc.old)")

        if Lc_new <= max(Lc_old):
            print("Lc.new <= max(Lc.old), exiting")
            break

        pair = improvement['pair']
        s_i = [pair[0] if x == pair[1] else x for x in s_i]

        cluster1 = pair[0]
        cluster2 = pair[1]
        i_s[cluster1].extend(i_s[cluster2])  # merge the elements of the two clusters
        del i_s[cluster2]  # removes reference to merged cluster2

        ns[cluster1] += ns[cluster2]
        del ns[cluster2]

        cs[cluster1] = np.sum(C[i_s[cluster1]][:, i_s[cluster1]])  # sums C over the elements of cluster1
        del cs[cluster2]

        clusters.append({
            'Lc': max_likelihood_list(cs, ns),
            'pair_merged': pair,
            's_i': s_i,
            'i_s': i_s,
            'cs': cs,
            'ns': ns
        })

    last_clusters = clusters[-1]

    return last_clusters


# Louvain clustering algorithm

def LouvainCorrelationClustering(R):
    """
```

```
    Computes the Louvain clustering of the returns matrix R
    Args:
    - R: matrix of returns
    Returns:
    - DF: dataframe
    """
    N=R.shape[1]
    T=R.shape[0]

    q=N*1./T
    lambda_plus=(1.+np.sqrt(q))**2

    C=R.corr()
    lambdas, v = LA.eigh(C)
    C_s=compute_C_minus_C0(lambdas,v,lambda_plus)

    mygraph= nx.from_numpy_matrix(np.abs(C_s))
    partition = community.community_louvain.best_partition(mygraph)

    DF=pd.DataFrame.from_dict(partition,orient="index")

    return(DF)


def compute_C_minus_C0(lambdas,v,lambda_plus,removeMarketMode=True):
    """
    Computes the correlation matrix C minus C0
    Args:
    - lambdas: eigenvalues
    - v: eigenvectors
    - lambda_plus: threshold
    - removeMarketMode: boolean
    Returns:
    - C_clean: correlation matrix
    """
    N=len(lambdas)
    C_clean=np.zeros((N, N))

    order = np.argsort(lambdas)
    lambdas,v = lambdas[order],v[:,order]

    v_m=np.matrix(v)

    for i in range(1*removeMarketMode,N):
        if lambdas[i]>lambda_plus:
            C_clean=C_clean+lambdas[i] * np.dot(v_m[:,i],v_m[:,i].T)

    return C_clean
```

```python
# Portfolios construction
# Part of the following code comes from https://quantdare.com/risk-parity-in-python/ and has
# been adapted to our needs.

def portfolios_construction(clusters_constituents, returns, dates, sub_portfolios_returns, if_bahc = False):
    """
    Constructs the portfolios
    Args:
    - clusters_constituents: dictionary
    - returns: dataframe
    - dates: array
    - sub_portfolios_returns: dataframe
    - if_bahc: boolean
    Returns:
    - portfolio_cumprod: cumulative returns dataframe
    - portfolio_return: portfolio returns dataframe
    """
    for cluster in clusters_constituents.keys():
        df_without_0, stocks_to_be_removed = remove_bad_columns(returns[clusters_constituents[cluster]].loc[dates[0]])
        corr_clipped, cov_matrix, outside_bulk_eigenvalues = covariance_matrix(df_without_0, if_bahc)
        w_mv = compute_mv_weights(cov_matrix)

        for date in dates[1:]:
            clusters_constituents_without_stocks_to_be_removed = [name for name in clusters_constituents[cluster] if name not in stocks_to_be_removed]
            df_without_0, stocks_to_be_removed = remove_bad_columns(returns[clusters_constituents[cluster]].loc[date])
            corr_clipped, cov_matrix, outside_bulk_eigenvalues = covariance_matrix(df_without_0, if_bahc)

            series = pd.Series(np.dot(w_mv.T, returns[clusters_constituents_without_stocks_to_be_removed].loc[date].T))
            series.index = sub_portfolios_returns.loc[date].index
            sub_portfolios_returns[f'sub_pf_{cluster}'].loc[date] = series

            w_mv = compute_mv_weights(cov_matrix)

    # sub_portfolios_returns = pd.DataFrame(np.where(np.abs(sub_portfolios_returns) > 0.5, 0.5 * np.sign(sub_portfolios_returns), sub_portfolios_returns))
    date = dates[1]
    portfolio_return = pd.DataFrame(0.0, index = returns.index, columns = ['rp_pf', 'iv_pf', 'mv_pf'])
    rp_weights = pd.DataFrame(0.0, index = dates, columns = ["sub_pf_{}".format(i) for i in range(max(list(clusters_constituents.keys())) + 1)])
```

```python

    for date in dates[1:]:
        df_without_0, stocks_to_be_removed = remove_bad_columns(sub_portfolios_returns.loc[date])
        corr_clipped, cov_matrix, outside_bulk_eigenvalues = covariance_matrix(sub_portfolios_returns.loc[date], if_bahc)
        w_rp = compute_rp_weights(cov_matrix)
        w_iv = compute_iv_weights(cov_matrix)
        w_mv = compute_mv_weights(cov_matrix)

        series_rp = pd.Series(np.dot(w_rp.T, sub_portfolios_returns.loc[date].T))
        series_rp.index = portfolio_return.loc[date].index
        portfolio_return['rp_pf'].loc[date] = series_rp

        series_iv = pd.Series(np.dot(w_iv.T, sub_portfolios_returns.loc[date].T))
        series_iv.index = portfolio_return.loc[date].index
        portfolio_return['iv_pf'].loc[date] = series_iv

        series_mv = pd.Series(np.dot(w_mv.T, sub_portfolios_returns.loc[date].T))
        series_mv.index = portfolio_return.loc[date].index
        portfolio_return['mv_pf'].loc[date] = series_mv

        series_w_rp = pd.Series(w_rp)
        series_w_rp.index = rp_weights.loc[date].index
        rp_weights.loc[date] = series_w_rp

    for column in portfolio_return.columns:
        portfolio_return[column] = np.where(np.abs(portfolio_return[column]) > 0.3,  0.3 * np.sign(portfolio_return[column]), portfolio_return[column])

    portfolio_cumprod = (portfolio_return + 1).cumprod()

    for column in portfolio_cumprod.columns:
        portfolio_cumprod[column] = np.where(np.abs(portfolio_cumprod[column]) > 1.7, 1.7 * np.sign(portfolio_cumprod[column]), portfolio_cumprod[column])


    return portfolio_cumprod, portfolio_return


def compute_mv_weights(cov):
    """
    Computes the weights of a minimal variance portfolio given the covariance matrix
    Args:
    - cov: covariance matrix
    Returns:
    - w: weights
    """
```

```
    eigenvalues, eigenvectors = np.linalg.eig(cov)
    D = np.diag(eigenvalues)
    inv_D = np.linalg.inv(D)
    inv_cov = np.dot(np.dot(eigenvectors,inv_D),eigenvectors.T)
    w = inv_cov.dot(np.ones(len(cov))) / np.ones(len(cov)).T.dot(inv_cov).dot(np.ones(len(cov)))

    return w


def compute_iv_weights(cov):
    """
    Computes the weights of an inverse volatility portfolio given the covariance matrix
    Args:
    - cov: covariance matrix
    Returns:
    - w: weights
    """
    w = 1 / np.diag(cov)
    w = w / w.sum()

    return w


def compute_vol(w, cov):
    """
    Computes  the volatility of a portfolio given the weights and the covariance matrix
    Args:
    - w: weights
    - cov: covariance matrix
    Returns:
    - pf_risk: portfolio risk
    """
    pf_risk = np.dot(w, cov).dot(w.T)

    return pf_risk

def compute_assets_risk_contribution(w, cov):
    """
    Computes the contribution of each asset to the risk of the whole portfolio
    Args:
    - w: weights
    - cov: covariance matrix
    Returns:
    - assets_risk_contribution: array of assets risk contribution
    """
    pf_risk = compute_vol(w, cov)
    assets_risk_contribution = np.multiply(w.T, np.dot(cov,w.T))/pf_risk
```

```python

    return assets_risk_contribution


def objective_function(w, args):
    """
    Objective function to minimize
    Args:
    - w: weights
    - args: arguments
    Returns:
    - error: error
    """
    cov = args[0]
    assets_risk_budget = args[1]
    w = np.matrix(w)
    pf_risk = compute_vol(w, cov)
    assets_risk_contribution = compute_assets_risk_contribution(w, cov)
    assets_risk_target = np.asmatrix(np.multiply(pf_risk, assets_risk_budget))
    error = sum(np.square(assets_risk_contribution - assets_risk_target.T))[0, 0]

    return error

def compute_rp_weights(cov, constraint_type = 'long'):
    """
    Computes the weights of a risk parity portfolio given the covariance matrix
    Args:
    - cov: covariance matrix
    - constraint_type: type of constraint
    Returns:
    - w: weights
    """
    initial_weights = np.ones(cov.shape[0]) * (1.0 / cov.shape[0])
    assets_risk_budget = np.ones(cov.shape[0]) * (1.0 / cov.shape[0])
    if constraint_type == 'long':
        #constraints for long-only portfolio. sum of weights = 1
        constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1.0},{'type': 'ineq', 'fun': lambda x: x})
    elif constraint_type == 'long_short':
        #constraints for long-short portfolio. sum of weights = 0
        constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x)})
    else:
        raise ValueError('Unknown constraint_type. Only possible values for the get_risk_parity_weights constraint_type argument are "long" and "long_short"')

    sol = minimize(fun=objective_function, x0=initial_weights, args=[cov, assets_risk_budget], method='SLSQP', constraints=constraints, tol=1e-15, options={'disp': False})
    w = sol.x
```

```
    return w


def remove_bad_columns(dataframe):
    """
    Removes the columns of a dataframe that are constant, NaN or infinite
    Args:
    - dataframe: dataframe
    Returns:
    - new_df: new dataframe
    - list(zero_columns): list of removed columns
    """
    for column in dataframe.columns:
        if dataframe[column].nunique() == 1:
            dataframe[column] = 0
    dataframe.fillna(0, inplace=True)
    dataframe.replace([np.inf, -np.inf], 0, inplace=True)
    zero_columns = dataframe.columns[dataframe.eq(0).all()]
    new_df = dataframe.loc[:, ~dataframe.columns.isin(zero_columns)]

    return new_df, list(zero_columns)


def SP500_returns(path):
    """
    Computes the returns of the S&P500 index
    Args:
    - path: path
    Returns:
    - portfolio_cumprod_SP500: cumulative returns dataframe
    """
    df_sp500 = pd.read_csv(path)
    df_reversed = df_sp500.iloc[::-1]
    df_reversed.reset_index(inplace=True)
    df_reversed.drop(columns=['index'], inplace=True)
    df_reversed['Date'] = pd.to_datetime(df_reversed['Date'])
    df_reversed.set_index('Date', inplace=True)
    df = df_reversed['Price'].to_frame()
    df['Price'] = df['Price'].str.replace(',', '').astype(float)
    df = (df / df.shift(1) - 1).dropna()
    df.rename(columns={'Price': 'Returns S&P500'}, inplace=True)
    portfolio_cumprod_SP500 = (df + 1).cumprod()
    # save dataframe into a parquet file
    portfolio_cumprod_SP500.to_parquet('data/clean/SP500returns.parquet', engine='pyarrow')

    return portfolio_cumprod_SP500
```

```
# stock_tickers contains the name of all the files/stocks
```

```python
stock_tickers_bbo = get_file_names('data/raw/sp100_2004-8/bbo')
stock_tickers_bbo.remove('.DS_Store')
for file_name_bbo in stock_tickers_bbo:
    file_path_bbo = f"data/raw/sp100_2004-8/bbo/{file_name_bbo}/{file_name_bbo}_bbo.tar"
    output_path_bbo = f"data/raw/sp100_2004-8/bbo/{file_name_bbo}/"
    extract_tar(file_path_bbo, output_path_bbo)
```

```python
stock_tickers_trade = get_file_names('data/raw/sp100_2004-8/trade')
for file_name_trade in stock_tickers_trade:
    file_path_trade = f"data/raw/sp100_2004-8/trade/{file_name_trade}/{file_name_trade}_trade.tar"
    output_path_trade = f"data/raw/sp100_2004-8/trade/{file_name_trade}/"
    extract_tar(file_path_trade, output_path_trade)
```

```python
tickers = get_file_names("data/raw/sp100_2004-8/trade/")
tickers.remove('MSFT.O')
tickers.remove('ORCL.N')
for ticker in tickers:
    data_to_parquet(ticker)
```

```python
trading_returns = pd.DataFrame()
tickers = get_file_names("data/clean/sp100_2004-8/")
tickers.remove('DVN.N.parquet')
tickers.remove('MA.N.parquet')
tickers.remove('MS.N.parquet')
tickers.remove('NOV.N.parquet')
tickers.remove('PM.N.parquet')
tickers.remove('V.N.parquet')
trading_returns_df = process_parquet_files(tickers, trading_returns)
trading_returns_df.to_parquet('data/clean/returns.parquet', engine='pyarrow')
```

```python
# run this cell only if you want to recreate the file already stored with the name "portfolio_cumprod_louvain_bahc.parquet"
returns = pd.read_parquet(f"data/clean/returns.parquet")
dates = returns.index.strftime("%Y-%m-%d").unique()
clusters_constituents, sub_portfolios, sub_portfolios_returns = clustering(returns, if_marsili_giada=False)
portfolio_cumprod = portfolios_construction(clusters_constituents, returns, dates, sub_portfolios_returns, if_bahc=True)
portfolio_cumprod.to_parquet('data/clean/portfolio_cumprod_louvain_bahc.parquet')
```

```python
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_louvain_bahc.parquet')
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
```

```
plt.show()
```

```
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_louvain_bahc.parquet')
path = "data/raw/sp100_2004-8/S&P 500 Historical Data.csv"
portfolio_cumprod_SP500 = SP500_returns(path)
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.plot(portfolio_cumprod_SP500, label = 'S&P 500')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
plt.show()
```

```
# run this cell only if you want to recreate the file already stored with the name "portfolio_cumprod_marsili_clipped.parquet"
returns = pd.read_parquet(f"data/clean/returns.parquet")
dates = returns.index.strftime("%Y-%m-%d").unique()
clusters_constituents, sub_portfolios, sub_portfolios_returns = clustering(returns, if_marsili_giada=True)
portfolio_cumprod, portfolio_return = portfolios_construction(clusters_constituents, returns, dates, sub_portfolios_returns, if_bahc=False)
portfolio_cumprod.to_parquet('data/clean/portfolio_cumprod_marsili_clipped.parquet')
```

```
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_marsili_clipped.parquet')
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
plt.show()
```

```
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_marsili_clipped.parquet')
path = "data/raw/sp100_2004-8/S&P 500 Historical Data.csv"
portfolio_cumprod_SP500 = SP500_returns(path)
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.plot(portfolio_cumprod_SP500, label = 'S&P 500')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
plt.show()
```

```python
# run this cell only if you want to recreate the file already stored with the name "portfolio_cumprod_louvain_clipped.parquet"
returns = pd.read_parquet(f"data/clean/returns.parquet")
dates = returns.index.strftime("%Y-%m-%d").unique()
clusters_constituents, sub_portfolios, sub_portfolios_returns = clustering(returns, if_marsili_giada=False)
portfolio_cumprod, portfolio_return = portfolios_construction(clusters_constituents, returns, dates, sub_portfolios_returns, if_bahc=False)
portfolio_cumprod.to_parquet('data/clean/portfolio_cumprod_louvain_clipped.parquet')
```

```python
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_louvain_clipped.parquet')
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
plt.show()
```

```python
portfolio_cumprod = pd.read_parquet('data/clean/portfolio_cumprod_louvain_clipped.parquet')
path = "data/raw/sp100_2004-8/S&P 500 Historical Data.csv"
portfolio_cumprod_SP500 = SP500_returns(path)
plt.plot(portfolio_cumprod['rp_pf'], label = 'risk parity')
plt.plot(portfolio_cumprod['iv_pf'], label = 'inverse volatility')
plt.plot(portfolio_cumprod['mv_pf'], label = 'minimum variance')
plt.plot(portfolio_cumprod_SP500, label = 'S&P 500')
plt.xlabel('Date')
plt.ylabel('Cumulative return')
plt.legend()
plt.show()
```