Machine Learning for Finance

-Final Project Report-

Student Name: Nguyen Ngoc Truong
Student ID: [20235264]

Student Name: Khalil Hassan
Student ID: [20234229]

Student Name: Alexandre Maroussiak
Student ID: [20216059]

Student Name: Amaury Manzione
Student ID: [20216814]

August 1, 2024

Contents

List of Figures

List of Tables

1. Introduction

In the dynamic realm of financial markets, the ability to predict the optimal portfolio for trading instruments within short timeframes holds an immense significance for traders and investors. Such predictions enable efficient portfolio management, risk mitigation, and capital optimization in volatile environments. This project delves into a comprehensive methodology aimed at predicting optimal weights for trading instruments within a 10-minute timeframe, leveraging datasets from the Kaggle competition titled **"Optiver Realized Volatility Prediction"**.

The dataset provides intricate insights into the order and trade books of various stocks for the first 10-minute period, serving as a foundation for predicting optimal weights. However, for the subsequent 10-minute period, only the Realized Volatility is provided, necessitating the development of a robust predictive model to derive optimal weight allocations.

Our approach encompasses two significant phases. The initial phase revolves around the synthesis of optimal weights using traditional methods such as Markowitz's Portfolio Optimization and Hierarchical Risk Parity (HRP). These established methodologies serve as benchmarks, providing targets for the subsequent predictive model.

In the second phase, we employ advanced machine learning techniques, specifically a Neural Network model, to predict these optimal weights genertaed in the first phase. This phase embodies a supervised learning framework, where the model is trained on the dataset comprising order and trade book data from the first 10-minute period, alongside the synthesized optimal weights derived from traditional methods.

By integrating traditional financial techniques with modern machine learning algorithms, our approach aims to harness the strengths of both domains to enhance the accuracy and efficiency of optimal weight prediction.

The general steps of our solution are shown in the flowchart below:

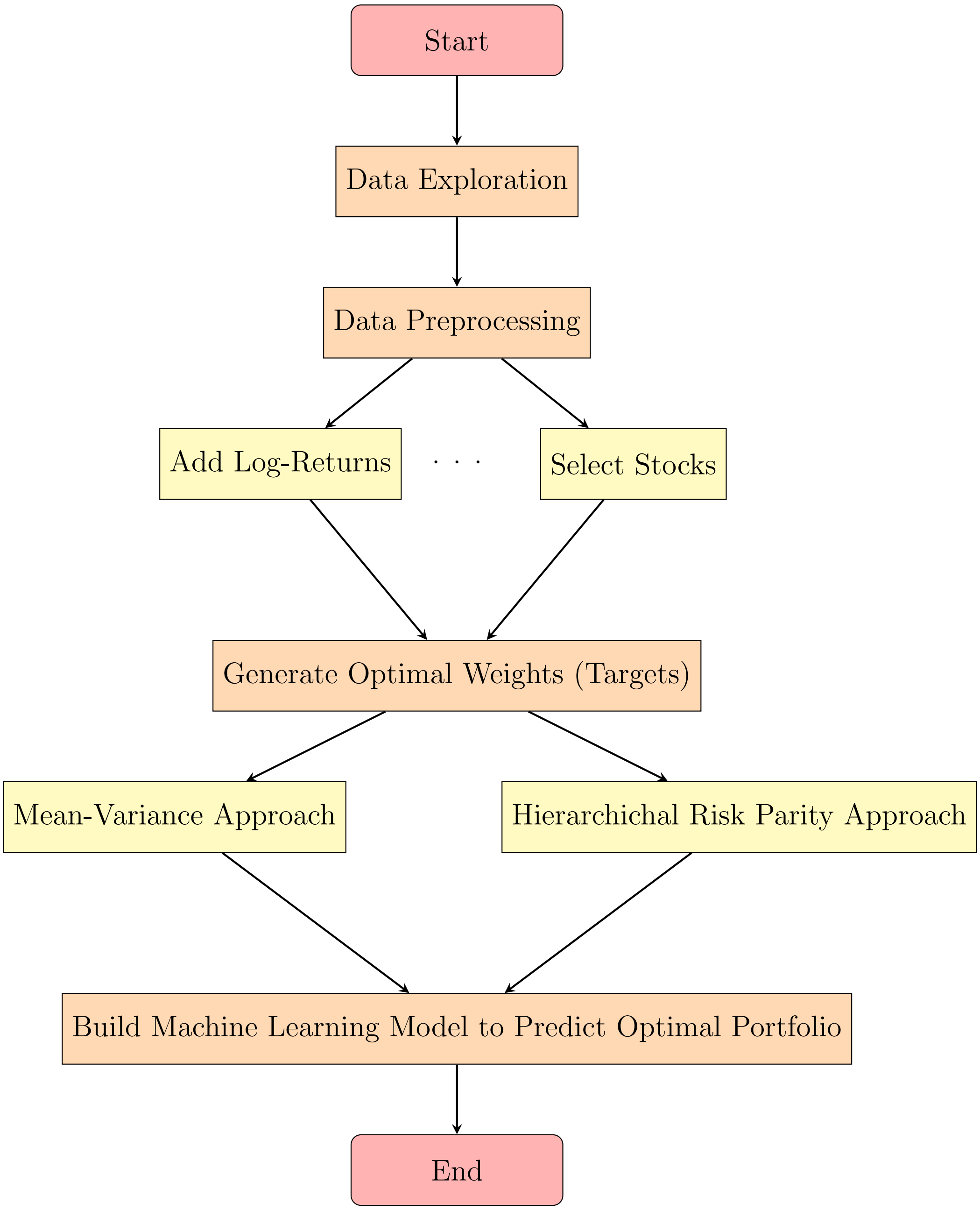

Start
Data Exploration
Data Preprocessing
Add Log-Returns
· · ·
Select Stocks
Generate Optimal Weights (Targets)
Mean-Variance Approach
Hierarchichal Risk Parity Approach
Build Machine Learning Model to Predict Optimal Portfolio
End

Figure 1.1: General Steps in the Project

The details of these steps are mentioned in the following sections.

2. Data Exploration

The Kaggle competition gives the dataset ”Optiver Realized Volatility Prediction.” It contains stock market data relevant to the practical execution of trades in the financial markets. In particular, it includes order book snapshots and executed trades.

Here is an overview of the provided datasets:

- **book_train.parquet**: for each stock_id, it contains order book data on the most competitive buy and sell orders entered into the market. Here are the given features:

Features	**Description**
stock_id	ID code for the stock.
time_id	ID code for the time bucket.
seconds_in_bucket	Number of seconds from the start of the bucket, always starting from 0.
bid_price[1/2]	Normalized prices of the most/second most competitive buy level.
ask_price[1/2]	Normalized prices of the most/second most competitive sell level.
bid_size[1/2]	The Number of shares on the most/second most competitive buy level.
ask_size[1/2]	The Number of shares on the most/second most competitive sell level.

Table 2.1: List of Features of book_train.parquet

- **trade_train.parquet**: for each stock_id it contains data on trades that actually executed. Here are the given features:

Features	**Description**
stock_id	ID code for the stock.
time_id	ID code for the time bucket.
seconds_in_bucket	Number of seconds from the start of the bucket, always starting from 0.
price	The average price of executed transactions happening in one second.
size	The sum number of shares traded.
order_count	The Number of unique trade orders taking place.

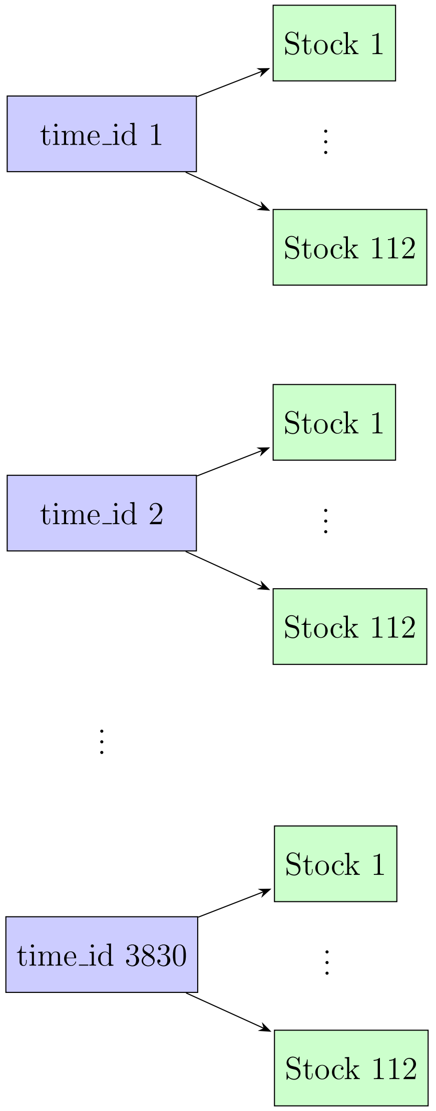
Table 2.2: List of Features of trade_train.parquet

- **train.csv**: for each stock_id it contains the ground truth values of Realized Volatilities for the training set.:

Features	**Description**
stock_id	ID code for the stock.
time_id	ID code for the time bucket.
target	The realized volatility computed over the 10-minute window following the feature data under the same stock/time_id.

Table 2.3: List of Features of train.csv

The given dataset contains 3830 time_ids, each time_id represents a **random fixed 20 minutes window**, with all the trade and book data available for the first 10 minutes window and only the Realized Volatility available for the second 10 minutes window. For each time_id, we have data of 112 stocks, as explained in the graph below.

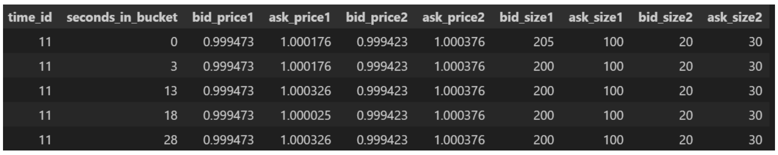


3. Data Pre-Processing

Fill in missing seconds, add log returns and RV, and Select 10 Stocks ...

3.1. Fill the missing seconds

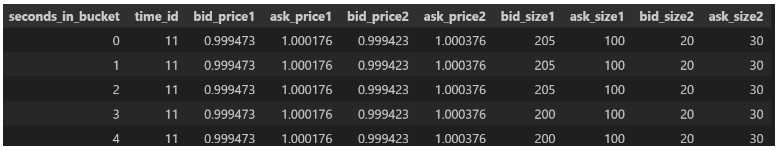
The order book data is given as below:



time_id	seconds_in_bucket	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2
11	0	0.999473	1.000176	0.999423	1.000376	205	100	20	30
11	3	0.999473	1.000176	0.999423	1.000376	200	100	20	30
11	13	0.999473	1.000326	0.999423	1.000376	200	100	20	30
11	18	0.999473	1.000025	0.999423	1.000376	200	100	20	30
11	28	0.999473	1.000326	0.999423	1.000376	200	100	20	30

Figure 3.1: Book data snapshot (time_id = 11, stock_id = 0)

As we notice in figure 5.5, some seconds in the seconds_in_bucket column are missing (for example, between second 0 and second 3, which means that there was no change in the order book in the seconds 1 and 2). To consistently calculate and compare the different statistics over all the stocks and time_ids, we suggest filling the gaps with the same information of the last provided second (for example, for seconds 1 and 2, we complete the other features with the information of second 0), as shown in the figure 3.2 below:



seconds_in_bucket	time_id	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2
0	11	0.999473	1.000176	0.999423	1.000376	205	100	20	30
1	11	0.999473	1.000176	0.999423	1.000376	205	100	20	30
2	11	0.999473	1.000176	0.999423	1.000376	205	100	20	30
3	11	0.999473	1.000176	0.999423	1.000376	200	100	20	30
4	11	0.999473	1.000176	0.999423	1.000376	200	100	20	30

Figure 3.2: Book data snapshot (time_id = 11, stock_id = 0) after filling the missing seconds

As a result, all the order_book data of all the stocks contain seconds from 0 to 600 seconds in each bucket.

3.2. Calculate the Order book statistics

We can derive many statistics from raw order book data to reflect market liquidity and stock valuation. These stats are proven to be fundamental inputs of any market prediction algorithms. We selected two common and essential stats to include in our solution:

3.2.1. Weighted Average Price (WAP)

The order book is also one of the primary sources for stock valuation. Given the order book, a fair stock valuation considers the level and the size of orders; that's why we used the weighted average price, or WAP, to calculate the instantaneous stock.

The formula of WAP can be written as below, which takes the top-level price and volume information into account:

$$WAP = \frac{BidPrice_1 * AskSize_1 + AskPrice_1 * BidSize_1}{BidSize_1 + AskSize_1} \tag{3.1}$$

The Weighted Average Price plot for stock_id = 43 and time_id = 5 is given below:

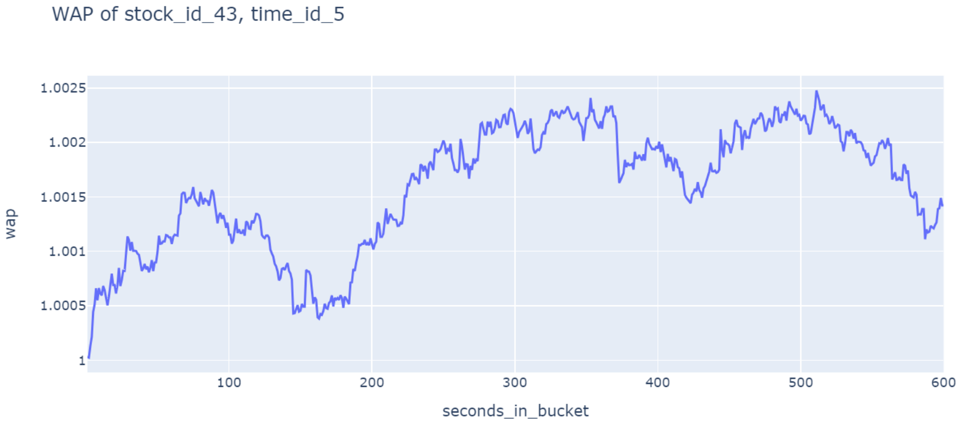


Figure 3.3: Weighted Average Prices of stock_id_43, time_id_5

3.2.2. Log-Return

Returns are widely used in finance; however, **log returns** are preferred whenever some mathematical modeling is required. Calling S_t the price of the stock S at time t, we can define the log return between t1 and t2 as:

$$r_{t_1,t_2} = log\left(\frac{S_{t_2}}{S_{t_1}}\right) \tag{3.2}$$

The log return plot for stock_id = 43 and time_id = 5 is given below:

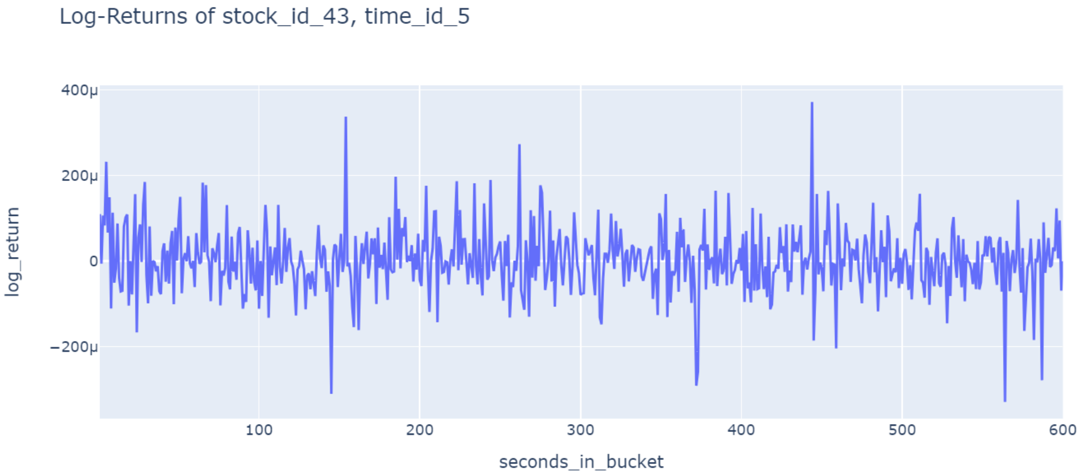


Figure 3.4: Log-Returns of stock_id_43, time_id_5

3.2.3. Bid Ask Spread

As different stocks trade on various levels on the market, we take the ratio of the best offer price and best bid price to calculate the bid-ask spread.

The formula of bid/ask spread can be written in below form:

$$BidAskSpread = \frac{AskPrice_1}{BidPrice_1} - 1 \tag{3.3}$$

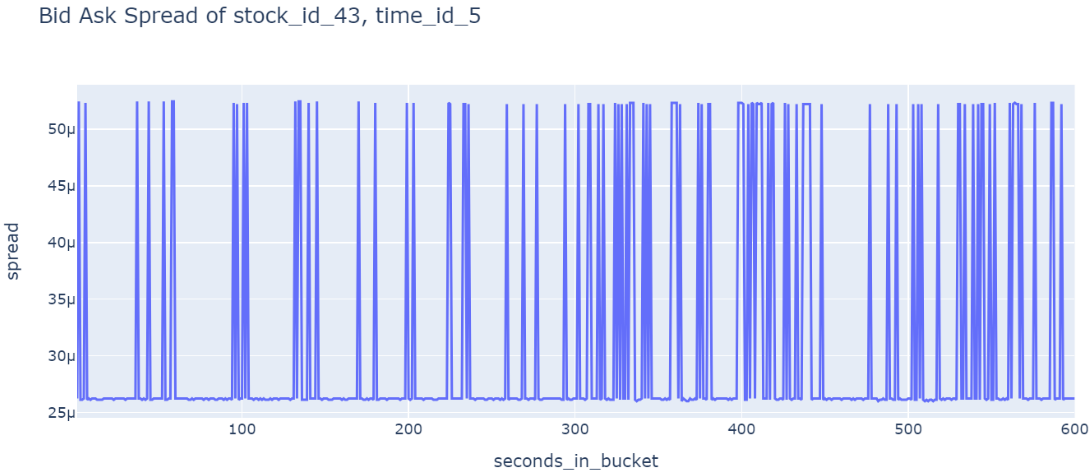


Figure 3.5: Bid Ask Spread of stock_id_43, time_id_5

3.2.4. Mid Price

The mid-price is the average of the best bid and ask prices. It represents the fair price at a given moment.

It's given by:

$$MidPrice = \frac{BidPrice_1 + AskPrice_1}{2} \tag{3.4}$$

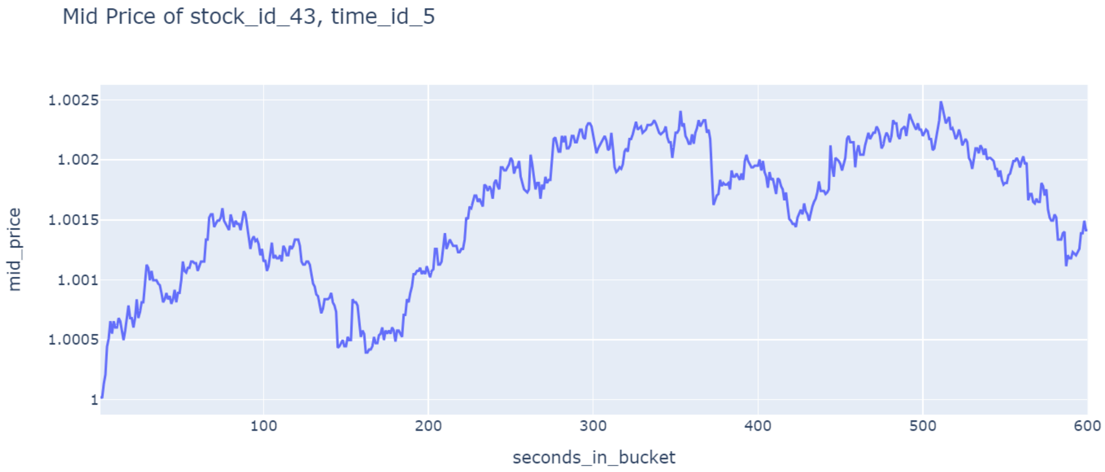


Figure 3.6: Mid Price of stock_id_43, time_id_5

3.2.5. Order Imbalance

Order imbalance measures the difference between the volumes of buy and sell orders, providing insight into market sentiment.

It's given by:

$$OrderImbalance = \frac{BidSize_1 - AskSize_1}{BidSize_1 + AskSize_1} \tag{3.5}$$

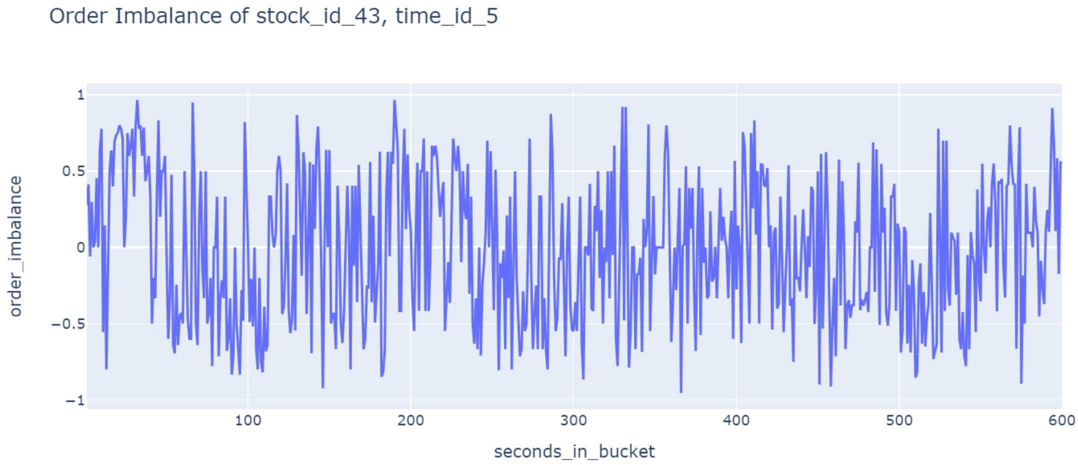


Figure 3.7: Order Imbalance of stock_id_43, time_id_5

3.2.6. Market Depth

Market depth can be measured by summing the sizes of the bid and ask orders at the top levels. This can indicate market liquidity.

It's given by:

$$MerketDepth = BidSize_1 + AskSize_1 \tag{3.6}$$

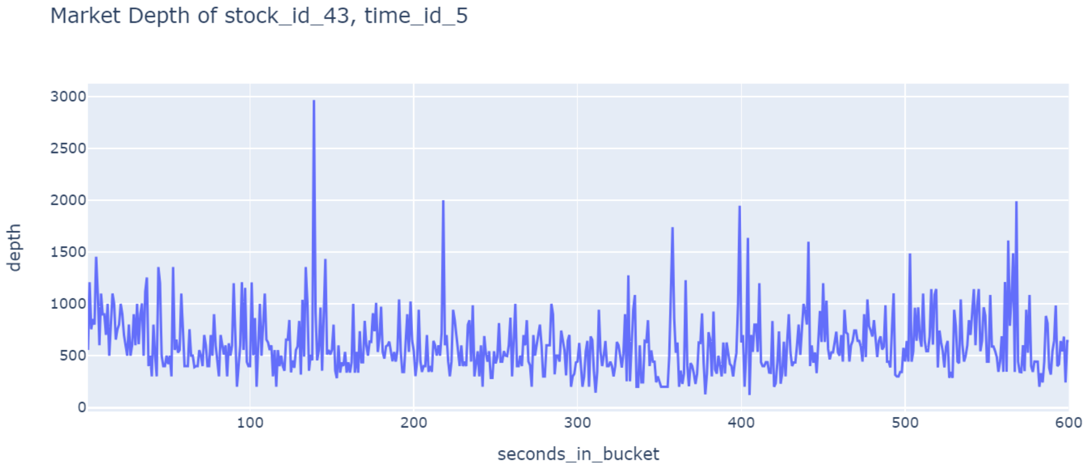


Figure 3.8: Market Depth of stock_id_43, time_id_5

3.2.7. Bid Ask Slope

This measures the difference in prices and sizes between the first and second levels of the order book, indicating how quickly the price changes with volume.

It's given by:

$$BidAskSlope = \frac{AskPrice_2 - AskPrice_1}{AskSize_1} - \frac{BidPrice_1 - BidPrice_2}{BidSize_1} \tag{3.7}$$

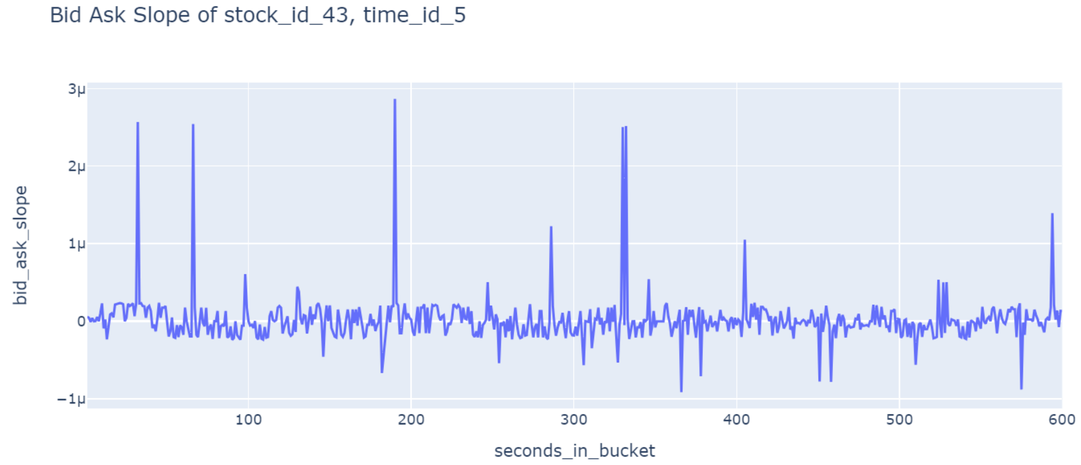


Figure 3.9: Bid Ask Slope of stock_id_43, time_id_5

The correlation matrix between the different features for stock_id = 43 and time_id = 11 is given below:

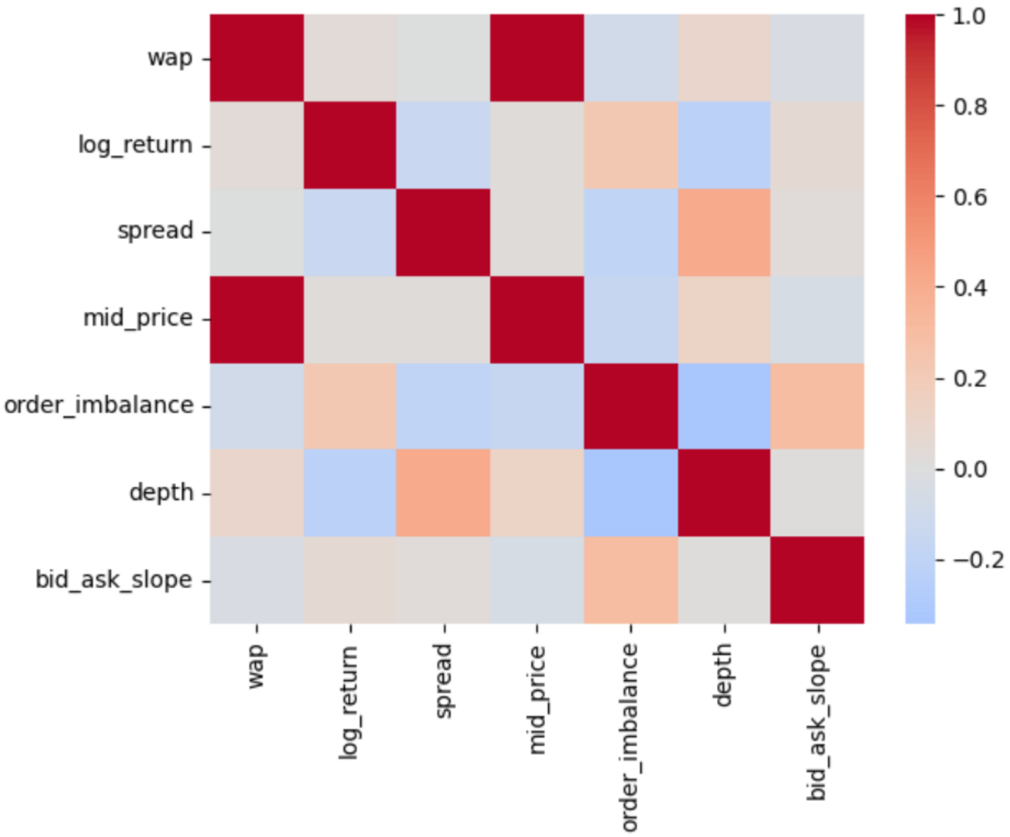


Figure 3.10: Correlation matrix between the different features for stock_id = 43 and time_id = 11

As we can see, the features are not highly correlated, except for the wap and mid_price, which makes sense since both features are used for the same purpose: to get a fair price.

3.3. Select the investment universe

We have strategically selected 10 stocks from a pool of 112 available stocks to streamline stock management and enhance algorithmic optimization efficiency. This targeted approach improves portfolio performance by focusing on high-quality, liquid stocks, enabling more precise and effective optimization and monitoring. The selection was based on trading data for the various stocks. **Here, we do not want to use the feature from the order book, and this exclusion is crucial to prevent potential biases in model training and to ensure the independence of distinct and vital steps within our methodology.**

According to the research conducted by Ahuja (2015) [1], Irala and Patil (2007) [3], and Domian et al. (2003) [2], a portfolio size of 10-20 stocks is frequently recommended. This range strikes an optimal balance among diversification benefits, cost efficiency, and management feasibility. We opted for the minimalistic approach by selecting 10 stocks for our project. The following outlines the step-by-step process employed in choosing these 10 stocks for investment:

1. Calculate key metrics for each stock. For each stock, within each time_id, we compute the following metrics:

    - **Average Return**: Measure of the average return from the prices of the executed trades in the different stocks; this measures the performance of the stocks.
    - **Volatility of Returns**: Measure of risk of the stocks.
    - **Average Trade Size**: Measure of liquidity of the stocks.
    - **Order Count**: Measure of market participation.

2. Normalize the Metrics:

   We normalize these metrics to bring them onto a standard scale using min-max normalization:

   $$X' = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

   Where:

   - X is the original value.
   - X' is the normalized value.
   - X_{min} is the minimum value of the feature.
   - X_{max} is the maximum value of the feature.

3. Compute Composite Score:

   We combine the normalized metrics into a single composite score:

   $Score = \mathbf{w_1}.Average\_Return + \mathbf{w_2}.Volatility\_of\_Return + \mathbf{w_3}.Total\_Size + \mathbf{w_4}.Order\_Count$

4. Aggregate Scores Across Periods:

   We compute an overall score for each stock by averaging its scores across all time_ids.

Weights	**Values**
w_1	0.4
w_2	0.2
w_3	0.2
w_4	0.2

   Typically, determining the optimal combination of weights involves training a model with the four metrics to derive the best possible allocation. However, we aim to maintain simplicity and practicality in this instance, avoiding unnecessary complexity at each step. Therefore, we have chosen a straightforward and reasonable set of weights that balances the critical factors without complicating the process.

5. Rank Stocks and Select Top 10:

   We rank the stocks based on their scores and select the top 10.

Result

After performing these steps, we obtain the 10 stock_ids to invest in:

selected_stocks = {43, 69, 124, 29, 31, 50, 111, 32, 41, 47}.

The investment universe is reduced to these 10 selected stocks for all that follows.

4. Optimal Portfolio Computation

After pre-processing the datasets (order and trade books) of the first 10-minute window, we aim to generate the targets for our model, which are the optimal weights for the 10 stocks in **the second 10-minute period**.

To do that, we used two traditional approaches, the Markowitz and the Hierarchical Risk Parity Approaches.

4.1. Markowitz Approach

To compute the optimal weights using the Markowitz approach, we require the expected returns of the assets and the covariance matrix of their returns for the second 10-minute window. We generate these parameters based on the Realized Volatilities provided in **train.csv** file.

4.1.1. Generate the expected returns

To generate the expected returns, we will use the Black & Scholes model for the stock price evolution. Since the log-returns are assumed to have a 0 mean (which is also verifiable in the provided order_books), the stock prices evolution is given by:

$$dln(S_t) = \sigma dB_t \tag{4.1}$$

With:

- σ: The 1 second volatility of the log-returns.
- B_t: a standard Brawnian Motion.

We also know that the Realized volatiVolatilityven by:

$$d(RV^2) =< dln(S_t) >= \sigma^2 dt \tag{4.2}$$

And since σ is constant and the given Realized VolatiVolatilityr 10 mins with a 1-second frequency of returns, σ is then given by:

$$\sigma = \frac{RV}{\sqrt{600}} \tag{4.3}$$

This enables us to simulate synthetic expected returns of the different stocks for the second 10-minute window using Monte Carlo.

Here is a step-by-step process to generate the expected return of a stock in the second 10 mins period for a given Bucket for which the target Realized Volatility is RV:

1. Simulate the 10 mins returns:

   Using the Black & Scholes model, we have that the log returns are normally distributed:

   $$ln(S_{t+1}) - ln(S_t) = \sigma(B_{t+1} - B_t) \sim \mathcal{N}\left(0, \frac{RV^2}{600}\right)$$

   We also simulate multiple 10-minute returns (the number of returns is the number of the Monte Carlo simulations). Since we are working with log-returns, the 10 mins returns are given by the sum of the 1-second returns (600 one-second returns), which means that the 10 mins returns $\sim \mathcal{N}(0, RV^2)$.

2. Calculate the expected return:

   We average the 10 mins returns obtained in the previous step to get the synthetic expected return of the stock.

4.1.2. Calculate the covariance matrix

To calculate the covariance matrix for the second 10-minute period, we need the **correlation matrix** and the **volatilities** of the log returns for this period.

- **Correlations between the log-returns**: Since the returns of the first 10 mins period are not available, a reasonable choice is to use those of the first 10 mins period to calculate the correlations since they represent a good indicator of the links between the stocks.
- **vector of volatilities D**: Since we have the realized volatilities of the different stocks for the second 10 mins period (for each bucket), we use the formula 4.3 to calculate the vector of volatilities D.

Given these two values, the **covariance matrix** of each bucket (for the second 10 mins period) is calculated as follows:

$$\Sigma = D.Corr.D^T \tag{4.4}$$

4.1.3. Calculate the Optimal Weights

To calculate the optimal portfolio weights using the Markowitz approach, we use the algorithm defined in the "compute_optimal_portfolio" function (in the provided code). This function computes the optimal weights by solving a quadratic programming problem, balancing between maximizing returns and minimizing risk (variance). Here is the QP formulation of the mean-variance optimization problem [5]:

$$x^* = \arg\min \frac{1}{2}x^T\Sigma x - \gamma x^T r \tag{4.5}$$

$$\text{s.t.} \begin{cases} \sum_{i=1}^n x_i = 1 \\ 0 \leq x_i \leq 1 \end{cases} \tag{4.6}$$

With:

- **x**: Represents the weights vector.
- Σ: Represents the covariance matrix.
- **r**: Represents the expected returns vector.
- γ: Represents the risk aversion parameter (explained below).

The key steps of the algorithm are as follows:

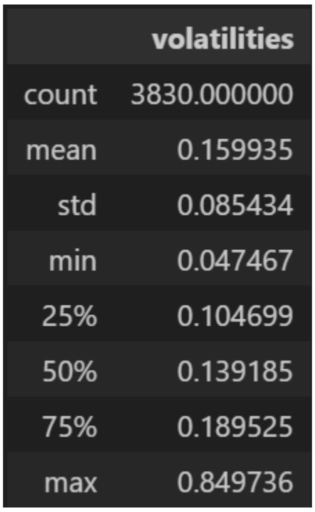
1. **Define Optimization Variable**: Define the portfolio weights as the optimization variable.
2. **Objective Function**: Define the objective function to minimize the quadratic form of the weights (representing variance) minus γ times the expected returns.
3. **Constraints**: Define constraints to ensure that the sum of the weights equals 1 (to invest all the capital), and each weight lies between the specified bounds (we choose the lower and upper bounds 0 and 1 respectively, to not allow for short selling or leverage). Additionally, a constraint is added to limit the portfolio volatility to 100%.
4. **Solve the Problem**: Use a convex optimization solver to find the optimal weights.

Gamma and the Risk and Return Trade-Off

The parameter γ in the objective function plays a crucial role in determining the risk and return trade-off of the optimal portfolio:

- **Low Gamma**: A low value of γ implies high risk aversion. In this case, the optimization will prioritize minimizing the portfolio variance (risk) over maximizing returns. This results in a more conservative portfolio with lower potential returns but also lower risk.

  Below, we have a summary statistics table of the optimal portfolio's annualized volatilities for $\gamma = 0$ for the different time_ids:

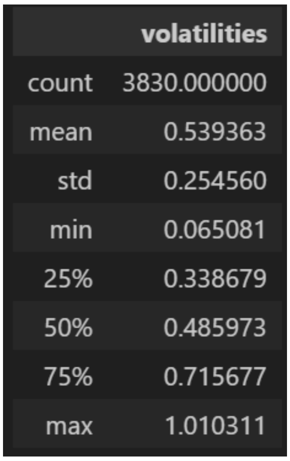
	volatilities
count	3830.000000
mean	0.159935
std	0.085434
min	0.047467
25%	0.104699
50%	0.139185
75%	0.189525
max	0.849736

  Figure 4.1: Descriptive statistics table of the optimal portfolios annualized volatilities for $\gamma = 0$.

  We notice that the optimal portfolio annualized volatility for 75% of the buckets (time_ids) is less than 19%. For the time_ids in the last quartile, since the stocks realized volatilities are high (for these time_ids), the obtained optimal portfolios still have a high volatility that reached 84%.

- **High Gamma**: A high value of γ implies low risk aversion. The optimization will prioritize maximizing the expected returns over minimizing the variance. This results in a more aggressive portfolio with higher potential returns and risk.

  Below, we have a summary statistics table of the optimal portfolio volatilities for $\gamma = 1$ for the different time_ids:

	volatilities
count	3830.000000
mean	0.539363
std	0.254560
min	0.065081
25%	0.338679
50%	0.485973
75%	0.715677
max	1.010311

  Figure 4.2: Descriptive statistics table of the optimal portfolios annualized volatilities for $\gamma = 1$.

  We notice that in this case, the optimal portfolios' annualized volatilities are higher than in the previous case (with $\gamma = 0$). We can see that for 50% of time_ids, the annualized volatilities of the optimal portfolios are higher than 48% and reach 100% for some of them.

4.2. Hierarchical Risk Parity Approach

One big problem with the traditional Markowitz problem is that we need to account for the inverse of the covariance matrix. At some point, the condition number is so high that numerical errors make the inverse matrix too unstable: A slight change in any entry will lead to a very different inverse. This is Markowitz's curse: The more correlated the investments, the greater the need for diversification, and yet the more likely we will receive unstable solutions. One of the reasons for the instability of quadratic optimizers we used above is that the vector space is modeled as a complete (fully connected graph), where every node is a potential candidate to substitute another .

Marcos López de Prado [4] addressed this problem using graph theory combined with machine learning. This The hierarchical Risk Parity method uses the information in the covariance matrix without requiring inversion or positive-de-nativeness.HRP can even compute a portfolio based on a singular covariance matrix. The algorithm operates in three main stages: tree clustering,quasi-diagonalization, and recursive bisection.

We need to note that this method does not directly optimize for maximum return with a given level of risk or minimize risk with a given level of return. Instead, it focuses on risk allocation through hierarchical clustering and recursive bisection, aiming to achieve balanced risk contributions across the portfolio. Here, we only consider the risk dimension. The main idea is that the performance dimension is too complicated to encompass because the forecasting step is generally not robust. This is why we focus on the patterns of the portfolio risk

I will state the main idea here with the corresponding code in **"HRP_class.py"** about the HRP class and link it with our portfolio.

Stage 1: Tree Clustering

Consider a $T \times N$ matrix of observations X, which returns a series of N variables over T periods. We aim to combine these N column vectors into a hierarchical structure of clusters.

Correlation Matrix and Distance

First, compute the $N \times N$ correlation matrix ρ_{ij} with entries $\rho_{ij} = \rho(X_{\cdot i}, X_{\cdot j})$. Define the distance measure:

$$d_{ij} = \sqrt{2(1 - \rho_{ij})}$$

This gives us a distance matrix $D = \{d_{ij}\}$.

Euclidean Distance of Correlation Distances

Compute the Euclidean distance between column-vectors of D:

$$d_{ij}^E = \left(\sum_{k=1}^{N} (d_{ik} - d_{jk})^2 \right)^{\frac{1}{2}}$$

Clustering Procedure

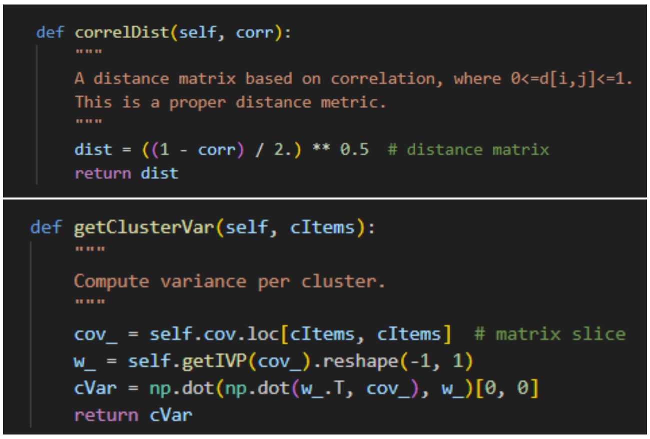
1. **Initialize**: Calculate initial distance matrix $\{d^E_{ij}\}$.
2. **Iterative Clustering**:

- Find the pair (i^*, j^*) that minimizes d^E_{ij}.
- Merge clusters i^* and j^* into a new cluster $u[1]$.
- Update distances using the linkage criterion:

$$d_{u[1],k} = \min\{d_{i^*k}, d_{j^*k}\}$$

- Update the distance matrix.

3. **Repeat** until all items are clustered.



```python
def correlDist(self, corr):
    """
    A distance matrix based on correlation, where 0<=d[i,j]<=1.
    This is a proper distance metric.
    """
    dist = ((1 - corr) / 2.) ** 0.5  # distance matrix
    return dist
```

```python
def getClusterVar(self, cItems):
    """
    Compute variance per cluster.
    """
    cov_ = self.cov.loc[cItems, cItems]  # matrix slice
    w_ = self.getIVP(cov_).reshape(-1, 1)
    cVar = np.dot(np.dot(w_.T, cov_), w_)[0, 0]
    return cVar
```

Figure 4.3: Code for Clustering Procedure

For our case, we will do this cluster for each correlation matrix of each time ID; we will represent one of our cluster links to the original covariance matrix for the selected stock of time ID 4132

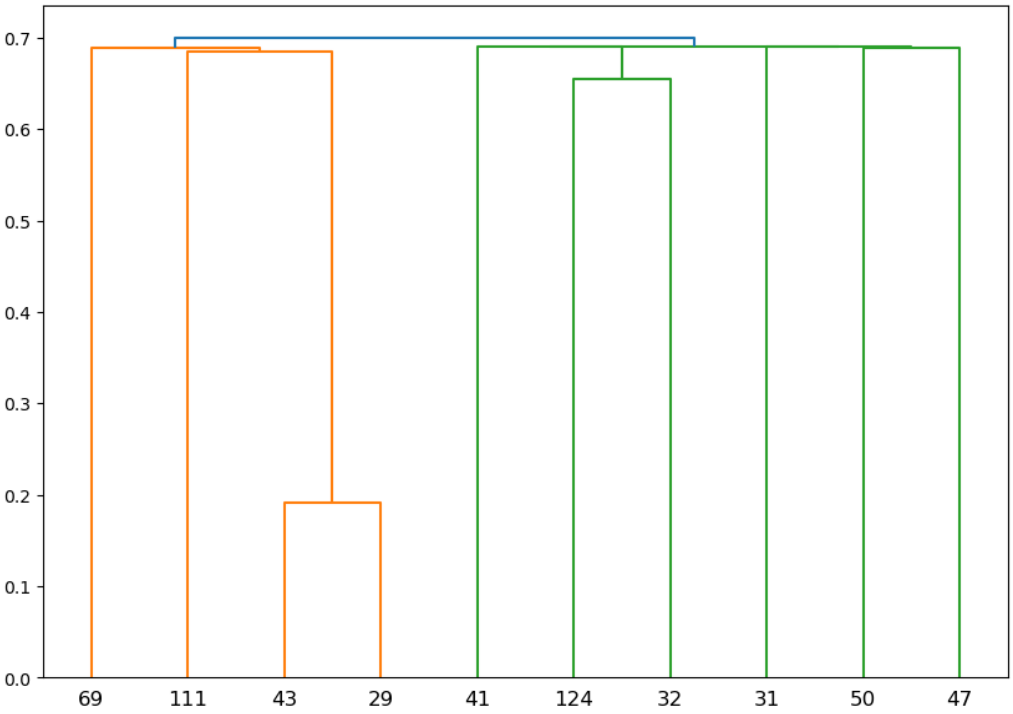


Figure 4.4: Dendogram of cluster formation for time ID 4132

- We have two main clusters here (69,11,43,29) and the other six stocks.
- The hierarchical structure shows that within the second group, series 32 and 124 form a sub-cluster, and within the first group, series 43 and 29 form another sub-cluster. This nested clustering indicates varying degrees of similarity within the larger group.

Stage 2: Quasi-Diagonalization

In this stage, we reorganize the rows and columns of the covariance matrix to ensure that the most significant values align along the diagonal. This process, known as quasi-diagonalization, does not require a change of basis and has the advantage of grouping similar investments while separating dissimilar ones. The algorithm operates as follows: each row of the linkage matrix merges two branches into one. We recursively replace clusters with their components until all clusters are resolved. This method preserves the original clustering order, resulting in a sorted list of the original (unclustered) items.

First, we have the original correlation matrix for time id 4132

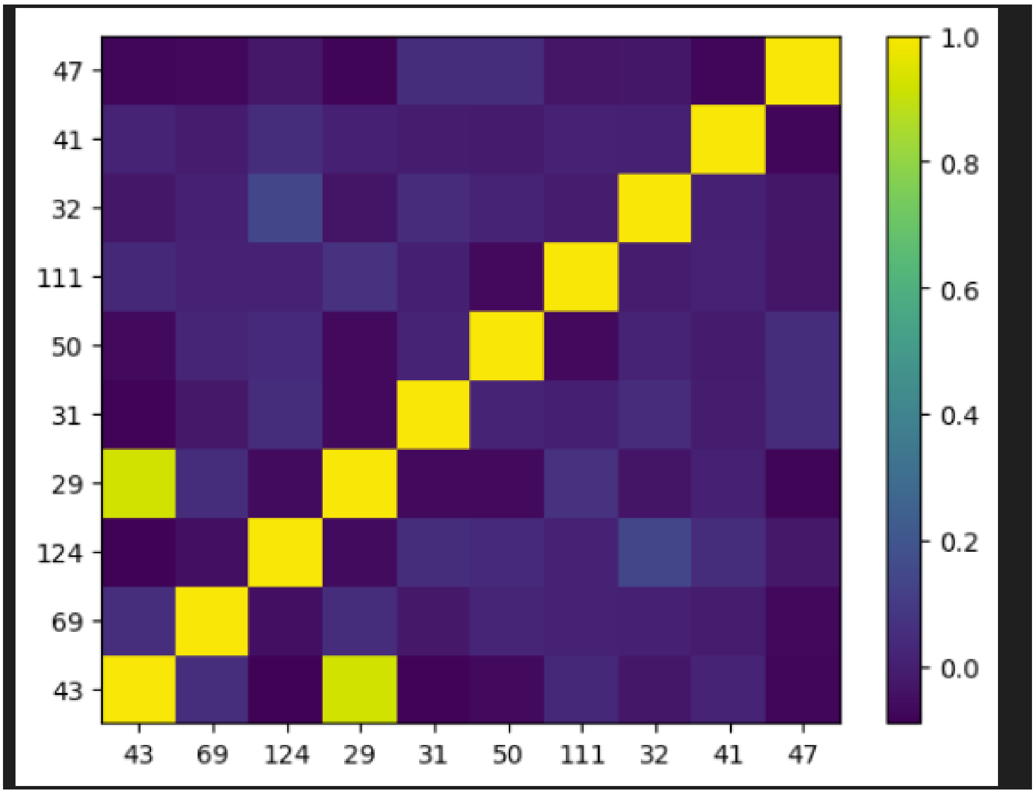
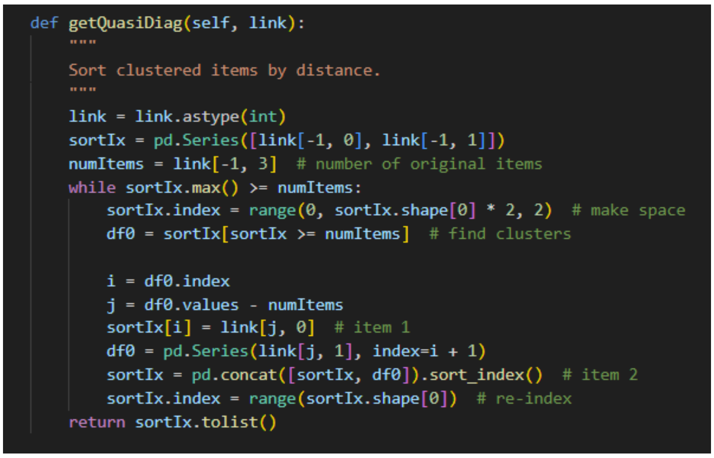


Figure 4.5: Original covariance matrix for time ID 4132



```python
def getQuasiDiag(self, link):
    """
    Sort clustered items by distance.
    """
    link = link.astype(int)
    sortIx = pd.Series([link[-1, 0], link[-1, 1]])
    numItems = link[-1, 3]  # number of original items
    while sortIx.max() >= numItems:
        sortIx.index = range(0, sortIx.shape[0] * 2, 2)  # make space
        df0 = sortIx[sortIx >= numItems]  # find clusters

        i = df0.index
        j = df0.values - numItems
        sortIx[i] = link[j, 0]  # item 1
        df0 = pd.Series(link[j, 1], index=i + 1)
        sortIx = pd.concat([sortIx, df0]).sort_index()  # item 2
        sortIx.index = range(sortIx.shape[0])  # re-index
    return sortIx.tolist()
```

Figure 4.6: Code for Quasi- Diagonalization

Then, we quasi-diagonalize the matrix for time ID 4132 using the code in the HRP class:

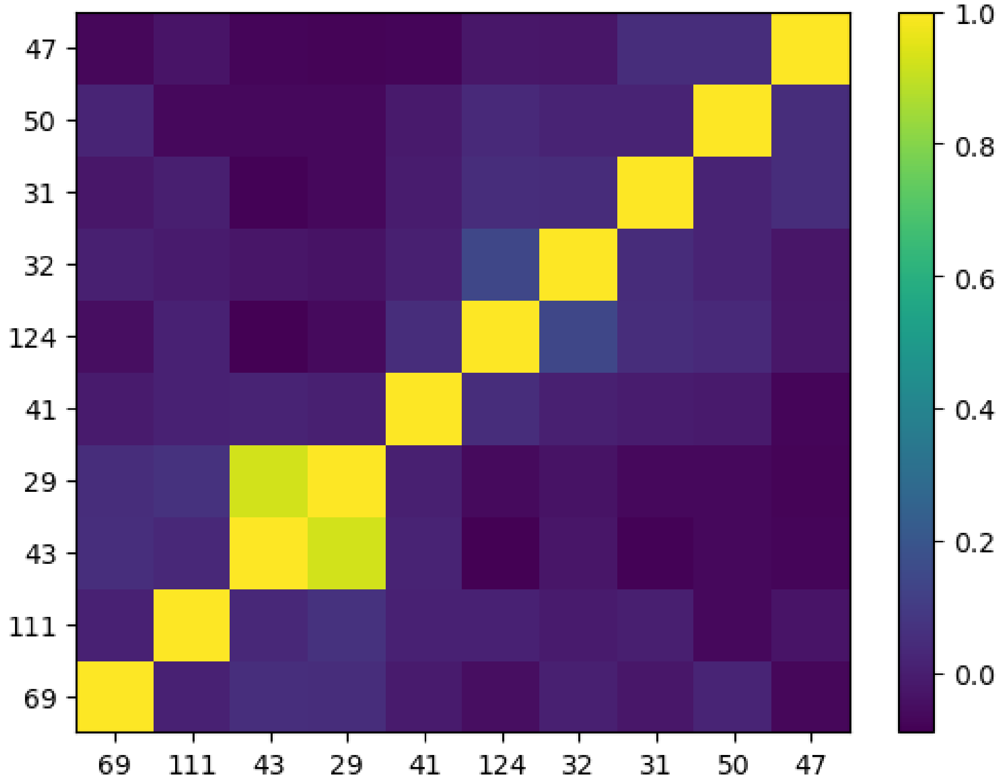


Figure 4.7: Clustered covariance matrix for time ID 4132

Stage 3: Recursive Bisection

Before continuing to Stage 3, we will state the optimal weight allocation for an Inverse Variance Portfolio. This approach is optimal when the covariance matrix is diagonal. The problem can be framed as a quadratic optimization problem:

$$\min_{\omega} \omega^\top V \omega \quad \text{s.t.} \quad \omega^\top \mathbf{a} = 1_I$$

With the solution:

$$\omega = \frac{V^{-1}\mathbf{a}}{\mathbf{a}^\top V^{-1}\mathbf{a}}$$

For the characteristic vector $\mathbf{a} = \mathbf{1}_N$, this results in the minimum variance portfolio. If V is diagonal, $\omega_n = \frac{V_{n,n}^{-1}}{\sum_{i=1}^{N} V_{i,i}^{-1}}$. For $N = 2$:

$$\omega_1 = \frac{\frac{1}{V_{1,1}}}{\frac{1}{V_{1,1}} + \frac{1}{V_{2,2}}} = 1 - \frac{V_{1,1}}{V_{1,1} + V_{2,2}}$$

This formulation shows how Stage 3 splits weights between two bisections of a subset, ensuring an optimal allocation based on the subset variances.

Stage 2 has produced a quasi-diagonal matrix. As mentioned above, the inverse-variance allocation is optimal for this diagonal covariance matrix. We utilize this in two ways: bottom-up to define variance for continuous subsets and top-down to split allocations.

Algorithm

1. **Initialization**:

- Set the list of items: $L = \{L_0\}$ with $L_0 = \{n\}_{n=1,\dots,N}$.
- Assign equal weight to all items: $w_n = 1$ for $n = 1, \dots, N$.

2. **Check for Single Item**:
   - If $|L_i| = 1$ for all $L_i \in L$, then stop.
3. **Bisect and Allocate**:
   - Bisect each L_i into two subsets if $|L_i| > 1$.
   - Define the variance for each subset:

$$V_{L_i^{(j)}} = \tilde{w}_{L_i^{(j)}}^\top V_{L_i} \tilde{w}_{L_i^{(j)}}$$

   where $\tilde{w}_{L_i^{(j)}}$ is a vector of weights for the subset $L_i^{(j)}$, normalized by their diagonal elements and trace.

   - Compute the split factor:

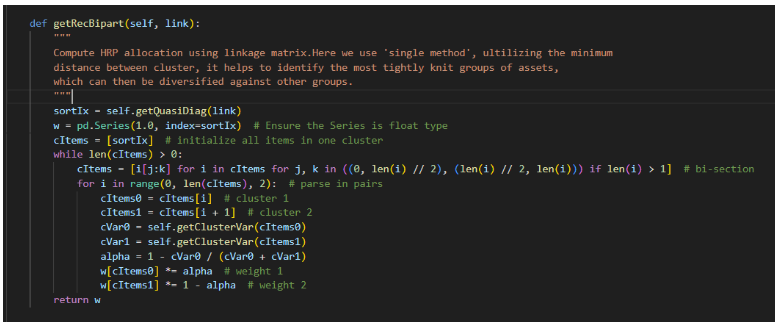
$$\alpha_i = 1 - \frac{V_{L_i^{(1)}}}{V_{L_i^{(1)}} + V_{L_i^{(2)}}}$$

   - Re-scale allocations for each subset:

$$w_n \leftarrow w_n \times \alpha_i \quad \text{for all } n \in L_i^{(1)}$$

$$w_n \leftarrow w_n \times (1 - \alpha_i) \quad \text{for all } n \in L_i^{(2)}$$

4. **Repeat**: Loop back to step 2 until all items are singletons.



```python
def getRecBipart(self, link):
    """
    Compute HRP allocation using linkage matrix.Here we use 'single method', ultilizing the minimum
    distance between cluster, it helps to identify the most tightly knit groups of assets,
    which can then be diversified against other groups.
    """
    sortIx = self.getQuasiDiag(link)
    w = pd.Series(1.0, index=sortIx)  # Ensure the Series is float type
    cItems = [sortIx]  # initialize all items in one cluster
    while len(cItems) > 0:
        cItems = [i[j:k] for i in cItems for j, k in ((0, len(i) // 2), (len(i) // 2, len(i))) if len(i) > 1]  # bi-section
        for i in range(0, len(cItems), 2):  # parse in pairs
            cItems0 = cItems[i]  # cluster 1
            cItems1 = cItems[i + 1]  # cluster 2
            cVar0 = self.getClusterVar(cItems0)
            cVar1 = self.getClusterVar(cItems1)
            alpha = 1 - cVar0 / (cVar0 + cVar1)
            w[cItems0] *= alpha  # weight 1
            w[cItems1] *= 1 - alpha  # weight 2
    return w
```

Figure 4.8: code to obtain the optimal weight of HRP

Objective and Benefits

Initial IVP Definition: Start with an equal weight allocation, similar to an Inverse Variance Portfolio (IVP).
Adjustment Based on Clustering: Weights are adjusted based on the clustering structure derived from the covariance matrix.
Optimization: The recursive bisection optimizes the allocation by refining the weight distribution, ensuring the final allocation reflects the data's underlying structure.

We will show a weight based on both methods for time id 4132. Here, we can see quite a clear pattern:

- Markowitz's approach concentrates weights on a few investments, hence becoming exposed to idiosyncratic shocks
- Otherwise, the HRP approach compromises between diversifying across all investments and diversifying across clusters, which makes it more resilient against not only idiosyncratic shocks but also systematic shocks

Stock	HRP Weights	Markowitz Weights
43	0.205146	0.00
69	0.089586	0.00
124	0.060147	0.00
29	0.173880	0.00
31	0.057380	0.00
50	0.114950	0.00
111	0.101603	0.00
32	0.080386	0.83
41	0.019713	0.00
47	0.097209	0.17

Table 4.1: Asset Weights Comparison

5. Forecasting of the Optimal Portfolio

5.1. Overview

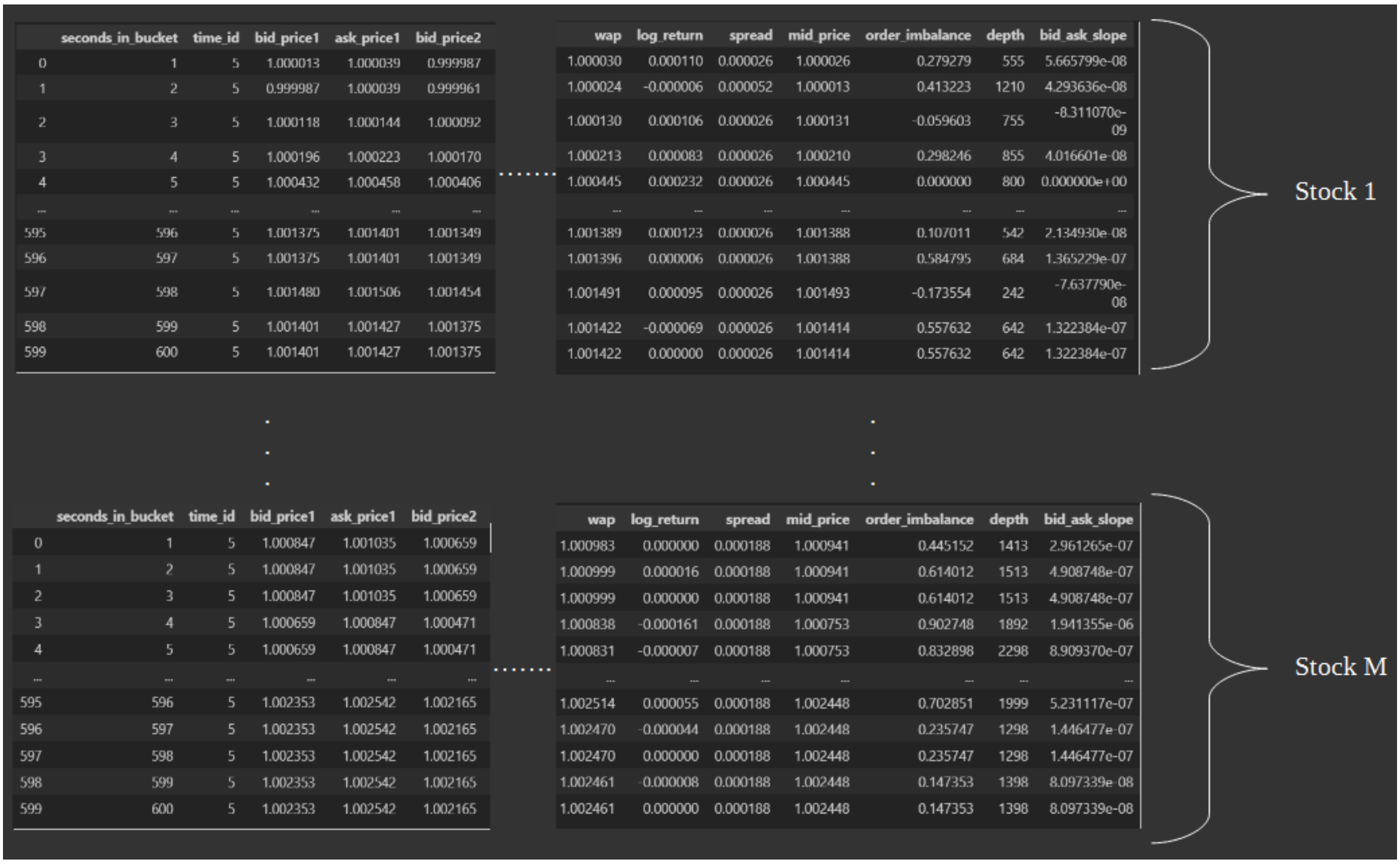
In the previous part, we approximated the optimal weights (by Markovitz, Hierarchical Risk Parity Approach, etc.) for all the time is $t_i, \forall i \in [1, 3830]$ and for a certain number of stocks $M << 112$. Indeed, we first choose to model the market with $M << 112$ because the computations may be heavy, and we want to analyze our results quickly.

That gives the following target weights :

$$\hat{w_i} = [\omega_1, \omega_2, ..., \omega_M], i = 1, 2, ..., 3830$$

The goal is to build a model to predict these weights $\hat{w_i}$ having the first ten minutes corresponding data X_i as an input. If we manage to do that, we get rid of the Monte Carlo simulations.

For every time id t_i (i.e., every time bucket), we have the following input data:



	seconds_in_bucket	time_id	bid_price1	ask_price1	bid_price2	wap	log_return	spread	mid_price	order_imbalance	depth	bid_ask_slope
0	1	5	1.000013	1.000039	0.999987	1.000030	0.000110	0.000026	1.000026	0.279279	555	5.665799e-08
1	2	5	0.999987	1.000039	0.999961	1.000024	-0.000006	0.000052	1.000013	0.413223	1210	4.293636e-08
2	3	5	1.000118	1.000144	1.000092	1.000130	0.000106	0.000026	1.000131	-0.059603	755	-8.311070e-09
3	4	5	1.000196	1.000223	1.000170	1.000213	0.000083	0.000026	1.000210	0.298246	855	4.016601e-08
4	5	5	1.000432	1.000458	1.000406	1.000445	0.000232	0.000026	1.000445	0.000000	800	0.000000e+00
...	...	...	...	...	...	...	...	...	...	...	...	...
595	596	5	1.001375	1.001401	1.001349	1.001389	0.000123	0.000026	1.001388	0.107011	542	2.134930e-08
596	597	5	1.001375	1.001401	1.001349	1.001396	0.000006	0.000026	1.001388	0.584795	684	1.365229e-07
597	598	5	1.001480	1.001506	1.001454	1.001491	0.000095	0.000026	1.001493	-0.173554	242	-7.637790e-08
598	599	5	1.001401	1.001427	1.001375	1.001422	-0.000069	0.000026	1.001414	0.557632	642	1.322384e-07
599	600	5	1.001401	1.001427	1.001375	1.001422	0.000000	0.000026	1.001414	0.557632	642	1.322384e-07

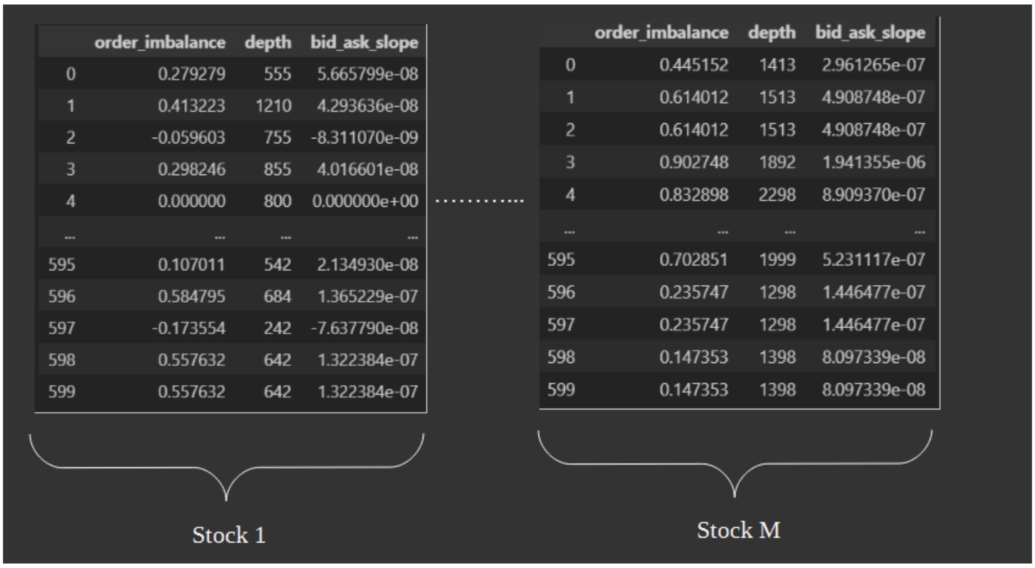
	seconds_in_bucket	time_id	bid_price1	ask_price1	bid_price2	wap	log_return	spread	mid_price	order_imbalance	depth	bid_ask_slope
0	1	5	1.000847	1.001035	1.000659	1.000983	0.000000	0.000188	1.000941	0.445152	1413	2.961265e-07
1	2	5	1.000847	1.001035	1.000659	1.000999	0.000016	0.000188	1.000941	0.614012	1513	4.908748e-07
2	3	5	1.000847	1.001035	1.000659	1.000999	0.000000	0.000188	1.000941	0.614012	1513	4.908748e-07
3	4	5	1.000659	1.000847	1.000471	1.000838	-0.000161	0.000188	1.000753	0.902748	1892	1.941355e-06
4	5	5	1.000659	1.000847	1.000471	1.000831	-0.000007	0.000188	1.000753	0.832898	2298	8.909370e-07
...	...	...	...	...	...	...	...	...	...	...	...	...
595	596	5	1.002353	1.002542	1.002165	1.002514	0.000055	0.000188	1.002448	0.702851	1999	5.231117e-07
596	597	5	1.002353	1.002542	1.002165	1.002470	-0.000044	0.000188	1.002448	0.235747	1298	1.446477e-07
597	598	5	1.002353	1.002542	1.002165	1.002470	0.000000	0.000188	1.002448	0.235747	1298	1.446477e-07
598	599	5	1.002353	1.002542	1.002165	1.002461	-0.000008	0.000188	1.002448	0.147353	1398	8.097339e-08
599	600	5	1.002353	1.002542	1.002165	1.002461	0.000000	0.000188	1.002448	0.147353	1398	8.097339e-08

Figure 5.1: Available data X_i for a certain time id t_i

5.2. An example of Data filtering

The first step is to filter our input data. Indeed, we do not need all the columns as some themes are highly correlated or irrelevant.

For example, we believe that the most relevant columns are the order imbalance, the depth, and the bid-ask slope (of course, this is debatable, but as the number of possible combinations of variables is high, we must make choices this way).



	order_imbalance	depth	bid_ask_slope
0	0.279279	555	5.665799e-08
1	0.413223	1210	4.293636e-08
2	-0.059603	755	-8.311070e-09
3	0.298246	855	4.016601e-08
4	0.000000	800	0.000000e+00
...	...	...	...
595	0.107011	542	2.134930e-08
596	0.584795	684	1.365229e-07
597	-0.173554	242	-7.637790e-08
598	0.557632	642	1.322384e-07
599	0.557632	642	1.322384e-07

	order_imbalance	depth	bid_ask_slope
0	0.445152	1413	2.961265e-07
1	0.614012	1513	4.908748e-07
2	0.614012	1513	4.908748e-07
3	0.902748	1892	1.941355e-06
4	0.832898	2298	8.909370e-07
...	...	...	...
595	0.702851	1999	5.231117e-07
596	0.235747	1298	1.446477e-07
597	0.235747	1298	1.446477e-07
598	0.147353	1398	8.097339e-08
599	0.147353	1398	8.097339e-08

Figure 5.2: Previous data X_i once filtered

5.3. First model

Assume we decided to keep the three last columns, so we have the following data shape :

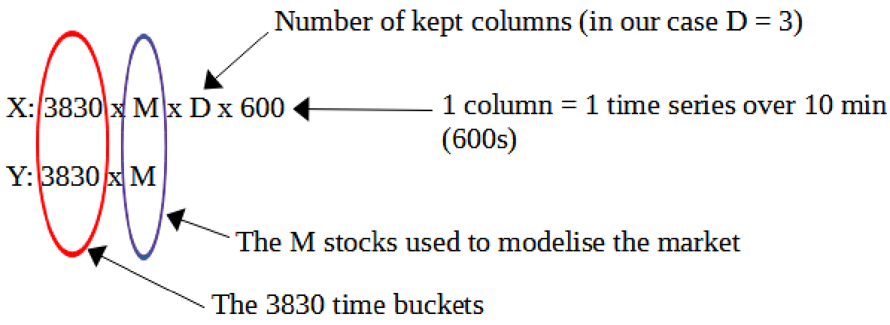

Number of kept columns (in our case D = 3)
X: 3830 x M x D x 600
Y: 3830 x M
1 column = 1 time series over 10 min (600s)
The M stocks used to modelise the market
The 3830 time buckets

Figure 5.3: Example of the shape of the filtered data

Since each time bucket represents an input, the goal is to predict an output vector of size M from a 3D input ($M \times D \times 600$).

An intuitive idea is to use a neural network (CNN) to achieve this goal. Remark that the input is analogous to 2D images (we can imagine that $M \times 600$ is the dimension, while $D = 3$ represents the colours). However, the output is a vector of size M to predict (usually, when working with images, we try to classify them here, but the task is much harder).

The appropriate activation functions for this task are:

- ReLU functions ($ReLU(x) = max(0, x)$) for the hidden layers. This is a common choice for most neural networks.

- SoftMax function for the output layer. For a vector $z = (z_1, ..., z_M)$, the softmax function σ is defined by :

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}} > 0$$

Moreover, $\sum_{i=1}^{M} \sigma(z)_i = 1$. Often used to predict probabilities, the softmax function ensures that the output vector consists of positive values that add up to 1. This normalization is exactly what is needed for our problem, as we require the predicted weights to be in a normalized form (i.e., weights that add up to 1).

In addition to choosing appropriate columns, the main challenge is tuning the model's parameters. In our implementation, we try to optimize:

- The number of layers

- The dropout level (between 0 and 1): During training, dropout randomly ”drops out” or sets to zero a fraction of the units (neurons) in the neural network. This means that each iteration of training may ignore a different set of neurons, forcing the network to not rely on specific neurons and instead learn more robust features, preventing overfitting.

- The learning rate that determines the step size at each iteration while moving toward a minimum of the loss function. It controls how much to change the model in response to the estimated error each time its weights are updated.

5.4. Training and model selection

The data (X, Y) is split between a train set and a test set (80 % train - 20 % test) (when we use the term ”validation set,” it refers to the test set). The parameters are optimized through grid research: for each combination of parameters, the model is trained on the train set, and then we look at the loss of the trained model on the test set and keep the combination with the lowest loss.

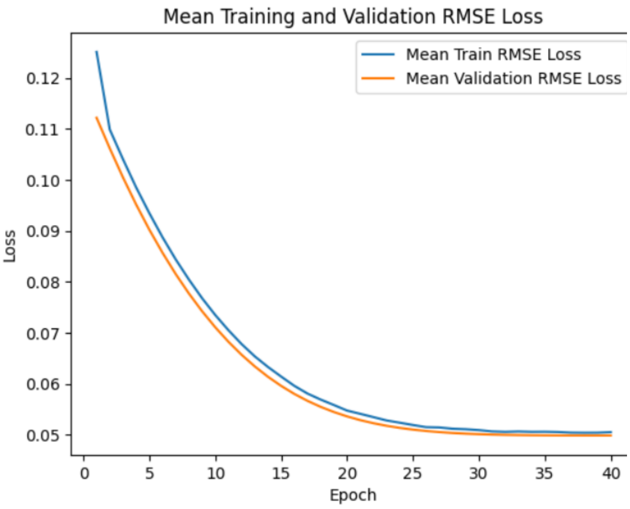
We used the following grid.

$$\text{Number of layers}: [1, 2, 3, 4]$$
$$\text{Dropout}: [0.5, 0.6, 0.7]$$
$$\text{Learning rate}: [0.001, 0.0001]$$

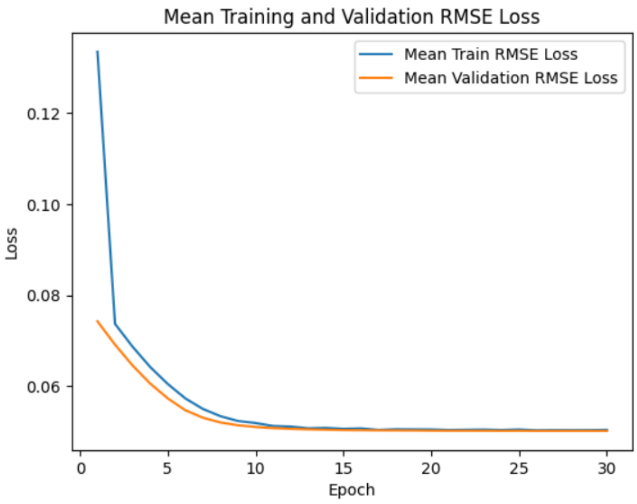
5.5. Results

Finding the best model is a non-trivial task, requiring finding the best combination between optimal data filtering and optimal parameters grid.

Our results are nonetheless promising and demonstrate that our methodology has considerable potential, provided the neural networks under study are further optimized. In the following part, we have $M = 10$. We also worked with weights generated from the Hierarchical Risk Approach. Here are some examples with different data filtering / neural networks:



(a) Num layers = 2, dropout = 0.6, learning rate = 0.0001
ReLU functions for hidden layers and output layer
Retained columns = order imbalance, depth, bid-ask slope



(b) Num layers = 5, dropout = 0.85, learning rate = 0.001
ReLU functions for hidden layers, SoftMax for output layer
Retained columns = Mid price, order imbalance, depth, bid-ask slope

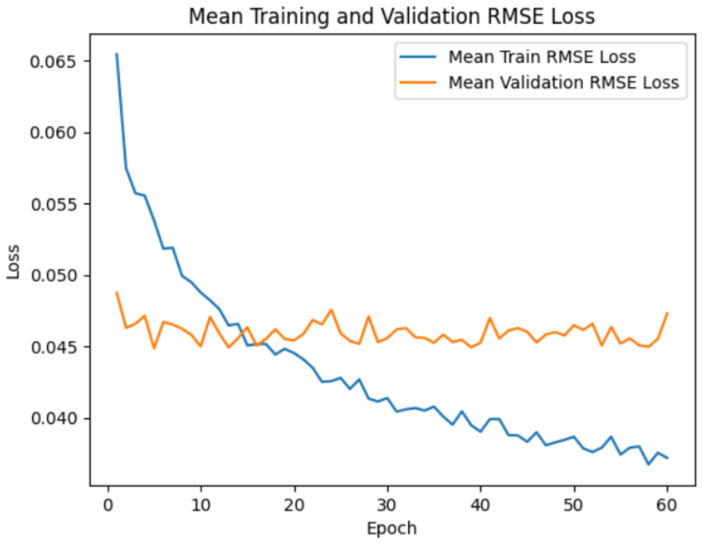
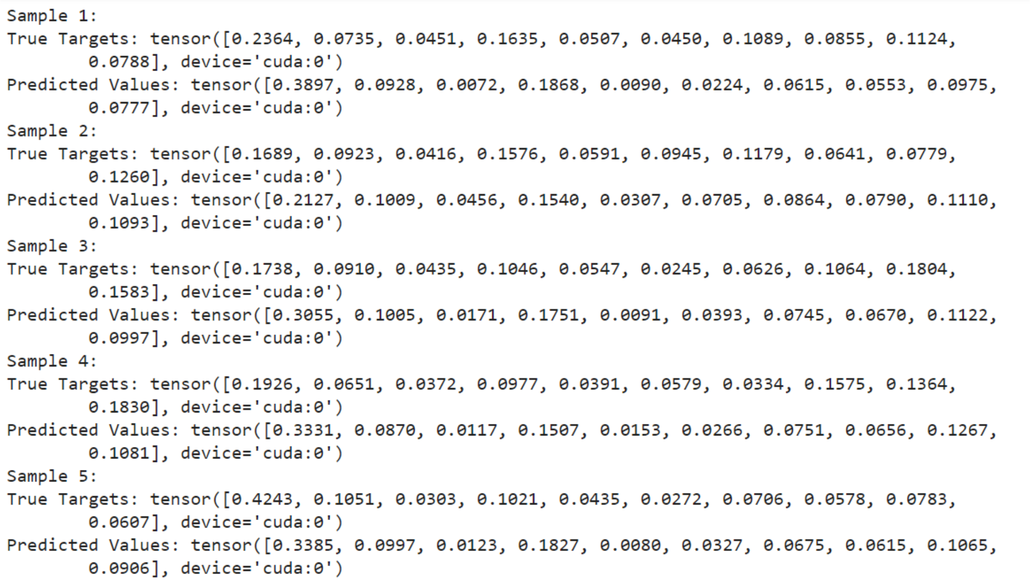


Figure 5.5: A good performance over test, but quite unstable
Num layers = 4, dropout = 0.5, learning rate = 0.0001
ReLU functions for hidden layers, SoftMax for output layer
Retained columns = Mid price, order imbalance, depth, bid-ask slope

Over each epoch, we compute the mean RMSE distance between the target and the predictions. We also tested some models with other activation functions (for example, ReLU); if the loss over the test set could seem satisfying, we have to be very careful as it does not ensure that weights are positive with a sum equal to 1, only the SoftMax function ensures coherent results.

The associated notebook is defined to work only with models of type (hidden layers: ReLU / output layer: SoftMax)

Here are some sample results to compare the true optimal weights with the predicted optimal weights:



```
Sample 1:
True Targets: tensor([0.2364, 0.0735, 0.0451, 0.1635, 0.0507, 0.0450, 0.1089, 0.0855, 0.1124,
        0.0788], device='cuda:0')
Predicted Values: tensor([0.3897, 0.0928, 0.0072, 0.1868, 0.0090, 0.0224, 0.0615, 0.0553, 0.0975,
        0.0777], device='cuda:0')
Sample 2:
True Targets: tensor([0.1689, 0.0923, 0.0416, 0.1576, 0.0591, 0.0945, 0.1179, 0.0641, 0.0779,
        0.1260], device='cuda:0')
Predicted Values: tensor([0.2127, 0.1009, 0.0456, 0.1540, 0.0307, 0.0705, 0.0864, 0.0790, 0.1110,
        0.1093], device='cuda:0')
Sample 3:
True Targets: tensor([0.1738, 0.0910, 0.0435, 0.1046, 0.0547, 0.0245, 0.0626, 0.1064, 0.1804,
        0.1583], device='cuda:0')
Predicted Values: tensor([0.3055, 0.1005, 0.0171, 0.1751, 0.0091, 0.0393, 0.0745, 0.0670, 0.1122,
        0.0997], device='cuda:0')
Sample 4:
True Targets: tensor([0.1926, 0.0651, 0.0372, 0.0977, 0.0391, 0.0579, 0.0334, 0.1575, 0.1364,
        0.1830], device='cuda:0')
Predicted Values: tensor([0.3331, 0.0870, 0.0117, 0.1507, 0.0153, 0.0266, 0.0751, 0.0656, 0.1267,
        0.1081], device='cuda:0')
Sample 5:
True Targets: tensor([0.4243, 0.1051, 0.0303, 0.1021, 0.0435, 0.0272, 0.0706, 0.0578, 0.0783,
        0.0607], device='cuda:0')
Predicted Values: tensor([0.3385, 0.0997, 0.0123, 0.1827, 0.0080, 0.0327, 0.0675, 0.0615, 0.1065,
        0.0906], device='cuda:0')
```

Figure 5.6: Comparision between the true and predicted optimal weights

We have plotted a comparison between the Sharpe ratios corresponding to the true optimal weights and the ones corresponding to the predicted optimal weights:

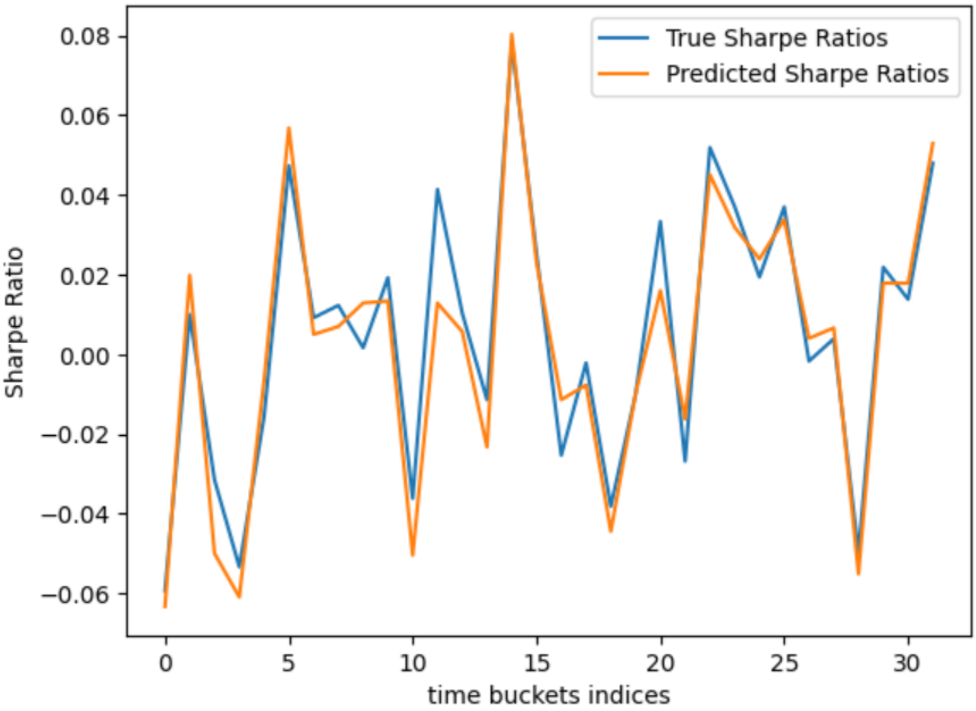


Figure 5.7: Comparision between the Sharpe ratios of the true and predicted optimal weights

We notice that the weights and the Sharpe ratios of the actual optimal portfolios are comparable to those of the predicted portfolios.

References

[1] Aakash Ahuja. Portfolio diversification in the karachi stock exchange. *Journal of Engineering and Technology*, 1, 2015.

[2] Dale L. Domian, David Louton, and Marie D. Racine. Portfolio diversification for long holding periods: How many stocks do investors need? *Studies in Economics and Finance*, 21:40–64, 2003.

[3] Lokanandha Reddy Irala and Prakash Patil. Portfolio size and diversification. *Monetary Economics*, 2007.

[4] Marcos Lopez de Prado. Building diversified portfolios that outperform out-of-sample. *Journal of Portfolio Management*, 2016.

[5] Roncalli Thierry. Portfolio allocation and asset management course. 2024.