Applying Machine Learning to Volatility Forecasting in Volatility-Managed Portfolios

Ilyas SBAI

Abstract

In this thesis, we replicate the volatility-managed portfolio strategy developed by DeMiguel, Martín-Utrera, and Uppal (2024), with a modification to the volatility timing mechanism. Instead of utilizing the market's current month's realized volatility, we employ the next month's predicted volatility to adjust the portfolio weights. The volatility predictions are generated using two advanced machine learning techniques: Long-Short-Term Memory (LSTM) Recurrent Neural Networks and ν-Support Vector Machine for Regression (ν-SVR). Our findings indicate that using predicted volatility does not enhance portfolio performance, likely due to estimation errors in the volatility forecasts.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mirco Rubin, for his guidance and support. I am also deeply thankful to Professor Raman Uppal for his enlightening course on Quantitative Portfolio Management and for his willingness to answer my questions during the preparation of this thesis.

1 Introduction

In finance, the fundamental principle that higher risk yields higher returns has long been accepted. This concept, deeply rooted in the Capital Asset Pricing Model (CAPM) introduced by Sharpe (1964), posits a direct relationship between risk and expected return. However, this traditional view has been challenged by recent research, particularly the work of Moreira and Muir (2017). They propose that by reducing exposure to risks during periods of high volatility, investors can achieve returns exceeding those of an unconditional strategy. They introduce the concept of volatility-managed factors, defined as $f^{\sigma}_{t+1} = \frac{c}{\sigma^2_t} \cdot f_{t+1}$, with f_{t+1} being the original factor return at time t, $\sigma^2_t(t)$ the factor's previous month's realized variance and c a constant such that f^{σ} has the same unconditional volatility as f. Their findings indicate that these volatility-managed factors exhibit significant betas and alphas relative to their original counterparts, even after adjusting for the Fama-French three factors. This suggests a weakening of the risk-return trade-off during high volatility periods.

The implications of such findings are profound for both academics and practitioners. For portfolio managers, incorporating volatility management into their strategies could mean achieving better risk-adjusted returns, which is crucial in today's unpredictable market environment. Building on this premise, DeMiguel, Martín-Utrera, and Uppal (2024) introduced multi-factor volatility-managed portfolios, incorporating transaction costs into their model, thus making them relevant for real-world use. They adjust each factor's weight according to market volatility, with weights decreasing variably as volatility increases. The portfolio at time t can thus be described by $w_t(\theta_{k,t}) = \sum_{k=1}^{K} x_{k,t}\theta_{k,t} = \sum_{k=1}^{K} x_{k,t}(a_{k,t} + \frac{b_{k,t}}{\sigma_t})$, where $x_{k,t}$ is the original factor, $a_{k,t}$ is the weight on the original factor, $b_{k,t}$ is the weight on the volatility managed factor ($\frac{x_{k,t}}{\sigma_t}$) and K the number of factors.

One critical aspect of this approach is the role of transaction costs, which tend to rise during periods of high volatility. By anticipating the rise of transaction costs, we could potentially improve the returns of portfolios net of transaction costs. This raises the question: could using the predicted next month's volatility help enhance the performance of multi-factor portfolios by reducing transaction costs? Moreover, can forecasting volatility help us anticipate periods when the risk-return trade-off breaks down, allowing for more strategic adjustments in portfolio management?

To explore these questions, we aim to replicate the methodology of DeMiguel, Martín-Utrera, and Uppal (2024) by reconstructing the factors used, modeling their transaction costs, and deriving weights through mean-variance optimization. We will then implement a machine learning model to predict next month's volatility as accurately as possible. Finally, we will apply the forecasted volatility in our mean-variance optimization.

Machine learning techniques, particularly Long Short Term Memory (LSTM) networks, have shown promise in forecasting financial time series due to their ability to capture temporal dependencies. For instance, Liu (2019) demonstrated the efficacy of non-linear regression models in predicting the volatility of the S&P 500, outperforming traditional models such as GARCH(1,1). However, our findings indicate that performances are not improved by using machine learning, possibly due to estimation error and poorer reactivity.

Our study will unfold as follows: first, we review relevant literature to provide a theoretical foundation and context for our research. Next, we describe the data used and the methodology followed, detailing the machine learning models and optimization techniques employed. Subsequently, we present our results, analyzing the performance of the volatility-managed portfolios against benchmarks. Finally, we conclude with insights derived from our findings.

2 Literature Review

Factor models were first introduced by Ross (1976), through the Arbitrage Pricing Theory (APT). APT posits that stock returns can be represented by a factor model expressed as: $R_i = \mathbb{E}[R_i] + \beta_{i,1}F_1 + \beta_{i,2}F_2 + \cdots + \beta_{i,K}F_K + \varepsilon_i$, where R_i is the stock return, $\mathbb{E}[R_i]$ the expected returns, F_k for $k \in 1, \ldots, K$ the common factors and β_k for $k \in 1, \ldots, K$ the sensitivity of the asset to these factors and ε_i an error term.

Expanding on factor models, Brandt, Santa-Clara, and Valkanov (2009) proposed their Parametric Portfolio Policies, which apply factor models to portfolio weights rather than returns. The portfolio weights are defined as: $w_{i,t} = \bar{w}_i + \frac{1}{N_t}\theta^T \widehat{x}_{i,t}$, where $\bar{w}_i$ is the weight in a benchmark portfolio, N_t is the number of stocks, $\widehat{x}_{i,t}$ are stock characteristics, akin to factors, and θ is a vector of weights. This method requires estimating fewer parameters compared to the APT-based portfolio construction, namely the weight coefficients in θ.

One specific approach to parameterizing these weights involves adjusting them for volatility, a concept explored by Moreira and Muir (2017) who introduced volatility-managed factors. They introduced volatility-managed factors defined as: $f^{\sigma}_{t+1} = \frac{c}{\sigma^2_t(t)} \cdot f_{t+1}$, where f_{t+1} is the original factor return at time t, $\sigma^2_t(t)$ the factor's previous month's variance and c a constant such that f^{σ} has the same unconditional volatility as f. They found that these factors achieved higher Sharpe ratios, attributing this to a lessened risk-return trade-off during high volatility periods. They also created volatility-managed portfolios by designing a mean-variance efficient portfolio, choosing the factor weights that maximise the in-sample Sharpe ratio of the portfolio: $F^{MVE}_t = \theta^T F_t$, where θ is the weight vector and F a vector containing the factor returns at time t. They then defined the volatility-timed Mean-Variant Efficient portfolio as $f^{MVE,\sigma}_{t+1} = \frac{c}{\sigma^2_t(F^{MVE}_{t+1})} F^{MVE}_{t+1}$, where c is again a normalising constant. Here, the relative weights of the portfolio (θ) remain constant. Their findings indicated that these volatility-managed portfolios produced significant positive alphas when regressed on the original MVE portfolios.

However, this approach is not implementable for a real-time investor. Cederburg et al. (2020) split the sample into a training period, in which the weights attributed to the volatility-managed and the original factor are determined, and a testing period. They find that volatility-managed strategy fail to systematically outperform the original factor, as measured by their Sharpe ratio. Similarly, Barroso and Detzel (2021) examined the "limits to arbitrage" and found that, excluding the market factor, most volatility-managed factors are not profitable after accounting for transaction costs.

Addressing these limitations, DeMiguel, Martín-Utrera, and Uppal (2024) proposed volatility-managed multifactor portfolios that allow relative factor weights to vary with volatility, which we will elaborate on in the next section. Transaction costs are applied after considering the netting of position across factors, which greatly reduces the turnover and thus cost. Their out-of-sample analysis showed that these portfolios outperformed (net of transaction costs) both the original volatility-managed multifactor portfolios and the market portfolio, achieving higher Sharpe ratios.

Regarding the Machine Learning part of this thesis, we build on Liu (2019) who forecasted next-day and third-day volatility of the S&P500 using non-linear regression methods, namely Long Short Term Memory Recurrent Neural Networks (LSTM) and ν-Support Vector Machine for

Regression. His results, compared with a GARCH(1,1) model, showed that non-linear regression methods were more accurate in almost all cases, as measured by Root Mean Square Error.

Similarly, Moon and Kim (2019) used LSTM to predict volatility of indices like the S&P 500 and NASDAQ. They discovered that increasing the number of predictors of the stock market's volatility did not necessarily improved accuracy; volatility itself was one of the best predictors, reinforcing the foundational premise of successful GARCH models (Bollerslev (1986)).

3 Data and Methodology

3.1 Data

We use CRSP's monthly and daily data, and Compustat annual and quarterly data for all stocks traded on the NYSE, AMEX and NASDAQ between January 1967 and December 2022. We drop stocks with a negative book-to-market ratio. Additionally, we use CRSP's daily market returns and 30-Day Bill Returns. Finally, we use the monthly risk-free rate provided by Kenneth French on his website. We use this data to replicate the 9 factors used in Moreira and Muir (2017). Additionally, we use CBOE's daily VIX close value between January 1990 and December 2022 in our volatility prediction model.

3.2 Factor Replication

We replicate the 5 factors from Fama and French (2015). For this, we only use stocks that have a share code of 10 or 11. With the exception of the $R_m - R_f$ factor, all portfolios are rebalanced in June and returns are computed from the following July until the end of the next year's June.

- $R_m - R_f$: return of all US-incorporated stocks, weighted by their previous month's market capitalisation. We directly use the risk-free rate provided on Kenneth French's website.
- SMB: return of a value-weighted portfolio of small stocks minus the return of a value-weighted portfolio of big stocks. Stocks are divided into small and big groups based on the median market capitalization of stocks incorporated on the NYSE to form the size characteristic. Market capitalization is calculated as Price $\times$ Shares Outstanding, aggregated at the PERMCO level.
- HML: return of a value-weighted double-sorted portfolio on size and book-to-market ratio.

We compute $HML = \frac{1}{2}(\text{Small Value} + \text{Big Value}) - \frac{1}{2}(\text{Small Growth} + \text{Big Growth})$. Stocks are divided into 3 groups representing the low 30% (Value), middle 40%, high 30% (Growth) of stocks based on the quantiles of the book-to-market ratio of NYSE-traded stocks. The book-to-market characteristic is computed as $\frac{\text{Book Equity in June of year } t}{\text{Market Equity as of December of year } t-1}$. Book Equity is defined as the book value of stockholders' equity, plus balance sheet deferred taxes and investment tax credit (if available), minus the book value of preferred stock.

- RMW: return of a value-weighted double-sorted portfolio on size and operating profitability. We compute $RMW = \frac{1}{2}(\text{Small Robust} + \text{Big Robust}) - \frac{1}{2}(\text{Small Weak} + \text{Big Weak})$. Stocks are divided into 3 groups representing the low 30% (Weak), middle 40%, high 30% (Robust) of stocks based on the quantiles of the operating profitability ratio of of NYSE-traded stocks. The operating profitability ratio is computed as $\frac{\text{Operating Profits as of June of year } t}{\text{Book Equity of year } t-1}$. Operating Profits is defined as Revenue minus COGS plus Interest Expense plus Selling, General, and Administrative Expense.

- CMA: return of a value-weighted double-sorted portfolio on size and investment. We compute $CMA = \frac{1}{2}(\text{Small Conservative} + \text{Big Conservative}) - \frac{1}{2}(\text{Small Aggressive} + \text{Big Aggressive})$. Stocks are divided into 3 groups representing the low 30% (Conservative), middle 40%, high 30% (Aggressive) of stocks based on the quantiles of the investment ratio of NYSE-traded stocks. The investment ratio is computed as $\frac{\text{Total Asset of year } t-1}{\text{Total Asset of year } t-2} - 1$.

We replicate the momentum factor from Carhart (1997), as described on Kenneth French's website, as the return of a value-weighted, double-sorted portfolio on size and prior returns. We compute $UMD = \frac{1}{2}(\text{Small High Returns+Big High Returns}) - \frac{1}{2}(\text{Small Low Returns+Big Low Returns})$. Stocks are divided into 3 groups representing the low 30% (Low returns), middle 40%, high 30% (High Returns) of stocks based on the quantiles of the operating profitability ratio of NYSE-traded stocks. For month t, we use returns from $t-2$ to $t-12$.

Additionally, we replicate the profitability and investment factors from Hou, Xue, and Zhang (2015). We triple-sort stocks based on size, Return on Equity (ROE) and Investment-to-Asset ratio. ROE is defined as $\frac{\text{Income before Extraordinary Items}}{\text{1-quarter Lagged Book Equity}}$. The size quantiles and Investment-to-Asset quantiles are the same as the size and Investment ratio from Fama and French (2015) and are similarly determined at the end of June every year using breakpoints determined from stocks

trading on the NYSE. At the beginning of each month, stocks are also independently divided into 3 groups representing the low 30% (Low ROE), middle 40%, high 30% (High ROE) of stocks based on the ROE quantiles of NYSE-traded stocks. Eighteen portfolios, rebalanced monthly, are formed from this triple sort, and the factors are defined as follows:

- ROE:

$$\begin{aligned} ROE = \frac{1}{6}&((\text{Small Conservative HighROE}) + (\text{Small Aggressive HighROE}) \\ &+ (\text{Small MidInv HighROE}) + (\text{Big Conservative HighROE}) \\ &+ (\text{Big Aggressive HighROE}) + (\text{Big MidInv HighROE})) \\ &- (\frac{1}{6}(\text{Small Conservative LowROE}) + (\text{Small Aggressive LowROE}) \\ &+ (\text{Small MidInv LowROE}) + (\text{Big Conservative LowROE}) \\ &+ (\text{Big Aggressive LowROE}) + (\text{Big MidInv LowROE})). \end{aligned}$$

- I/A:

$$\begin{aligned} I/A = \frac{1}{6}&((\text{Small Conservative HighROE}) + (\text{Small Conservative LowROE}) \\ &+ (\text{Small Conservative MidROE}) + (\text{Big Conservative HighROE}) \\ &+ (\text{Big Conservative LowROE}) + (\text{Big Conservative MidROE})) \\ &- \frac{1}{6}((\text{Small Aggressive HighROE} + (\text{Small Aggressive LowROE}) \\ &+ (\text{Small Aggressive MidROE}) + (\text{Big Aggressive HighROE}) \\ &+ (\text{Big Aggressive LowROE}) + (\text{Big Aggressive MidROE})). \end{aligned}$$

Finally, we replicate the Betting-Against-Beta factor from Frazzini and Pedersen (2014). For each stock i, we compute β as $\hat{\beta}_i^{ts} = \hat{\rho}\frac{\hat{\sigma}_i}{\hat{\sigma}_m}$, where m represents the market portfolio. Volatilities are 1-year rolling standard deviation of log returns, with at least 6 months of non-missing data. The correlation is the 5-year rolling correlation of 3-day overlapping log returns ($r_{i,t}^{3d} = \sum_{k=0}^{2}\log(r_{i,t+k})$), with at least 3 years of non-missing data. βs are then shrunk towards a cross-sectional mean set to 1: $\hat{\beta}_i = 0.6 \times \hat{\beta}_i^{ts} + 0.4 \times 1$. Stocks are then divided into a low beta portfolio and high beta portfolio, and their weights are determined by their β: the portfolio weights of the two portfolios are given by $w_L = \frac{2}{1_n|z-\bar{z}|}(z-\bar{z})^+$ and $w_H = \frac{2}{1_n|z-\bar{z}|}(z-\bar{z})^+$,

where z is the vector of ranked betas, $\bar{z}$ is the average rank and x^- and x^+ indicate respectively the negative and positive elements of a vector. The betting-against-beta is finally defined as $BAB_{t+1} = \frac{1}{\beta_t^L}(r_{t+1}^L - r^f) - \frac{1}{\beta_t^H}(r_{t+1}^H - r^f)$, where $\beta_t^L = \beta_t^{'} w_L$, $\beta_t^H = \beta_t^{'} w_H$, $r_{t+1}^L = r_{t+1}^{'} w_L$ and $r_{t+1}^H = r_{t+1}^{'} w_H$.

The code implementation in this thesis was inspired by various sources, including Song Drechsler (2020) for the SMB and HML factors, Ropotos (2022) for the ROE and CMA factors, and Li (2021) for the BAB factor. Although the final code is original, these sources served as valuable starting points in the development process.

Here are the results of our replication, measured by the correlation of the replicated factor returns with the original factors' returns.

Factor	**Market**	**SMB**	**HML**	**RMW**	**CMA**	**UMD**	**ROE**	**I/A**	**BAB**
Correlation	99%	96%	91%	88%	93%	97%	74%	69%	88%

Table 1: Correlation of returns of the replicated factors with the original factors

3.3 Volatility-Managed Multifactor Portfolios

We construct conditional multi-factor portfolios modeled after those introduced in DeMiguel, Martín-Utrera, and Uppal (2024). These portfolios incorporate seven of the nine factors introduced previously along with the volatility-managed factors, allowing their weights to vary according to volatility. Volatility-managed factors are defined as

$$\frac{x_{k,t}}{\sigma_t}$$

where $x_{k,t} \in \mathbb{R}^{N_t}$ represents the individual stock weights from factor k, with N_t being the number of stocks in our universe, and σ_t is the realized market volatility estimated as the standard deviation of the daily market returns in month t. The portfolio at time t is thus defined as

$$w_t(\theta_{k,t}) = \sum_{k=1}^{K} x_{k,t}\theta_{k,t} = \sum_{k=1}^{K} x_{k,t}(a_{k,t} + \frac{b_{k,t}}{\sigma_t}) \tag{1}$$

where K is the number of factors, $a_{k,t}$ is the weight given to the regular factor and $b_{k,t}$ is the weight given to the volatility-time factor at time t. Let $r_{k,t+1} \equiv x_{k,t}(r_{t+1} - r_{t+1}^f e_{N_t}) \in \mathbb{R}$ be the

k-th factor return in month $t+1$, where r_{t+1} are the stocks returns, r^f_{t+1} is the risk-free return and e_{N_t} is the N_t-dimensional vector of ones.

The return of a conditional multi-factor portfolio is then:

$$r_{p,t+1}(\theta_{k,t}) = \sum_{k=1}^{K} r_{k,t+1}\theta_{k,t} = \sum_{k=1}^{K} r_{k,t+1}(a_k + \frac{b_k}{\sigma_t}) \tag{2}$$

Following DeMiguel, Martín-Utrera, and Uppal (2024), we introduce the extended factor portfolio-weights matrix $X_{ext,t}$, factor returns vector $r_{ext,t+1}$ and factor-weight vector η:

$$X_{ext,t} = \begin{bmatrix} x_{1,t}^T \\ x_{2,t}^T \\ \vdots \\ x_{K,t}^T \\ x_{1,t}^T \times \frac{1}{\sigma_t} \\ x_{2,t}^T \times \frac{1}{\sigma_t} \\ \vdots \\ x_{K,t}^T \times \frac{1}{\sigma_t} \end{bmatrix}^{\mathrm{T}}, \quad r_{ext,t+1} = \begin{bmatrix} r_{1,t+1} \\ r_{2,t+1} \\ \vdots \\ r_{K,t+1} \\ r_{1,t+1} \times \frac{1}{\sigma_t} \\ r_{2,t+1} \times \frac{1}{\sigma_t} \\ \vdots \\ r_{K,t+1} \times \frac{1}{\sigma_t} \end{bmatrix}, \quad \eta_t = \begin{bmatrix} a_{1,t} \\ a_{2,t} \\ \vdots \\ a_{K,t} \\ b_{1,t} \\ b_{2,t} \\ \vdots \\ b_{K,t} \end{bmatrix} \tag{3}$$

respectively. The conditional multi-factor portfolio is obtained by optimizing the mean-variance utility of an investor with a risk-aversion parameter γ, net of the average transaction costs:

$$\max_{\eta_t \geq 0} \widehat{\mu}_{ext,t}\eta_t - \widehat{\mathrm{TC}}(\eta_t) - \frac{\gamma}{2}\eta_t^T \widehat{\Sigma}_{ext,t}\eta_t \tag{4}$$

where $\widehat{\mu}_{ext,t}$ and $\widehat{\Sigma}_{ext,t}$ are the sample mean and sample covariance of the extended factor returns computed using the data up until month t. $\widehat{\mu}_{ext,t}\eta_t$ and $\eta_t^T\widehat{\Sigma}_{ext,t}\eta_t$ represent the sample mean and covariance of the conditional multi-factor portfolio and $\widehat{\mathrm{TC}}(\eta_t)$ is the the average sample transaction cost, whose estimation is detailed in the following section. The constraint $\eta_t \geq 0$, meaning $a_{k,t} \geq 0$ and $b_{k,t} \geq 0$ for $k \in \{1, 2 \ldots, K\}$, is imposed to limit estimation errors.

3.4 Transaction Costs

Stock-level transaction costs $\kappa_{i,t}$ are estimated following the two-day corrected methodology outlined by Abdi and Ranaldo (2017). These costs are defined as half the bid-ask spread, where the bid-ask spread is estimated as

$$\hat{s}_{i,t} = \frac{1}{D}\sum_{d=1}^{D}\hat{s}_{i,d} = \frac{1}{D}\sum_{d=1}^{D}\sqrt{\max\left\{4(\text{cls}_{i,d} - \text{mid}_{i,d})(\text{cls}_{i,d+1} - \text{mid}_{i,d+1}), 0\right\}} \tag{5}$$

where D is the number of days in month t, $\text{cls}_{i,d}$ is the log closing price on day d and $\text{mid}_{i,d}$ is the average of the mean of the high and low log price of day d.

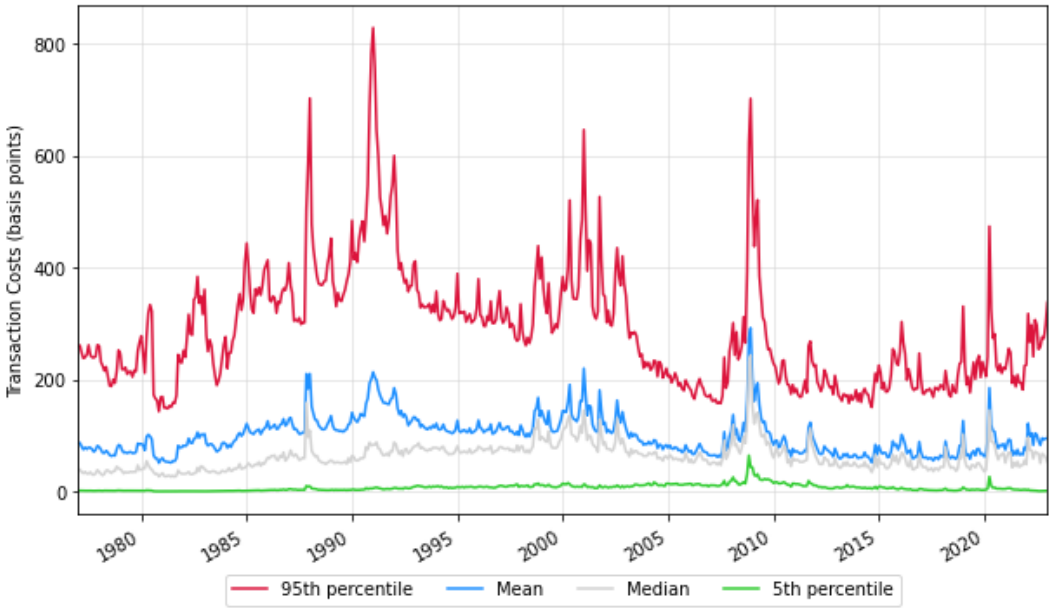
When transaction costs are not available, we use the methodology described in Novy-Marx and Velikov (2016). For each missing transaction cost, we find the stock's closest match and use its transaction cost instead. The closest match is defined as the stock with the shortest Euclidean distance in size and idiosyncratic volatility rank space: for stock i, it is the stock j that minimizes $\sqrt{(\text{rankME}_i - \text{rankME}_j)^2 + (\text{rankIVOL}_i - \text{rankIVOL}_j)^2}$. Idiosyncratic volatility is the standard deviation of the residuals from a regression of the stock's past three months of daily returns on the market's excess returns. If idiosyncratic volatility is not available, we only use the size rank to find the closest match. If that one is also unavailable, we use the cross-sectional average transaction cost.

We can see the stock level transaction costs on Fig. 1a. Periods associated with economic downturns display spikes in transaction costs, as around 1991, 2001, 2008 and 2020. When we fill the missing transaction costs values in Fig. 1b, we can see that the 95th percentile transaction costs gets lower, as stocks with very high transaction costs represent a smaller proportion of our data, and that the average transaction cost gets lower, due to the fact that we filled some missing values with cross-sectional average.

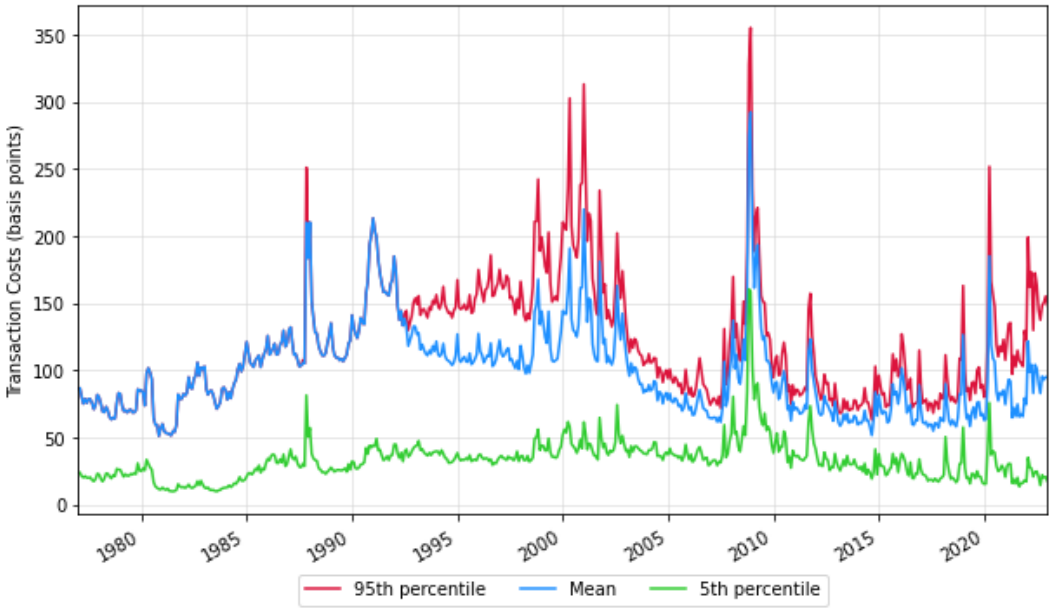
Figure 1: Transaction costs at the stock level.

This figure shows the transaction costs at the stock levels from January 1977 until December 2022. Panel (a) shows transaction costs obtained by using the two-day corrected methodology of Abdi and Ranaldo (2017) to compute the half spread. Panel (b) shows the same transaction costs in which missing values have been filled using the steps described in Novy-Marx and Velikov (2016).

(a) Transaction costs obtained following Abdi and Ranaldo (2017)



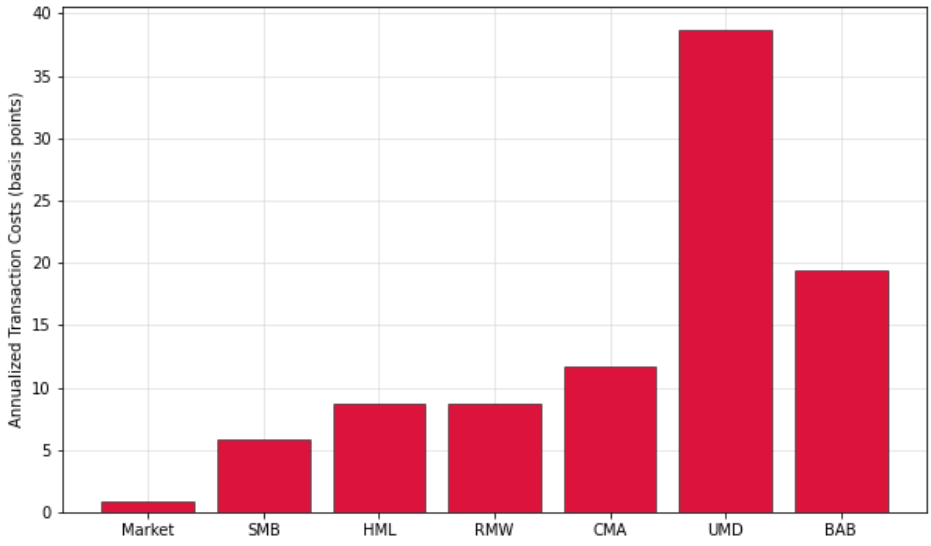
(b) Transaction costs obtained after applying the correction from Novy-Marx and Velikov (2016)



We represent the annualised transaction costs per factor in Fig. 2. The market factor has the lowest transaction costs, as the largest stocks usually have the lower transaction costs. The momentum factor has the highest transaction costs. Overall, the transaction costs are in line with those reported in DeMiguel, Martín-Utrera, and Uppal (2024), although our transaction costs are slightly lower. From here on, we decided not to include the I/A and ROE factors from Hou, Xue, and Zhang (2015), as the transaction costs obtained confirmed that our replication was not satisfactory. Additionally, our replicated CMA factor has a correlation higher than 85% with the original I/A factor.

Figure 2: Annualised transaction costs per factor, excluding I/A and ROE

This figure shows the annualised transaction costs associated with the rebalancing of each factor from January 1977 until December 2022. Transaction costs are computed using the methodology of Abdi and Ranaldo (2017), complemented with the methodology of Novy-Marx and Velikov (2016) at the stock levels.



We then aggregate the stock level transaction costs in a diagonal matrix:

$$\Lambda_t = \begin{pmatrix} \kappa_{1,t} & & \\ & \ddots & \\ & & \kappa_{n,t} \end{pmatrix}$$

The portfolio's average transaction costs are defined following DeMiguel, Martín-Utrera, and Uppal

(2024) up until time T as:

$$\hat{\mathrm{TC}}(\eta_t) = \frac{1}{T-1}\sum_{t=1}^{T}\|\Lambda_t \Delta w_{t+1}(\eta_t)\|_1 \tag{6}$$

where $\|a\|_1 = \sum_{i=1}^{N_t}|a_i|$ is the 1-norm of the N_t dimensional vector. $\Delta w_{t+1}(\eta_t)$ corresponds to the change in weights and accounts for the netting across factors: the change in weight for each stock across each factor is summed, and then the transaction cost is applied. It is defined as

$$\Delta w_{t+1}(\eta_t) = w_{t+1}(\eta_{t+1}) - w_t(\eta_t)^+ \tag{7}$$

where

$$w_{t+1}(\eta_{t+1}) = X_{ext,t+1}\eta_{t+1} \tag{8}$$

is the conditional multi-factor portfolio at time $t+1$, and

$$w_t(\eta_t)^+ = w_t(\eta_t) \circ (e_t + r_{t+1}) \tag{9}$$

is the conditional multi-factor before the rebalancing at time $t+1$, $w_t(\eta_t) \circ (e_t + r_{t+1})$ representing the component-wise product between $w_t(\eta_t)$ and $(e_t + r_{t+1})$.

3.5 Volatility Forecasting

In order to extend the results from DeMiguel, Martín-Utrera, and Uppal (2024), we introduce the use of volatility forecasting to manage the factor weights. Instead of using the realized market volatility estimated as the standard deviation of the daily market returns in month, we use the next month's forecasted volatility. We therefore introduce the matrix $X^{\text{forc}}_{ext,t}$, factor returns vector

$r^{\text{forc}}_{ext,t+1}$:

$$X^{\text{forc}}_{ext,t} = \begin{bmatrix} x^T_{1,t} \\ x^T_{2,t} \\ \vdots \\ x^T_{K,t} \\ x^T_{1,t} \times \frac{1}{\sigma^{\text{forc}}_{t+1}} \\ x^T_{2,t} \times \frac{1}{\sigma^{\text{forc}}_{t+1}} \\ \vdots \\ x^T_{K,t} \times \frac{1}{\sigma^{\text{forc}}_{t+1}} \end{bmatrix}^{\text{T}}, \quad r^{\text{forc}}_{ext,t+1} = \begin{bmatrix} r_{1,t+1} \\ r_{2,t+1} \\ \vdots \\ r_{K,t+1} \\ r_{1,t+1} \times \frac{1}{\sigma^{\text{forc}}_{t+2}} \\ r_{2,t+1} \times \frac{1}{\sigma^{\text{forc}}_{t+2}} \\ \vdots \\ r_{K,t+1} \times \frac{1}{\sigma^{\text{forc}}_{t+2}} \end{bmatrix}, \tag{10}$$

respectively, where $\sigma^{\text{forc}}_{t+1}$ denotes the forecasted volatility of month $t+1$.

Following Liu (2019), we implement two machine learning models to forecast volatility: Long Short-Term Memory Re-current Neural Networks and ν-support vector machine for regression.

The models are trained using daily data of market returns, VIX index values and rolling volatility with a 22-day window. Indeed, the VIX index has been found to be an important predictor of market volatility by Mittnik, Robinzonov, and Spindler (2015). Additionally, we create lagged features with a lag of up to 5 days: if x_t is one of the feature described, we also include x_{t-i} for $i \in \{1, 2, \ldots, 5\}$. We can therefore write our feature for each day as:

$$X^{\text{mret}}_t = \begin{bmatrix} x^{\text{mret}}_t \\ x^{\text{mret}}_{t-1} \\ x^{\text{mret}}_{t-2} \\ x^{\text{mret}}_{t-3} \\ x^{\text{mret}}_{t-4} \\ x^{\text{mret}}_{t-5} \end{bmatrix}, \quad X^{\text{mvol}}_t = \begin{bmatrix} x^{\text{mvol}}_t \\ x^{\text{mvol}}_{t-1} \\ x^{\text{mvol}}_{t-2} \\ x^{\text{mvol}}_{t-3} \\ x^{\text{mvol}}_{t-4} \\ x^{\text{mvol}}_{t-5} \end{bmatrix}, \quad X^{\text{vix}}_t = \begin{bmatrix} x^{\text{vix}}_t \\ x^{\text{vix}}_{t-1} \\ x^{\text{vix}}_{t-2} \\ x^{\text{vix}}_{t-3} \\ x^{\text{vix}}_{t-4} \\ x^{\text{vix}}_{t-5} \end{bmatrix} \tag{11}$$

where x^{mret}_t is the market return on day t, x^{mvol}_t the rolling market volatility on day t and x^{vix}_t the value of the VIX index on day t; and our training data is thus

$$X_t = \begin{bmatrix} X^{\text{mret}}_t & X^{\text{mvol}}_t & X^{\text{vix}}_t \end{bmatrix}. \tag{12}$$

Using this data, we create sequences of length of 22 days, with each sequence labeled by the

end-of-month volatility. For each month, the sequence S_t and its corresponding label y_t are:

$$S_t = [X_{t-22+1}, X_{t-22+2}, \ldots, X_t] \tag{13}$$

$$y_t = \sigma_{t+1} \tag{14}$$

where σ_{t+1} is the volatility at the end of the next month. The model are trained using an expanding window approach: we start the training from the beginning of our data for the VIX index, that is January 1990, and use all the sequences of data up to the current end-of-month date for training to predict the next month's end-of-month volatility. The first window is 2-year long and outputs the volatility of February 1992. In each training, each feature in the training data is normalized to a range $[0, 1]$ using Min-Max scaling to improve the performance of the model. For a feature X, the scaled value $X^{'}$ is

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

3.5.1 Long Short-Term Memory Re-current Neural Networks

The first method to forecast volatility used relies on Long Short-Term Memory Re-current Neural Networks (LSTM RNN). A Recurrent Neural Network (RNN) is a type of artificial neural network characterized by connections between units that form directed cycles, allowing the network to use past inputs to influence future outputs, thus incorporating a form of memory. Long Short-Term Memory, introduced by Hochreiter and Schmidhuber (1997), is a specialized type of RNN capable of learning long-term dependencies and mitigating the exploding or vanishing gradient problem by incorporating gates that regulate access to the cell state, as explained by Gers, Schmidhuber, and Cummins (2000). This makes LSTM particularly suitable for time-series predictions.

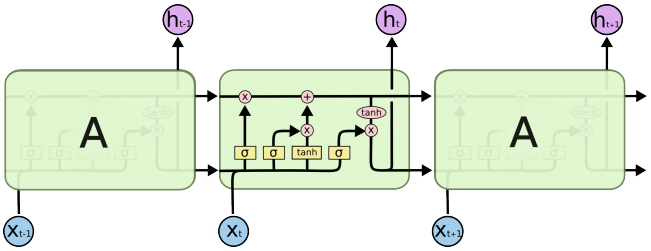


Figure 3: Repeating module in an LSTM Olah (2015)

An LSTM RNN is composed of an input layer, hidden layers, and an output layer. To

understand the workings of an LSTM, we refer to Fig. 3, designed and explained by Olah (2015). Let x_t be an input vector, h_t the state of the output units, C_t the cell state vector, W_x and b_x be the weights and bias associated with the transformation x, respectively. The core idea behind LSTM networks is to enable the network to remember long-term dependencies and mitigate the vanishing gradient problem encountered in traditional RNNs. This is achieved through the use of gates and cell states that control the flow of information. Each cell in the LSTM network decides how much of the previous information from cell C_{t-1} (top line of the diagram) to retain through a series of gates.

- Forget Gate Layer: This layer determines what information from the previous cell state C_{t-1} should be discarded using a sigmoid activation function, which outputs values between 0 and 1, representing the retention rate of each component of the cell state:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f).$$

  Here, σ denotes the sigmoid function. The forget gate's output f_t modulates the cell state:

$$C_t = C_{t-1} \times f_t$$

- Input Gate Layer: This layer decides which new information will be stored in the cell state. It also employs a sigmoid function to regulate this flow:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

  In parallel, a tanh function generates a vector of new candidate values $\tilde{C}_t$ that could be added to the cell state:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C).$$

  The cell state C_t is updated as a combination of the old cell state and the new candidate values, modulated by the forget gate and input gate, respectively: $C_t = C_{t-1} \times f_t + i_t \times \tilde{C}_t$

- Output Gate Layer: This layer determines the output of the current cell. It uses a sigmoid

function to decide which parts of the cell state should be output:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o).$$

The actual output h_t of the cell is obtained by modulating the cell state C_t with the output gate and applying the tanh function to the cell state:

$$h_t = o_t \times tanh(C_t)$$

Let us now delve into our specific implementation of the LSTM to predict volatility. We describe here a base implementation, and we will show how changing parameters affect our results in the next section. The LSTM network architecture includes:

- An input layer of shape (T, d), where T is the number of time steps (i.e. 22) and d is the number of features,
- A first LSTM layer in which 20% of the units are randomly dropped during training for regularization, similar to Liu (2019). The layer has an input shape of (T, d) and an output shape of $(T, 50)$, where 50 is the number of LSTM units in the layer and in which 10 (i.e. 20%) have been set to 0.
- A second LSTM layer with a 20% dropout rate. The layer has an input shape of $(T, 50)$ and an output shape of (50), representing the 50 LSTM units. This layer processes the sequence from the first LSTM layer and outputs a single vector representing the entire sequence.
- A Dense layer with an input shape of 50 and an output shape of 1, representing the end-of-next-month volatility.

The model is trained using the mean squared error (MSE) loss function:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2$$

where y_i are the actual values and $\widehat{y}_i$ are the predicted values.

The number of epochs is set to 50, which means that the model iterates 50 times over the

entire dataset during training to adjust the weights and bias of all components of the LSTM cells, in order to minimize the loss function. The batch size is set to 32, meaning that 32 sequences are processed before the model's weights and biases are updated based on the gradients of the loss function.

3.5.2 ν-Support Vector Machine for Regression

The ν-Support Vector Machine for Regression, introduced by Schölkopf et al. (2000) (ν-SVR), is an extension to the Support Vector Machine for Regression (SVR). In SVR, the goal is to find a linear function $f(x) = x^T w + b$ that deviates from the actual target values y by at most ε for all the training data (i.e. $\left|y_i - (x_i^T w + b)\right| \leq \varepsilon$), and is as flat as possible. Since it is not always possible to find such function, slack variables are introduced to handle points that are outside of the ε region.

In ν-SVR, the parameter ε is not specified directly and is instead controlled by the parameter $\nu \in (0, 1]$. As demonstrated by Schölkopf et al. (2000), ν sets an upper bound on the fraction of margin errors and a lower bound on the fraction of support vectors, thus adjusting the model's complexity.

To estimate function f, the following minimization problem is solved:

$$\min_{w,\xi^*,\varepsilon} \frac{1}{2}\|w\|^2 + C \cdot (\nu\varepsilon + \frac{1}{l}\sum_{i=1}^{l}(\xi_i + \xi_i^*)). \tag{15}$$

subject to:

$$((w \cdot x_i) + b) - y_i \leq \varepsilon + \xi_i \tag{16}$$

$$y_i - ((w \cdot x_i) + b) \leq \varepsilon + \xi_i^* \tag{17}$$

$$\xi_i^* \geq 0, \quad \xi_i^* \geq 0, \quad \varepsilon_i \geq 0 \tag{18}$$

Here, w is the weight vector, b is the bias term, C is a regularization parameter that determines the trade-off between the flatness of $f(x)$ and the amount up to which deviations larger than ε are tolerated, ξ_i and ξ_i^* are the slack variables.

As explained by Smola and Schölkopf (2004), we can introduce Lagrange multipliers to solve this problem to get a Lagrangian. Deriving this Lagrangian with respect to its variables gives us

the following dual formulation that is easier to solve than the initial problem:

$$\max_{\alpha,\alpha^*} \begin{cases} -\frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)(x_i \cdot x_j) \\ \\ -\varepsilon\sum_{i=1}^{l} y_i(\alpha_i - \alpha_i^*) + \sum_{i=1}^{l} y_i(\alpha_i - \alpha_i^*) \end{cases} \tag{19}$$

subject to,

$$\sum_{i=1}^{l}(\alpha_i - \alpha_i^*) = 0 \tag{20}$$

$$\alpha_i, \alpha_i^* \in [0, C] \tag{21}$$

The regression function from the solution of the dual problem is:

$$f(x) = \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)(x_i \cdot x) + b \tag{22}$$

Still noted by Smola and Schölkopf (2004), the objective function in the dual problem involves the dot product of the feature vectors, $\phi(x)\cdot\phi(x^T)$, which can be computationally expensive. We can replace this computation by using a kernel function, which computes the dot product in the feature space implicitly, without explicitly mapping the input vectors to that space. The kernel function is thus defined as

$$k(x, x^T) := \phi(x) \cdot \phi(x^T).$$

Like Liu (2019), we use the Gaussian (RBF) Kernel, defined as:

$$k(x_i, x_j) := \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2}).$$

In the following we replace $\frac{1}{2\sigma^2}$ by γ. The kernel is thus written $k(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$, and we need to estimate three parameters: ν, C, and γ.

4 Results and Discussion

4.1 Tuning of the Models' Parameters

We ran several versions of the LSTM and ν-SVR models with different parameters, in order to choose the combination of parameters that give us best out-of-sample predictions. We evaluate the accuracy using the root mean square error metric (RMSE), defined as the the square root of the mean square error:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2}.$$

The most accurate model is the one which will minimise the root mean square error.

4.1.1 LSTM model

Due to the computation-intensive nature of the LSTM model, we could only run a few models in order to select the best-performing one. Indeed, one model takes several hours to run.

We look at different combinations of units per cells and drop-out rate for an LSTM network.

Table 2: Parameters of the models tested and their root mean squared error

This table displays different LSTM models, their parameters and the RMSE computed on the out-of-sample periods from January 1992 until September 2022. The different parameters tried are the number of cells, the number of units per cells and the number of epochs used during the training.

Parameter	**Cells**	**Units**	**Dropout**	**Epochs**	**RMSE**
Model 1	1	30	0.2	50	0.3284%
Model 2	1	50	0.2	30	0.3264%
Model 3	1	100	0.2	30	0.2988%
Model 4	2	30; 30	0.2 ; 0.2	50	0.3243%

We can observe from Table 2 that Model 3 is the best performing among the tested models. The results indicate that the number of cells is the most crucial factor in determining the accuracy of the model, with an increase in the number of cells per unit leading to a decrease in RMSE. Additionally, we notice that increasing the number of epochs also contributes to a reduction in RMSE. Interestingly, reducing the number of epochs during the training phase did not adversely affect the performance, suggesting a certain robustness in the model's training process.

Figure 4: Comparison of the predictions of the next month's volatility generated by the LSTM model 3 with actual values over the period from January 1992 until September 2022.

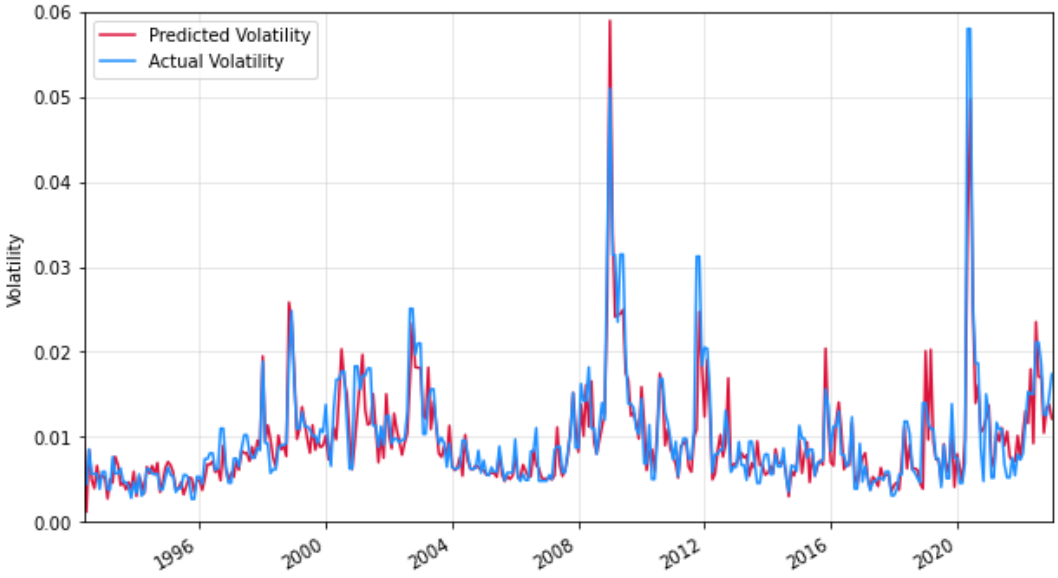


Fig. 4 illustrates how the best-performing LSTM model tracks the variations in actual next-month market volatility, especially during periods of low volatility. The model accurately predicted the smaller spikes between 1998 and 2003. However, it overestimated the spike following the 2008 financial crisis and slightly underestimated the spike in 2020.

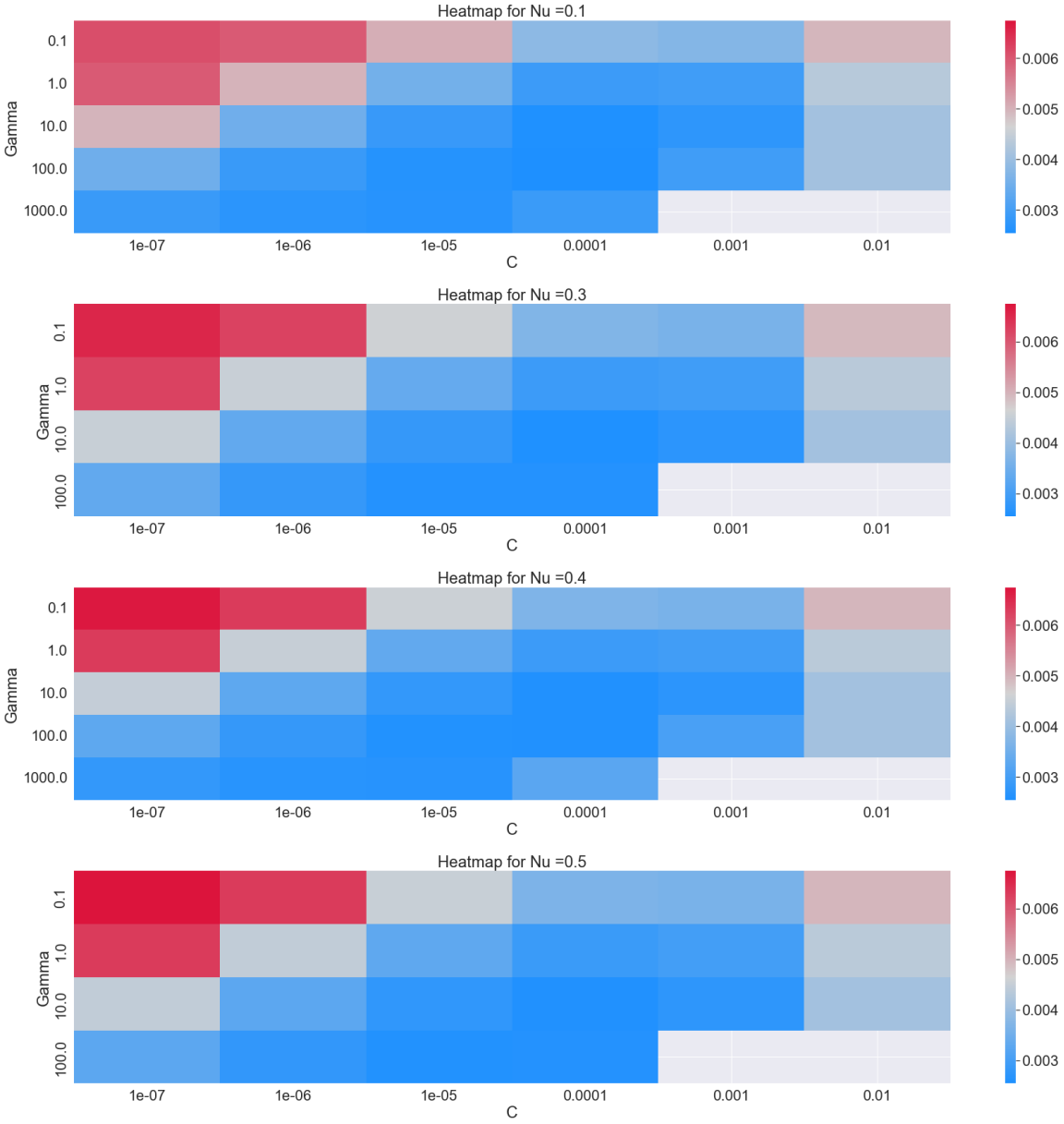
4.1.2 ν-SVR model

Given that the ν-SVR model is quicker to train than the LSTM model, we conducted a traditional grid-search over the parameter space to identify the optimal parameter combination. This involved iterating over different values of ν, C, and γ and selecting the combination that minimized the RMSE. The grid-search was performed on a sub-sample covering 2/3 of our data (1992-2012), a period representative of various market conditions, including high volatility periods in 2001 and 2008. We test values for ν in $\{0.1, 0.3, 0.4, 0.5\}$, for C in $\{0.1, 1, 10, 100, 1000\}$ and for γ in $\{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{2}, 10^{-1}\}$.

The results of the grid-search are visualized in Fig. 5, where RMSE is color-coded with blue indicating low values and red indicating high values. Each sub-figure corresponds to a different ν value, with the x-axes representing C values and the y-axes representing γ values. The patterns indicate that ν has a secondary role in prediction accuracy, with the same C and γ combinations yielding the best results across different ν values. Notably, the five best performing models had γ

values of 10^{-4}; two of these models had C values of 100, while three had C values of 10.

Figure 5: Heatmap of the RMSE obtained from running the ν-SVR model with different combination of parameters



To validate our findings, we ran the five best models over the entire sample and reported the results in Table 3.

Table 3: Parameters of the ν-SVR models tested and their root mean squared error

This table displays the 5 ν-SVR models which had the the smallest RMSE in the grid-search. The RMSE is computed on the periods between January 1992 and December 2022.

Model	ν	C	γ	**RMSE**
Model 1	0.1	10	0.0001	0.3230%
Model 2	0.1	100	0.0001	0.3117%
Model 3	0.3	10	0.0001	0.3110%
Model 4	0.4	10	0.0001	0.3228%
Model 5	0.4	100	0.0001	0.3117%

All models performed similarly in terms of RMSE, with Model 3 emerging as the best at predicting next-month's volatility. However, even the best ν-SVR model performed worse than the best LSTM model.

Figure 6: Comparison of the predictions of the next month's volatility generated by the ν-SVR model 3 with actual values over the period from January 1992 until September 2022.

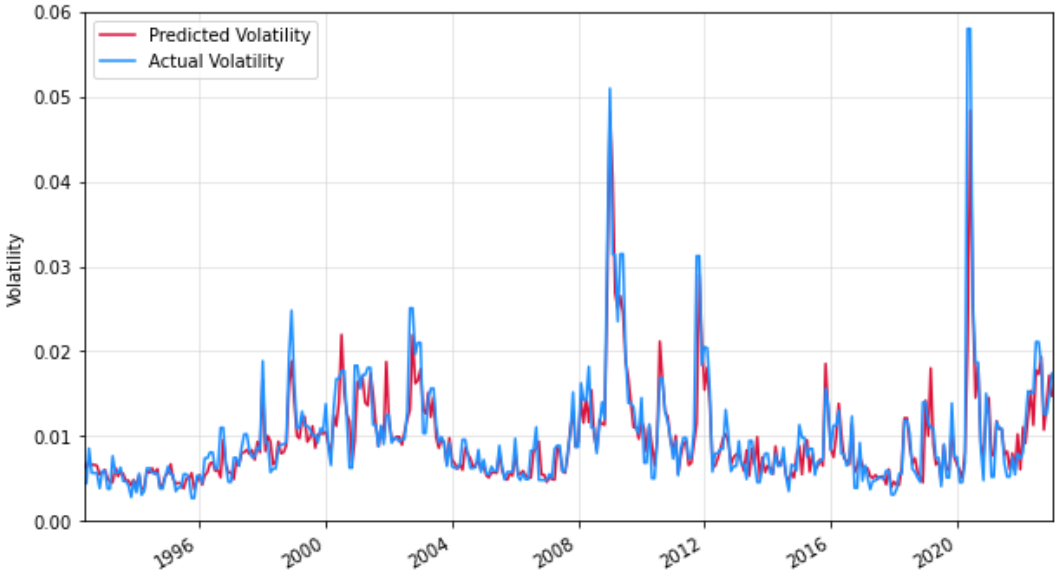


Fig. 6 compares the out-of-sample predictions of Model 3 with actual realized volatility. The predictions closely follow the actual values, although the model tends to overestimate volatility in low-volatility periods and underestimate it in high-volatility periods. This contrasts with the

LSTM model predictions shown in Fig. 4 where no systematic bias was noticeable.

To address our research question, we need to focus on periods of increased volatility, as these are when transaction costs are most significant and when the breakdown in the risk-return tradeoff is the most severe. While the RMSE was initially evaluated over the entire out-of-sample period, we now consider the RMSE during high-volatility periods: 1998-2003, 2008-2012, and 2020.

Table 4: RMSE of ν-SVR models during periods of high volatility

This table displays the 5 ν-SVR models that have the best performance overall and shows their RMSE during periods of increased volatility. The periods tested are 1998-2003, 2008-2012 and 2020. The average RMSE over the three periods is also displayed.

Period	**1998-2003**	**2008-2012**	**2020**	**Avg. RMSE**
Model 1	0.6383%	0.7508%	0.4532%	0.61141%
Model 2	0.6553%	0.7394%	0.4579%	0.6175%
Model 3	0.6573%	0.7627%	0.4323%	0.6174%
Model 4	0.6641%	0.7694%	0.4291%	0.6209%
Model 5	0.6850%	0.7570%	0.4024%	0.6148%

Table 4 shows that during these high-volatility periods, Model 1 provides the best predictions, with a lower average RMSE compared to other models. Nonetheless, the performance differences among the models are marginal, with Model 3 ranking third but still in line with other models.

We select Model 3 among the LSTM models and Model 3 among the ν-SVR models for further analysis. We then assess their usefulness in the context of volatility-managed portfolios.

4.2 Portfolio's Performance

Initially, we replicate the results obtained by DeMiguel, Martín-Utrera, and Uppal (2024). We calculate the cumulative returns of a portfolio whose weights are determined by solving Eq. (4) using an expanding window of data. The first window spans 10 years, with out-of-sample results obtained between January 1977 and December 2022.

Figure 7: Comparison of the Cumulative Performance of the Volatility-Managed Portfolio and the Market Portfolio

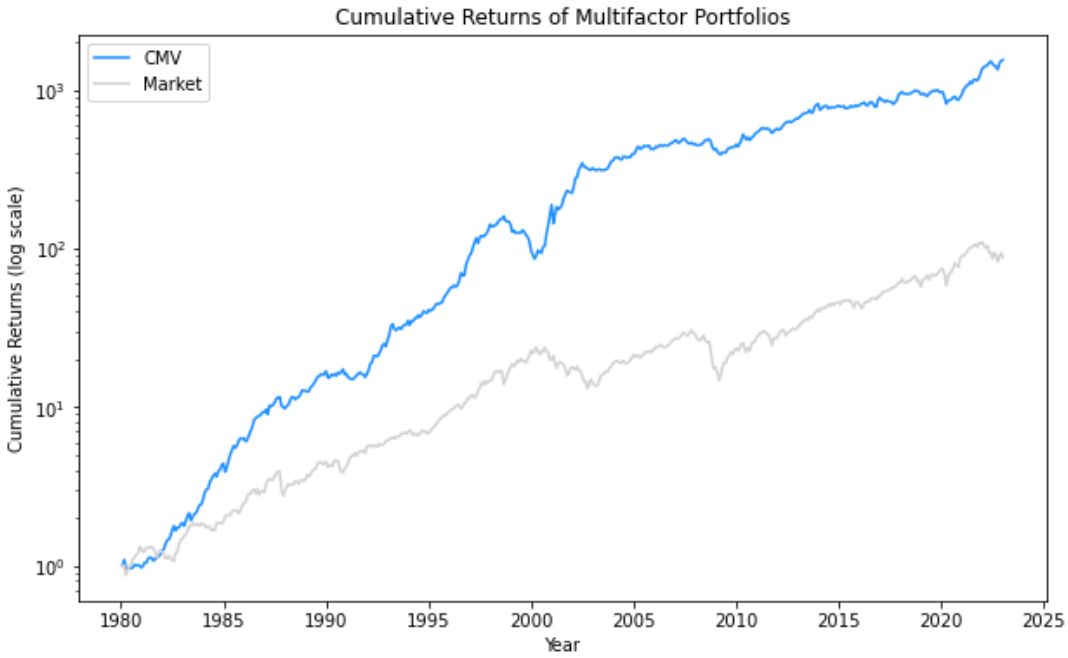


Fig. 7 displays the cumulative returns of the volatility-managed portfolio based on realized market volatility. The portfolio returns are scaled to match the volatility of market returns. Our results align with those of DeMiguel, Martín-Utrera, and Uppal (2024), confirming that removing factors such as ROE and I/A does not significantly impact the findings.

Next, we replace the realized market volatility with forecasted market volatility. The portfolio weights are determined by solving:

$$\max_{\eta_t \geq 0} \widehat{\mu}_{ext,t}^{\text{forc}} \eta_t - \widehat{\text{TC}}(\eta_t) - \frac{\gamma}{2} \eta_t^T \widehat{\Sigma}_{ext,t}^{\text{forc}} \eta_t \tag{23}$$

in which $\widehat{\mu}_{ext,t}^{forc}$ and $\widehat{\Sigma}_{ext,t}^{forc}$ are respectively the sample mean and sample covariance of the extended factor returns $r_{ext,t+1}^{\text{forc}}$ computed using the data up until month t and the volatility of month $t+1$ forecasted using the LSTM and ν-SVR models. The estimation of the average transaction costs is adapted accordingly: $\hat{\text{TC}}(\eta_t) = \frac{1}{T-1} \sum_{t=1}^{T} \|\Lambda_t \Delta w_{t+1}(\eta_t)\|_1$ with $\Delta w_{t+1}(\eta_t) = w_{t+1}(\eta_t) - w_t(\eta_{t-1})^+ = X_{ext,t+1}^{\text{forc}}(\eta_t) - w_t(\eta_{t-1})^+$.

We compute the cumulative returns of a portfolio using an expanding window of data, starting with a 5-year window, and obtain out-of-sample results from May 1997 to December 2022. For a fair comparison, we re-compute the original volatility-managed portfolio based on a 5-year

window, with data starting in 1992. We scale all returns in Fig. 8 so that the series have the same volatility as the market returns.

Figure 8: Comparison of the Cumulative Performance of the Volatility-Managed Portfolio with and without forecasted volatility the Market Portfolio between May 1997 and December 2022.

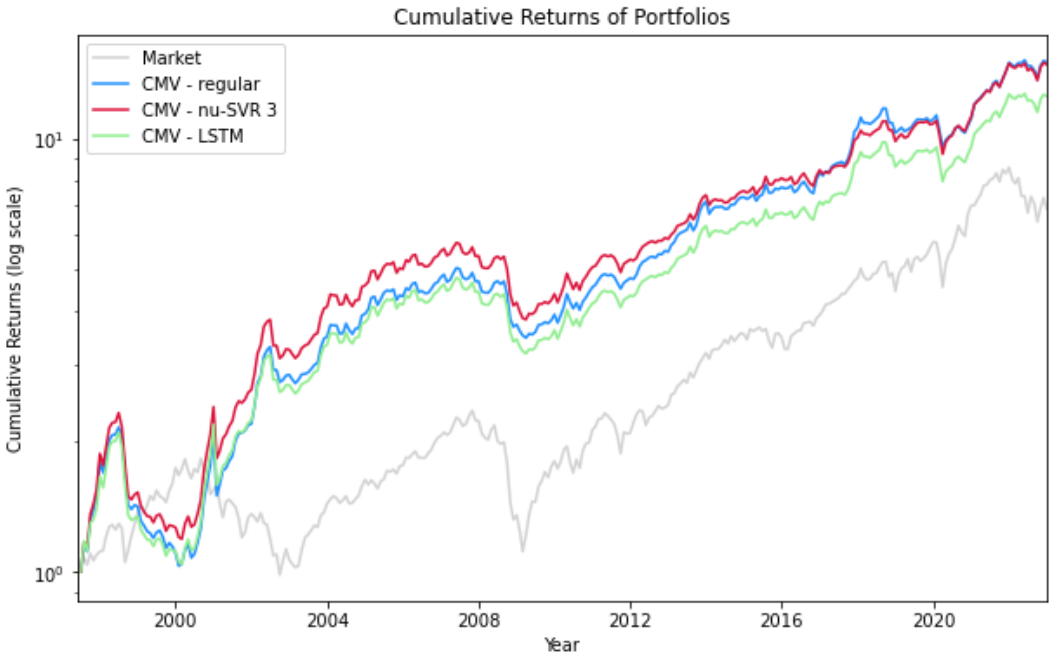


Fig. 8 shows that using forecasted volatility instead of realized volatility does not enhance the performance of volatility-managed portfolios. In fact, it worsens performance in the case of the LSTM-based portfolio, which diverges from the regular volatility-managed portfolio (CMV) as early as 2002. The ν-SVR-based initially outperforms the CMV, and then tracks it closely.

Table 5: Performance of the Portfolios

This table displays for the three portfolios the return's mean, standard deviation, Sharpe ratio, p-value and average transaction costs TC between May 1997 and December 2022.
We compare the difference in Sharpe ratio by using the same one-sided p-value methodology like DeMiguel, Martín-Utrera, and Uppal (2024). We generate 10,000 bootstrap samples of the returns of the two portfolios that we want to compare using the stationary block-bootstrap method of Politis and Romano (1994) with an average block size of five, using the implementation from the arch library in python (Sheppard (2024, Apr 16)). We then subsequently construct from the 10,000 bootstrap samples the empirical distribution of the difference between the Sharpe ratios of the returns of the conditional volatility-managed portfolio based realized market volatility and on forecasted market volatility. We finally compute the p-value as the frequency with which this difference is smaller than zero across the bootstrap samples.

Model	**CMV**	**LSTM**	**ν-SVR**
Mean	0.0099	0.0311	0.0350
Standard Deviation	0.0442	0.1614	0.1664
Sharpe Ratio	0.2247	0.1929	0.2101
p-value		0.0001	0.0001
TC	0.0247	0.0207	0.0198

We can see in Table 5 that the portfolios based on predicted volatility do incur less transaction costs on average, confirming our initial hypothesis. However, the outperformance of the portfolio based on the ν-SVR model over the one based on the LSTM model is counter-intuitive, as the latter one had a smaller RMSE. Examining the error distribution in Fig. 9, shows that the ν-SVR model's errors are more tightly clustered around zero, indicating smaller average errors. This contributes to its superior performance despite a few extreme errors that inflate its RMSE. Conversely, the LSTM model has fewer extreme outliers but larger average errors. Thus, the ν-SVR-based portfolio outperforms the LSTM-based one due to more accurate volatility predictions.

Figure 9: Box Plots of distribution of errors of the predictions of the LSTM model and the ν-SVR model

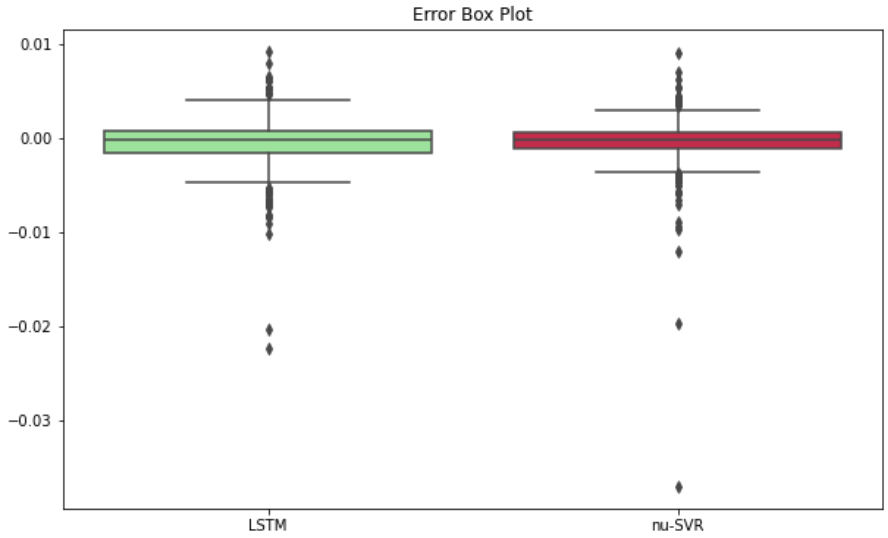
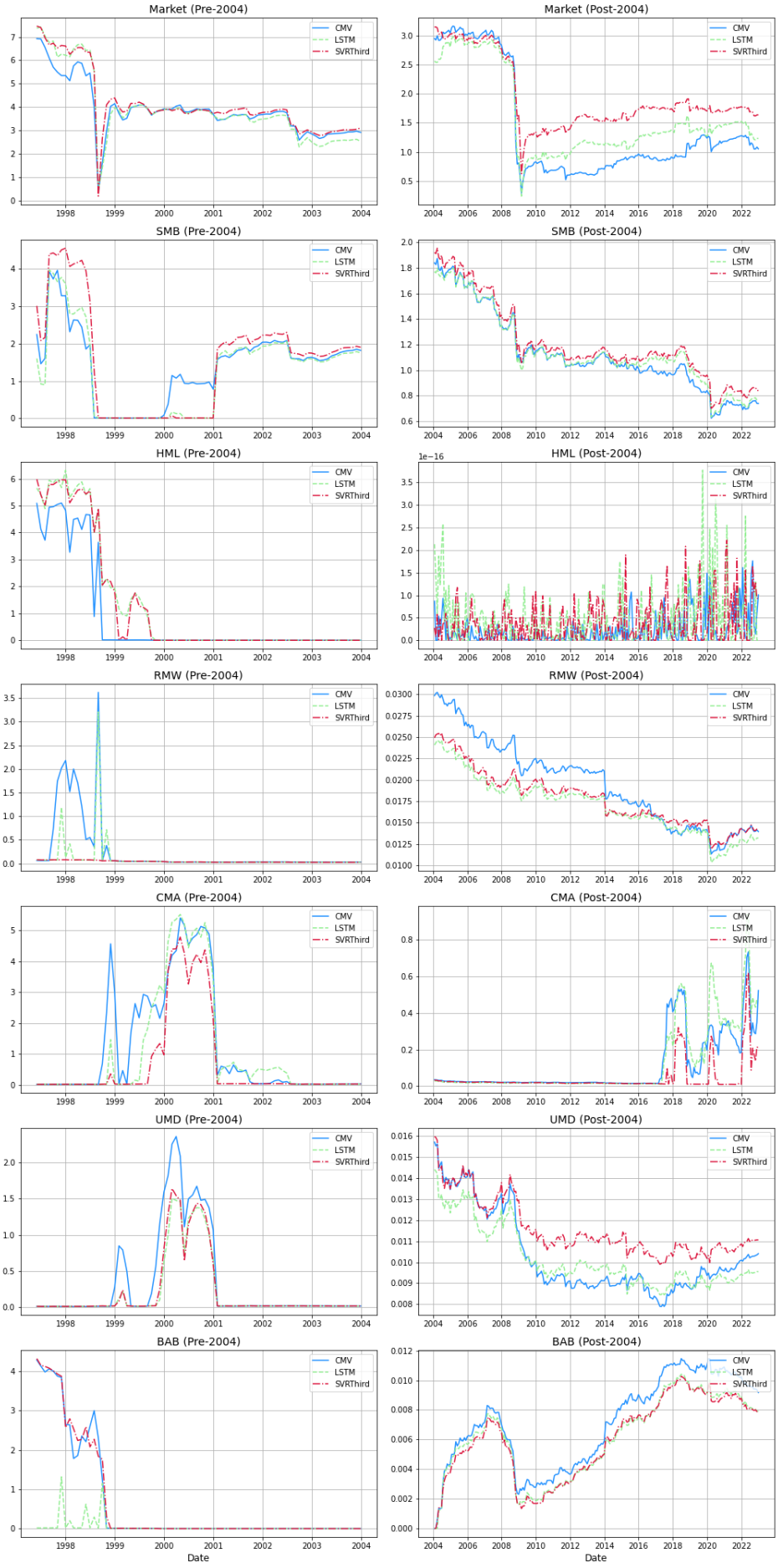


Fig. 10 illustrates the impact of the models on weight distribution. Given significant changes in weight scale over time, we split the observations at the 2004 mark. We observe that the regular volatility-managed portfolio typically has lower or higher weights than those based on predicted volatility, suggesting more reactive weights. Otherwise, the weight patterns for each factor are similar.

Figure 10: Weight distribution per factor from May 1997 to December 2022



4.3 Robustness Check

As a robustness check, we conducted the maximization from Eq. (23) without the non-negativity constraint on the weights, with the results presented in Table 6. Removing the non-negativity constraint led to much lower Sharpe ratios compared to the regular volatility-managed portfolio. This constraint was beneficial in minimizing the impact of estimation errors in predicted volatility. Additionally, we observed a substantial increase in transaction costs for the ν-SVR portfolio, which likely contributed to its markedly poor performance compared to its constrained version.

Table 6: Performance of the Portfolios

This table displays for the three portfolios the return's mean, standard deviation, Sharpe ratio, and average transaction costs TC between May 1997 and December 2022.

Model	**CMV**	**LSTM**	**ν-SVR**
Mean	0.0097	0.0375	0.0739
Standard Deviation	0.0442	0.2297	0.5381
Sharpe Ratio	0.2201	0.1635	0.1375
TC	0.0334	0.0295	0.0365

5 Conclusion

We replicated the setup of volatility-managed portfolios, accounting for transaction costs, as developed by DeMiguel, Martín-Utrera, and Uppal (2024). Subsequently, we substituted the prediction of next month's volatility for the realized market volatility in the mean-variance optimization to determine the portfolio weights. Following the methodology of Liu (2019), next month's market volatility was estimated using two machine learning techniques: Long Short-Term Memory (LSTM) Recurrent Neural Networks and ν-Support Vector Regression (SVR) models.

Our analysis revealed that, although the Sharpe ratios of portfolios using forecasted volatility were close to those of the regular volatility-managed portfolios, their overall performance was still inferior. This underperformance was observed in both the LSTM and ν-SVR models. More-

over, when the non-negativity constraint on the weights was removed, the portfolios became more vulnerable to volatility prediction errors, resulting in significantly worse performance compared to portfolios based on realized market volatility. Consequently, even though using forecasted volatility in the constrained version led to lower transaction costs, it did not improve the anticipation of the breakdown of the risk-return trade-off, as evidenced by Sharpe ratios that were not higher than those of the regular volatility-managed portfolio.

These findings highlight the challenges of using predicted volatility in portfolio management. Despite advancements in machine learning techniques, the accuracy of volatility predictions remains a critical factor, and even slight errors can substantially impact portfolio performance. Therefore, relying on realized market volatility, which does not suffer from prediction errors, remains a more robust approach for volatility-managed portfolios.

Reference

Abdi, Farshid and Angelo Ranaldo (2017). "A simple estimation of bid-ask spreads from daily close, high, and low price". In: *Review of Financial Studies* 30, pp. 4437–4480.

Barroso, Pedro and Andrew L. Detzel (2021). "Do limits to arbitrage explain the benefits of volatility-managed portfolios?" In: *Journal of Financial Economics* 140, pp. 744–767.

Bollerslev, Tim (1986). "Generalized autoregressive conditional heteroskedasticity". In: *Journal of Econometrics* 31, pp. 307–327.

Brandt, Michael W., Pedro Santa-Clara, and Rossen Valkanov (2009). "Parametric Portfolio Policies: Exploiting Characteristics in the Cross-Section of Equity Returns". In: *The Review of Financial Studies* 22, pp. 3411–3447.

Carhart, Mark M. (1997). "On persistence in mutual fund performance". In: *Journal of Finance* 52, pp. 57–82.

Cederburg, Scott et al. (2020). "On the performance of volatility-managed portfolios". In: *Journal of Financial Economics* 138, pp. 95–117.

DeMiguel, Victor, Alberto Martín-Utrera, and Raman Uppal (2024). "A Multifactor Perspective on Volatility-Managed Portfolios". In: *Journal of Finance* Forthcoming.

Fama, Eugene F. and Kenneth R. French (2015). "A five-factor asset pricing model". In: *Journal of Financial Economics* 116, pp. 1–22.

Frazzini, Andrea and Lasse H. Pedersen (2014). "Betting against beta". In: *Journal of Financial Economics* 111, pp. 1–25.

Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins (2000). "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12, pp. 2451–2471.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9, pp. 1735–1780.

Hou, Kewei, Cheng Xue, and Lu Zhang (2015). "Digesting anomalies: An investment approach". In: *Review of Financial Studies* 28, pp. 650–705.

Li (2021). *Betting_against_beta*. https://github.com/LittleLi2020/Betting_against_beta.git.

Liu, Yang (2019). "Novel volatility forecasting using deep learning–Long Short Term Memory Recurrent Neural Networks". In: *Expert Systems With Applications* 132, pp. 99–109.

Mittnik, Stefan, Nikolay Robinzonov, and Martin Spindler (2015). "Stock market volatility: Identifying major drivers and the nature of their impact". In: *Journal of Banking and Finance* 58, pp. 1–14.

Moon, Kyoung-Sook and Hongjoong Kim (2019). "Performance of deep learning in prediction of stock market volatility". In: *Economic Computation and Economic Cybernetics Studies and Research* 53, pp. 77–92.

Moreira, Alan and Tyler Muir (2017). "Volatility-managed portfolios". In: *Journal of Finance* 72, pp. 1611–1644.

Novy-Marx, Robert and Mihail Velikov (2016). "A taxonomy of anomalies and their trading costs". In: *Review of Financial Studies* 29, pp. 104–147.

Olah, Christopher (2015). *Understanding LSTM Networks*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Politis, Dimitris N. and Joseph P. Romano (1994). "The stationary bootstrap". In: *Journal of the American Statistical Association* 89, pp. 1303–1313.

Ropotos, Ioannis (2022). *FamaFrench2015FF5*. https://github.com/ioannisrpt/FamaFrench2015FF5.git.

Ross, Stephen. A. (1976). "The arbitrage theory of capital asset pricing". In: *Journal of Economic Theory* 13, pp. 341–360.

Schölkopf, Bernhard et al. (2000). "New Support Vector Algorithms". In: *Neural Computation* 12, pp. 1207–1245.

Sharpe, William F. (1964). "Capital asset prices: a theory of market equilibrium under conditions of risk". In: *Journal of Finance* 19, pp. 425–442.

Sheppard, Kevin (2024, Apr 16). *arch library*. https://doi.org/10.5281/zenodo.593254. bashtage/arch: Release 7.00 (Version v7.00). Zenodo.

Smola, Alex J. and Bernhard Schölkopf (2004). "A tutorial on support vector regression". In: *Statistics and Computing* 14, pp. 199–222.

Song Drechsler, Qingyi (2020). *Fama-French 3 Factor Model*. https://www.fredasongdrechsler.com/data-crunching/fama-french.

Appendix

We append the code used to replicate the factors (including the beta computation), compute the transaction costs, run the portfolio optimisation, and the two machine-learning models.

Factor Replication

```
import itertools
import numpy as np
import pandas as pd
from pandas.tseries.offsets import MonthEnd
import sqlite3


################################################################
########################## CRSP DATA ###########################
################################################################

# Access the database
```

```
crsp_compustat =
↪  sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/crsp_compustat.sqlite")


# Import the CRSP monthly data
crsp =  pd.read_sql_query(
                            sql="SELECT * FROM crsp_m",
                            con=crsp_compustat
                            )


# Rename columns
ccols = {'permno': 'PERMNO',
         'permco' : 'PERMCO',
         'ret' : 'RET',
         'prc' : 'PRC',
         'shrout' : 'SHROUT',
         'exchcd' : 'EXCHCD',
         'shrcd' : 'SHRCD'}
crsp = crsp.rename(columns = ccols)


# Dictionary to type 32bit
dictotype32 = {'PERMNO': np.int32,
               'PERMCO': np.int32,
               'EXCHCD': np.int32,
               'SHRCD': np.int32,
               'RET': np.float32}


# Convert the variables to the required type
crsp = crsp.astype(dictotype32)


# Convert the date to a datetime format
crsp['date'] = pd.to_datetime(crsp['date'])
```

```
# Line up date to be end of month
crsp['date'] = crsp['date'] + MonthEnd(0)

# Ensure that price data is positive
crsp['PRC'] = np.abs(crsp['PRC'])

# Compute the market cap at the PERMNO level, and set it to NaN if it is 0
crsp['CAP'] = crsp['PRC'] * crsp['SHROUT']
crsp['CAP'] = np.where(crsp['CAP']==0, np.nan, crsp['CAP'])

# Drop duplicates
crsp.drop_duplicates(subset=['date', 'PERMNO'], ignore_index=True, inplace=True)
crsp = crsp.sort_values(by=['PERMNO', 'date']).reset_index(drop=True)

# Define the month and year column
crsp['month'] = crsp['date'].dt.month
crsp['year'] = crsp['date'].dt.year

# Returns that are -66, -77, -88, -99 are mapped to null
crsp['RET'] = np.where(crsp['RET'] < -1, np.nan, crsp['RET'])

# Isolate the December and June market cap
crsp['DEC_CAP'] = crsp.loc[crsp['month'] == 12, 'CAP']
crsp['JUN_CAP'] = crsp.loc[crsp['month'] == 6, 'CAP']

# Fill the missing values with the preceding year's market cap
crsp['DEC_CAP'] = crsp.groupby('PERMNO')['DEC_CAP'].ffill(limit=11)
crsp['JUN_CAP'] = crsp.groupby('PERMNO')['JUN_CAP'].ffill(limit=11)



################################################################
#################### COMPUSTAT ANNUAL DATA #####################
```

```
################################################################

# Import the Compustat data
comp = pd.read_sql_query(
                              sql="SELECT * FROM comp",
                              con=crsp_compustat
                              )

# Dictionary to type 32bit
comptotype32 = {'cusip': str,
                'gvkey' : np.int32,
                'at' : np.float32,
                'ceq' : np.float32,
                'cogs' : np.float32,
                'lt' : np.float32,
                'mib' : np.float32,
                'pstk' : np.float32,
                'pstkl' : np.float32,
                'pstkrv' : np.float32,
                'sale' : np.float32,
                'seq' : np.float32,
                'txdb' : np.float32,
                'txditc' : np.float32,
                'xint' : np.float32,
                'xsga' : np.float32}

# Convert the variables to the required type
comp = comp.astype(comptotype32)

# Rename the date column and convert it to an datetime format
comp.rename(columns={'datadate':'date_compa'},inplace=True)
comp['date_compa'] = pd.to_datetime(comp['date_compa'])
```

```
# Line up date to be end of month
comp['date_compa'] = comp['date_compa'] + MonthEnd(0)

# Drop duplicates and sort by gvkey and date_jun
comp = comp.drop_duplicates(subset=['gvkey', 'date_compa'])
comp = comp.sort_values(by=['gvkey', 'date_compa'])

# Counting the number of years a company has been in Compustat
comp['count'] = comp.groupby(['gvkey']).cumcount()

## Compute Operating Profitability
# Compute the book value of equity (BE)
# Calculate SBE (SEQ, or CEQ + PSTK, or CEQ + AT - LT - MIB, or NaN)
comp['SBE'] = np.where(comp['seq'].notna(), comp['seq'],
                       np.where(comp['ceq'].notna(),
                                np.where(comp['pstk'].notna(), comp['ceq'] + comp['pstk'],
                                ↪  comp['ceq'] + comp['at'] - comp['lt'] - comp['mib']),
                                np.nan))

# Calculate BVPS (PSTKRV, or PSTKL, or PSTK, or NaN)
comp['BVPS'] = np.where(comp['pstkrv'].notna(), comp['pstkrv'],
                        np.where(comp['pstkl'].notna(), comp['pstkl'],
                                 np.where(comp['pstk'].notna(), comp['pstk'], np.nan)))

# Calculate DT (TXDITC, or TXDB + ITCB)
comp['DT'] = np.where(comp['txditc'].notna(), comp['txditc'], comp['txdb'] + comp['itcb'])

# BE
comp['BE'] = comp['SBE'] - comp['BVPS'].fillna(0) + comp['DT']

# Ensure positive book value of equity
```

```
comp['BE'] = np.where(comp['BE'] < 0, np.nan, comp['BE'])


# Compute Operating Profits

# Check if at least one of cogs, xsga, or xint exists

one_exists = comp[['cogs', 'xsga', 'xint']].notna().any(axis=1)


# operpro

comp['operpro'] = np.where(comp['sale'].notna() & one_exists,

                           comp['sale'] - comp['cogs'].fillna(0) - comp['xsga'].fillna(0)

                           ↪  - comp['xint'].fillna(0),

                           np.nan)


# Operating Profitability

comp['OP'] = np.where(comp['BE'] > 0, comp['operpro'] / comp['BE'], np.nan)


## Compute Investment

# Calculate the lagged total assets by one year

comp['at_lag1'] = comp.groupby('gvkey')['at'].shift(1)


# Compute the investment variable

comp['INV'] = np.where((comp['at'] > 0) & (comp['at_lag1'] > 0),

                       comp['at'] / comp['at_lag1'] - 1,

                       np.nan)


# Rename GVKEY

comp.rename(columns={'gvkey':'GVKEY'}, inplace=True)



#################################################################

################### COMPUSTAT QUARTERLY DATA ####################

#################################################################
```

```
# Import the Compustat Quarterly data
compq = pd.read_sql_query(
                            sql="SELECT * FROM compq",
                            con=crsp_compustat
                            )

# Dictionary to type 32bit
compqtotype32 = {'cusip' : str,
                 'atq' : np.float32,
                 'ceqq' : np.float32,
                 'ltq' : np.float32,
                 'pstkq' : np.float32,
                 'seqq' : np.float32,
                 'txditcq' : np.float32,
                 'ibq' : np.float32}

# Convert the variables to the required type
compq = compq.astype(compqtotype32)

# Rename the date column and convert it to an datetime format
compq.rename(columns={'datadate':'date_compq'},inplace=True)
compq['date_compq'] = pd.to_datetime(compq['date_compq'])

# Line up date to be end of month
compq['date_compq'] = compq['date_compq'] + MonthEnd(0)

# Drop duplicates and sort by cusip and datadate
compq = compq.drop_duplicates(subset=['cusip', 'date_compq'])
compq = compq.sort_values(by=['cusip', 'date_compq'])

## Compute the book value of equity (BEQ)
# Calculate SBES (SEQQ, or CEQQ + PSTKQ, or ATQ - LTQ, or NaN)
```

```
condition1 = compq['ceqq'].notna() & compq['pstkq'].notna()
result1 = np.where(condition1, compq['ceqq'] + compq['pstkq'], compq['atq'] -
↪  compq['ltq'])

compq['SBEQ'] = np.where(compq['seqq'].notna(), compq['seqq'],
                         np.where(compq['ceqq'].notna(), result1, np.nan))

# Calculate BVPSS (PSTKRQ, or PSTKQ, or NaN)
compq['BVPSQ'] = np.where(compq['pstkrq'].notna(), compq['pstkrq'],
                         np.where(compq['pstkq'].notna(), compq['pstkq'], np.nan))

# Calculate DTS
compq['DTQ'] = compq['txditcq']

# Compute BEQ
compq['BEQ'] = compq['SBEQ'] - compq['BVPSQ'].fillna(0) + compq['DTQ'].fillna(0)

# Keep only positive book value of equity
#compq['BEQ'] = np.where(compq['BEQ'] < 0, np.nan, compq['BEQ'])

# Lag the Book Value of Equity by one quarter
compq['BEQ_lag1'] = compq.groupby('cusip')['BEQ'].shift(1)

# Compute the ROE
compq['ROE'] = compq['ibq'] / compq['BEQ_lag1']



#####################################################################
################## MERGE COMPUSTAT AND CRSP DATA ##################
#####################################################################

# Import the link table
```

```
link_table = pd.read_sql_query(
                                sql="SELECT * FROM ccm",
                                con=crsp_compustat
                                )

# Rename the columns
link_table.rename(columns={'gvkey':'GVKEY', 'permno':'PERMNO', 'lpermco': 'PERMCO',
↪  'linkdt':'LINKDT', 'linkenddt':'LINKENDDT'}, inplace=True)

# Convert the GVKEY to a consistent type and LINKDT and LINKENDDT to a datetime format
link_table['GVKEY'] = link_table['GVKEY'].astype(np.int32)
link_table['LINKDT'] = pd.to_datetime(link_table['LINKDT'])
link_table['LINKENDDT'] = pd.to_datetime(link_table['LINKENDDT'])
link_table['LINKENDDT'] = link_table['LINKENDDT'].fillna(value = crsp.date.max())

# Inner merge between the link_table and the CRSP data where dates are in the bounds
↪  LINKDT and LINKENDDT
crsp_ccm = (crsp
            .merge(link_table, how="left", on=['PERMNO', 'PERMCO'])
            .query("(date >= LINKDT) & (date <= LINKENDDT)")
            )

# Left merge crsp_ccm with Annual Compustat
crsp_comp_ccm = pd.merge(crsp_ccm, comp, how='left', left_on=['GVKEY','date'],
↪  right_on=['GVKEY', 'date_compa'])


# Left merge crsp_comp_ccm  with Quarterly Compustat
crsp_compq = pd.merge(crsp_comp_ccm, compq, how='left', left_on=['cusip','date'],
↪  right_on=['cusip', 'date_compq'])

# Deleting unused variables
```

```
del ccols, dictotype32, crsp, comp, comptotype32, one_exists, compq, compqtotype32
del condition1, result1, link_table, crsp_ccm, crsp_comp_ccm


################################################################
############# SORT PORTFOLIOS INTO CHARACTERISTICS ##############
################################################################


# Keep only ordinary shares
crsp_compq = crsp_compq[crsp_compq['SHRCD'].isin(set([10,11]))].copy()

# Create a column that contains 1 if the stock trades on the NYSE
crsp_compq['NYSE'] = np.where(crsp_compq['EXCHCD'] == 1, 1, 0)

# Get the market equity aggregated at the PERMCO level
crsp_compq['ME'] = crsp_compq.groupby(['GVKEY', 'date'])['JUN_CAP'].transform('sum')
crsp_compq['ME_dec'] = crsp_compq.groupby(['GVKEY', 'date'])['DEC_CAP'].transform('sum')

# Book to Market
crsp_compq['BtM'] = crsp_compq['BE'] / crsp_compq['ME_dec']

# Fill the missing values up to 1 year ahead so that they are available for the bucket
↪  attribution
crsp_compq['BtM'] = crsp_compq.groupby('PERMNO')['BtM'].ffill(limit=11)
crsp_compq['OP'] = crsp_compq.groupby('PERMNO')['OP'].ffill(limit=11)
crsp_compq['INV'] = crsp_compq.groupby('PERMNO')['INV'].ffill(limit=11)


# Filter the data for June and NYSE equals 1
june_nyse_data = crsp_compq[(crsp_compq['date'].dt.month == 6) & (crsp_compq['NYSE'] ==
↪  1)]
```

```
## Size characteristic

# Group by date and compute the median of 'ME'

median_me_by_date = june_nyse_data.groupby('date')['ME'].median().reset_index()

median_me_by_date.rename(columns={'ME': 'Size breakp'}, inplace=True)


# Merge the median values back to the original DataFrame

crsp_compq = crsp_compq.merge(median_me_by_date, on='date', how='left')


# Create a column that contains Small if the stock has a small cap and Big if it has a big
↪  cap

crsp_compq['Size_bucket'] = np.where(crsp_compq['ME'] <= crsp_compq['Size breakp'],
↪  'Small',
                                     np.where(crsp_compq['ME'] > crsp_compq['Size
                                     ↪  breakp'], 'Big', 0))


# Ensure that NaNs are correctly recognized as np.nan in the 'Size_bucket' column

crsp_compq['Size_bucket'] = crsp_compq['Size_bucket'].replace('0', np.nan)



## Book to Market characteristic

# Get the breakpoints for the Book to Market

quantiles_BtM_by_date = june_nyse_data.groupby('date')['BtM'].quantile(q=[.3,
↪  .7]).unstack().reset_index()

quantiles_BtM_by_date.rename(columns={0.3: 'BtM breakp 0.3', 0.7: 'BtM breakp 0.7'},
↪  inplace=True)


# Merge the quantile values back to the original DataFrame

crsp_compq = crsp_compq.merge(quantiles_BtM_by_date, on='date', how='left')


# Create a column that contains Low if the stock has a small BtM and High if it has a big
↪  BtM
```

```
crsp_compq['BtM_bucket'] = np.where(crsp_compq['BtM'] <= crsp_compq['BtM breakp 0.3'],
↪  'Low',
                                    np.where(crsp_compq['BtM'] > crsp_compq['BtM breakp
                                    ↪  0.7'], 'High',
                                             np.where((crsp_compq['BtM'] > crsp_compq['BtM
                                             ↪  breakp 0.3']) & (crsp_compq['BtM'] <=
                                             ↪  crsp_compq['BtM breakp 0.7']), 'Mid BtM',
                                             ↪  0) ))
```

```
## Operating Profitability characteristic
# Get the breakpoints for the Operating Profitability
quantiles_OP_by_date = june_nyse_data.groupby('date')['OP'].quantile(q=[.3,
↪  .7]).unstack().reset_index()
quantiles_OP_by_date.rename(columns={0.3: 'OP breakp 0.3', 0.7: 'OP breakp 0.7'},
↪  inplace=True)

# Merge the quantile values back to the original DataFrame
crsp_compq = crsp_compq.merge(quantiles_OP_by_date, on='date', how='left')

# Create a column that contains Weak if the stock has a small OP and Robust if it has a
↪  big OP
crsp_compq['OP_bucket'] = np.where(crsp_compq['OP'] <= crsp_compq['OP breakp 0.3'],
↪  'Weak',
                                   np.where(crsp_compq['OP'] > crsp_compq['OP breakp
                                   ↪  0.7'], 'Robust',
                                            np.where((crsp_compq['OP'] > crsp_compq['OP
                                            ↪  breakp 0.3']) & (crsp_compq['OP'] <=
                                            ↪  crsp_compq['OP breakp 0.7']), 'Mid OP',
                                            ↪  0) ))
```

```
## Investment Characteristic

# Get the breakpoints for the Investment

quantiles_INV_by_date = june_nyse_data.groupby('date')['INV'].quantile(q=[.3,
↪  .7]).unstack().reset_index()

quantiles_INV_by_date.rename(columns={0.3: 'INV breakp 0.3', 0.7: 'INV breakp 0.7'},
↪  inplace=True)


# Merge the quantile values back to the original DataFrame

crsp_compq = crsp_compq.merge(quantiles_INV_by_date, on='date', how='left')


# Create a column that contains Conservative if the stock has a small INV and Aggressive
↪  if it has a big INV

crsp_compq['INV_bucket'] = np.where(crsp_compq['INV'] <= crsp_compq['INV breakp 0.3'],
↪  'Conservative',
                                    np.where(crsp_compq['INV'] > crsp_compq['INV breakp
                                    ↪  0.7'], 'Aggressive',
                                             np.where((crsp_compq['INV'] > crsp_compq['INV
                                             ↪  breakp 0.3']) & (crsp_compq['INV'] <=
                                             ↪  crsp_compq['INV breakp 0.7']), 'Mid INV',
                                             ↪  0) ))


crsp_compq['INV_bucket'] = crsp_compq['INV_bucket'].replace('0', np.nan)


# Deleting unused variables

del june_nyse_data, median_me_by_date, quantiles_BtM_by_date, quantiles_OP_by_date,
↪  quantiles_INV_by_date



######################################################################

######################## COMPUTE FF FACTORS ##########################

######################################################################
```

```
def compute_cap_weighted_returns(crsp_compq, characteristics, columns):
    """
    Compute monthly cap-weighted returns for portfolios rebalanced annually formed based
    ↪  on given characteristics and
    output an additional DataFrame with weights for the factors.

    Parameters
    ----------
    crsp_compq : Pandas DataFrame
        Contains the data to compute the returns (date, PERMNO, RET, ME).
    characteristics : List of tuples
        Characteristics on which to do the portfolio sorts.
    columns : List of strings
        Contains in which are stored the buckets of stocks.

    Returns
    -------
    pivoted_returns : Pandas DataFrame
        Returns of portfolios sorted independently on characteristics.
    weights_dfs : Dictionary of Pandas DataFrame
        Weights of the sorted portfolios

    """

    # Initialize an empty DataFrame to store the cap-weighted returns
    cap_weighted_returns = pd.DataFrame(columns=['date'] + columns +
    ↪  ['cap_weighted_return'])

    # Initialize a dictionary to store weights for each characteristic
    weights_data = {char: [] for char in characteristics}

    # Extract unique years from the 'date' column
```

```
years = crsp_compq['date'].dt.year.unique()


# Loop over each year
for year in years:
    # Loop over each set of characteristic values
    for characteristic_values in characteristics:
        # Filter stocks based on the date and characteristic values
        filter_condition = (crsp_compq['date'] == pd.Timestamp(year, 6, 30))
        for col, val in zip(columns, characteristic_values):
            filter_condition &= (crsp_compq[col] == val)

        # Select stocks that meet the filter condition
        june_stocks = crsp_compq.loc[filter_condition]

        if not june_stocks.empty:
            # Get unique stock identifiers (PERMNO)
            stock_ids = june_stocks['PERMNO'].unique()

            # Define the start and end date for the next 12 months
            start_date = pd.Timestamp(year, 7, 1)
            end_date = pd.Timestamp(year + 1, 6, 30)

            # Filter stocks that fall within the 12-month period and are in the
            ↪  selected stock IDs
            period_stocks = crsp_compq.loc[
                (crsp_compq['date'] >= start_date) &
                (crsp_compq['date'] <= end_date) &
                (crsp_compq['PERMNO'].isin(stock_ids))
            ]

            # Merge with June stocks to carry forward June characteristics
```

```
period_stocks = period_stocks.merge(june_stocks[['PERMNO']], on='PERMNO',
↪  how='left')

# Compute the total market capitalization (cap) of the period stocks
total_cap = period_stocks['JUN_CAP'].sum()

# Calculate the weight for each stock based on its June capitalization
period_stocks = period_stocks.assign(weight=period_stocks['JUN_CAP'] /
↪  total_cap)

# Append the weights data to the corresponding characteristic value
weights_data[characteristic_values].append(period_stocks[['date',
↪  'PERMNO', 'weight']])

# Compute the weighted return for each stock
period_stocks['weighted_return'] = period_stocks['RET'] *
↪  period_stocks['weight']

# Group by date and sum the weighted returns to get the cap-weighted
↪  return for each date
grouped =
↪  period_stocks.groupby('date')['weighted_return'].sum().reset_index()

# Rename columns and add characteristic values to the grouped DataFrame
grouped.columns = ['date', 'cap_weighted_return']
for col, val in zip(columns, characteristic_values):
    grouped[col] = val

# Append the grouped DataFrame to the cap_weighted_returns DataFrame
cap_weighted_returns = pd.concat([cap_weighted_returns, grouped],
↪  ignore_index=True)
```

```
    # Sort the cap_weighted_returns DataFrame by date
    cap_weighted_returns.sort_values(by='date', inplace=True)
    cap_weighted_returns.reset_index(drop=True, inplace=True)

    # Pivot the cap_weighted_returns DataFrame to have dates as rows and characteristic
    ↪  combinations as columns
    pivoted_returns = cap_weighted_returns.pivot_table(
        index='date',
        columns=columns,
        values='cap_weighted_return'
    )

    # Flatten the multi-index columns
    pivoted_returns.columns = ['_'.join(map(str, col)).replace(' ', '').replace('_', '')
    ↪  for col in pivoted_returns.columns.values]
    pivoted_returns.reset_index(inplace=True)

    # Create a dictionary of DataFrames containing weights for each characteristic
    ↪  combination
    weights_dfs = {}
    for characteristic_values, weight_list in weights_data.items():
        if weight_list:
            weights_df = pd.concat(weight_list)
            weights_df = weights_df.pivot_table(index='date', columns='PERMNO',
            ↪  values='weight')
            weights_dfs[characteristic_values] = weights_df

    return pivoted_returns, weights_dfs


# Import the list of PERMNOs
PERMNOs = pd.read_sql_query("SELECT * FROM PERMNOs", crsp_compustat)
```

```
PERMNOs = PERMNOs.astype(int)

PERMNOs = PERMNOs.iloc[:,0].to_list()


def reindex_weights(weights_dict, stock_ids):
    """
    Reindex the dataframes in the weights dictionary to have the same columns.

    Parameters
    ----------
    weights_dict: Dictionary
        Dictionary containing dataframes with portfolio weights.
    stock_ids: List
        List of stock IDs to use for reindexing columns.

    Returns
    ----------
    dict: Dictionary of pandas DataFrames
        The reindexed weights dictionary.
    """

    reindexed_weights_dict = {}
    for key in weights_dict.keys():
        reindexed_weights_dict[key] = weights_dict[key].reindex(columns=PERMNOs).fillna(0)
    return reindexed_weights_dict


# Size factor
size_buckets = [('Small',), ('Big',)]
size_column = ['Size_bucket']
size_factor, weightsSMBdict = compute_cap_weighted_returns(crsp_compq, size_buckets,
↪  size_column)
size_factor['SMB'] = size_factor['Small'] - size_factor['Big']
```

```
# Compute the SMB weights
weightsSMBdict = reindex_weights(weightsSMBdict, PERMNOs)
weightsSMB = weightsSMBdict[('Small',)].fillna(0) - weightsSMBdict[('Big',)].fillna(0)



# HML factor
size_btm_buckets = [('Small', 'Low'), ('Small', 'High'), ('Big', 'Low'), ('Big', 'High')]
size_btm_columns = ['Size_bucket', 'BtM_bucket']
HML_factor, weightsHMLdict = compute_cap_weighted_returns(crsp_compq, size_btm_buckets,
↪  size_btm_columns)
HML_factor['HML'] = 0.5 * (HML_factor['SmallHigh'] + HML_factor['BigHigh']) - 0.5 *
↪  (HML_factor['SmallLow'] + HML_factor['BigLow'])

# Compute the HML weights
weightsHMLdict = reindex_weights(weightsHMLdict, PERMNOs)
weightsHML = 0.5 * (weightsHMLdict[('Small', 'High')].fillna(0) + weightsHMLdict[('Big',
↪  'High')].fillna(0)) - 0.5 * (weightsHMLdict[('Small', 'Low')].fillna(0) +
↪  weightsHMLdict[('Big', 'Low')].fillna(0))



# RMW factor
size_op_buckets = [('Small', 'Robust'), ('Small', 'Weak'), ('Big', 'Robust'), ('Big',
↪  'Weak')]
size_op_columns = ['Size_bucket', 'OP_bucket']
RMW_factor, weightsRMWdict = compute_cap_weighted_returns(crsp_compq, size_op_buckets,
↪  size_op_columns)
RMW_factor['RMW'] = 0.5 * (RMW_factor['SmallRobust'] + RMW_factor['BigRobust']) - 0.5 *
↪  (RMW_factor['SmallWeak'] + RMW_factor['BigWeak'])

# Compute the RMW weights
weightsRMWdict = reindex_weights(weightsRMWdict, PERMNOs)
```

```
weightsRMW = 0.5 * (weightsRMWdict[('Small', 'Robust')].fillna(0) + weightsRMWdict[('Big',
↪  'Robust')].fillna(0)) - 0.5 * (weightsRMWdict[('Small', 'Weak')].fillna(0) +
↪  weightsRMWdict[('Big', 'Weak')].fillna(0))



# CMA Factor
size_inv_buckets = [('Small', 'Conservative'), ('Small', 'Aggressive'), ('Big',
↪  'Conservative'), ('Big', 'Aggressive')]
size_inv_columns = ['Size_bucket', 'INV_bucket']
CMA_factor, weightsCMAdict = compute_cap_weighted_returns(crsp_compq, size_inv_buckets,
↪  size_inv_columns)
CMA_factor['CMA'] = 0.5 * (CMA_factor['SmallConservative'] +
↪  CMA_factor['BigConservative']) - 0.5 * (CMA_factor['SmallAggressive'] +
↪  CMA_factor['BigAggressive'])

# Compute the CMA weights
weightsCMAdict = reindex_weights(weightsCMAdict, PERMNOs)
weightsCMA = 0.5 * (weightsCMAdict[('Small', 'Conservative')].fillna(0) +
↪  weightsCMAdict[('Big', 'Conservative')].fillna(0)) - 0.5 * (weightsCMAdict[('Small',
↪  'Aggressive')].fillna(0) + weightsCMAdict[('Big', 'Aggressive')].fillna(0))



# Set the date as the index
size_factor.set_index('date', inplace=True)
HML_factor.set_index('date', inplace=True)
RMW_factor.set_index('date', inplace=True)
CMA_factor.set_index('date', inplace=True)

# Deleting unused variables
del size_buckets, size_column, weightsSMBdict, size_btm_buckets, size_btm_columns,
↪  weightsHMLdict
```

```
del size_op_buckets, size_op_columns, weightsRMWdict, size_inv_buckets, size_inv_columns,
↪  weightsCMAdict
```

```
######################################################################
######################## COMPUTE FF FACTORS ##########################
######################################################################

# Load the Fama French Factors
ff_original_fact = pd.read_csv("/Users/Ilyas/Documents/Mémoire/_FF 5 Factors_2x3.csv",
↪  skiprows=3, nrows=729)
ff_original_fact.rename(columns={"Unnamed: 0":"date"}, inplace="True")
ff_original_fact["date"] = pd.to_datetime(ff_original_fact["date"], format="%Y%m") +
↪  MonthEnd(0)
ff_original_fact.set_index(ff_original_fact["date"], drop=True, inplace=True)

# Merge the Risk-Free rate with the stock data
crsp_compq = crsp_compq.merge(ff_original_fact.RF, how='left', on='date')

# Shift the market cap to get the weights from the previous month
crsp_compq['ME_shifted'] = crsp_compq.groupby('PERMNO')['ME'].shift(1)

ME_shifted_total = crsp_compq.groupby('date')['ME_shifted'].apply(lambda x: x.sum())
ME_shifted_total.name = 'ME_shifted_total'
crsp_compq = crsp_compq.merge(ME_shifted_total, how='left', left_on='date',
↪  right_index=True)

# Calculate weights
crsp_compq['weight'] = crsp_compq['ME_shifted'].div(crsp_compq['ME_shifted_total'])
```

```
# Calculate portfolio returns
crsp_compq['Mkt-RF'] = (crsp_compq['RET'] - crsp_compq['RF'].div(100)) *
↪  crsp_compq['weight']

# Compute the portfolio return for each month
market_factor = crsp_compq.groupby('date')['Mkt-RF'].sum()



# Create a DataFrame to store weights for each month
weightsMarket = crsp_compq.pivot_table(index='date', columns='PERMNO',
↪  values='weight').fillna(0)



#####################################################################
######### COMPARE THE RESULTS WITH THE ORIGINAL FACTORS #########
#####################################################################

# Merge the replicated factors with the original ones into a single dataframe
ffcomp = pd.merge(ff_original_fact, market_factor, how='inner', left_index=True,
↪  right_index=True)
ffcomp = pd.merge(ffcomp, size_factor['SMB'], how='inner', left_index=True,
↪  right_index=True)
ffcomp = pd.merge(ffcomp, HML_factor['HML'], how='inner', left_index=True,
↪  right_index=True)
ffcomp = pd.merge(ffcomp, RMW_factor['RMW'], how='inner', left_index=True,
↪  right_index=True)
ffcomp = pd.merge(ffcomp, CMA_factor['CMA'], how='inner', left_index=True,
↪  right_index=True)

# Get the correlation between the original and the replicated factor (after 1966)
ffcomp66 = ffcomp[ffcomp['date']>='01/01/1966']
```

```
print('Correlation with the original series')
print('Market', f"{np.corrcoef(ffcomp66['Mkt-RF_x'], ffcomp66['Mkt-RF_y'])[1][0]:.0%}")
print('SMB', f"{np.corrcoef(ffcomp66['SMB_x'], ffcomp66['SMB_y'])[1][0]:.0%}")
print('HML', f"{np.corrcoef(ffcomp66['HML_x'], ffcomp66['HML_y'])[1][0]:.0%}")
print('RMW', f"{np.corrcoef(ffcomp66['RMW_x'], ffcomp66['RMW_y'])[1][0]:.0%}")
print('CMA', f"{np.corrcoef(ffcomp66['CMA_x'], ffcomp66['CMA_y'])[1][0]:.0%}")



# Removing unused variables
crsp_compq.drop(['month', 'year', 'linktype', 'linkprim', 'LINKDT', 'LINKENDDT',
↪  'date_compa', 'at',
'pstkl', 'txditc', 'pstkrv', 'seq', 'pstk', 'ceq', 'lt', 'txdb', 'itcb',
'cusip', 'capx', 'oancf', 'sale', 'cogs', 'xint', 'xsga', 'dp', 'mib',
'fyr', 'fyear', 'SBE', 'BVPS', 'DT','operpro','seqq', 'txditcq', 'ceqq', 'pstkq', 'atq',
'ltq', 'pstkrq', 'SBEQ', 'BVPSQ', 'DTQ'], inplace=True, axis=1)

del ffcomp, ffcomp66

######################################################################
#################### REPLICATE HXZ FACTORS ####################
######################################################################

# Removing financial firms and firms with negative book equity
crsp_compq = crsp_compq[~crsp_compq['siccd'].between(6000, 6999)]

crsp_compq = crsp_compq[~crsp_compq['BEQ'].lt(0)]

# Ensure 'rdq' is in datetime format
crsp_compq.loc[:, 'rdq'] = np.where((pd.isnull(crsp_compq['rdq'])) &
↪  (~pd.isnull(crsp_compq['ibq'])), crsp_compq['date'], crsp_compq['rdq'])
crsp_compq['rdq'] = pd.to_datetime(crsp_compq['rdq'])
```

```
# Calculate the target date for ROE
crsp_compq['target_date'] = crsp_compq['rdq'] + MonthEnd(2)

# Filter for NYSE stocks
crsp_compq_nyse = crsp_compq[crsp_compq['NYSE'] == 1]

# Prepare a DataFrame that shifts the ROE values based on the target_date
crsp_compq_shifted = crsp_compq[['PERMNO', 'ROE', 'target_date']].copy()
crsp_compq_shifted.rename(columns={'ROE': 'shifted_ROE', 'target_date': 'date'},
↪  inplace=True)

# Merge shifted ROE back to the main DataFrame
crsp_compq = crsp_compq.merge(crsp_compq_shifted, on=['PERMNO', 'date'], how='left')

# Fill the ROE column until the last date where it can be used
crsp_compq['ROE'] = crsp_compq.groupby('PERMNO')['ROE'].ffill(limit=5)
crsp_compq['shifted_ROE'] = crsp_compq.groupby('PERMNO')['shifted_ROE'].ffill(limit=5)

# Filter the data to ensure the ROE value is valid within the 6-month window
crsp_compq['ROE_valid'] = np.where(crsp_compq['ROE'] == np.nan, -np.inf,
↪  crsp_compq['shifted_ROE'])
crsp_compq['ROE_valid'] = crsp_compq['ROE_valid'].replace(-np.inf, np.nan)

# Filter the data for June and NYSE equals 1
nyse_data = crsp_compq[(crsp_compq['NYSE'] == 1)]

## Return On Equity characteristic
# Get the breakpoints for the ROE
```

```
quantiles_ROE_by_date = nyse_data.groupby('date')['ROE_valid'].quantile(q=[.3,
↪  .7]).unstack().reset_index()
quantiles_ROE_by_date.rename(columns={0.3: 'ROE breakp 0.3', 0.7: 'ROE breakp 0.7'},
↪  inplace=True)

# Merge the quantile values back to the original DataFrame
crsp_compq = crsp_compq.merge(quantiles_ROE_by_date, on='date', how='left')

# Create a column that contains LowROE if the stock has a small ROE and HighROE if it has
↪  a big ROE
crsp_compq['ROE_bucket'] = np.where(crsp_compq['ROE_valid'] <= crsp_compq['ROE breakp
↪  0.3'], 'LowROE',
                                    np.where(crsp_compq['ROE_valid'] > crsp_compq['ROE
                                    ↪  breakp 0.7'], 'HighROE',
                                             np.where((crsp_compq['ROE_valid'] >
                                             ↪  crsp_compq['ROE breakp 0.3']) &
                                             ↪  (crsp_compq['ROE_valid'] <=
                                             ↪  crsp_compq['ROE breakp 0.7']), 'Mid ROE',
                                             ↪  0) ))


# Forward fill the yearly buckets for Size and INV
crsp_compq['Size_bucket'] = crsp_compq.groupby('PERMNO')['Size_bucket'].ffill(limit=11)
crsp_compq['INV_bucket'] = crsp_compq.groupby('PERMNO')['INV_bucket'].ffill(limit=11)



def compute_cap_weighted_returns_monthly_rebalance(crsp_compq, characteristics, columns):
    """
    Compute monthly cap-weighted returns for portfolios rebalanced monthly formed based on
    ↪  given characteristics and
    output an additional DataFrame with weights for the factors.
```

```
    Parameters:
    crsp_compq: Pandas DataFrame
        DataFrame containing stock data with necessary columns.
    characteristics (list of tuples): List of tuples where each tuple contains
    ↪  characteristic values.
    columns (list of str): List of columns corresponding to the characteristic values.

    Returns:
    pivoted_returns: Pandas DataFrame
        DataFrame with cap-weighted returns for each combination of characteristics.
    weights_dfs: Dictionary of DataFrames
        Weights for each characteristic combination.
    """
    # Initialize the list to store results
    results = []
    weights_data = {char: [] for char in characteristics}

    # Extract unique periods (months)
    crsp_compq['period'] = crsp_compq['date'].dt.to_period('M')

    # Compute the cap-weighted returns for each combination of characteristics
    for characteristic_values in characteristics:
        # Create filter conditions for each combination of characteristics
        filter_conditions = [crsp_compq[col] == val for col, val in zip(columns,
        ↪  characteristic_values)]
        combined_filter = filter_conditions[0]
        for condition in filter_conditions[1:]:
            combined_filter &= condition

        # Apply the filter to the DataFrame
        filtered_data = crsp_compq.loc[combined_filter].copy()
```

```
# Check if filtered_data is not empty
if filtered_data.empty:
    print(f"No data for characteristics {characteristic_values}")
    continue

# Compute market cap weights for each period
filtered_data['total_cap'] =
↪  filtered_data.groupby('date')['ME_shifted'].transform('sum')
filtered_data['weight'] = filtered_data['ME_shifted'] / filtered_data['total_cap']

# Store weights data
weights_data[characteristic_values].append(filtered_data[['date', 'PERMNO',
↪  'weight']])

# Compute weighted returns for each period
filtered_data['weighted_return'] = filtered_data['RET'] * filtered_data['weight']

# Aggregate the weighted returns by date
aggregated_returns =
↪  filtered_data.groupby(['date'])['weighted_return'].sum().reset_index()

#aggregated_returns = filtered_data.groupby(['date',
↪  'period'])['weighted_return'].sum().reset_index()

# Assign characteristic values
for col, val in zip(columns, characteristic_values):
    aggregated_returns[col] = val

# Append the results to the list
results.append(aggregated_returns)
```

```
# Concatenate all results into a single DataFrame
if not results:
    return pd.DataFrame(), {}

cap_weighted_returns = pd.concat(results, ignore_index=True)

# Sort the final DataFrame by date
cap_weighted_returns.sort_values(by='date', inplace=True)
cap_weighted_returns.reset_index(drop=True, inplace=True)

# Pivot the DataFrame to get one row per month and separate columns for each
↪  characteristic combination
pivoted_returns = cap_weighted_returns.pivot_table(
    index='date',
    columns=columns,
    values='weighted_return'
)

# Flatten the columns
pivoted_returns.columns = ['_'.join(map(str, col)).replace(' ', '').replace('_', '')
↪  for col in pivoted_returns.columns.values]

# Reset index to make 'date' a column again
pivoted_returns.reset_index(inplace=True)

# Combine weights data into dataframes for each characteristic combination
weights_dfs = {}
for characteristic_values, weight_list in weights_data.items():
    if weight_list:
        weights_df = pd.concat(weight_list)
        weights_df = weights_df.pivot_table(index='date', columns='PERMNO',
        ↪  values='weight')
```

```
            weights_dfs[characteristic_values] = weights_df

    return pivoted_returns, weights_dfs



# List of individual characteristics
size_buckets = ['Small', 'Big']
inv_buckets = ['Conservative', 'Aggressive', 'Mid INV']
roe_buckets = ['LowROE', 'HighROE', 'Mid ROE']

# Generate combinations of characteristics
characteristics = list(itertools.product(size_buckets, inv_buckets, roe_buckets))

columns = ['Size_bucket', 'INV_bucket', 'ROE_bucket']
triple_sort, weights_dict = compute_cap_weighted_returns_monthly_rebalance(crsp_compq,
↪  characteristics, columns)

triple_sort.loc[:,'date'] = pd.to_datetime(triple_sort['date'])
triple_sort.set_index('date', inplace=True)

# Get the I/A Factor
triple_sort['I/A'] = (
    triple_sort['SmallConservativeLowROE'].fillna(0) +
    triple_sort['SmallConservativeHighROE'].fillna(0) +
    triple_sort['SmallConservativeMidROE'].fillna(0) +
    triple_sort['BigConservativeLowROE'].fillna(0) +
    triple_sort['BigConservativeHighROE'].fillna(0) +
    triple_sort['BigConservativeMidROE'].fillna(0)
) / 6 - (
    triple_sort['SmallAggressiveLowROE'].fillna(0) +
    triple_sort['SmallAggressiveHighROE'].fillna(0) +
    triple_sort['SmallAggressiveMidROE'].fillna(0) +
```

```
    triple_sort['BigAggressiveLowROE'].fillna(0) +
    triple_sort['BigAggressiveHighROE'].fillna(0) +
    triple_sort['BigAggressiveMidROE'].fillna(0)
) / 6

# Get the ROE Factor
triple_sort['ROE'] = (
    triple_sort['SmallConservativeHighROE'].fillna(0) +
    triple_sort['SmallAggressiveHighROE'].fillna(0) +
    triple_sort['SmallMidINVHighROE'].fillna(0) +
    triple_sort['BigConservativeHighROE'].fillna(0) +
    triple_sort['BigAggressiveHighROE'].fillna(0) +
    triple_sort['BigMidINVHighROE'].fillna(0)
) / 6 - (
    triple_sort['SmallConservativeLowROE'].fillna(0) +
    triple_sort['SmallAggressiveLowROE'].fillna(0) +
    triple_sort['SmallMidINVLowROE'].fillna(0) +
    triple_sort['BigConservativeLowROE'].fillna(0) +
    triple_sort['BigAggressiveLowROE'].fillna(0) +
    triple_sort['BigMidINVLowROE'].fillna(0)
) / 6



# Reindex the weights_dict so that all dataframes have the same columns
weights_dict = reindex_weights(weights_dict, PERMNOs)

# Calculate the weights for the I/A factor
weightsIA = (
    weights_dict[('Small', 'Conservative', 'LowROE')].fillna(0) +
    weights_dict[('Small', 'Conservative', 'HighROE')].fillna(0) +
    weights_dict[('Small', 'Conservative', 'Mid ROE')].fillna(0) +
    weights_dict[('Big', 'Conservative', 'LowROE')].fillna(0) +
```

```
    weights_dict[('Big', 'Conservative', 'HighROE')].fillna(0) +
    weights_dict[('Big', 'Conservative', 'Mid ROE')].fillna(0)
) / 6 - (
    weights_dict[('Small', 'Aggressive', 'LowROE')].fillna(0) +
    weights_dict[('Small', 'Aggressive', 'HighROE')].fillna(0) +
    weights_dict[('Small', 'Aggressive', 'Mid ROE')].fillna(0) +
    weights_dict[('Big', 'Aggressive', 'LowROE')].fillna(0) +
    weights_dict[('Big', 'Aggressive', 'HighROE')].fillna(0) +
    weights_dict[('Big', 'Aggressive', 'Mid ROE')].fillna(0)
) / 6



# Calculate the weights for the ROE factor
weightsROE = (
    weights_dict[('Small', 'Conservative', 'HighROE')].fillna(0) +
    weights_dict[('Small', 'Aggressive', 'HighROE')].fillna(0) +
    weights_dict[('Small', 'Mid INV', 'HighROE')].fillna(0) +
    weights_dict[('Big', 'Conservative', 'HighROE')].fillna(0) +
    weights_dict[('Big', 'Aggressive', 'HighROE')].fillna(0) +
    weights_dict[('Big', 'Mid INV', 'HighROE')].fillna(0)
) / 6 - (
    weights_dict[('Small', 'Conservative', 'LowROE')].fillna(0) +
    weights_dict[('Small', 'Aggressive', 'LowROE')].fillna(0) +
    weights_dict[('Small', 'Mid INV', 'LowROE')].fillna(0) +
    weights_dict[('Big', 'Conservative', 'LowROE')].fillna(0) +
    weights_dict[('Big', 'Aggressive', 'LowROE')].fillna(0) +
    weights_dict[('Big', 'Mid INV', 'LowROE')].fillna(0)
) / 6



# Import the 18 portfolio returns series from Hou, Xue, Zhang
```

```
qfactors = pd.read_csv('/Users/Ilyas/Documents/Mémoire/qfactors.csv', sep=',')
qfactors['date'] = pd.to_datetime(qfactors[['year', 'month']].assign(day=1)) + MonthEnd(0)

# Rename the columns with explicit casting to avoid dtype issues
qfactors['rank_ME'] = np.where(qfactors['rank_ME'] == 1, 'Small', 'Big').astype(str)
qfactors['rank_IA'] = np.where(qfactors['rank_IA'] == 1, 'Conservative',
                               np.where(qfactors['rank_IA'] == 2, 'Mid INV',
                               ↪  'Aggressive')).astype(str)
qfactors['rank_ROE'] = np.where(qfactors['rank_ROE'] == 1, 'LowROE',
                                np.where(qfactors['rank_ROE'] == 2, 'Mid ROE',
                                ↪  'HighROE')).astype(str)

# Compute the returns of the I/A factor
pivot1 = qfactors.pivot(columns=['rank_IA','rank_ME', 'rank_ROE'], index='date',
↪  values='ret_vw')
HXZ_factors = pd.DataFrame(index=pivot1.index.tolist())
HXZ_factors['I/A'] = pivot1.loc[:,'Conservative'].mean(axis=1) -
↪  pivot1.loc[:,'Aggressive'].mean(axis=1)

# Compute the returns of the ROE factor
pivot2 = qfactors.pivot(columns=['rank_ROE','rank_IA','rank_ME'], index='date',
↪  values='ret_vw')
HXZ_factors['ROE'] = pivot2.loc[:,'HighROE'].mean(axis=1) -
↪  pivot2.loc[:,'LowROE'].mean(axis=1)

# Assuming triple_sort and CMA_factor are already defined DataFrames
# Merge the replicated factors with the original ones into a single dataframe
HXZcomp = pd.merge(HXZ_factors, triple_sort[['I/A','ROE']], how='inner', left_index=True,
↪  right_index=True)

# Get the correlation between the original and the replicated factor
print('ROE', f"{np.corrcoef(HXZcomp['ROE_x'], HXZcomp['ROE_y'])[1][0]:.0%}")
```

```
print('I/A', f"{np.corrcoef(HXZcomp['I/A_x'], HXZcomp['I/A_y'])[1][0]:.0%}")


# Removing unused variables
del crsp_compq_nyse, crsp_compq_shifted, nyse_data, quantiles_ROE_by_date
del size_buckets, inv_buckets, roe_buckets, characteristics, columns, weights_dict
del qfactors, pivot1, pivot2, HXZ_factors, HXZcomp



######################################################################
######################### MOMENTUM FACTOR ############################
######################################################################


# Define the lagged returns
crsp_compq['1 + RET'] = np.where((crsp_compq['RET'] < -1) & (crsp_compq['RET'] != -99),
↪  np.nan, crsp_compq['RET'] + 1)
crsp_compq.loc[:, '1 + RET'] = np.where(crsp_compq['RET'] == -99, 1, crsp_compq['1 +
↪  RET'])


def rolling_cumprod(x):
    if x.isna().any():
        return np.nan
    else:
        return np.prod(x)


crsp_compq['11-month RET'] = crsp_compq.groupby('PERMNO')['1 +
↪  RET'].rolling(window=11).apply(rolling_cumprod, raw=False).reset_index(level=0,
↪  drop=True)
crsp_compq['11-month RET'] -= 1
crsp_compq['11-month RET_lag2'] = crsp_compq.groupby('PERMNO')['11-month RET'].shift(2)


# Variables used in conditions: price at the end of month t-13 and good return at the end
↪  of month t-2
```

```
crsp_compq['PRC_lag13'] = crsp_compq.groupby('PERMNO')['PRC'].shift(13)
crsp_compq['RET_lag2'] = crsp_compq.groupby('PERMNO')['RET'].shift(2)
crsp_compq['RET_lag2'] = crsp_compq['RET_lag2'].fillna(-100)

# Compute the breakpoints for the MOM factor
mom_nyse_data = crsp_compq[(crsp_compq['NYSE'] == 1) & (crsp_compq['PRC_lag13'].notna()) &
↪  (crsp_compq['RET_lag2'] > -2 )]

quantiles_MOM_by_date = mom_nyse_data.groupby('date')['11-month RET_lag2'].quantile(q=[.3,
↪  .7]).unstack().reset_index()
quantiles_MOM_by_date.rename(columns={0.3: 'MOM breakp 0.3', 0.7: 'MOM breakp 0.7'},
↪  inplace=True)

# Merge the quantile values back to the original DataFrame
crsp_compq = crsp_compq.merge(quantiles_MOM_by_date, on='date', how='left')

# Create a column that contains Weak if the stock has a small OP and Robust if it has a
↪  big OP
crsp_compq['MOM_bucket'] = np.where(crsp_compq['11-month RET_lag2'] <= crsp_compq['MOM
↪  breakp 0.3'], 'Low MOM',
                                    np.where(crsp_compq['11-month RET_lag2'] >
                                    ↪  crsp_compq['MOM breakp 0.7'], 'High MOM',
                                            np.where((crsp_compq['11-month RET_lag2'] >
                                            ↪  crsp_compq['MOM breakp 0.3']) &
                                            ↪  (crsp_compq['11-month RET_lag2'] <=
                                            ↪  crsp_compq['MOM breakp 0.7']), 'Mid MOM',
                                            ↪  0) ))

crsp_compq['Size_bucket'] = crsp_compq.groupby('PERMNO')['Size_bucket'].ffill(limit=11)
crsp_compq['MOM_bucket'] = crsp_compq.groupby('PERMNO')['MOM_bucket'].ffill(limit=11)
```

```
size_MOM_buckets = [('Small', 'Low MOM'), ('Small', 'High MOM'), ('Big', 'Low MOM'),
↪  ('Big', 'High MOM')]
size_MOM_columns = ['Size_bucket', 'MOM_bucket']
MOM_factor, weightsMOMdict = compute_cap_weighted_returns_monthly_rebalance(crsp_compq,
↪  size_MOM_buckets, size_MOM_columns)
MOM_factor['MOM'] = 0.5 * (MOM_factor['SmallHighMOM'] + MOM_factor['BigHighMOM']) - 0.5 *
↪  (MOM_factor['SmallLowMOM'] + MOM_factor['BigLowMOM'])

# Compute the CMA weights
weightsMOMdict = reindex_weights(weightsMOMdict, PERMNOs)
weightsMOM = 0.5 * (weightsMOMdict[('Small', 'High MOM')].fillna(0) +
↪  weightsMOMdict[('Big', 'High MOM')].fillna(0)) - 0.5 * (weightsMOMdict[('Small', 'Low
↪  MOM')].fillna(0) + weightsMOMdict[('Big', 'Low MOM')].fillna(0))



MOM_factor.set_index('date', inplace=True)

# Compare with the Momentum factor from French's website
# Load the Fama French Factor
ff_MOM_fact = pd.read_csv("/Users/Ilyas/Documents/Mémoire/F-F_Momentum_Factor.csv",
↪  skiprows=13, nrows=1167)
ff_MOM_fact.rename(columns={"Unnamed: 0":"date", 'Mom   ':'MOM'}, inplace="True")
ff_MOM_fact["date"] = pd.to_datetime(ff_MOM_fact["date"], format="%Y%m") + MonthEnd(0)
ff_MOM_fact.set_index(ff_MOM_fact["date"], drop=True, inplace=True)

# Merge the replicated factors with the original ones into a single dataframe
ffcompMOM = pd.merge(ff_MOM_fact, MOM_factor['MOM'], how='inner', left_index=True,
↪  right_index=True)



# Get the correlation between the original and the replicated factor (after 1966)
ffcompMOM66 = ffcompMOM[ffcompMOM['date']>='01/01/1966']
```

```
print('MOM', f"{np.corrcoef(ffcompMOM66['MOM_x'], ffcompMOM66['MOM_y'])[1][0]:.0%}")


# Removing unused variables

del mom_nyse_data, quantiles_MOM_by_date, size_MOM_buckets, size_MOM_columns,
↪  weightsMOMdict

del ff_MOM_fact, ffcompMOM, ffcompMOM66

}
```

Beta Computation

```
import pandas as pd

import numpy as np

import sqlite3


######################################################################

######################### COMPUTE BETAS ##########################

######################################################################


# Access the database

crsp_compustat =
↪  sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/crsp_compustat.sqlite")


crsp_d = pd.read_sql_query(

    sql="SELECT * FROM crsp_d",

    con=crsp_compustat

)

crsp_d['date'] = pd.to_datetime(crsp_d['date'])


# Rename the columns

crsp_d.rename(columns={'permno':'PERMNO','prc':'PRC'}, inplace=True)


# Dictionary to type32

linktotype = {'PERMNO': int,
```

```
                'PRC': float}


# Set the type of the variables
crsp_d = crsp_d.astype(linktotype)


# Load the tbill data
treasury = pd.read_csv('/Users/Ilyas/Documents/Mémoire/TBill 2.csv')
treasury.rename(columns={'caldt':'date', 't30ret' : 'TBill'}, inplace=True)
treasury['date'] = pd.to_datetime(treasury['date'])


# Load the market index data
market_index = pd.read_csv('/Users/Ilyas/Documents/Mémoire/Market Index.csv')
market_index.rename(columns={'caldt':'date', 'vwretd':'Market RET'}, inplace=True)
market_index['date'] = pd.to_datetime(market_index['date'])
market_index = market_index.merge(treasury, how='left', on='date')
market_index.set_index('date', inplace=True)


market_index.loc[:,'TBill'] = market_index['TBill'].ffill(limit=31)


# Compute the log returns and overlapping returns
market_index['Exc RET'] = market_index['Market RET'] - market_index['TBill']
market_index['1 + RET'] = market_index['Exc RET'].add(1)
market_index['log(1+RET)'] = np.log(market_index['1 + RET'])
market_index['Overlapping Market RET'] =
↪  market_index['log(1+RET)'].rolling(window=3).sum().shift(-3)


# Compute the rolling market volatility
market_index['Market Vol'] =  market_index['log(1+RET)'].rolling(window=120,
↪  min_periods=120).std()



# Merge the data
```

```
data = pd.merge(crsp_d, market_index.rename(columns={'vwretd':'Market_ret'}), how='left',
↪  on='date')

# Set the prices to be positive
data['PRC'] = data['PRC'].abs()

# Get the list of PERMNOs:
PERMNOs = crsp_d['PERMNO'].unique().tolist()

# Set the date and the PERMNO as index
data.set_index(['PERMNO','date'], inplace=True)

# Create an empty dataframe that will hold the transaction costs of every stock
betas = pd.DataFrame(index=pd.date_range(start='1966-01-01',end='2022-12-31', freq="ME"))



# Loop over every stock
for PERMNO in PERMNOs:
    # Get the data of the stock PERMNO
    stock = data.loc[PERMNO].copy()

    # Set the dates as the index
    stock.reset_index(inplace=True)
    stock.set_index('date', inplace=True)

    # Compute the stock's log returns
    stock['Stock Exc RET'] = (stock['PRC'] / stock['PRC'].shift(1)) -stock['TBill']
    stock['Stock Log_RET'] = np.log(1 + stock['Stock Exc RET'])

    # Compute the stock's volatiliy
    stock['Stock Vol'] =  stock['Stock Log_RET'].rolling(window=250,
    ↪  min_periods=120).std()
```

```
    # Compute the stock's 3-day overlapping returns
    stock['Overlapping Stock RET'] = stock['Stock
    ↪  Log_RET'].rolling(window=3).sum().shift(-3)

    # Compute the correlation with the market's returns
    stock['Correlation'] = stock['Overlapping Stock RET'].rolling(window=1250,
    ↪  min_periods=750).corr(stock['Overlapping Market RET'])

    # Compute the stock's beta
    stock['beta'] = stock['Correlation'] * (stock['Stock Vol'] / stock['Market Vol'])

    # Get the mean two-day bid-ask spread per month
    beta = stock['beta'].resample('ME').mean()

    # Add the average to the dataframe
    betas = betas.merge(beta.rename(PERMNO), how='left', left_index=True,
    ↪  right_index=True)


# Split the columns into 50 parts
n_splits = 50
columns_split = np.array_split(betas.columns, n_splits)

# Save each part to the SQL database
for i, cols in enumerate(columns_split, 1):
    part_df = betas[cols]
    part_df.to_sql(name=f'betas_part_{i}',
                con=crsp_compustat,
                if_exists="replace",
                index=True)
```

Transaction Costs

```
import numpy as np

import pandas as pd

import sqlite3

import statsmodels.api as sm

from pandas.tseries.offsets import MonthEnd


#################################################################

########################### LOAD DATA ###########################

#################################################################


# Access the database

crsp_compustat =

↪  sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/crsp_compustat.sqlite")


# Import the CRSP monthly data

crspd =  pd.read_sql_query(

                            sql="SELECT * FROM crsp_d",

                            con=crsp_compustat

                            )


# Rename the columns

crspd.rename(columns={'permno':'PERMNO','bidlo':'LOW','askhi':'HIGH','prc':'CLOSE'},

↪  inplace=True)


# Dictionary to type32

linktotype = {'PERMNO': int,

              'LOW': float,

              'HIGH': float,

              'CLOSE': float}


# Set the type of the variables
```

```
crspd = crspd.astype(linktotype)


# Create a month column and get the list years

crspd['date'] = pd.to_datetime(crspd['date'], errors='coerce')


# Get the list of PERMNOs:

PERMNOs = crspd['PERMNO'].unique().tolist()


# Compute the log high, low and close prices

crspd.loc[:,'HIGH'] = np.log(crspd.loc[:,'HIGH'].abs())

crspd.loc[:,'LOW'] = np.log(crspd.loc[:,'LOW'].abs())

crspd.loc[:,'CLOSE'] = np.log(crspd.loc[:,'CLOSE'].abs())


# Create the mid-price column

crspd['MID'] = (crspd['HIGH'] + crspd['LOW']) / 2


# Set the date and the PERMNO as index

#crspd.set_index('PERMNO', inplace=True)

crspd.set_index(['PERMNO','date'], inplace=True)


# Delete unused columns

crspd.drop(columns=['LOW', 'HIGH'], inplace=True)


######################################################################

################### COMPUTE TRANSACTION COSTS ###################

######################################################################


# Create an empty dataframe that will hold the transaction costs of every stock

t_costs = pd.DataFrame(index=pd.date_range(start='1966-01-01',end='2022-12-31',

↪  freq="ME"))


# Loop over every stock
```

```
for PERMNO in PERMNOs:

    # Get the data of the stock PERMNO
    data = crspd.loc[PERMNO].copy()

    # Set the dates as the index
    data.reset_index(inplace=True)
    data.set_index('date', inplace=True)

    # Create the shifted (d+1) close and high log prices
    data['shift_MID'] = data['MID'].shift(-1)
    data['shift_CLOSE'] = data['CLOSE'].shift(-1)

    # Compute the two-day squared bid-ask spread
    data['squared_spread'] = (4 * (data['CLOSE'] - data['MID']) * (data['shift_CLOSE'] -
    ↪  data['shift_MID'])).clip(lower=0)

    # Compute the two-day bid-ask spread
    data['spread'] = np.sqrt(data['squared_spread'])

    # Get the mean two-day bid-ask spread per month
    avg_spread = data['spread'].resample('ME').mean()

    # Add the average to the dataframe
    t_costs = t_costs.merge(avg_spread.rename(PERMNO), how='left', left_index=True,
    ↪  right_index=True)

# Save to SQL
n_splits = 50 # Split the columns into 50 parts
columns_split = np.array_split(t_costs.columns, n_splits)

# Save each part to the SQL database
```

```
for i, cols in enumerate(columns_split, 1):
    part_df = t_costs[cols]
    part_df.to_sql(name=f't_costs_filled_part_{i}',
                con=crsp_compustat,
                if_exists="replace",
                index=True)

#####################################################################
############### COMPUTE MISSING TRANSACTION COSTS ###############
#####################################################################

##  Compute the idiosyncratic volatility

# Load Market data
market_returns = pd.read_csv('/Users/Ilyas/Documents/Mémoire/Market Index.csv')
market_returns.rename(columns={'caldt':'date', 'vwretd':'Market RET'}, inplace=True)
market_returns['date'] = pd.to_datetime(market_returns['date'])

# Load the risk-free rate data from French's website
FF_data =
↪  pd.read_csv("/Users/Ilyas/Documents/Mémoire/F-F_Research_Data_Factors_daily.csv",
↪  skiprows=3, nrows=25732)
FF_data.rename(columns={"Unnamed: 0":"date"}, inplace="True")
FF_data["date"] = pd.to_datetime(FF_data["date"], format="%Y%m%d")

FF_data.RF = FF_data.RF/100

#Load stock data
crspd = pd.read_sql_query(
    sql="SELECT * FROM t_costs_crspd",
    con=crsp_compustat
)
```

```
crspd['date'] = pd.to_datetime(crspd['date'])


# Compute the daily returns for each stock

crspd['daily_return'] = crspd.groupby('PERMNO')['CLOSE'].pct_change(fill_method=None)


# Add the market data and the risk-free rate to the crspd dataframe

crspd = crspd.merge(market_returns, how='left', on='date')

crspd = crspd.merge(FF_data[['date', 'RF']], how='left', on='date')


# Compute the market's excess returns

crspd['Excess Market return'] = crspd['Market RET'] - crspd['RF']

crspd = crspd.dropna(subset=['Excess Market return'])


# Sort stocks by PERMNO and date

crspd = crspd.sort_values(['PERMNO','date'])


def rolling_std_residuals(group, window=63, min_periods=60):

    """

    Calculate the rolling standard deviation of residuals from a regression of daily

    ↪  returns on excess market returns.


    Parameters

    ----------

    group : Pandas DataFrame

        DataFrame containing the daily returns and excess market returns for a group of

        ↪  stocks.

    window : int, optional

        The size of the rolling window (default is 63).

    min_periods : int, optional

        Minimum number of observations in the window required to have a value (default is

        ↪  60).
```

```
    Returns
    -------
    Pandas Series
        Series containing the rolling standard deviation of the residuals.
    """

    residuals_std = []  # Initialize a list to store the rolling standard deviations of
    ↪ the residuals

    # Loop over each row in the group DataFrame
    for i in range(len(group)):
        # If the number of periods is less than the minimum required, append NaN to the
        ↪ results
        if i + 1 < min_periods:
            residuals_std.append(np.nan)
        else:
            # Select the subset of data within the rolling window
            subset = group.iloc[max(0, i+1-window):i+1]

            # If there are any missing values in the required columns, append NaN to the
            ↪ results
            if subset['Excess Market return'].isnull().sum() > 0 or
            ↪ subset['daily_return'].isnull().sum() > 0:
                residuals_std.append(np.nan)
                continue

            # Set up the regression model with a constant and the excess market returns
            X = sm.add_constant(subset['Excess Market return'])
            y = subset['daily_return']

            try:
```

```
                # Fit the OLS regression model
                model = sm.OLS(y, X).fit()
                # Get the residuals from the model
                residuals = model.resid
                # Calculate the standard deviation of the residuals and append to the
                ↪ results
                residuals_std.append(np.std(residuals))
            except:
                # If there's an error in the regression process, append NaN to the results
                residuals_std.append(np.nan)

    # Return the results as a Pandas Series with the same index as the input group
    ↪ DataFrame
    return pd.Series(residuals_std, index=group.index)

# Apply the rolling calculation for each group of 'PERMNO'
crspd['std_residuals'] = crspd.groupby('PERMNO').apply(
    lambda x: rolling_std_residuals(x, window=63, min_periods=60)
).reset_index(level=0, drop=True)

# Compute the monthly Market Equity

# Grouping by stock, year, and month, and selecting the last trading day of each month for
↪ each stock
last_trading_days = crspd.groupby(['PERMNO', crspd['date'].dt.year,
↪ crspd['date'].dt.month]).apply(lambda x: x.iloc[-1])
last_trading_days = last_trading_days.rename_axis(['PERMNO2', 'year',
↪ 'month']).reset_index()

last_trading_days.drop(columns=['level_0','index', 'PERMNO2', 'year', 'month', 'vwretx'],
↪ inplace=True)
```

```
# Save data of the last trading day of each month
last_trading_days.to_sql(name='crspd_last_trading_days',
          con=crsp_compustat,
          if_exists="replace")

# Load monthly data
crspm = pd.read_sql_query(
    sql="SELECT * FROM crsp_m",
    con=crsp_compustat
)
crspm.rename(columns={'permno': 'PERMNO'}, inplace=True)
crspm = crspm.astype({'PERMNO':int})
crspm['date'] = pd.to_datetime(crspm['date'])

# Merge the data with the idiosyncratic volatility data
crspm = pd.merge(crspm,last_trading_days, how='inner', on=['PERMNO','date'])

def load_transaction_costs():
    parts = []
    for i in range(1, 51):  # Assuming 50 parts
        #part_df = pd.read_sql_query(f"SELECT * FROM t_costs_filled_part_{i}", conn,
        ↪  index_col='date')
        part_df = pd.read_sql_query(f"SELECT * FROM t_costs_filled_part_{i}",
        ↪  crsp_compustat,index_col='index')
        parts.append(part_df)
    return pd.concat(parts, axis=1)

# Load transaction costs
transaction_costs = load_transaction_costs()

# Modify the transaction costs dataframe to have the date as indices and PERMNOs as
↪  columns
```

```
df_reset = transaction_costs.reset_index()

df_long = df_reset.melt(id_vars='index', var_name='PERMNO', value_name='t_cost')

df_long.rename(columns={'index': 'date'}, inplace=True)


t_cost = df_long

t_costs = t_cost

t_costs = t_costs.astype({'PERMNO':int})

t_costs.date = pd.to_datetime(t_costs.date)


# Ensure that price data is positive

crspm['prc'] = crspm['prc'].abs()


# Compute the market equity at the PERMNO level

crspm['ME'] = crspm['prc'] * crspm['shrout']


crsp_combined = crspm


# Delete unused dataframes

del crspd, crspm,df_long


# Delete unused columns

crsp_combined.drop(columns=['shrcd','exchcd', 'ret', 'retx', 'shrout','prc',

↪  'siccd','level_0', 'index','CLOSE','vwretx',

                            'Market RET','RF', 'Excess Market return', 'daily_return'],

                            ↪  inplace=True)



# Rank ME and IVOL

crsp_combined['rank_ME'] = crsp_combined.groupby('date')['ME'].rank()

crsp_combined['rank_IVOL'] = crsp_combined.groupby('date')['std_residuals'].rank()


# Process each date individually
```

```
unique_dates = t_costs['date'].unique()


# Initialize the result DataFrame

result_df = pd.DataFrame()


crsp_combined['date'] = crsp_combined['date'] + MonthEnd(0)


# Find the closest match of each stock with missing TCs for each date

for date in unique_dates:

    # Filter data for the specific date

    t_costs_date = t_costs[t_costs['date'] == date].copy()

    crsp_combined_date = crsp_combined[crsp_combined['date'] == date].copy()


    # Merge data for the specific date

    merged = pd.merge(t_costs_date, crsp_combined_date, on=['PERMNO', 'date'], how='left')



    # Drop unused columns

    merged.drop(columns=[ 'ME', 'std_residuals', 'spread'], inplace=True)


    # Calculate Euclidean distance for each missing spread

    def calculate_closest_match(group):

        """

        Fill missing transaction cost ('t_cost') values by finding the closest match based

        ↪  on

        'rank_ME' and 'rank_IVOL' values within the group.


        Parameters

        ----------

        group : Pandas DataFrame

            DataFrame containing columns 'rank_ME', 'rank_IVOL', and 't_cost'.
```

```
    Returns
    -------
    group : Pandas DataFrame
        DataFrame with missing 't_cost' values filled based on the closest match.
    """

    # Identify rows with missing and present 't_cost' values
    missing_spread = group['t_cost'].isnull()
    present_spread = group['t_cost'].notnull()

    # If there are no missing 't_cost' values, return the group as is
    if missing_spread.sum() == 0:
        return group

    # Extract the rows with present 't_cost' values and relevant columns
    present_data = group.loc[present_spread, ['rank_ME', 'rank_IVOL', 't_cost']]

    # Loop over each row with missing 't_cost'
    for idx in group[missing_spread].index:
        row = group.loc[idx]

        # Check if both 'rank_ME' and 'rank_IVOL' are not null
        if pd.notnull(row['rank_ME']) and pd.notnull(row['rank_IVOL']):
            # Calculate the Euclidean distance based on both 'rank_ME' and 'rank_IVOL'
            distances = np.sqrt((present_data['rank_ME'] - row['rank_ME']) ** 2 +
                                (present_data['rank_IVOL'] - row['rank_IVOL']) ** 2)
        elif pd.notnull(row['rank_ME']):
            # Calculate the absolute distance based on 'rank_ME' only
            distances = np.abs(present_data['rank_ME'] - row['rank_ME'])
        else:
            # If 'rank_ME' is null, use the mean 't_cost' from the present data
            group.loc[idx, 't_cost'] = present_data['t_cost'].mean()
```

```
                continue

            # Find the index of the closest match
            closest_index = distances.idxmin()

            # Fill the missing 't_cost' value with the closest match's 't_cost'
            group.loc[idx, 't_cost'] = present_data.loc[closest_index, 't_cost']

        # Return the group with filled 't_cost' values
        return group


    # Apply the calculation to the merged data
    filled_t_costs_date = calculate_closest_match(merged)

    # Append the result to the final result DataFrame
    result_df = pd.concat([result_df, filled_t_costs_date])

# Save the intermediate results
result_df.to_sql(name='result_df',
          con=crsp_compustat,
          if_exists="replace",
          index=True)

# Update t_costs with the filled transaction costs
# Merge the result back to the original t_costs DataFrame
t_costs = t_costs.merge(result_df, on=['PERMNO', 'date'], how='left', suffixes=('',
↪  '_filled'))

# Use the filled t_cost where original t_cost is NaN
t_costs['t_cost'] = t_costs['t_cost'].combine_first(t_costs['t_cost_filled'])
```

```
# Drop the temporary filled column
t_costs.drop(columns=['t_cost_filled'], inplace=True)

# Clean up placeholder columns
t_costs.drop(['rank_ME', 'rank_IVOL'], axis=1, inplace=True)


# Calculate the cross-sectional average of 't_cost' by date
average_t_costs = t_costs.groupby('date')['t_cost'].transform('mean')

# Fill missing values in 't_cost' with the cross-sectional average by date
t_costs['t_cost'].fillna(average_t_costs, inplace=True)

transaction_costs = t_costs.pivot(index='date', columns='PERMNO', values='t_cost')

# Save to SQL
n_splits = 50
columns_split = np.array_split(transaction_costs.columns, n_splits)

# Save each part to the SQL database
for i, cols in enumerate(columns_split, 1):
    part_df = transaction_costs[cols]
    part_df.to_sql(name=f't_costs_filled_part_{i}',
                con=crsp_compustat,
                if_exists="replace",
                index=True)
```

Portfolio Optimization

```
import sqlite3
import numpy as np
import pandas as pd
```

```
from pandas.tseries.offsets import MonthEnd
from scipy.optimize import minimize
import matplotlib.pyplot as plt
from arch.bootstrap import StationaryBootstrap
from numpy.random import RandomState

######################################################################
########################### LOAD DATA ###############################
######################################################################

def load_data():
    conn = sqlite3.connect("/Users/Ilyas/Documents/Mémoire/crsp_compustat.sqlite")

    # Load factor returns and market volatility
    factor_returns = pd.read_sql_query("SELECT * FROM factors", conn, index_col='index')
    market_volatility = pd.read_sql_query("SELECT * FROM market_vol", conn,
    ↪  index_col='date')

    # Function to load and concatenate split weights data
    def load_weights(name):
        parts = []
        for i in range(1, 51):
            part_df = pd.read_sql_query(f"SELECT * FROM weights_{name}_part_{i}", conn,
            ↪  index_col='date')
            parts.append(part_df)
        return pd.concat(parts, axis=1)

    # Load all factor weights
    #factor_weights_names = ['Market', 'SMB', 'HML', 'RMW', 'CMA','MOM', 'IA', 'ROE',
    ↪  'BAB']
    factor_weights_names = ['Market', 'SMB', 'HML', 'RMW', 'CMA','MOM', 'BAB'] # Exclude
    ↪  I/A and ROE
```

```
factor_weights = [load_weights(name) for name in factor_weights_names]


# Function to load and concatenate split transaction costs data
def load_transaction_costs():
    parts = []
    for i in range(1, 51):  # Assuming 50 parts
        part_df = pd.read_sql_query(f"SELECT * FROM t_costs_filled_part_{i}", conn,
        ↪  index_col='date')


        parts.append(part_df)
    return pd.concat(parts, axis=1)


# Load transaction costs
transaction_costs = load_transaction_costs()


# Function to load and concatenate split stock returns data
def load_stock_returns():
    parts = []
    for i in range(1, 51):  # Assuming 50 parts
        part_df = pd.read_sql_query(f"SELECT * FROM stock_ret_part_{i}", conn,
        ↪  index_col='date', parse_dates='date')
        parts.append(part_df)
    return pd.concat(parts, axis=1)


# Load transaction costs
stock_returns = load_stock_returns()


# Load the list of PERMNOs
PERMNOs = pd.read_sql_query("SELECT * FROM PERMNOs", conn)
PERMNOs = PERMNOs.astype(int)
PERMNOs = PERMNOs.iloc[:,0].to_list()
```

```
    conn.close()

    return factor_returns, market_volatility, transaction_costs, factor_weights,
    ↪  stock_returns, PERMNOs

# Load data
factor_returns, market_volatility, transaction_costs, factor_weights, stock_returns,
↪  PERMNOs = load_data()

# Exclude I/A and ROE factors
factor_returns.drop(columns=['I/A', 'ROE'], inplace=True)



# Access the database
crsp_compustat =
↪  sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/crsp_compustat.sqlite")

# Import the CRSP monthly data
stock_prices =  pd.read_sql_query(
                            sql="SELECT permno, date, prc from crsp_m",
                            con=crsp_compustat
                            )

stock_prices.drop_duplicates(subset=['permno','date'],inplace=True)
stock_prices = stock_prices.astype({'permno': int, 'prc':float})

stock_prices = stock_prices.pivot_table(index='date', columns='permno', values='prc')
stock_prices = stock_prices.abs()
stock_prices.dropna(how='all', axis=1)
stock_prices.index = pd.to_datetime(stock_prices.index) + MonthEnd(0)

################################################################
```

```
########################## PREPARE DATA ##########################
##################################################################

# Reindex the data so that they all share the same row index

# Get the list of unique dates
unique_dates = pd.date_range(start='1969-11-01', end='2022-12-31', freq="ME")

# Redfinition to get the comparison with the other series
unique_dates = pd.date_range(start='1992-04-01', end='2022-12-31', freq="ME")

def reindex_weights(df, row, column=None, fill=0):
    # Reindex the rows on the list of all dates
    df.index = pd.to_datetime(df.index)
    if fill == 0:
        reindexed_df = df.reindex(index=row)
    else:
        reindexed_df = df.reindex(index=row, method='ffill', limit=fill)
    reindexed_df.rename_axis('date', inplace=True)

    # Reindex the columns on the list of all PERMNOs
    if column is not None:
        reindexed_df.columns = reindexed_df.columns.astype(int)
        reindexed_df = reindexed_df.reindex(columns=column)

    return reindexed_df

# Reindex the dataframes so that they all share the same row and column indices
for i in range(len(factor_weights)):
    factor_weights[i] = reindex_weights(factor_weights[i], row=unique_dates,
    ↪  column=PERMNOs, fill=0)
```

```
transaction_costs = reindex_weights(transaction_costs, column=PERMNOs, row=unique_dates,
↪  fill=0)
stock_returns = reindex_weights(stock_returns, column=PERMNOs, row=unique_dates, fill=0)
factor_returns = reindex_weights(factor_returns, row=unique_dates, fill=0)
market_volatility = reindex_weights(market_volatility, row=unique_dates, fill=0)
stock_prices = reindex_weights(stock_prices, column=PERMNOs, row=unique_dates, fill=0)

# Fill with 0 for the matrix multiplications
factor_returns.fillna(0.0, inplace=True)
stock_returns.fillna(0.0, inplace=True)
stock_prices.fillna(0.0, inplace=True)

# Transaction costs are half of the spread
transaction_costs = transaction_costs.div(2)

######################################################################
###################### PRECOMPUTE VARIABLES ######################
######################################################################

# Compute the volatility-managed returns
def precompute_extended_factor_returns(factor_returns, market_volatility):

    # Number of factors
    K = factor_returns.shape[1]

    #c = 1/(1/market_volatility).mean().values.item() #Constant to rescale the volatility
    ↪  managed portfolios

    # Get the volatility as a Pandas Series
    market_volatility = market_volatility.iloc[:,0].shift(1)
```

```
    # Initialise the lists that will hold the original factors' returns and
    ↪ volatility-managed factors' returns
    extended_factor_returns = []
    extended_factor_vol_returns = []

    # Loop over the factors returns series
    for k in range(K):
        extended_factor_returns.append(factor_returns.iloc[:,k].fillna(0))

        ↪ extended_factor_vol_returns.append(factor_returns.iloc[:,k].fillna(0).div(market_volatility,
        ↪ axis=0))

    # Combine the returns in a dataframe
    extended_factor_returns = pd.concat(extended_factor_returns, axis=1)
    extended_factor_vol_returns = pd.concat(extended_factor_vol_returns, axis=1)




    ↪ extended_factor_vol_returns.rename(columns={0:'MarketVol',1:'SMBVol',2:'HMLVol',3:'RMWVol',4:'CMAVol
    ↪ 6:'BABVol'}, inplace=True)

    # Concatenate the original and volatility managed factors so that they have the same
    ↪ order
    extended_factor_returns = extended_factor_returns.merge(extended_factor_vol_returns,
    ↪ how='left', left_index=True, right_index=True)

    return extended_factor_returns

# Compute r_ext, shape: (T, 2 x K)
extended_factor_returns = precompute_extended_factor_returns(factor_returns,
↪ market_volatility)
```

```python
# Get the sample mean of each factor for the in-sample analysis, shape: (T, 2 x K)
mu_ext = extended_factor_returns.expanding(min_periods=1).mean()



# Desired order of the factors, to facilitate further analysis
desired_order = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA', 'MOM', 'BAB',
                 'MarketVol', 'SMBVol', 'HMLVol', 'RMWVol', 'CMAVol', 'MOMVol','BABVol']


# Function to compute expanding window covariance
def expanding_window_covariance(df, desired_order):

    # Initialise the list that will hold the covariance matrixes
    cov_list = []

    # Compute the r_ext covariance for the in-sample analysis
    for end in range(1, len(df) + 1):
        # Data up to the end of the in-sample window
        window_df = df.iloc[:end]

        if end > 1:  # Ensure there are at least 2 data points
            cov_matrix = window_df.cov()
            cov_matrix['Date'] = window_df.index[-1]
            cov_list.append(cov_matrix)

    # Concatenate all covariance matrices
    all_cov = pd.concat(cov_list)

    # Create a multi-level index
    all_cov = all_cov.set_index(['Date'], append=True)
    all_cov = all_cov.swaplevel(0, 1)
    all_cov = all_cov.sort_index(level=0)
```

```
    # Reorder rows and columns
    all_cov = all_cov.loc[(slice(None), desired_order), desired_order]

    return all_cov

# Get the sample covariance of each factor for the in-sample analysis, shape: (T, 1), and
↪  each cell contains a matrix of shape (2 x K, 2 x K)
sigma_ext = expanding_window_covariance(extended_factor_returns, desired_order)

# Function to compute the volatility-managed weights matrices
def precompute_weights(factor_weights, market_volatility):
    # Number of factors
    K = len(factor_weights)

    # Get the volatility as a Pandas Series
    market_volatility = market_volatility.iloc[:,0]

    # Initialise the lists that will contain the weights for the original and volatility
    ↪  managed factors
    factor_weights_list = []
    factor_weights_vol = []

    # Loop over each factor
    for k in range(K):
        # Store the weights, and set the weights of other stocks to 0 for the matrix
        ↪  multiplications
        factor_weights_list.append(factor_weights[k].fillna(0.0))
        factor_weights_vol.append(factor_weights[k].fillna(0.0).div(market_volatility,
        ↪  axis=0))

    # Add the factor weights of volatility-managed factors, in the same order as the
    ↪  original factors
```

```
    factor_weights_list.extend(factor_weights_vol)

    return factor_weights_list

# Compute the volatility-managed weights, shape: (2 x K), and each element inside has a
↪  shape (T, N)
factor_weights = precompute_weights(factor_weights, market_volatility)



# Function to compute the extended weight matrix
def create_weights_per_dates(factor_weights):

    # Check if all dataframes have the same indices
    if not all(df.index.equals(factor_weights[0].index) and
    ↪  df.columns.equals(factor_weights[0].columns) for df in factor_weights):
        raise ValueError("All dataframes in factor_weights must have the same indices and
        ↪  columns")

    # Extract common row indices (dates) and column indices (PERMNOs)
    dates = factor_weights[0].index
    permnos = factor_weights[0].columns

    # Initialize a dictionary to store numpy arrays by date
    weights_per_dates_dict = {}

    # Iterate over each date
    for date in dates:
        # Create a numpy array for the current date of shape (N, 2 x K)
        weights_array = np.zeros((len(permnos), len(factor_weights)))

        # Populate the numpy array with values from factor_weights
        for i, df in enumerate(factor_weights):
```

```
            weights_array[:, i] = df.loc[date].values

        # Add the numpy array to the dictionary with the date as the key
        weights_per_dates_dict[date] = weights_array

    # Convert the dictionary to a pandas Series
    weights_per_dates = pd.Series(weights_per_dates_dict, name='weights_per_dates')

    return weights_per_dates

# Compute X_ext for each date, shape: (T, 1), and each cell has a shape (N, 2 x K)
weights_per_dates = create_weights_per_dates(factor_weights)

######################################################################
###################### WEIGHTS OPTIMIZATION ######################
######################################################################


# Objective function to minimize
def objective_function(eta, mu_ext, sigma_ext, Lambda, compounded_weights, X_ext, t, TC,
↪   gamma):
    """


    Parameters
    ----------
    eta : Numpy array, shape: (2 x K, )
        Factor portfolio weights.
    mu_ext : Numpy array, shape: (1, 2 x K)
        Sample mean of factor returns up to date t.
    sigma_ext : Numpy array, shape: (2 x K, 2 x K)
        Sample covariance of factor returns up to date t.
```

```
    Lambda : Numpy array, shape: (N, 1)
        Individual transaction cost at date t.
    compounded_weights : Numpy array, shape: (N, 1)
        Position before the rebalancing.
    X_ext : Numpy array, shape: (N, 2 x K)
        Stock weights associated to each factor.
    t : int, shape: 1
        Number of trading periods.
    TC : float, shape: 1
        Cumulative transaction costs.
    gamma : float, shape: 1
        Risk aversion parameter.

    Returns
    -------
    float
        Opposite of an investor mean-variance utility.

    """

    # Reshape eta for matrix multiplications and make it a contiguous array
    eta = eta.reshape(-1, 1)

    # Sample portfolio mean, shape: 1
    mu_eta = (mu_ext @ eta).item() # Shapes: (1, 2 x K) @ (2 x K, 1)

    # Sample portfolio variance, shape: 1
    eta_sigma_eta = np.transpose(eta) @ sigma_ext @ eta # Shapes: (1, 2 x K) @ (2 x K, 2 x
    ↪  K) @ (2 x K, 1)

    if t == 1:
        TC_avg = 0
```

```
        #print(f"mu_eta: {mu_eta}, eta_sigma_eta: {eta_sigma_eta}")
    else:
        # Compute the change in weights (deltaw), shape: (N, 1)
        change_weights = np.abs(X_ext @ eta - compounded_weights) # Shapes: (N, 2 x K) @
        ↪  (2 x K, 1) - (N, 1)

        # Compute the transaction costs associated with the rebalancing, shape: 1
        TC_eta = np.sum(change_weights * Lambda).item() # Shapes: (N, 1) @ (N, 1)

        # Add the previous transaction costs, shape: 1
        TC_avg = (TC_eta + TC) / (t - 1)

        #print(f"mu_eta: {mu_eta}, eta_sigma_eta: {eta_sigma_eta}, TC_eta: {TC_eta},
        ↪  TC_avg: {TC_avg}")

    return -(mu_eta - TC_avg - 0.5 * gamma * eta_sigma_eta).item()

# Function to find the weights that minimize the opposite of the utility function
def optimize_eta(mu_ext,sigma_ext, Lambda, compounded_weights, X_ext, gamma, TC, t,
↪  wgt_eta):
    """

    Parameters
    ----------
    mu_ext : Numpy array, shape: (1, 2 x K)
        Sample mean of factor returns up to date t.
    sigma_ext : Numpy array, shape: (2 x K, 2 x K)
        Sample covariance of factor returns up to date t.
    Lambda : Numpy array, shape: (N, 1)
        Individual transaction cost at date t.
    compounded_weights : Numpy array, shape: (N, 1)
        Position before the rebalancing.
```

```
    X_ext : Numpy array, shape: (N, 2 x K)
        Stock weights associated to each factor.
    gamma : float, shape: 1
        Risk aversion parameter.
    TC : float, shape: 1
        Cumulative transaction costs.
    t : int, shape: 1
        Number of trading periods.
    wgt_eta : Numpy array, shape: (1, 2 x K)
        Previous weights.

    Returns
    -------
    Results of the minimization of the objective function.

    """

    # # Initial guess (previous month's weights), shape: (2 x K, 1)
    initial_eta = wgt_eta

    # Define bounds for each variable to be non-negative
    bounds = tuple((0.001, 999) for _ in range(K))
    #bounds = tuple((0.00001, None) for _ in range(len(wgt_eta))]
    #lb = np.append(1, np.repeat(0, 13))
    #ub = np.append(1, np.repeat(9999, 13))

    #constraints = ({'type': 'eq', 'fun': lambda eta: np.sum(eta) - 1})
    # Minimization
    result = minimize(
        objective_function,
        initial_eta,
        args=(mu_ext, sigma_ext, Lambda, compounded_weights, X_ext, t, TC, gamma),
```

```
        method='SLSQP',
        bounds=bounds,
        options={'ftol': 0.000000000001}
    )

    return result

gamma = 5  # Risk-aversion parameter

# Initialisation of the dataframe that will contain the weights of the factors
optimal_eta_df = pd.DataFrame(index=unique_dates, columns=desired_order)

# Initialisation of the dataframe that will contain the portfolio transaction costs
portfolio = pd.DataFrame(index=unique_dates, columns=['Portfolio Value','Transaction
↪  Costs'])

# Number of factors
K = len(factor_weights)

# First guess: equal-weighted, shape: (1, 2 x K)
#wgt_eta = np.array([1.3277229465404619, 0.837389381165301, 1.0460160658429223e-16, 0.0,
↪  4.569037125016756, 0.3472973323564199, 5.4119064118295655e-18        ,
↪  1.9278842141570563e-05,         0.0032872695669896755, 0.017182812730402397,
↪  0.04950084105413666, 1.9972369364017215e-18,
↪  0.005329323296599978,0.0045307431142720634])
wgt_eta = np.ones(K)

# Initialise the cumulative transaction costs
TC = 0

# Initial position, shape: (N, 1)
weights = weights_per_dates.loc[unique_dates[121]] @ wgt_eta.reshape(-1, 1)
```

```
weights = weights_per_dates.loc[unique_dates[61]] @ wgt_eta.reshape(-1, 1)

# Initialise the number of months of trading, to compute the average transaction costs
t = 1

# Loop over each month of the sample, with an initial window of 10 years
#for month in unique_dates[121:]:
for month in unique_dates[61:]:
    # Keep track of the month processed
    print(f"{month} is being processed.")

    # Compute the updated weights (w^+ = X_ext_(t-1) • (1 + r_t)), shape: (N, 1)
    compounded_weights = weights.reshape(-1, 1) *
    ↪  (stock_returns.loc[month].add(1)).values.reshape(-1, 1) # Shapes: (N, 1) • (N, 1)
    #compounded_weights = weights * (1 + stock_returns.loc[month].values.reshape(-1, 1))

    # Compute the optimal weights through the minimisation
    optimal_eta = optimize_eta(mu_ext.loc[month].values.reshape(1, -1),
    ↪  sigma_ext.loc[month].values, transaction_costs.loc[month].values.reshape(-1, 1),
    ↪  compounded_weights, weights_per_dates.loc[month], gamma, TC, t, wgt_eta)

    # Keep the weights as an initial guess for the next optimisation, shape: (2 x K, )
    wgt_eta = optimal_eta.x

    # Store the weights
    optimal_eta_df.loc[month] = wgt_eta


    # Update the cumulative transaction costs
    change_weights = (weights_per_dates.loc[month] @ wgt_eta.reshape(-1, 1)) -
    ↪  compounded_weights # Shape: (N, 2 x K) @ (2 x K, 1) - (N, 1)
```

```
    TC_eta = np.sum(np.abs(change_weights) *
    ↪  transaction_costs.loc[month].values.reshape(-1, 1)) # Shape: sum(|(N, 1) • (N,
    ↪  1)|) = (1, 1)

    # Do not consider the transaction costs to get the first position
    if t == 1:
        TC = 0
    else: # Add the transaction costs associated with the rebalancing to the cumulative
    ↪  transaction costs
        TC += TC_eta.item()

    # Store the transaction costs
    portfolio.loc[month,'Transaction Costs'] = TC_eta.item()

    # Update the out-of-sample portfolio values
    portfolio_value = np.sum((weights_per_dates.loc[month] @ wgt_eta.reshape(-1, 1)) *
    ↪  stock_prices.loc[month].values.reshape(-1,1))
    print(f'Portfolio Value: {portfolio_value}')
    portfolio.loc[month,'Portfolio Value'] = portfolio_value

    # Compute X_ext @ eta, shape: (N, 1)
    weights = weights_per_dates.loc[month] @ wgt_eta.reshape(-1, 1) # Shapes: (N, 2 x K) @
    ↪  (2 x K, 1)

    # Update the month counter
    t += 1

######################################################################
###################### DISPLAY THE RESULTS ######################
######################################################################

# Compute the portfolio returns
```

```
ret = optimal_eta_df.shift(1).dropna(axis=0).mul(extended_factor_returns)

# Get the returns net of transaction costs
port_ret = ret.dropna().sum(axis=1).sub(portfolio["Transaction Costs"])

cmv_ret = port_ret

cumret = port_ret.dropna(axis=0).add(1).cumprod().div(10000)

######################################################################
####################### COMPARISON MODELS ############################
######################################################################

def SR_pval(model1ret, model2ret, n):
    rs = RandomState(1234)
    bs = StationaryBootstrap(5, model1ret, model2ret, random_state=rs)

    # Array to store the differences in Sharpe ratios
    diff_distr = np.zeros(n)

    for data in bs.bootstrap(n):
        bs_model1ret = data[0][0]
        bs_model2ret = data[0][1]

        # Computing the Sharpe ratio of each series
        sr1 = bs_model1ret.mean() / bs_model1ret.std()
        sr2 = bs_model2ret.mean() / bs_model2ret.std()

        # Store the difference in Sharpe ratios
        diff_distr[i] = sr1 -  sr2

    # Calculate the p-value
```

```
    negative_count = (diff_distr < 0).sum()
    p_value = negative_count / n

    return p_value, diff_distr

p_val, diff_distr = SR_pval(port_ret.dropna(), cmv_ret.dropna().iloc[:-1], 10000)
print(f'The p-value is {p_val}')
```

LSTM Model

```
import pandas as pd
import numpy as np
import sqlite3
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Input
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

######################################################################
############################ LOAD DATA ###############################
######################################################################

# Load the VIX data
vix = pd.read_csv('/Users/Ilyas/Documents/Mémoire/VIX.csv')
vix.rename(columns={'Date':'date'}, inplace=True)
vix.set_index('date', inplace=True)

vix.dropna(inplace=True)
vix.index = pd.to_datetime(vix.index)

# Load the market data
```

```
market_index = pd.read_csv('/Users/Ilyas/Documents/Mémoire/Market Index.csv')
market_index.rename(columns={'caldt': 'date', 'vwretd': 'Market RET'}, inplace=True)
market_index['date'] = pd.to_datetime(market_index['date'])
market_index.set_index('date', inplace=True)

market_index['volatility'] = market_index['Market RET'].rolling(22).std()
market_index = market_index.merge(vix, how='left', left_index=True, right_index=True)
market_index.loc[:, 'vix'] = market_index.ffill()
market_index.dropna(subset=['Market RET', 'volatility', 'vix'], inplace=True)

# Extract end-of-month volatility
end_of_month_vol = market_index['volatility'].resample('ME').last()

######################################################################
######################### PREPARE DATA ###########################
######################################################################

# Function to create lagged features
def create_lagged_features(df, lag=5):
    for i in range(1, lag+1):
        df[f'Market RET_lag_{i}'] = df['Market RET'].shift(i)
        df[f'vix_lag_{i}'] = df['vix'].shift(i)
        df[f'volatility_lag_{i}'] = df['volatility'].shift(i)
    return df

# Create lagged features
market_index = create_lagged_features(market_index, lag=5)
market_index.dropna(inplace=True)
market_index.drop_duplicates(subset=['Market RET', 'volatility', 'vix'],inplace=True)

# Function to have the last trading have the index of the last day of the month
def adjust_last_day_to_eom(df):
```

```
    # Group by year and month to find the last data point for each month
    df['month'] = df.index.to_period('M')
    last_per_month = df.groupby('month').tail(1)

    # Adjust the index to the end of the month if needed
    adjusted_idx = []
    for original_idx in last_per_month.index:
        eom_idx = original_idx.to_period('M').to_timestamp('M')
        if original_idx != eom_idx:
            adjusted_idx.append((original_idx, eom_idx))

    # Apply the adjustments
    for original_idx, eom_idx in adjusted_idx:
        df.loc[eom_idx] = df.loc[original_idx]
        df = df.drop(original_idx)

    df = df.sort_index()  # Ensure the DataFrame is sorted by index
    return df.drop(columns='month')


# Apply the function to modify the index of the last trading data
market_index = adjust_last_day_to_eom(market_index)

# Prepare the dataset
features = ['Market RET', 'vix', 'volatility'] + [f'Market RET_lag_{i}' for i in range(1,
↪  6)] + [f'vix_lag_{i}' for i in range(1, 6)] + [f'volatility_lag_{i}' for i in range(1,
↪  6)]
dataset = market_index[features].values

def create_sequences_with_eom_target(data, vol_end_of_month, time_steps=22):
    sequences = []
    labels = []
```

```
    eom_dates = vol_end_of_month.index

    for i in range(len(data) - time_steps):
        current_date = market_index.index[i + time_steps - 1]
        eom_current_month = current_date - pd.offsets.MonthEnd(0)
        eom_next_month = eom_current_month + pd.DateOffset(months=1)
        eom_next_month = eom_next_month - pd.offsets.MonthEnd(1)

        if eom_next_month in eom_dates:
            sequences.append(data[i:(i + time_steps)])
            eom_idx = np.where(eom_dates == eom_next_month)[0][0]
            labels.append(vol_end_of_month.iloc[eom_idx])

    print(f"Total sequences created: {len(sequences)}")
    return np.array(sequences), np.array(labels)

time_steps = 22
X, y = create_sequences_with_eom_target(dataset, end_of_month_vol, time_steps)
print(f"Shape of X: {X.shape}, Shape of y: {y.shape}")

######################################################################
######################### MODEL DEFINITION ###########################
######################################################################

# Model definition with additional LSTM layers and dropout
def create_model(input_shape):
    model = Sequential()
    model.add(Input(shape=input_shape))
    #model.add(LSTM(30, return_sequences=True))
    #model.add(Dropout(0.20))
    model.add(LSTM(50))
    model.add(Dropout(0.20))
```

```
    model.add(Dense(1))
    model.compile(optimizer='adamax', loss='mean_squared_error')
    return model

######################################################################
#################### OUT-OF-SAMPLE PREDICTION ####################
######################################################################

# Expanding window training and prediction
start_date = '1992-01-31'

# Lists to store the predictions and the actual values
predictions = []
actuals = []

input_shape = (X.shape[1], X.shape[2])
model = create_model(input_shape)

eom_dates = end_of_month_vol.index
eom_start_index = eom_dates.get_loc(start_date)

# Loop over each date to get the predictions
for eom_date in eom_dates[eom_start_index:]:

    # Get the index of the data
    loc = market_index.index.get_loc(eom_date)
    if isinstance(loc, slice):
        current_index = loc.start
    else:
        current_index = loc

    if current_index + time_steps > len(X):
```

```
    break

# Keep track of the month being processed
print(f"Processing month: {eom_date.strftime('%Y-%m')}")

# Get the training and test data
X_train, X_test = X[:current_index], X[current_index:current_index + 1]
y_train, y_test = y[:current_index], y[current_index:current_index + 1]

# Scale the features to be in the range [0,1]
scaler = MinMaxScaler(feature_range=(0, 1))
X_train_scaled = scaler.fit_transform(X_train.reshape(-1,
↪  X_train.shape[-1])).reshape(X_train.shape)
X_test_scaled = scaler.transform(X_test.reshape(-1,
↪  X_test.shape[-1])).reshape(X_test.shape)

if X_train.shape[0] == 0 or X_test.shape[0] == 0 or y_train.shape[0] == 0 or
↪  y_test.shape[0] == 0:
    continue

early_stopping = EarlyStopping(monitor='loss', patience=5)

# Fit the model
model.fit(X_train_scaled, y_train, epochs=30, batch_size=32, verbose=0,
↪  callbacks=[early_stopping])

# Get the prediction
pred = model.predict(X_test_scaled, verbose=0)

# Store the results in the list
predictions.append(pred[0][0])
actuals.append(y_test[0])
```

```
if predictions:
    predictions = np.array(predictions).reshape(-1, 1)
    actuals = np.array(actuals).reshape(-1, 1)

# Define the root mean squared error (RMSE)
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_pred - y_true)**2))
print('RMSE', rmse(actuals,predictions))

# Save predictions
predictions = pd.DataFrame(predictions)
pred_vol = sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/pred_vol.sqlite")
predictions.to_sql("MarketVol1Cell502", pred_vol)

######################################################################
######################### PLOT RESULTS ###########################
######################################################################

# Store the predictions and the actual values in dataframes with the same index
unique_dates = pd.to_datetime(eom_dates)

pred = pd.DataFrame(predictions.iloc[:,0].values, index= unique_dates[27:])
act = pd.DataFrame(actuals, index= unique_dates[27:])
plt.figure(figsize=(10, 6))
plt.plot(pred.index, pred, label='Predicted Volatility', c='crimson')
plt.plot(act.index, act, label='Actual Volatility', c='dodgerblue')
plt.grid(color='lightgrey', linestyle='-', linewidth=0.5, zorder=0)
plt.ylabel("Volatility")

# Set the x-axis to start in 1992
plt.xlim(unique_dates[27], unique_dates[-1])
```

```
plt.ylim([0,0.06])


# Formatting the x-axis to show dates nicely
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
plt.gcf().autofmt_xdate()  # Rotate date labels


# Display the legend
plt.legend()


# Save the figure
plt.savefig("MarketVol1Cell502.png", bbox_inches="tight")
plt.show()
```

ν-SVR Model

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.svm import NuSVR
import sqlite3
import matplotlib.pyplot as plt
import matplotlib.dates as mdates


######################################################################
############################ LOAD DATA ###############################
######################################################################


# Load the VIX data
vix = pd.read_csv('/Users/Ilyas/Documents/Mémoire/VIX.csv')
vix.rename(columns={'Date':'date'}, inplace=True)
vix.set_index('date', inplace=True)
```

```
vix.dropna(inplace=True)
vix.index = pd.to_datetime(vix.index)

# Load the market data
market_index = pd.read_csv('/Users/Ilyas/Documents/Mémoire/Market Index.csv')
market_index.rename(columns={'caldt': 'date', 'vwretd': 'Market RET'}, inplace=True)
market_index['date'] = pd.to_datetime(market_index['date'])
market_index.set_index('date', inplace=True)

market_index['volatility'] = market_index['Market RET'].rolling(22).std()
market_index = market_index.merge(vix, how='left', left_index=True, right_index=True)
market_index.loc[:, 'vix'] = market_index.ffill()
market_index.dropna(subset=['Market RET', 'volatility', 'vix'], inplace=True)

# Extract end-of-month volatility
end_of_month_vol = market_index['volatility'].resample('ME').last()

######################################################################
######################### PREPARE DATA ###########################
######################################################################

# Function to create lagged features
def create_lagged_features(df, lag=5):
    for i in range(1, lag+1):
        df[f'Market RET_lag_{i}'] = df['Market RET'].shift(i)
        df[f'vix_lag_{i}'] = df['vix'].shift(i)
        df[f'volatility_lag_{i}'] = df['volatility'].shift(i)
    return df

# Create lagged features
market_index = create_lagged_features(market_index, lag=5)
market_index.dropna(inplace=True)
```

```
market_index.drop_duplicates(subset=['Market RET', 'volatility', 'vix'],inplace=True)


# Function to have the last trading have the index of the last day of the month
def adjust_last_day_to_eom(df):
    # Group by year and month to find the last data point for each month
    df['month'] = df.index.to_period('M')
    last_per_month = df.groupby('month').tail(1)

    # Adjust the index to the end of the month if needed
    adjusted_idx = []
    for original_idx in last_per_month.index:
        eom_idx = original_idx.to_period('M').to_timestamp('M')
        if original_idx != eom_idx:
            adjusted_idx.append((original_idx, eom_idx))

    # Apply the adjustments
    for original_idx, eom_idx in adjusted_idx:
        df.loc[eom_idx] = df.loc[original_idx]
        df = df.drop(original_idx)

    df = df.sort_index()  # Ensure the DataFrame is sorted by index
    return df.drop(columns='month')



# Apply the function to modify the index of the last trading data
market_index = adjust_last_day_to_eom(market_index)

# Prepare the dataset
features = ['Market RET', 'vix', 'volatility'] + [f'Market RET_lag_{i}' for i in range(1,
↪  6)] + [f'vix_lag_{i}' for i in range(1, 6)] + [f'volatility_lag_{i}' for i in range(1,
↪  6)]
dataset = market_index[features].values
```

```
def create_sequences_with_eom_target(data, vol_end_of_month, time_steps=22):
    sequences = []
    labels = []
    eom_dates = vol_end_of_month.index

    for i in range(len(data) - time_steps):
        current_date = market_index.index[i + time_steps - 1]
        eom_current_month = current_date - pd.offsets.MonthEnd(0)
        eom_next_month = eom_current_month + pd.DateOffset(months=1)
        eom_next_month = eom_next_month - pd.offsets.MonthEnd(1)

        if eom_next_month in eom_dates:
            sequences.append(data[i:(i + time_steps)])
            eom_idx = np.where(eom_dates == eom_next_month)[0][0]
            labels.append(vol_end_of_month.iloc[eom_idx])

    print(f"Total sequences created: {len(sequences)}")
    return np.array(sequences), np.array(labels)

time_steps = 22
X, y = create_sequences_with_eom_target(dataset, end_of_month_vol, time_steps)
print(f"Shape of X: {X.shape}, Shape of y: {y.shape}")

# Flatten the sequences for SVR
X_flattened = X.reshape(X.shape[0], -1)

######################################################################
######################### MODEL DEFINITION ###########################
######################################################################

# Model definition with additional LSTM layers and dropout
```

```
svr = NuSVR(nu=0.5, C=2, kernel='rbf', gamma=0.001)

#####################################################################
#################### OUT-OF-SAMPLE PREDICTION ######################
#####################################################################

# Expanding window training and prediction
start_date = '1992-01-31'

# Lists to store the predictions and the actual values
predictions = []
actuals = []

input_shape = (X.shape[1], X.shape[2])

eom_dates = end_of_month_vol.index
eom_start_index = eom_dates.get_loc(start_date)

# Loop over each date to get the predictions
for eom_date in eom_dates[eom_start_index:]:

    # Get the index of the data
    loc = market_index.index.get_loc(eom_date)
    if isinstance(loc, slice):
        current_index = loc.start
    else:
        current_index = loc

    if current_index + time_steps > len(X):
        break

    # Keep track of the month being processed
```

```
print(f"Processing month: {eom_date.strftime('%Y-%m')}")


# Get the training and test data

X_train, X_test = X_flattened[:current_index], X_flattened[current_index:current_index
↪  + 1]

y_train, y_test = y[:current_index], y[current_index:current_index + 1]



scaler_X = StandardScaler()

scaler_y = StandardScaler()


X_train_scaled = scaler_X.fit_transform(X_train)

X_test_scaled = scaler_X.transform(X_test)

y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1)).ravel()

y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).ravel()


if X_train.shape[0] == 0 or X_test.shape[0] == 0 or y_train.shape[0] == 0 or
↪  y_test.shape[0] == 0:

    continue


# Fit the model

svr.fit(X_train_scaled, y_train_scaled)


# Get the prediction

pred = svr.predict(X_test_scaled, verbose=0)

pred_scaled = svr.predict(X_test_scaled)

pred = scaler_y.inverse_transform(pred_scaled.reshape(-1, 1))



# Store the results in the list

predictions.append(pred[0][0])

actuals.append(y_test[0])
```

```
if predictions:
    predictions = np.array(predictions).reshape(-1, 1)
    actuals = np.array(actuals).reshape(-1, 1)

# Define the root mean squared error (RMSE)
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_pred - y_true)**2))
print('RMSE', rmse(actuals,predictions))



# Save predictions
predictions = pd.DataFrame(predictions)
pred_vol = sqlite3.connect(database="/Users/Ilyas/Documents/Mémoire/pred_vol.sqlite")
predictions.to_sql("MarketVolNuSVR", pred_vol)

######################################################################
######################### PLOT RESULTS ###########################
######################################################################

# Store the predictions and the actual values in dataframes with the same index
unique_dates = pd.to_datetime(eom_dates)

pred = pd.DataFrame(predictions.iloc[:,0].values, index= unique_dates[27:])
act = pd.DataFrame(actuals, index= unique_dates[27:])
plt.figure(figsize=(10, 6))
plt.plot(pred.index, pred, label='Predicted Volatility', c='crimson')
plt.plot(act.index, act, label='Actual Volatility', c='dodgerblue')
plt.grid(color='lightgrey', linestyle='-', linewidth=0.5, zorder=0)
plt.ylabel("Volatility")

# Set the x-axis to start in 1992
```

```
plt.xlim(unique_dates[27], unique_dates[-1])

plt.ylim([0,0.06])


# Formatting the x-axis to show dates nicely

plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))

plt.gcf().autofmt_xdate()  # Rotate date labels


# Display the legend

plt.legend()


# Save the figure

plt.savefig("MarketVolNuSVR.png", bbox_inches="tight")

plt.show()
```