Erasmus University Rotterdam

Erasmus School of Economics

Bachelor Thesis

Econometrics and Operations Research

# Application of recurrent neural networks to momentum trading

*Author:*
Suet Yin Wong
406994

*Supervisor:*
S.H.L.C.G. Vermeulen
*Second assessor:*
X. Xiao

July 4, 2019

## Abstract

In recent years, deep learning techniques have been developed to handle complexity in the data which can not be achieved with traditional econometric techniques. However, the application of these techniques to real world problems have not been studied extensively. In this paper, we aim to investigate whether recurrent neural networks using a LSTM-autoencoder can contribute to financial time series forecasting and especially, trading strategies. We will combine different regression methods with a modified momentum strategy suggested by Kim (2019) and evaluate its performance on S&P 500 data. Our finding indicates that regression methods using LSTM-autoencoders lead to an improved profitability performance and predictive accuracy performance compared to regression methods based on shallow learning or non-recurrent deep learning algorithms. This implies that preserving sequential information is crucial in time series forecasting, for which recurrent neural networks are designed.

## Contents

## 1 Introduction

Historically speaking, momentum has been a strong strategy among investors. This asset pricing anomaly is also known as *relative strength* strategy and is based on the assumption that stocks that have an upward (downward) trend in the look-back period will continue to rise (fall). Portfolios based on this strategy have been proven to generate abnormal returns across different markets and assets, as reported in Fama and French (1996).

The momentum strategy can be implemented by constructing equally weighted decile portfolios based on the ranking of the stock returns of the look-back period, by means of buying the top decile and selling the bottom decile. This strategy is known as the winner-minus-loser (WML) strategy. Jegadeesh and Titman (1993, 2001) have shown that another variant of momentum strategy, the winner only (WO) strategy, performs well too. In this strategy, only stocks in the top decile are bought without going short on stocks in the bottom decile.

The contribution of this research to existing literature is that we implement the selective long/short momentum strategy proposed by Kim (2019) using time series predictions obtained with recurrent neural networks. In addition, we will compare our findings with the predictions that are made with shallow learning models as well as other deep learning models. The choice of using deep features is supported by the ability of artificial neural networks to handle even more complexity in the data (Sutskever and Hinton (2008)). Hence, our research question can be formulated as "Can financial time series predictions based on RNN models improve the performance of momentum strategies?"

The outline of this research paper is as follows. In Section 2, a brief literature review will be given about the existing momentum strategies. In Section 3, we present a description of our obtained data. Furthermore, in Section 4 we will explain the models that we have used for prediction in combination with our trading strategy. Then, in Section 6 we will discuss our findings that we have obtained from implementing different momentum strategies. To conclude this paper, we will give a brief summary and provide a clear answer on our research question in Section 7.

## 2 Literature

Momentum for investment purposes has a long history, but throughout the years there have been different approaches and improvements on this strategy, resulting in many sub-variants.

Before this strategy was properly introduced, Jegadeesh and Titman (1993) have shown that relative strength strategies that buy past winners and sell past losers simultaneously realize significant excess returns over 3- to 12-month holding periods. They have shown that the profitability of these strategies are not due to their systematic risk or to delayed stock price reactions to common factors. Except for the 3-month/3-month strategy, meaning that both look-back period and holding period are equal to 3 months, all other strategies led to significant excess returns.

These findings were still supported in their follow-up study Jegadeesh and Titman (2001) and they argue that the momentum effect represents the strongest evidence against the efficient markets hypothesis, which says that if there exists any predictable patterns in returns, it will be exploited by investors instantly, until the source of predictability is eliminated.

Existing literature about momentum strategy focuses mainly on cross-sectional momentum strategies, which contains the above-mentioned strategies. This type of strategy compares assets and buys (sells) relative winners (losers). On the other hand, strategies that only look at absolute returns, where the return of assets in the look-back period are examined, are called time-series momentum strategies. In this case, buy trades are executed if and only if the return of the asset is positive in the look-back period, and sell trades are executed otherwise. Thus, in cross-sectional strategies, portfolios are based on the relative performances of the underlying assets, whereas in the time-series strategies, these portfolios are based on the absolute performances as stated in Moskowitz et al. (2012).

Under most economic circumstances, the performances of both types of momentum strategies are exceptional. In contrast to the relatively high Sharpe ratio of momentum compared to other factors, Barroso and Santa-Clara (2015) have shown that momentum is associated with high crash risk, which results in reduction of the accumulated returns to zero. Needless to say, investors are unlikely to use such strategies that experience devastating crashes. The momentum crashes in 1932 and 2009, as reported in Daniel and Moskowitz (2016), occurred during rebound periods following bear markets. Moreover, Grundy and Martin (2015) have shown that the WML strategy has an overall negative beta, which is a measure of systematic risk, due to the fact that past winners (losers) have a low (high) beta. Controlling for this time-varying beta would then result in stable returns. Daniel and Moskowitz (2016) and Barroso and Santa-Clara (2015) contradicted this finding by proving that beta hedging solely does not avoid these crashes. They used a different method, by estimating the risk with the realized variance (RV) of the daily returns. The RV turned out to be highly predictable. Hence, they proposed a volatility-scaling technique for risk management that would lead to elimination of the crashes and significant improvement of the Sharpe ratios simultaneously.

In Kim (2019), an improved variant of the existing momentum strategies is introduced, by means of including only the positive aspects from both WML and WO strategies. In other words, when even the bottom decile generates positive returns, this decile is also identified as a 'winner', and stocks in both bottom and top deciles are bought. On the other hand, when the top decile realizes negative returns too, this decile is identified as a 'loser', which results in selling stocks in both bottom and top deciles. Therefore, the performance of this strategy depends on the accuracy of the forecasts. This implies that for this strategy, it is possible to have two winners or losers simultaneously. Using several machine learning techniques, they have implemented models to estimate the expected returns to construct their strategies.

In recent years, machine learning algorithms have been widely used in time series predictions. Especially support vector regression (SVR), a shallow regression algorithm, has been suggested to perform forecasts accurately when underlying processes are unknown. Alternatively, with the development of deep learning techniques, Heaton et al. (2016) has shown that results obtained from regression schemes that are based on deep-learning methods are more pragmatic than standard methods used in finance.

Kim (2019) has used shallow learning algorithms as well as deep learning algorithms to predict the returns, and implied that deep learning techniques would lead to improved accuracy of predictions compared to SVR. These models are designed with deep neural networks (DNN) and therefore can extract hidden features in the data.

However, DNNs do not take any time ordering in the data into account, and therefore, relevant sequential information will go lost. Introducing long-short term memory (LSTM) cells into the network would solve this problem. Bao et al. (2017) have implemented a combination of LSTM and stacked autoencoders to forecast stock prices and showed that their model outperforms other models in terms of accuracy.

## 3 Data

The data consists of equal-weighted (EW) daily and monthly returns of the momentum portfolios from January 1927 to March 2019 and is obtained from Kenneth French's data library (French). These portfolios contain NYSE, AMEX and NASDAQ stocks with prior return data, and are constructed by ranking the cumulative returns of stocks from month $t-12$ to $t-2$. By skipping one month in the ranking period, the short-term reversal associated with momentum will be avoided. According to this ranking, the firms will be put in the corresponding momentum portfolio. The descriptive statistics of the monthly returns are shown in Table 1, ranked from the lowest to the highest 10%.

Table 1: Descriptive statistics of the monthly returns

| Portfolio | Mean (%) | Min (%) | Max (%) | Median (%) | St. dev. | Skewness | Kurtosis | Obs. |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.964 | -38.920 | 113.980 | 0.480 | 0.110 | 2.824 | 22.414 | 1107 |
| 2 | 1.100 | -38.420 | 112.130 | 0.900 | 0.090 | 3.150 | 32.641 | 1107 |
| 3 | 1.125 | -36.840 | 74.440 | 1.170 | 0.077 | 2.085 | 20.550 | 1107 |
| 4 | 1.221 | -31.850 | 76.850 | 1.240 | 0.074 | 2.529 | 25.375 | 1107 |
| 5 | 1.213 | -32.670 | 75.390 | 1.450 | 0.068 | 1.742 | 20.788 | 1107 |
| 6 | 1.326 | -29.060 | 62.380 | 1.580 | 0.065 | 1.408 | 16.407 | 1107 |
| 7 | 1.335 | -30.520 | 62.930 | 1.730 | 0.063 | 1.104 | 15.229 | 1107 |
| 8 | 1.435 | -29.180 | 57.350 | 1.760 | 0.062 | 0.965 | 14.102 | 1107 |
| 9 | 1.578 | -29.730 | 42.230 | 1.900 | 0.063 | 0.094 | 5.883 | 1107 |
| 10 | 1.776 | -32.320 | 56.600 | 2.280 | 0.073 | 0.106 | 5.809 | 1107 |

The monthly returns range from -38.920% to 113.980%, while the mean of these portfolios range from 0.964% to 1.776%. The standard deviation remains fairly constant across the portfolios. Given this large range of returns, a momentum strategy that is implemented accordingly would be able to generate good excess returns. Furthermore, the skewness of the portfolios decreases as the momentum portfolio is ranked higher, which is remarkable. In general, a positive skewness is preferred since this indicates an asymmetric distribution with a tail extending towards positive values.

## 4 Methodology

In this section we will discuss three time series prediction models that are used by Kim (2019) and introduce a model based on recurrent neural networks in Section 4.4.

## 4.1 Look-back period model

The first model that we will discuss, is the look-back period model, This model does not require any regression methods.

$$\prod_{i=1}^{12}(1+r_{t-i})-1 \tag{1}$$

In Equation 1, monthly returns in the formation period (previous 12 months) are used to estimate the sign of the monthly return in month $t$. Then, the sign of the predictions determines which action is being executed in the corresponding month. Hence, this is a simple, but an effective approach to incorporate information about past trends in this trading strategy, denoted by SLSa.

## 4.2 SVR model

Another method to predict the monthly returns involves a shallow machine learning technique, which is the support vector regression (SVR). We denote the momentum strategy that corresponds with this prediction method as SLSb. In this model, monthly returns of the formation period $(r_{t-1},...,r_{t-12})$ are used to forecast $r_t$. For each optimization, the sample that is going to be used for training and validating contains the most recent 300 input-output pairs, with each pair consisting of the set $(r_{t'-1},...,r_{t'-12})$ as input, and $r_{t'}$ as output, for $t'=t-1,...,t-300$. Thus, we will create a rolling window of 300 input-output pairs for each one-step ahead forecast.

First, we consider our sample $(\boldsymbol{x_{t'}}, y_{t'}) \in \mathbb{R}^{12}\times\mathbb{R}$ of 300 input-output pairs, where $\boldsymbol{x_{t'}}$ and $y_{t'}$ correspond to $(r_{t'-1},...,r_{t'-12})$ and $r_{t'}$, respectively. This sample is split into a training set of 270 observation pairs and a validation set of 30 observation pairs. The goal of $\epsilon$-SVR is to obtain a function $f(x)$ that maps the input, in our case $(r_{t'-1},...,r_{t'-12})$, to the target variable, $r_{t'}$, with at most $\epsilon$ deviation (Smola and Schölkopf (2004)). In other words, with this configuration we still accept errors that are smaller than the value of $\epsilon$. We define the $\epsilon$-insensitive loss function as $|y-f(\boldsymbol{x})|_\epsilon := \max\{0,|y-f(\boldsymbol{x})|-\epsilon$ (Law and Shawe-Taylor (2017)). Incorporating this loss function in our linear regression $f(x)=\boldsymbol{w}\cdot\boldsymbol{x}+b$ is equivalent to minimizing Equation 2,

$$\frac{C}{N}\sum_{t=1}^{N}|y_t-f(\boldsymbol{x_t})|_\epsilon+\frac{1}{2}||\boldsymbol{w}||^2, \tag{2}$$

where $C$ is defined as a regularization constant to prevent the regression from over-fitting and $N$ is the length of our forecast horizon. By introducing slack variables and Karush-Kuhn-Tucker (KKT) conditions, the optimal weight $\boldsymbol{w^*}$ is given by $\sum_{t=1}^{N}(\alpha_t-\alpha_t^*)K(\boldsymbol{x},\boldsymbol{x_t})$[1], where $\alpha_t$ and $\alpha_t^*$ are Lagrangian multipliers and $K(\boldsymbol{x},\boldsymbol{x_t})$ is the kernel function. Thus, our SVR can be written in the following form,

$$f(\boldsymbol{x},\boldsymbol{v})=f(\boldsymbol{x},\alpha,\alpha^*)=\sum_{t=1}^{N}(\alpha_t-\alpha_t^*)K(\boldsymbol{x},\boldsymbol{x_t})+b, \tag{3}$$

where the Guassian kernel function is defined as $K(\boldsymbol{x},\boldsymbol{x_t})=exp\left(\frac{-||x_t-x_s||^2}{2\gamma^2}\right)$, with the kernel

[1]For in-depth derivations of the optimization problem, please refer to Lu et al. (2009)

parameter $\gamma$. This function serves for mapping the input into a higher dimension space that can describe the non-linearity (Henrique et al. (2018)).

As performance measure for our model, we used the mean squared error (MSE) which we define as $\frac{1}{30}\sum_{i=1}^{30}(r_{t-i} - \hat{r}_{t-i})^2$, where $\hat{r}_t$ is the monthly return estimated with SVR in month $t$. Using this criterion, we have tuned the parameter set $(C, \epsilon, \gamma)$ for each point estimator, where $C \in \{10^i, 5{\cdot}10^i : -2 \leq i \leq 3\}$, $\epsilon \in \{10^i, 5{\cdot}10^i : -5 \leq i \leq -1\}$ and $\gamma \in \{10^i, 5{\cdot}10^i : -5 \leq i \leq -1\}$. Hence, the optimal combination consists of the values of the hyperparameters that minimizes the MSE.

## 4.3 Deep neural network (DNN) using stacked denoising autoencoders (SdAE)

Deep learning applications such as autoencoders are based on machine learning theories, in the sense that a model is being trained to replicate the input data. Prior to implementation, the dataset is split up into two smaller samples used for training and validating as in Section 4.2. The purpose of the training set is to choose the optimal weights of $W_{(\cdot)}$ and $b_{(\cdot)}$. Second, we use the validating set to prevent over-fitting of the model (Heaton et al. (2016)). The sample sizes of the training and validating were set identical to the sizes used for SVR-based prediction for comparison purposes.

In addition to dimensionality reduction, a denoising autoencoder is able to handle corrupted input data and is therefore more robust than an ordinary autoencoder. Here, we assume that the original input data follows a stochastic process, where an arbitrary proportion $v$ of the data is forced to 0. For this so-called “masking noise” process, we have considered noise levels equal to 0, 0.1, 0.25 and 0.4. Then, the network is being trained with the noise-containing data.

In every hidden layer, a non-linear transformation is applied to the noisy input to obtain a latent representation of the original data and map this into a reconstructed version, formulated in Equation 4 and 5,

$$y = f(W_y x' + b_y), \tag{4}$$

$$z = f(W_z y + b_z), \tag{5}$$

where $x' \in \mathbb{R}^d$ represents the corrupted input vector, $y \in \mathbb{R}^{d'}$ the latent representation of $x$ and $z \in \mathbb{R}^d$ the reconstruction of $x$, with $d = 12$ in the first layer and $d \neq d'$. Here, $f$ is a non-linear activation function, i.e. a sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ with a range from 0 to 1. Lastly, $W_{(\cdot)}$ and $b_{(\cdot)}$ are weight matrices and weight vectors respectively. To increase the learning speed during pre-training, we normalize our input data with a min-max scaler by applying the following transformation:

$$x \to (x - x_{min})/(x_{max} - x_{min}),$$

where $x_{min}$ and $x_{max}$ are the minimum and maximum values of the data. This transformation is applied on the first layer only. For the intermediate layers, a “batch normalization” is implemented to standardize the inputs within the network. This has a several advantages, such as accelerating the network, allowing for higher learning rates for convergence and solving the vanishing gradients problem. The latter occurs when the activation function gets saturated as the value of the input increases. This will lead to exponentially decreasing gradients, hence the name “vanishing gradients problem”.

Since $z$ is merely a representation of $x$, it contains some uncertainty. Therefore, the goal is to minimize the loss function $L_s(x, z) = ||x - z||^2$ in each layer.

A stacked denoising autoencoder (SdAE) contains multiple layers, where the latent representation of the previous layer is used as input for the next layer. Pre-training of the SdAE is performed in a greedy layer-wise way, meaning that every autoencoder is being trained individually and sequentially, while freezing the weights $W_{(\cdot)}$ and biases $b_{(\cdot)}$ of the previous layers. This routine is repeated until the final layer is reached. Until now, our construction defines a neural network with an unsupervised learning algorithm. Thus, the last step of this process is fine-tuning our SdAE with a supervised regression in the final layer, which predicts the monthly returns in month $t$.

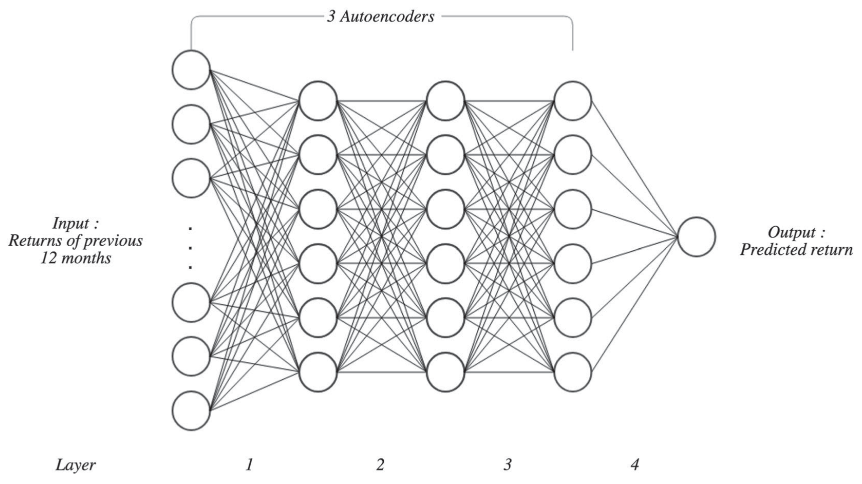
3 Autoencoders
Input : Returns of previous 12 months
Output : Predicted return
Layer
1
2
3
4

Figure 1: Stacked autoencoder containing three hidden layers
Source: Kim (2019)

Our SdAE consists of three sequential autoencoders, with one hidden layer per autoencoder to extract the hidden features. More precisely, the architecture of the whole SdAE is as follows: twelve input nodes, three hidden layers with six nodes each (6-6-6) and one output node, which correspond to the monthly returns in the formation period $(r_{t'-1}, ..., r_{t'-12})$, the latent representations $(\boldsymbol{y_1}, .., \boldsymbol{y_6})$ and the monthly return of month $t$ $r_{t'}$ respectively (See Figure 1). The mapping in the last layer, which is done using a linear regression, provides the estimated monthly returns, given by $\hat{r}_{t'} = \sum_{j=1}^{6} w_j y_j + b$. Lastly, with our validation data, consisting of the most recent 10% of our training data for each point estimator, we will select the parameter for the noise level that minimizes the MSE.

As for the other hyperparameters in the pre-training as well as in the fine-tuning process, we have set the learning rate, number of epochs and batch size to 0.0005, 300 and 100, respectively.

## 4.4 Recurrent neural network (RNN) using LSTM autoencoders

Regular autoencoders do not take the long term dependency of the data into consideration. Since we are dealing with time series, a recurrent neural network (RNN) will be preferred over "regular" autoencoders (DNN). Non-recurrent neural networks assume independency between all of the input vectors, which means that sequential information is not incorporated in the implementation. However, Bengio et al. (1994) have shown that standard RNNs are solely capable of handling sequences, but are not suitable for learning long term dependencies in the data. Therefore, LSTM networks differentiate itselves from standard RNNs by creating a “hidden state” based on the sequential information. These LSTM cells attack the long term dependency problem in recurrent neural networks, and can be decomposed into four parts: an input gate, output gate, forget gate and a cell state vector (See Figure 2).

This neural network has the construction of a deep autoencoder, which is a single autoencoder with multiple layers, rather than a stacked autoencoder. Within a layer, each individual node represents a LSTM cell, which is shown in detail in Figure 3. Each cell processes one time step and emits a signal (the cell state) to the subsequent cell. If we have a stacked LSTM autoencoder, only the state of the last LSTM cell will be passed on to the next autoencoder. In a LSTM network with multiple layers, each LSTM cell will also emit a signal to the LSTM cell in the subsequent layer that corresponds to the time step, which means that the states of each time step will be passed on to the next layer (See Figure 2). Thus, the construction of a deep LSTM autoencoder makes sure that information about the whole sequence is preserved.

In addition, to remove the noise from the data, Wavelet transform[2] (WT) can be applied to the input vector in case of non-stationary characteristics in the data (Bao et al. (2017)). Moreover, to reduce the risk of over-fitting, we will apply a two-level wavelet to our time series data twice.

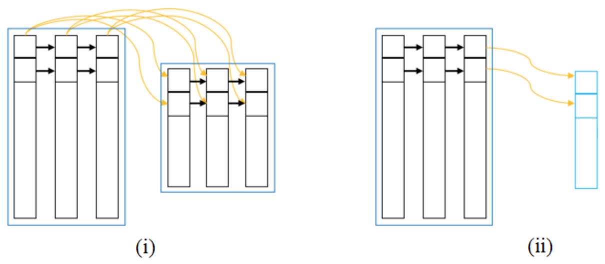

Figure 2: Difference between a deep LSTM autoencoder (i) and a stacked LSTM autoencoder (ii)

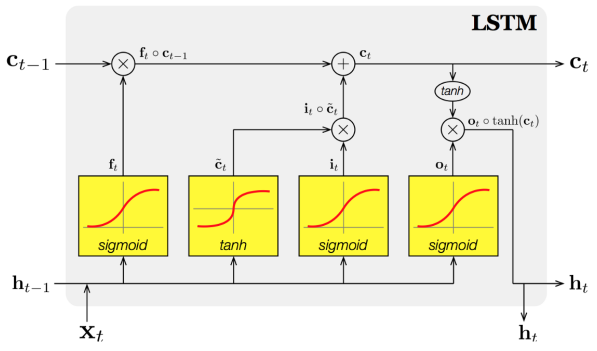

Figure 3: Process inside a LSTM memory cell
Source: Medium

In Equations 6 to 10, $i_t$, $f_t$ and $o_t$ are the input gate, forget gate and output gate at time $t$, respectively. $\tilde{C}_t$ and $C_t$ are defined as the candidate state and regular state of the memory cell. $W_{(\cdot)}$ and $U_{(\cdot)}$ are weight matrices corresponding to the gate. $\sigma(\cdot)$ and $tanh(\cdot)$ denote the sigmoid and hyperbolic tangent activation functions, respectively.

The input layer decides which part of the information will be stored in the cell state in two steps. The first step consist of the input gate layer and decides which values will be updated. In the second step, new candidate values for the cell state will be created with the *tanh* layer:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b) \tag{6}$$

$$\tilde{C}_t = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{7}$$

The forget layer decides which part of the information will be thrown away based on the inputs

[2]The computations of the Wavelet transform can be found in Appendix A

$h_{t-1}$ and $x_t$:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{8}$$

The next step is updating the old cell state with the information from Equation 6 and 8:

$$C_t = i_t \tilde{C}_t + f_t C_{t-1} \tag{9}$$

Finally, the output can be generated based on the cell state. Again, a *sigmoid* layer will be applied to decide which part of the cell state is used for the output. The final *tanh* layer outputs the desirable result:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o) \tag{10}$$

$$h_t = o_t tanh(C_t) \tag{11}$$

Since choosing the number of layers and its nodes is based on trial-and-error, we have found that three hidden layers with eight nodes each would lead to promising results. As for the hyperparameters of this network, we have set the learning rate, number of epochs and batch size to 0.05, 300 and 100, respectively.

## 5 Performance measurement

We will evaluate the models in Section 4 by implementing our trading strategy based on the corresponding predictions. To implement these strategies, only the bottom decile and top decile portfolios with monthly rebalancing for the period January 1954 to December 2018 are used. As mentioned earlier, not hedging the risk of momentum would lead to large crashes. Therefore, we will also consider the volatility-scaled momentum strategy, which manages this risk, in addition to the plain momentum strategy. First of all, we introduce the following variables,

$\{r_t\}_{t=1}^{T}$, monthly returns of plain momentum strategy,

$\{r_t(d)\}_{d=1}^{D}$, daily returns in month $t$ of plain momentum strategy

In order to scale the momentum, we need an estimate of the momentum risk. This can be computed by forecasting the variance using daily returns in the past 6 months for each month. $\hat{\sigma}_t^2$ denotes this variance forecast, with $t$ the corresponding month, as seen in Equation 12:

$$\hat{\sigma}_t^2 = \frac{21}{126} \sum_{\tau=1}^{6} \sum_{d=1}^{21} r_{t-\tau}^2(d), \tag{12}$$

where the number 21 corresponds to the number of trading days in a month.

Then, the scaled returns $r_t^*$ can be obtained by multiplying the returns of the plain momentum strategy with a ratio,

$$r_t^* = \frac{\sigma_{target}}{\hat{\sigma}_t} \cdot r_t \tag{13}$$

where the ratio between the volatilities corresponds to the weight of this scaled momentum strategy. Furthermore, $\sigma_{target}$ is a constant, which corresponds to the target level of volatility and is set to an annualized volatility of 12%.

## 5.1 Profitability performance

The aforementioned types of cross-sectional momentum strategies, WO and WML, are able to generate good profits during stable economic conditions, as shown by Jegadeesh and Titman (1993, 2001). However, when a market crash occurs, the WO strategy will be exposed to significant losses since even the top decile will generate negative returns. In contrast, the WML strategy corrects for different market conditions by buying stocks from the top decile and selling stocks from the bottom decile simultaneously, resulting in a market-neutral strategy. One pitfall of the latter strategy is that in times of bull markets, selling stocks from the bottom decile would also lead to losses.

The selective long/short strategy introduced in Kim (2019) is constructed by combining the strengths of the pure WML and WO strategies. This strategy can be described as follows

$$\begin{array}{ll} \text{Buy signal} & \text{if } \hat{r}_{top} > 0, \\ \text{Sell signal} & \text{if } \hat{r}_{bottom} < 0, \\ \text{No trade} & \text{otherwise,} \end{array}$$

for stocks in the top and bottom decile portfolio respectively. Here, $\hat{r}_{(\cdot)}$ denotes the predicted return of the portfolio in the holding period.

Our main goal is to find the model that is able to generate the highest profits, measured in cumulative returns, under the same trading strategy as described above. For comparison purposes, we will also report the performance of the existing WO and WML strategies in the results.

## 5.2 Predictive accuracy performance

Despite of the idea that our models are constructed such that their aim is to correctly predict the sign (or trend) of the returns, we are still interested in the predictive accuracy performance of these models. Earlier, we have used mean squared error (MSE) as our loss function to obtain the optimal hyperparameters for each forecasting model. To make a comparison across our models in Section 4.2, 4.3 and 4.4, we will use the root mean squared error (RMSE) and mean absolute error (MAE) to measure the predictive accuracy of our models. These measures are defined as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{t=1}^{N}(\hat{r}_t - r_t)^2}, \qquad MAE = \frac{1}{N}\sum_{t=1}^{N}|\hat{r}_t - r_t|,$$

where $r_t$ and $\hat{r}_t$ are the true and predicted returns in month $t$, respectively, and $N$ is the number of months which is equal to 780. Both measures are quite similar, but the essential difference is that for RMSE, the weight given to large errors is relatively higher.

Then, to test whether the predictions of two models are significantly different from each other in terms of accuracy, we will perform a Diebold-Mariano (DM) test (Diebold and Mariano (1995)). Given the actual returns $\{r_t : t = 1, ..., T\}$ and the predicted returns for model $i$ $\{\hat{r}_t : t = 1, ..., T\}$, the forecast errors are defined as:

$$e_{it} = \hat{r}_{it} - r_{it}$$

and the loss differential is defined as:

$$d_t = g(e_{it}) - g(e_{jt}), \qquad \text{for } i \neq j$$

where $g(\cdot)$ is denoted as a loss function, i.e. MSE or RMSE. Model $i$ and $j$ would have the same predictive accuracy if and only if the expectation of the loss differential is equal to zero. This boils down to testing the null hypothesis $H_0 : E(d_t) = 0 \, \forall t$ against the alternative $H_a : E(d_t) \neq 0$. For one-step ahead forecasts, the test statistic can by calculated as:

$$DM = \frac{\bar{d}}{\sqrt{\frac{2\pi \hat{f}_d(0)}{T}}},$$

where $\bar{d} = \sum_{t=1}^{T} d_t$ and $\hat{f}_d(0) = \frac{1}{2\pi} \sum_{k=-(T-1)}^{T-1} I(\frac{k}{h-1}) \frac{1}{T} \sum_{t=|k|+1}^{T} (d_t - \bar{d})(d_{t-|k|} - \bar{d})$. Under the null hypothesis, this test statistic is asymptotic normally distributed $(DM \to (N0, 1))$. Hence, we will reject $H_0$ on a 5% significance level if $|DM| > 1.96$.

## 6 Results

In this section, we will present results of the plain and volatility-scaled selective long/short momentum strategies based on different models. We denote these strategies as the look-back period based strategy (SLSa), the support vector regression based strategy (SLSb), the stacked denoising autoencoder based strategy (SLSc) and the LSTM autoencoder based strategy (SLSd). We will also report the performances of the pure WML and WO strategy. To investigate the performance under different economic conditions, we have divided the sample into three periods of approximately equal duration. During the period January 1954 to December 1978, the economy was relatively stable without the occurrence of significant market crashes. In the period January 1979 to December 1998, the stock market crash occurred in 1987. Lastly, the period January 1999 to December 2018 contains the dot-com bubble burst in 2000 and the financial crisis of 2008.

It is important to note that all returns that we have obtained after implementing different variants of momentum strategies are net profits by subtracting the transaction costs of 0.2% for the period January 1954 to December 1998 and 0.1% for the period January 1999 to December 2018 (Hurst et al. (2017), given a turnover rate of 0.75 (Barroso and Santa-Clara (2015)).

To evaluate the profitability performance of these strategies, the following performance measures were used: average (monthly) return, maximum (monthly) return, minumum (monthly) return, annualized return, annualized volatility, Sharpe ratio, kurtosis, skewness and maximum drawdown (MDD). MDD measures the maximum loss when a trough follows a peak and is an indicator of downside risk. In this paper, the definitions for the Sharpe ratio and MDD are as follows:

$$Sharpe\ Ratio = \frac{R_a}{\sigma_a}, \qquad MDD = \frac{Trough\, Value - Peak\, Value}{Peak\, Value},$$

where $R_a$ is the annualized return and $\sigma_a$ is the annualized volatility. A larger value indicates a better performance for all of these aforementioned measures, except for annualized volatility, kurtosis and MDD.

Table 2: Performance summary of plain momentum strategies

| | | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|---|
| 1954 - 1978 | WML | 0.009 | 0.175 | -0.280 | 0.094 | **0.166** | 0.568 | 9.993 | -1.534 | 0.399 |
| | WO | 0.018 | **0.232** | -0.260 | 0.206 | 0.206 | 1.002 | 5.185 | -0.405 | 0.465 |
| | SLSa | 0.015 | **0.232** | -0.460 | 0.151 | 0.258 | 0.595 | 10.065 | -1.580 | 0.579 |
| | SLSb | 0.016 | 0.200 | -0.260 | 0.236 | 0.191 | 0.990 | 6.279 | -0.792 | 0.460 |
| | SLSc | 0.008 | 0.232 | -0.279 | 0.082 | 0.200 | 0.409 | **4.836** | -0.773 | 0.629 |
| | SLSd | **0.024** | **0.232** | **-0.240** | **0.310** | 0.201 | **1.545** | 5.924 | **-0.217** | **0.299** |
| 1979 - 1998 | WML | 0.011 | 0.126 | -0.199 | 0.127 | **0.149** | 0.854 | 6.257 | -0.797 | 0.364 |
| | WO | 0.019 | 0.171 | -0.325 | 0.230 | 0.217 | 1.057 | 7.455 | -1.131 | 0.367 |
| | SLSa | 0.012 | 0.168 | -0.325 | 0.126 | 0.229 | 0.552 | 7.241 | -1.330 | 0.454 |
| | SLSb | 0.017 | **0.284** | **-0.127** | 0.239 | 0.178 | 1.122 | 5.866 | **0.625** | 0.301 |
| | SLSc | 0.012 | 0.232 | -0.225 | 0.133 | 0.195 | 0.679 | **3.688** | -0.225 | 0.400 |
| | SLSd | **0.028** | 0.171 | -0.225 | **0.365** | 0.184 | **1.978** | 5.324 | -0.637 | **0.225** |
| 1999 - 2018 | WML | 0.003 | 0.192 | -0.601 | -0.013 | 0.277 | -0.046 | 19.884 | -2.850 | 0.849 |
| | WO | 0.012 | **0.314** | **-0.211** | 0.118 | 0.235 | 0.501 | **5.286** | **0.000** | 0.599 |
| | SLSa | 0.005 | **0.314** | -0.648 | -0.017 | 0.350 | -0.048 | 11.240 | -1.698 | 0.786 |
| | SLSb | 0.008 | 0.221 | -0.254 | 0.126 | **0.212** | 0.429 | 6.400 | -0.537 | 0.671 |
| | SLSc | 0.004 | 0.314 | -0.648 | 0.133 | 0.263 | -0.767 | 24.332 | -2.774 | 0.767 |
| | SLSd | **0.023** | **0.314** | -0.273 | **0.287** | 0.226 | **1.270** | 7.825 | -0.027 | **0.386** |
| 1954 - 2018 | WML | 0.008 | 0.192 | -0.601 | 0.070 | 0.203 | 0.345 | 23.887 | -2.733 | 0.849 |
| | WO | 0.016 | **0.314** | -0.325 | 0.185 | 0.219 | 0.847 | **5.824** | -0.481 | 0.599 |
| | SLSa | 0.011 | **0.314** | -0.648 | 0.089 | 0.282 | 0.317 | 11.674 | -1.726 | 0.786 |
| | SLSb | 0.014 | 0.284 | **-0.260** | 0.204 | **0.194** | 0.835 | 6.352 | -0.385 | 0.671 |
| | SLSc | 0.008 | 0.314 | -0.648 | 0.073 | 0.220 | 0.333 | 17.601 | -1.758 | 0.767 |
| | SLSd | **0.025** | **0.314** | -0.273 | **0.319** | 0.204 | **1.566** | 6.740 | **-0.247** | **0.386** |

The columns represent the following performance measures: average monthly return (A), maximum return (B), minimum return (C), annualized return (D), annualized volatility (E), Sharpe ratio (F), kurtosis (G), skewness (H), maximum drawdown (I).

In Table 2 the performance summaries of all plain momentum strategies are shown of three different time periods plus the full time period. Compared to the traditional momentum strategies, the look-back period based SLSa does not perform necessarily better than WML given the statistics, and additionally, this strategy underperforms the WO strategy. Furthermore, we can see that an improvement in the performances has been made by introducing SVR as prediction method, since the SVR-based SLSb performs better than the look-back period based SLSa. On the other hand, implementing a deep learning model using stacked denoising autoencoders (SLSc) did not lead to any improvements compared to SLSa and SLSb.

During most periods, SLSd, the strategy based on LSTM-autoencoders, outperforms the other momentum strategies in the majority of the performance measures. The second best performing portfolio is the portfolio based on the WO strategy, and is followed by SLSb (SVR-based strategy). As mentioned before, the period between 1999 and 2018 experienced devastating market crashes, which is reflected by a relatively low Sharpe ratio and high maximum drawdown in most strategies. When the Sharpe ratio attains a value below zero, it means that investing in this portfolio generates lower returns than the risk-free rate. A maximum drawdown close to 1 indicates that the investment is worthless, since the cumulative return is reduced to nihil. However, SLSd is still able to perform reasonably well under these economic conditions, given a Sharpe ratio of 1.270 and maximum drawdown of 0.386. Furthermore, if we only consider the average monthly returns, it is remarkable that SLSd performs almost twice as good as every other strategy.

In Figures 4-7 the log of the cumulative returns of the plain momentum strategies during period 1954-1978, 1979-1998, 1999-2018 and 1954-2018 are shown. It can easily be seen that SLSd is able to generate abnormal returns, given a log cumulative return of approximately 18 over the full period, which corresponds to a cumulative return of $8 \times 10^7$. Moreover, SLSd performs clearly better than the traditional momentum strategies as well as the strategies based on shallow learning (SLSb) and deep learning (SLSc). Nonetheless, we can see that both WO strategy and SLSb perform extremely well in Figure 7. However, during the financial crisis in 2008, returns of both strategies experienced a large depreciation, and it took years to recover from this crash.

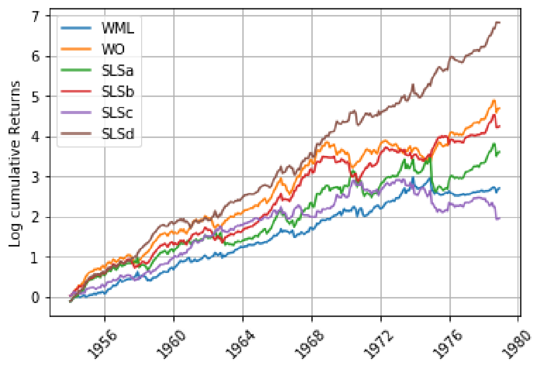

Figure 4: Log cumulative returns of plain momentum strategies in period 1953-1978

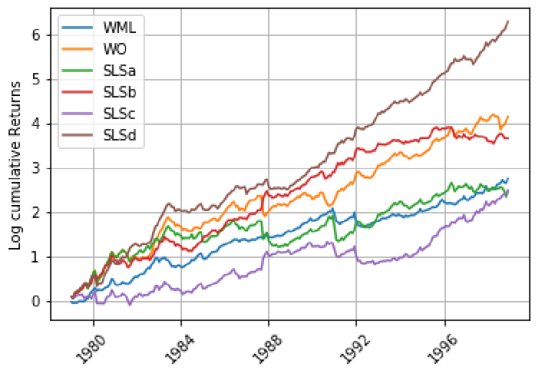

Figure 5: Log cumulative returns of plain momentum strategies in period 1979-1998

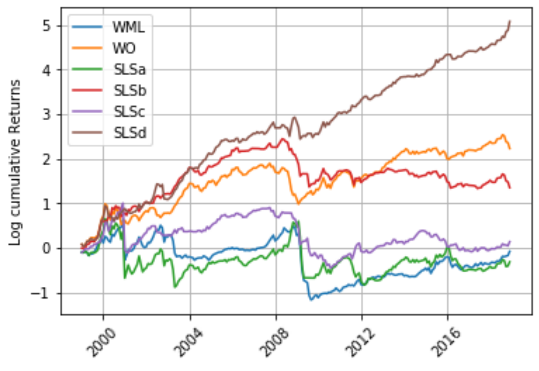

Figure 6: Log cumulative returns of plain momentum strategies in period 1999-2018

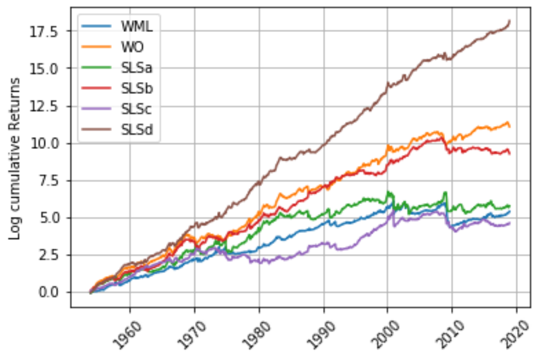

Figure 7: Log cumulative returns of plain momentum strategies in period 1954-2018

In Table 3, the statistics of the trades executed in the selective long/short momentum strategies are presented. During all time periods, the frequency of the long only trade is the highest among other trades for every strategy (SLSa-SLSd). The simultaneous long and short trade and no trade has been picked most often in the SLSc strategy. On the other hand, the short only trade is executed most frequently in the SLSa strategy. Moreover, we notice that the frequencies of different trades did not vary a lot across the three time periods.

Table 3: Long/short statistics of momentum strategies

| | | Long/short | Long only | Short only | No trade |
|---|---|---|---|---|---|
| 1954 - 1978 | SLSa | 0.21 | 0.60 | 0.19 | 0.00 |
| | SLSb | 0.20 | 0.71 | 0.02 | 0.07 |
| | SLSc | 0.27 | 0.41 | 0.13 | 0.19 |
| | SLSd | 0.19 | 0.58 | 0.17 | 0.06 |
| 1979 - 1998 | SLSa | 0.20 | 0..61 | 0.18 | 0.01 |
| | SLSb | 0.27 | 0.47 | 0.13 | 0.13 |
| | SLSc | 0.27 | 0.46 | 0.11 | 0.16 |
| | SLSd | 0.20 | 0.58 | 0.16 | 0.06 |
| 1999 - 2018 | SLSa | 0.19 | 0.56 | 0.22 | 0.03 |
| | SLSb | 0.23 | 0.63 | 0.04 | 0.10 |
| | SLSc | 0.25 | 0.40 | 0.17 | 0.18 |
| | SLSd | 0.25 | 0.52 | 0.16 | 0.07 |
| 1954 - 2018 | SLSa | 0.20 | 0.59 | 0.20 | 0.01 |
| | SLSb | 0.23 | 0.61 | 0.06 | 0.10 |
| | SLSc | 0.27 | 0.42 | 0.14 | 0.17 |
| | SLSd | 0.21 | 0.56 | 0.17 | 0.06 |

We have reported the predictive accuracy performances of the three forecasting models in Table 4. Since the predicted returns were obtained for the bottom and top decile portfolios separately, accuracy measures were available for both portfolios. The values for RMSE and MAE are the lowest for the LSTM model, given values of 0.087 (0.061) and 0.057 (0.045) for the bottom (top) decile, implying that the predictions made with this model are the most accurate compared to the other two models. The differences in accuracy measures between the SVR model and the SdAE model are small, but in general the SdAE model performs slightly better in terms of accuracy. To check whether the performances are significantly different across the models, we perform a DM test, assuming the sample size is large enough. If we test for the differences in predictive accuracies between the SVR model and the SdAE model, we obtain p-values of 0.380 (0.000) based on RMSE and 0.298 (0.000) based on MAE for the bottom (top) decile. Based on these results, we can not reject the null hypothesis at a 5% significance level, if we assume that both series of returns follow the same data generating process. In addition, testing the LSTM model against the SdAE model gives us p-values of 0.000 (0.000) based on RMSE and 0.000 (0.000) based on MAE, which confirms that the LSTM model has indeed the most predictive power.

Table 4: Predictive accuracy performance statistics of ML/DL models

| | RMSE | | MAE | |
|---|---|---|---|---|
| | Bottom | Top | Bottom | Top |
| SVR | 0.118 | 0.068 | 0.072 | 0.051 |
| SdAE | 0.107 | 0.079 | 0.076 | 0.059 |
| LSTM | 0..087 | 0.061 | 0.057 | 0.045 |

We would like to add that nor the long/short statistics, nor the predictive accuracy performances across the models depend on whether or not the momentum strategy has been managed for the risk. Hence, the same results as stated in Table 3 and 4 hold for the volatility-scaled momentum strategies.

Similarly to Table 2, the performance statistics of the different volatility-scaled momentum strategies are presented in Table 5. The purpose of volatility scaling is to generate more stable excess returns by managing the momentum risk. We can see that this is indeed the case since the maximum drawdown and annualized volatility have decreased for most of the momentum strategies. Among the traditional momentum strategies, it can be noticed that risk-managing did not have any (positive) effects on the WO strategy, while the returns, volatility, higher-order moment statistics, Sharpe ratio and maximum drawdown of the WML strategy have improved substantially. Our empirical results confirm that risk-management leads to a decrease in the amount of crashes and an increase in the Sharpe ratio for trading strategies associated with a negative beta. However, despite of this volatility-scaling technique, the WML strategy is still not able to outperform the WO strategy.

Furthermore, it is remarkable that the volatility-scaled strategies SLSa-SLSc are performing worse than the plain versions in terms of returns. The reason why volatility-scaling did not lead to increased returns in these strategies could be due to the fact that relatively more long trades have been executed compared to short trades, resulting in a small-positive or neutral beta portfolio. Since Daniel and Moskowitz (2016) and Barroso and Santa-Clara (2015) only showed the advantages of volatility-scaling on negative beta, further research should be done for portfolios with varying beta.

On the contrary, all performance statistics of SLSd did improve slightly after volatility-scaling. Referring to Table 3, the frequencies of the long and short trades of this strategy did not differ significantly from SLSa-SLSc. Therefore, again we want to emphasize that more research should be done in this field.

Table 5: Performance summary of volatility-scaled momentum strategies

| | | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|---|
| 1954 - 1978 | WML | 0.011 | 0.114 | **-0.197** | 0.130 | **0.131** | 0.997 | 6.751 | -1.118 | **0.259** |
| | WO | 0.017 | 0.172 | -0.284 | 0.199 | 0.193 | 1.034 | 6.019 | -0.668 | 0.423 |
| | SLSa | 0.014 | **0.238** | -0.460 | 0.141 | 0.261 | 0.541 | 9.781 | -1.414 | 0.573 |
| | SLSb | 0.016 | 0.172 | -0.284 | 0.186 | 0.173 | 1.076 | 8.046 | -0.746 | 0.293 |
| | SLSc | 0.009 | 0.232 | -0.279 | 0.100 | 0.187 | 0.537 | 8.818 | -0.611 | 0.521 |
| | SLSd | **0.024** | 0.200 | -0.211 | **0.310** | 0.192 | **1.619** | **4.756** | **-0.251** | 0.281 |
| 1979 - 1998 | WML | 0.017 | 0.155 | -0.251 | 0.208 | **0.156** | 1.330 | 7.886 | -0.789 | **0.315** |
| | WO | 0.018 | 0.218 | -0.387 | 0.212 | 0.223 | 0.950 | 9.712 | -1.271 | 0.404 |
| | SLSa | 0.011 | 0.168 | -0.325 | 0.117 | 0.225 | 0.519 | 7.397 | -1.298 | 0.518 |
| | SLSb | 0.018 | 0.218 | **-0.204** | 0.210 | 0.191 | 1.100 | **5.152** | -0.376 | 0.367 |
| | SLSc | 0.008 | **0.284** | -0.225 | 0.083 | 0.199 | 0.415 | 7.216 | **-0.358** | 0.406 |
| | SLSd | **0.026** | **0.284** | -0.225 | **0.338** | 0.200 | **1.689** | 6.783 | -0.500 | 0.344 |
| 1999 - 2018 | WML | 0.006 | 0.108 | -0.240 | 0.071 | **0.121** | 0.579 | 13.125 | -1.453 | 0.249 |
| | WO | 0.008 | 0.187 | **-0.124** | 0.095 | 0.141 | 0.669 | **5.066** | 0.218 | 0.311 |
| | SLSa | 0.002 | **0.314** | -0.600 | -0.051 | 0.353 | -0.145 | 10.131 | -1.492 | 0.852 |
| | SLSb | 0.007 | 0.187 | **-0.124** | 0.078 | 0.121 | 0.644 | 7.445 | 0.521 | **0.174** |
| | SLSc | 0.000 | **0.314** | -0.404 | -0.039 | 0.261 | -0.150 | 9.333 | -1.113 | 0.933 |
| | SLSd | **0.028** | **0.314** | -0.192 | **0.350** | 0.232 | **1.513** | 5.519 | **0.253** | 0.203 |
| 1954 - 2018 | WML | 0.011 | 0.155 | -0.251 | 0.135 | **0.137** | 0.983 | 8.647 | -0.971 | **0.314** |
| | WO | 0.015 | 0.218 | -0.387 | 0.170 | 0.189 | 0.897 | 8.386 | -0.803 | 0.423 |
| | SLSa | 0.010 | **0.314** | -0.600 | 0.071 | 0.284 | 0.251 | 11.216 | -1.569 | 0.852 |
| | SLSb | 0.013 | 0.218 | -0.284 | 0.159 | 0.166 | 0.959 | 6.936 | -0.350 | 0.367 |
| | SLSc | 0.006 | **0.314** | -0.404 | 0.050 | 0.216 | 0.232 | 9.697 | -0.899 | 0.933 |
| | SLSd | **0.026** | **0.314** | **-0.225** | **0.331** | 0.207 | **1.599** | **5.762** | **-0.095** | 0.344 |

The columns represent the following performance measures: average monthly return (A), maximum return (B), minimum return (C), annualized return (D), annualized volatility (E), Sharpe ratio (F), kurtosis (G), skewness (H), maximum drawdown (I).

Lastly, in Figures 8-11 the log cumulative returns of the volatility-scaled momentum strategies are presented. The first thing that we notice is that in Figure 10, the large decline in returns after the dot-com bubble burst and the financial crisis are less pronounced than in Figure 6 as a result of risk-managing. Secondly, momentum strategies SLSa and SLSc do show increased profits, in contrast to Table 5. Due to a sufficient decline in the maximum drawdown, while having constant returns (between the plain and volatility-scaled strategies), an increase in profits can be achieved. Regardless, our main strategy, SLSd, still outperforms the traditional momentum strategies as well as the modified momentum strategies (SLSa-SLSc) in every time period.

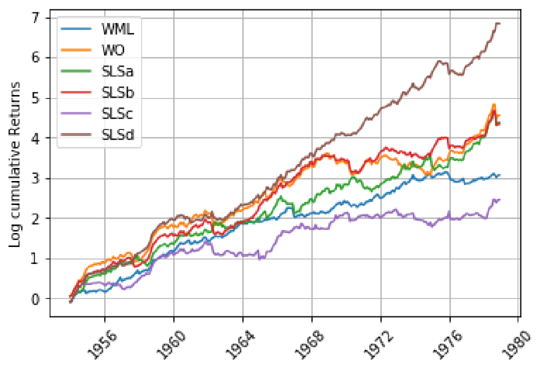

Figure 8: Log cumulative returns of volatility-scaled momentum strategies in period 1953-1978

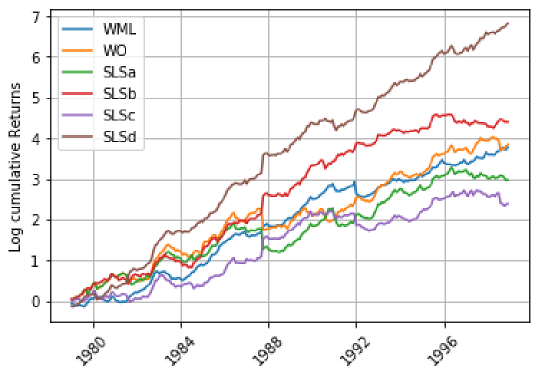

Figure 9: Log cumulative returns of volatility-scaled momentum strategies in period 1979-1998

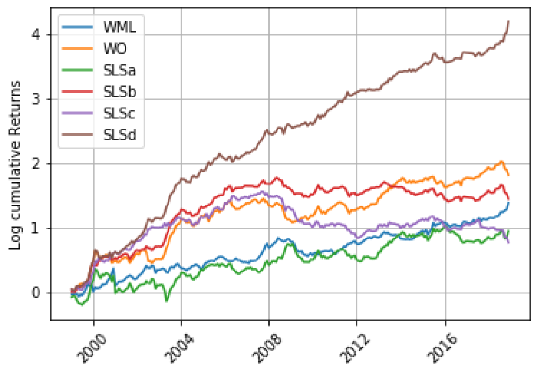

Figure 10: Log cumulative returns of volatility-scaled momentum strategies in period 1999-2018

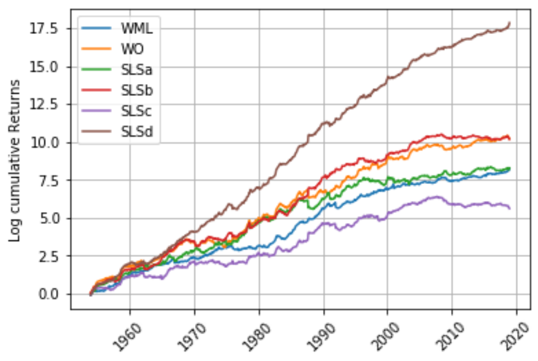

Figure 11: Log cumulative returns of volatility-scaled momentum strategies in period 1954-2018

## 7 Conclusion

The application of deep learning algorithms, and in particular recurrent neural networks, to investing strategies have not yet been studied extensively. Therefore, we have investigated the combination of a LSTM-autoencoder as a prediction model with a selective long/short momentum strategy, and compared this with the traditional momentum strategies, WO and WML, and with other existing strategies reported in Kim (2019). In this research, momentum portfolios consisting of S&P 500 data were used to implement the trading strategies.

Empirical results have shown that our proposed strategy is able to generate exceptionally high returns and can outperform existing strategies, regardless of whether these are based on traditional trading strategies or the selective long/short trading strategy. As we have seen, this finding still holds under different economic conditions.

Furthermore, the selective long/short strategy requires an accurate forecast of the sign of the return, rather than the actual value, which we have evaluated with profitability performance measures. However, we were also interested in investigating to what extent our model is capable of producing accurate forecasts of the returns. Given the relatively high values of the accuracy measures, we suggest that further research needs to be done in developing prediction models using recurrent deep learning algorithms for financial time series forecasting. Nevertheless, our model showed the best results in both profitability and predictive accuracy.

In accordance with our findings, we can conclude that a prediction model based on recurrent neural networks is suitable for the selective long/short momentum strategy. To investigate the generalization of RNN prediction models to other trading strategies, a follow-up study is highly recommended.

## References

W. Bao, J. Yue, and Y. Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLOS ONE*, 12:1–24, 2017.

P. Barroso and P. Santa-Clara. Momentum has its moments. *Journal of Financial Economics*, 116(1):111–120, 2015.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, Mar. 1994.

K. Daniel and T. J. Moskowitz. Momentum crashes. *Journal of Financial Economics*, 122(2):221–247, 2016.

F. Diebold and R. Mariano. Comparing predictive accuracy. *Journal of Business  Economic Statistics*, 13(3): 253–63, 1995.

E. Fama and K. French. Multifactor explanations of asset pricing anomalies. *The Journal of Finance*, 51(1): 55–84, 1996.

K. French. http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html. Accessed: 2019-05-06.

S. G. Mallat. A theory of multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence - PAMI*, 11, 01 1989.

B. D. Grundy and J. S. M. Martin. Understanding the Nature of the Risks and the Source of the Rewards to Momentum Investing. *The Review of Financial Studies*, 14(1):29–78, 2015.

J. B. Heaton, N. G. Polson, and J. H. Witte. Deep learning in finance. *CoRR*, abs/1602.06561, 2016.

B. M. Henrique, V. A. Sobreiro, and H. Kimura. Stock price prediction using support vector regression on daily and up to the minute prices. *The Journal of Finance and Data Science*, 4(3):183 – 201, 2018.

B. Hurst, Y. Hua Ooi, and L. Pedersen. A century of evidence on trend-following investing. *SSRN Electronic Journal*, 01 2017.

N. Jegadeesh and S. Titman. Returns to buying winners and selling losers: Implications for stock market efficiency. *Journal of Finance*, 48(1):65–91, 1993.

N. Jegadeesh and S. Titman. Profitability of momentum strategies: An evaluation of alternative explanations. *Journal of Finance*, 56:699–720, 04 2001.

S. Kim. Enhancing the momentum strategy through deep regression. *Quantitative Finance*, 0(0):1–13, 2019.

T. Law and J. Shawe-Taylor. Practical bayesian support vector regression for financial time series prediction and market condition change detection. *Quantitative Finance*, 17(9):1403–1416, 2017.

C.-J. Lu, T.-S. Lee, and C.-C. Chiu. Financial time series forecasting using independent component analysis and support vector regression. *Decision Support Systems*, 47(2):115 – 125, 2009.

T. J. Moskowitz, Y. H. Ooi, and L. H. Pedersen. Time series momentum. *Journal of Financial Economics*, 104 (2):228 – 250, 2012.

A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

I. Sutskever and G. E. Hinton. Deep, narrow sigmoid belief networks are universal approximators. *Neural Computation*, 20:2629–2636, 2008.

# Appendix

## A Wavelet transform

The Wavelet function for continuous wavelet transform (CWT) is given by:

$$\phi_{a,\delta}(t) = \frac{1}{\sqrt{a}}\phi\Big(\frac{t-\delta}{a}\Big), \tag{14}$$

where $a$ and $\delta$ are scale and translation factors respectively. Also, $\phi(t)$ is defined as the basis wavelet, which satisfies the condition in Equation 15:

$$C_\phi = \int_0^\infty \frac{|\Phi(\omega)|}{\omega} d\omega < \infty, \tag{15}$$

where $\omega$ is the frequency and $\Phi(\omega)$ the Fourier transfom of $\phi(t)$. Given that $x(t) \in L^2(\mathbb{R})$, we define CWT as

$$CWT_x(a,\delta) = \frac{1}{\sqrt{a}}\int_{-\infty}^{+\infty} x(t)\overline{\phi\Big(\frac{t-\delta}{a}\Big)}dt, \tag{16}$$

with the complex conjugate denoted as $\overline{\phi(\cdot)}$.

Then, the inverse transform of CWT is given in the following equation

$$x(t) = \frac{1}{C_\phi}\int_0^{+\infty}\frac{da}{a^2}\int_{-\infty}^{+\infty} CWT_x(a,\delta)\phi_{a,\delta}(t)d\delta \tag{17}$$

To remove the redundant information in the coefficients of CWT, we will use the orthogonal projection of our time series. This results in a discrete wavelet transform, which can be implemented by using the Mallat algorithm (G. Mallat (1989)). This algorithm uses "father wavelets" $\phi(t)$ and "mother wavelets" $\psi(t)$ that describe high-frequency and low-frequency parts of the data, respectively. We can formulate these two types of wavelets as:

$$\phi_{j,k}(t) = 2^{-\frac{j}{2}}\phi(2^{-j} - k) \tag{18}$$

$$\psi_{j,k}(t) = 2^{-\frac{j}{2}}\psi(2^{-j} - k) \tag{19}$$

where $j$ corresponds to the level.

The reconstructed time series consist of a sequence of projections on the mother wavelets and the father wavelets. The multiscale approximation of x(t) takes the following form:

$$x(t) = \sum_k s_{J,k}\phi_{J,k}(t) + \sum_k d_{J,k}\psi_{J,k}(t) + \sum_k d_{J-1,k}\psi_{J-1,k}(t) + ... + \sum_k d_{1,k}\psi_{1,k}(t) \tag{20}$$

where J is the number of multiresolution scales, $k\epsilon$\{0,1,2,..\}, $j\epsilon$\{0,1,2,..,J\} and the expansion coefficients are defined as:

$$s_{J,k} = \int \phi_{J,k}x(t)dt \tag{21}$$

$$d_{j,k} = \int \psi_{j,k}x(t)dt \tag{22}$$

## B Codes

### B.1 DataClass.py

```
import numpy as np
import pandas as pd
import math
from sklearn.preprocessing import MinMaxScaler
```

```
class DataClass:

    def __init__(self):
        pass

    @staticmethod
    def get_data():

        pf = ['Lo PRIOR', 'PRIOR 2', 'PRIOR 3', 'PRIOR 4', 'PRIOR 5', 'PRIOR 6'\
              , 'PRIOR 7', 'PRIOR 8', 'PRIOR 9', 'Hi PRIOR']

        #import daily equal weighted returns
        dr = pd.read_csv('C:/Users/suety/Dropbox/Thesis/Data/daily.csv',\
                                  skiprows=24364)
        dr['date']= pd.to_datetime(dr['date'], format='%Y%m%d').dt.date
        dr = dr.set_index('date')
        dr.index = pd.to_datetime(dr.index)

        #import monthly equal weighted returns
        mr = pd.read_csv('C:/Users/suety/Dropbox/Thesis/Data/monthly.csv',\
                                  skiprows=1121, nrows=1107)
        mr['date']= pd.to_datetime(mr['date'], format='%Y%m').dt.date
        mr[pf] = mr.loc[:,pf].divide(100)
        mr = mr.set_index('date')
        mr.index = pd.to_datetime(mr.index)

        dr = dr.drop(['PRIOR 2', 'PRIOR 3', 'PRIOR 4', 'PRIOR 5', 'PRIOR 6'\
              , 'PRIOR 7', 'PRIOR 8', 'PRIOR 9'], axis=1)
        mr = mr.drop(['PRIOR 2', 'PRIOR 3', 'PRIOR 4', 'PRIOR 5', 'PRIOR 6'\
              , 'PRIOR 7', 'PRIOR 8', 'PRIOR 9'], axis=1)

        return dr, mr

    @staticmethod
    def get_data_scaled(self):
        dr, mr = self.get_data()

        drh = pd.DataFrame(dr['Hi PRIOR'].copy())
        mrh = pd.DataFrame(mr['Hi PRIOR'].copy())

        drl = pd.DataFrame(dr['Lo PRIOR'].copy())
        mrl = pd.DataFrame(mr['Lo PRIOR'].copy())

        sigmaL, mrScaled_lo = self.volScaler(drl, mrl)
        sigmaH, mrScaled_hi = self.volScaler(drh, mrh)

        mr_scaled = pd.concat([mrScaled_lo, mrScaled_hi], axis=1)
        return mr_scaled, sigmaL, sigmaH


##############################################################################
# Scale returns based on monthly volatility

    @staticmethod
    def volScaler (dailyReturns, monthlyReturns):

        dailyReturns = dailyReturns.copy()['1927-07-01 00:00:00':'2018-11-30']
        newMonth = dailyReturns.index[0].month

        rm = np.zeros(shape=(1098,1))

        cumsum = 0
        idx = 0

        for d, row in dailyReturns.iterrows():

            if d.month == newMonth:
                cumsum += dailyReturns.loc[d].copy().values**2
                if idx == 1097:
                    rm[idx,:] = cumsum
            else:
                rm[idx,:] = cumsum
                cumsum = 0
                newMonth = d.month
                cumsum = dailyReturns.loc[d].copy()**2
                idx+=1

        idx=0

        predSig = pd.DataFrame(monthlyReturns['1928-01-01 00:00:00':'2018-12-01
            00:00:00'].copy())
        for m, row in predSig.iterrows():
            window = rm[idx:idx+6].copy()
            predSig.loc[m] = (21/126)*window.sum(axis=0)
            idx+=1

        mr_sc = pd.DataFrame(monthlyReturns.copy()['1928-01-01 00:00:00':'2018-12-01
            00:00:00'].copy())

        for month, row in mr_sc.iterrows():
            mr_sc.loc[month,:] = (math.sqrt(12)/(predSig.loc[month,:])**0.5)*row


        return (predSig, mr_sc)
```

```
###############################################################################
 # Create rolling window for train and test set
   @staticmethod
   def create_dataset(dataset, portfolio, lookback, scaler):

       dataX, dataY = [], []

       for i in range(len(dataset) - lookback):
           a = dataset[i:(i + lookback),portfolio]
           dataX.append(a)
           dataY.append(dataset[(i + lookback),portfolio])
       x_data = np.array(dataX)
       y_data = np.array(dataY)

       y_data = y_data.reshape(len(y_data),1)

       x_train=np.zeros(shape=(x_data.shape[0]-300,270,12))
       y_train=np.zeros(shape=(x_data.shape[0]-300,270,1))
       x_test=np.zeros(shape=(x_data.shape[0]-300,30,12))
       y_test=np.zeros(shape=(x_data.shape[0]-300,30,1))

       x_pred=np.zeros(shape=(x_data.shape[0]-300,1,12))

       x = np.zeros(shape=(x_data.shape[0]-300,300,12))
       y = np.zeros(shape=(x_data.shape[0]-300,300,1))

       for i in range(x_data.shape[0]-300):
           x_train[i,:,:] = x_data[i:i+270,:]
           y_train[i,:,:] = y_data[i:i+270]

           x_test[i,:,:] = x_data[i+270:i+300,:]
           y_test[i,:,:] = y_data[i+270:i+300]

           x_pred[i,:,:] = x_data[i+300,:]

           x[i,:,:] = x_data[i:i+300,:]
           y[i,:,:] = y_data[i:i+300]

           if scaler:
               sc = MinMaxScaler()
               x_train[i,:,:] = sc.fit_transform(x_train[i,:,:])
               x_test[i,:,:] = sc.transform(x_test[i,:,:])
               x_pred[i,:,:] = sc.transform(x_pred[i,:,:])
               x[i,:,:] = sc.transform(x[i,:,:])

       return x_data, y_data, x_train, y_train, x_test, y_test, x_pred, x, y

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

```
# Create matrix with look-back structure
    @staticmethod
    def create_x(dataset, lookback):

        dataX = []

        for i in range(len(dataset) - lookback):
            a = dataset[i:(i + lookback)]
            dataX.append(a)
        x_data = np.array(dataX)

        return x_data

###############################################################################
# Reshape into n samples x timesteps x n features matrix for LSTM
    @staticmethod
    def reshape3D(dataX):

        dataX = dataX.reshape((dataX.shape[0],dataX.shape[1],1))

        return dataX
```

## B.2 PerformanceClass.py

```
import pandas as pd
import numpy as np
import math
from dateutil.relativedelta import relativedelta
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score

class Performance:

    def __init__(self):
        pass
###############################################################################
# define WO strategy

    @staticmethod
    def WOstrat (monthlyReturns):

        mr = monthlyReturns['1954-01-01':'2018-12-01'].copy()
        pr = pd.DataFrame(0, index=np.arange(len(mr)), \
                  columns={'Return'} ).set_index(mr.index)

        for i, row in pr.iterrows():
            r2 = mr.loc[i,'Hi PRIOR']
            if i <= mr.index[539]:
                t = 0.002
            if i > mr.index[539]:
                t = 0.001

            pr.loc[i] = r2 - t*0.75

        return pr

##############################################################################
# define WML strategy

    @staticmethod
    def WMLstrat (monthlyReturns):

        mr = monthlyReturns['1954-01-01':'2018-12-01'].copy()
        pr = pd.DataFrame(0, index=np.arange(len(mr)), \
                  columns={'Return'} ).set_index(mr.index)

        for i, row in pr.iterrows():
            r1 = -mr.loc[i,'Lo PRIOR']
            r2 = mr.loc[i,'Hi PRIOR']

            if i <= mr.index[539]:
                t = 0.002
            if i > mr.index[539]:
                t = 0.001

            pr.loc[i] = r1 + r2 -t*0.75

        return pr

##############################################################################
# define selective long/short strategy

    @staticmethod
    def SLSstrat (predictedReturns, monthlyReturns):

        pr = pd.DataFrame(0, index=np.arange(len(predictedReturns)), \
                         columns={'Return'} ).set_index(predictedReturns.index)
        mr = monthlyReturns['1954-01-01':'2018-12-01'].copy()

        for i, row in predictedReturns.iterrows():
            r1=0
            r2=0

            if predictedReturns.loc[i,'Lo PRIOR'] < 0:
```

```
            r1 = -mr.loc[i,'Lo PRIOR']

          if predictedReturns.loc[i,'Hi PRIOR'] > 0:
            r2 = mr.loc[i,'Hi PRIOR']

          if i <= mr.index[539]:
            t = 0.002
          if i > mr.index[539]:
            t = 0.001

          pr.loc[i] = r1 + r2 - t*0.75

        return pr

###############################################################################
# calculate performance of subgroups

#0 1954-1978
#1 1979-1998
#2 1999-2018
#3 1954-2018

    @staticmethod
    def performanceStat (portfolioReturns):
        smpl_0 = portfolioReturns['1954-01-01':'1978-12-01']
        smpl_1 = portfolioReturns['1979-01-01':'1998-12-01']
        smpl_2 = portfolioReturns['1999-01-01':'2018-12-01']
        smpl_3 = portfolioReturns['1954-01-01':'2018-12-01']
        smpl = [smpl_0, smpl_1,smpl_2,smpl_3]

        info=['Mean','Max', 'Min','Ann. Vol.','Kurtosis','Skewness','Ann.
            Return','Sharpe Ratio','MDD']
        stat = pd.DataFrame(columns = info)
        for i, val in enumerate(smpl):
            meanReturn = val.mean()[0]
            maxReturn = val.max()[0]
            minReturn = val.min()[0]
            kurReturn = val.kurtosis()[0]+3
            skewReturn = val.skew()[0]
            volReturn = val.std()[0]*math.sqrt(12)
            stat.loc[i,0:6] =
                [meanReturn,maxReturn,minReturn,volReturn,kurReturn,skewReturn]

        for s in range(4):
            stat.loc[s,'Ann. Return'] =
                (smpl[s].copy().add(1).prod()**(12/len(smpl[s]))-1).values
            stat.loc[s,'Sharpe Ratio'] = stat.loc[s, 'Ann. Return']/stat.loc[s,'Ann.
                Vol.']
            plusOne = smpl[s].copy()+1
            d = (plusOne).cumprod()
            dd = 1 - d.div(d.cummax())
            stat.loc[s, 'MDD'] = dd.max()[0]

        return(stat)
##############################################################################
# calculate cumulative returns of strategies
# SWMLa
# WML
# WO

    @staticmethod
    def cumReturn(returns, period):
        if period == 0:
            returns = returns['1954-01-01':'2018-12-01']
        if period == 1:
            returns = returns['1954-01-01':'1978-12-01']
        if period == 2:
            returns = returns['1979-01-01':'1998-12-01']
        if period == 3:
            returns = returns['1999-01-01':'2018-12-01']

        creturns = (returns.copy()+1).cumprod() #cumulative returns
        return creturns

##############################################################################
# plot predicted and true returns

    @staticmethod
    def plot2(prediction, true):
        plt.plot(true)
        plt.plot(prediction)
        plt.legend(['True','Prediction'])
        return plt.show()

##############################################################################
# predictive accuracy measures

    @staticmethod
    def pred_accuracy(y_true, y_pred):
        mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
        r2 = r2_score(y_true, y_pred)
        return mape, r2

##############################################################################
# statistics for long/short in SLS strategy
```

```
    @staticmethod
    def longshort(predictedReturns):

        count_matrix = pd.DataFrame(0, index=np.arange(len(predictedReturns)),\
                                    columns = ['long/short','long','short','no trade'])\
                                    .set_index(predictedReturns.index)

        for i, row in count_matrix.iterrows():
            bottom = predictedReturns.loc[i,'Lo PRIOR']
            top = predictedReturns.loc[i,'Hi PRIOR']

            if bottom < 0 and top > 0:
                count_matrix.loc[i,'long/short'] = 1

            elif bottom < 0:
                count_matrix.loc[i,'short'] = 1

            elif top > 0:
                count_matrix.loc[i,'long'] = 1

            else:
                count_matrix.loc[i,'no trade'] = 1

        count_list = [count_matrix['1954-01-01':'1978-12-01'],\
                      count_matrix['1979-01-01':'1998-12-01'],\
                      count_matrix['1999-01-01':'2018-12-01'],
                      count_matrix['1954-01-01':'2018-12-01']]

        idx=0
        stat = np.zeros(shape=(4,4))
        for item in count_list:
            count0 = item.sum()
            total0 = count0.sum()
            stat[:,idx] = count0/total0
            idx+=1

        return stat
```

## B.3 main_lookback.py

```
import numpy as np
import pandas as pd
from dateutil.relativedelta import relativedelta
import matplotlib as plt

from PerformanceClass import *
from DataClass import *
```

```python
import pickle


def load(file1, file2, file3):
    fp = open(file1, "rb")
    pkl1 = pickle.load(fp)
    fp.close()

    fp = open(file2, "rb")
    pkl2 = pickle.load(fp)
    fp.close()

    fp = open(file3, "rb")
    pkl3 = pickle.load(fp)
    fp.close()

    data1 = pkl1.get("pred")
    data2 = pkl2.get("pred")
    data3 = pkl3.get("pred")

    data = [data1, data2, data3]

    return data

#%%###########################################################################
# Load predicted returns (plain)

values_plain_LB = load("plain_lb.pkl", "plain_wo.pkl", "plain_wml.pkl")

#%%###########################################################################
# Load predicted returns (volatility)

values_vol_LB = load("vol_lb.pkl", "vol_wo.pkl", "vol_wml.pkl")

#%%###########################################################################
# Get data

dt = DataClass()
lb = Performance()
dr, mr = dt.get_data()

#%%###########################################################################
# Get volatility scaled monthly returns (and predicted volatiliies)

mr_scaled, sigmaL, sigmaH = dt.get_data_scaled(dt)

#%%###########################################################################
# SWMLb: look-back period-based selective long/short strategy
```

```
def lookbackPrediction(monthlyreturn, skip = False):
    lbreturn = monthlyreturn['1954-01-01':'2018-01-01'].copy()

    # calculate predictions (signs) of monthly returns
    for i, row in monthlyreturn.loc['1954-01-01':'2018-12-01'].iterrows():
        lbreturn.loc[i,['Lo PRIOR','Hi PRIOR']]= \
        ((monthlyreturn.loc[i+relativedelta(months=-12):i+relativedelta(months=-1)\
                          ,['Lo PRIOR','Hi PRIOR']]+1).prod(axis=0)-1)

    if skip:
        for i, row in monthlyreturn.loc['1954-01-01':'2018-12-01'].iterrows():
            lbreturn.loc[i,['Lo PRIOR','Hi PRIOR']]= \
            ((monthlyreturn.loc[i+relativedelta(months=-12):i+relativedelta(months=-2)\
                              ,['Lo PRIOR','Hi PRIOR']]+1).prod(axis=0)-1)
    return lbreturn


# Implement WO, WML and SLS strategy on look-back based predictions
values_plain_LB=[lb.SLSstrat(lookbackPrediction(mr, skip = False), mr),\
               lb.WOstrat(mr),lb.WMLstrat(mr)]
values_vol_LB=[lb.SLSstrat(lookbackPrediction(mr_scaled, skip = False), mr),\
             lb.WOstrat(mr_scaled),lb.WMLstrat(mr_scaled)]


#%%##########################################################################
# Performance statistics of strategies implemented with plain monthly returns

stat_plain_LB = lb.performanceStat(values_plain_LB[0])
stat_plain_WO = lb.performanceStat(values_plain_LB[1])
stat_plain_WML = lb.performanceStat(values_plain_LB[2])

# Performance statistics of strategies implemented with volatility scaled monthly
    returns
stat_vol_LB = lb.performanceStat(values_vol_LB[0])
stat_vol_WO = lb.performanceStat(values_vol_LB[1])
stat_vol_WML = lb.performanceStat(values_vol_LB[2])

longshort_LB = lb.longshort(lookbackPrediction(mr))
longshort_LB_sc = lb.longshort(lookbackPrediction(mr_scaled))

#%%##########################################################################
# Save data

plain_dict = {"plain_0": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr), mr),0),\
             "plain_1": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr), mr),1),\
             "plain_2": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr), mr),2),\
             "plain_3": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr), mr),3),\
             "pred": values_plain_LB[0]}
fp = open("plain_lb.pkl","wb")
pickle.dump(plain_dict, fp)
fp.close()

plain_dict = {"plain_0": lb.cumReturn(lb.WOstrat(mr),0),\
             "plain_1": lb.cumReturn(lb.WOstrat(mr),1),\
             "plain_2": lb.cumReturn(lb.WOstrat(mr),2),\
             "plain_3": lb.cumReturn(lb.WOstrat(mr),3),\
             "pred": values_plain_LB[1]}
fp = open("plain_wo.pkl","wb")
pickle.dump(plain_dict, fp)
fp.close()

plain_dict = {"plain_0": lb.cumReturn(lb.WMLstrat(mr),0),\
             "plain_1": lb.cumReturn(lb.WMLstrat(mr),1),\
             "plain_2": lb.cumReturn(lb.WMLstrat(mr),2),\
             "plain_3": lb.cumReturn(lb.WMLstrat(mr),3),\
             "pred": values_plain_LB[2]}
fp = open("plain_wml.pkl","wb")
pickle.dump(plain_dict, fp)
fp.close()

vol_dict = {"vol_0": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr_scaled),
    mr_scaled),0),\
             "vol_1": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr_scaled),
                 mr_scaled),1),\
             "vol_2": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr_scaled),
                 mr_scaled),2),\
             "vol_3": lb.cumReturn(lb.SLSstrat(lookbackPrediction(mr_scaled),
                 mr_scaled),3),\
             "pred": values_vol_LB[0]}
fp = open("vol_lb.pkl","wb")
pickle.dump(vol_dict, fp)
fp.close()

vol_dict = {"vol_0": lb.cumReturn(lb.WOstrat(mr_scaled),0),\
             "vol_1": lb.cumReturn(lb.WOstrat(mr_scaled),1),\
             "vol_2": lb.cumReturn(lb.WOstrat(mr_scaled),2),\
             "vol_3": lb.cumReturn(lb.WOstrat(mr_scaled),3),\
             "pred": values_vol_LB[1]}
fp = open("vol_wo.pkl","wb")
pickle.dump(vol_dict, fp)
fp.close()

vol_dict = {"vol_0": lb.cumReturn(lb.WMLstrat(mr_scaled),0),\
             "vol_1": lb.cumReturn(lb.WMLstrat(mr_scaled),1),\
             "vol_2": lb.cumReturn(lb.WMLstrat(mr_scaled),2),\
            "vol_3": lb.cumReturn(lb.WMLstrat(mr_scaled),3),\
            "pred": values_vol_LB[2]}
fp = open("vol_wml.pkl","wb")
pickle.dump(vol_dict, fp)
fp.close()
```

### B.4 main_svr.py

```
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error , mean_absolute_error
from sklearn.model_selection import GridSearchCV, TimeSeriesSplit
from sklearn.metrics import make_scorer
from sklearn.preprocessing import MinMaxScaler, StandardScaler, MaxAbsScaler

import numpy as np
import pandas as pd
from PerformanceClass import *
from DataClass import *
import matplotlib.pyplot as plt
import pickle
from math import sqrt


#%%##############################################################################
# Load predicted returns (plain)

fp = open("plain_svr.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnSVR = pkl.get("pred")

#%%##############################################################################
# Load predicted returns (volatility)

fp = open("vol_svr.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnSVR = pkl.get("pred")

#%%##############################################################################
# Get data
dt = DataClass()
svr = Performance()
dr, mr = dt.get_data()

#%%##############################################################################
# Get volatility scaled monthly returns (and predicted volatiliies)
```

```
mr_scaled, sigmaL, sigmaH = dt.get_data_scaled(dt)
volscale = True
if volscale:
    mr = mr_scaled

#%%###############################################################################
# Data preprocessing

trueYL = mr.loc['1954-01-01':'2018-12-01', 'Lo PRIOR'].copy().values.reshape(780,1)
trueYH = mr.loc['1954-01-01':'2018-12-01', 'Hi PRIOR'].copy().values.reshape(780,1)

# prepare data
dataXL, dataYL, trainXL, trainYL, testXL, testYL, predXL, XL, YL =
    dt.create_dataset(mr.values, 0, 12, scaler = False)
dataXH, dataYH, trainXH, trainYH, testXH, testYH, predXH, XH, YH =
    dt.create_dataset(mr.values, 1, 12, scaler = False)

#%%###############################################################################
# SWMLb: SVR-based selective long/short strategy

from hypopt import GridSearch

# Grid-search all parameter combinations using a validation set.
def predictSVR2(trainX, trainY, testX, testY, predX):
    param_grid = [{'C': [1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3,\
                        5*1e-2, 5*1e-1, 5e0, 5*1e1, 5*1e2, 5*1e3],\
    'gamma': [1e-5, 1e-4, 1e-3, 1e-2, 1e-1,\
             5*1e-5, 5*1e-4, 5*1e-3, 5*1e-2, 5*1e-1],\
             'epsilon': [1e-5, 1e-4, 1e-3, 1e-2, 1e-1,\
                         5*1e-5, 5*1e-4, 5*1e-3, 5*1e-2, 5*1e-1],\
                         'kernel': ['rbf']}]

    final_pred = np.zeros(shape=(780,1))
    for i in range(final_pred.shape[0]):
        opt = GridSearch(model = SVR(), param_grid = param_grid)
        opt.fit(trainX[i], trainY[i].ravel(), testX[i], testY[i].ravel(), scoring =
            'neg_mean_squared_error')
        final_pred[i] = opt.predict(predX[i,:])
        print('Prediction ',i+1,' of ',780)

    return final_pred

pred_lo_SVR = predictSVR2(trainXL, trainYL, testXL, testYL, predXL)
pred_hi_SVR = predictSVR2(trainXH, trainYH, testXH, testYH, predXH)

returnSVR = mr['1954-01-01':'2018-12-01'].copy()
returnSVR.loc[:,'Lo PRIOR'] = pred_lo_SVR
returnSVR.loc[:,'Hi PRIOR'] = pred_hi_SVR
```

```
#%%###########################################################################
#Plot predicted and true returns of full dataset
svr.plot2(returnSVR.loc[:,'Lo PRIOR'].values, trueYL)
svr.plot2(returnSVR.loc[:,'Hi PRIOR'].values, trueYH)

#Predictive accuracy
mape_lo_SVR, r2_lo_SVR = svr.pred_accuracy(trueYL, returnSVR.loc[:,'Lo PRIOR'].values)
mape_hi_SVR, r2_hi_SVR = svr.pred_accuracy(trueYH, returnSVR.loc[:,'Hi PRIOR'].values)

rmse_lo_SVR = sqrt(mean_squared_error(returnSVR.loc[:,'Lo PRIOR'].values, trueYL))
rmse_hi_SVR = sqrt(mean_squared_error(returnSVR.loc[:,'Hi PRIOR'].values, trueYH))
mae_lo_SVR = mean_absolute_error(returnSVR.loc[:,'Lo PRIOR'].values, trueYL)
mae_hi_SVR = mean_absolute_error(returnSVR.loc[:,'Hi PRIOR'].values, trueYH)

print('rmse low', rmse_lo_SVR)
print('rmse high', rmse_hi_SVR)

#%%###########################################################################
# Implement SLS strategy on svr based predictions

mr_copy = mr['1954-01-01':'2018-12-01'].copy()

longshort_SVR = svr.longshort(returnSVR)

if volscale:
    stat_vol_SVR = svr.performanceStat(svr.SLSstrat(returnSVR, mr_copy))
else:
    stat_plain_SVR = svr.performanceStat(svr.SLSstrat(returnSVR, mr_copy))

#%%###########################################################################
# Save data

if volscale:

    vol_dict = {"vol_0": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),0),\
               "vol_1": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),1),\
               "vol_2": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),2),\
               "vol_3": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),3),\
               "pred": returnSVR}
    fp = open("vol_svr.pkl","wb")
    pickle.dump(vol_dict, fp)
    fp.close()

else:

    plain_dict = {"plain_0": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),0),\
                 "plain_1": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),1),\
                "plain_2": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),2),\
                "plain_3": svr.cumReturn(svr.SLSstrat(returnSVR, mr_copy),3),\
                "pred": returnSVR}
    fp = open("plain_svr.pkl","wb")
    pickle.dump(plain_dict, fp)
    fp.close()
```

## B.5 main_sdae.py

```
import keras
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PerformanceClass import *
from DataClass import *
import pickle
from sklearn.preprocessing import MinMaxScaler, StandardScaler, MaxAbsScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt

#Seed
np.random.seed(55)
#%%###########################################################################
# Load predicted returns (plain)

fp = open("plain_sae.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnDNN = pkl.get("pred")

#%%###########################################################################
# Load predicted returns (volatility)

fp = open("vol_sae.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnDNN = pkl.get("pred")

#%%###########################################################################
# Get data

dt = DataClass()
sae = Performance()
dr, mr = dt.get_data()

#%%###########################################################################
# Get volatility scaled monthly returns (and predicted volatiliies)
```

```
mr_scaled, sigmaL, sigmaH = dt.get_data_scaled(dt)
volscale = True
if volscale:
    mr = mr_scaled

#%%##############################################################################
# Data preprocessing

trueYL = mr.loc['1954-01-01':'2018-12-01', 'Lo PRIOR'].copy().values.reshape(780,1)
trueYH = mr.loc['1954-01-01':'2018-12-01', 'Hi PRIOR'].copy().values.reshape(780,1)

# prepare data
dataXL, dataYL, trainXL, trainYL, testXL, testYL, predXL, XL, YL =
    dt.create_dataset(mr.values, 0, 12, scaler = True)
dataXH, dataYH, trainXH, trainYH, testXH, testYH, predXH, XH, YH =
    dt.create_dataset(mr.values, 1, 12, scaler = True)

# denoising autoencoder (if True, this is SDAE with masking noise)
# default value
denoising = False

gaussian = False
masking = True
if gaussian:
    denoising = True
    noise = [0.1,0.15,0.3,0.5]
if masking:
    denoising = True
    noise = [0,0.1,0.25,0.4]

if denoising:
    def masking_noise(dataX, noiseparam):

        #Masking noise
        if masking:
            mat = dataX
            prop = int(mat.size * noiseparam)
            i = [np.random.choice(range(mat.shape[0])) for _ in range(prop)]
            j = [np.random.choice(range(mat.shape[1])) for _ in range(prop)]
            mat[i,j] = 0
            dataX = mat

        #Gaussian noise
        if gaussian:
            noise = np.random.normal(loc=0, scale=noiseparam, size=dataX.shape)
            dataX = dataX + noise
        return dataX

#%%###########################################################################
# SWMLc: DNN-using SAE-based selective long/short strategy

from keras.layers import Input, Dense, BatchNormalization, Dropout
from keras.models import Model, Sequential
from keras.optimizers import Adam, SGD, Adadelta
from keras import backend as K

# Hyperparameters
learning_rate = 0.0005
n_epochs = 300
batch_size = 100
n_inputs = 12
n_outputs = 12
n_hidden = 6

# Turn on drop out layers by setting dropout = True
dropout = False
dropout_rate = 0.1
if dropout:
    dropout_layers = 1
else:
    dropout_layers = 0


def neuralNetwork(dataX, dataY, trainX, trainY, testX, testY, dataXTrue, noise):

    # reset memory
    keras.backend.clear_session()

    ############################################################################
#    print('*************************** AUTOENCODER 1 ****************************')

    if denoising:
        dataXd = masking_noise(dataX, noise)
    else:
        dataXd = dataX

    in1 = Input(shape=(n_inputs,))

    e1 = Dense(n_hidden, activation = 'sigmoid') (in1)
    if dropout:
        x = Dropout(dropout_rate) (e1)
    else:
        x = e1
    b1 = BatchNormalization() (x)
    d1 = Dense(n_outputs, activation = 'sigmoid') (b1)

    encoder1 = Model(in1, e1)
    decoder1 = Model(in1, d1)

    decoder1.compile(loss='mean_squared_error', optimizer=Adam(lr=learning_rate))
    decoder1.fit(dataXd, dataX, batch_size=batch_size, \
                epochs=n_epochs, validation_split = 0.1, shuffle = False, verbose = 0)

    ################################################################################
#    print('*************************** AUTOENCODER 2 ****************************')

    # Get latent representation
    get_hidden1 = K.function([encoder1.layers[1].input],
                                [encoder1.layers[1].output])
    hidden1 = get_hidden1([dataX])[0]

    if denoising:
        hidden1d = masking_noise(hidden1, noise)
    else:
        hidden1d = hidden1

    in2 = Input(shape=(n_hidden,))

    e2 = Dense(n_hidden, activation = 'sigmoid') (in2)
    if dropout:
        x = Dropout(dropout_rate) (e2)
    else:
        x = e2
    b2 = BatchNormalization() (x)
    d2 = Dense(n_hidden, activation = 'sigmoid') (b2)

    encoder2 = Model(in2, e2)
    decoder2 = Model(in2, d2)


    decoder2.compile(loss='mean_squared_error', optimizer=Adam(lr=learning_rate))
    decoder2.fit(hidden1d, hidden1, batch_size=batch_size, \
                epochs=n_epochs, validation_split = 0.1, shuffle = False, verbose = 0)


    #################################################################################
#    print('*************************** AUTOENCODER 3 ***************************')

    # Get latent representation
    get_hidden2 = K.function([encoder2.layers[1].input],
                                [encoder2.layers[1].output])
    hidden2 = get_hidden2([hidden1])[0]
    if denoising:
        hidden2d = masking_noise(hidden2, noise)
    else:
        hidden2d = hidden2

    in3 = Input(shape=(n_hidden,))

    e3 = Dense(n_hidden, activation = 'sigmoid') (in3)
    if dropout:
        x = Dropout(dropout_rate) (e3)
    else:
        x = e3
    b3 = BatchNormalization() (x)
    d3 = Dense(n_hidden, activation = 'sigmoid') (b3)

    encoder3 = Model(in3, e3)
    decoder3 = Model(in3, d3)

    decoder3.compile(loss='mean_squared_error', optimizer=Adam(lr=learning_rate))
    decoder3.fit(hidden2d, hidden2, batch_size=batch_size, \
                epochs=n_epochs, validation_split = 0.1, shuffle = False, verbose = 0)


    ################################################################################
#    print('*************************** FINE TUNING *****************************')

    FineTune = Sequential()

    if denoising:
        trainX = masking_noise(trainX, noise)

    for layer in encoder1.layers: # exclude last layer from copying
        FineTune.add(layer)
        if dropout:
            FineTune.add(Dropout(dropout_rate))
        FineTune.add(BatchNormalization())

    for layer in encoder2.layers: # exclude last layer from copying
        FineTune.add(layer)
        if dropout:
            FineTune.add(Dropout(dropout_rate))
        FineTune.add(BatchNormalization())

    for layer in encoder3.layers: # exclude last layer from copying
        FineTune.add(layer)
        if dropout:
            FineTune.add(Dropout(dropout_rate))
        FineTune.add(BatchNormalization())

    for layer in FineTune.layers:
        layer.trainable = True

    FineTune.add(Dense(units=1, activation='linear', input_shape=(n_hidden,)))

    FineTune.compile(loss='mean_squared_error', optimizer=Adam(lr=learning_rate))
    FineTune.fit(trainX, trainY, validation_data = (testX, testY), shuffle = False, \
                epochs=n_epochs, batch_size = batch_size, verbose = 0)

    pred  = FineTune.predict(dataXTrue)

#    FineTune.summary()

    return (pred, FineTune)


def predictSAE(dataX, dataY, trainX, trainY, testX, testY, predX):
    final_pred = np.zeros(shape=(780,1))

    for i in range(final_pred.shape[0]):

        if denoising:
            mse = 1

            for n in noise:

                pred, model = neuralNetwork(dataX[i], dataY[i], trainX[i], trainY[i],
                    testX[i], testY[i], testX[i], n)

                if mean_squared_error(pred, testY[i]) < mse:
                    best_model = model
                    mse = mean_squared_error(pred, testY[i])
                    final_pred[i] = best_model.predict(predX[i])
                print('Noise: ',n)
        print('Prediction ',i+1,' of ',780)

    return final_pred

pred_lo_SAE = predictSAE(XL, YL, trainXL, trainYL, testXL, testYL, predXL)
pred_hi_SAE = predictSAE(XH, YH, trainXH, trainYH, testXH, testYH, predXH)

returnDNN = mr['1954-01-01':'2018-12-01'].copy()
returnDNN.loc[:,'Lo PRIOR'] = pred_lo_SAE
returnDNN.loc[:,'Hi PRIOR'] = pred_hi_SAE

#%%###############################################################################
```

```
# Plot predicted and true returns of full dataset
sae.plot2(returnDNN.loc[:,'Lo PRIOR'].values,trueYL)
sae.plot2(returnDNN.loc[:,'Hi PRIOR'].values,trueYH)

# Predictive accuracy
mape_lo_SAE, r2_lo_SAE = sae.pred_accuracy(trueYL, returnDNN.loc[:,'Lo PRIOR'].values)
mape_hi_SAE, r2_hi_SAE = sae.pred_accuracy(trueYH, returnDNN.loc[:,'Hi PRIOR'].values)

rmse_lo_SAE = sqrt(mean_squared_error(returnDNN.loc[:,'Lo PRIOR'].values, trueYL))
rmse_hi_SAE = sqrt(mean_squared_error(returnDNN.loc[:,'Hi PRIOR'].values, trueYH))
mae_lo_SAE = mean_absolute_error(returnDNN.loc[:,'Lo PRIOR'].values, trueYL)
mae_hi_SAE = mean_absolute_error(returnDNN.loc[:,'Hi PRIOR'].values, trueYH)

print('rmse low', rmse_lo_SAE)
print('rmse high', rmse_hi_SAE)

#%%############################################################################
# Implement SLS strategy on DNN-using SAE based predictions


mr_copy = mr['1954-01-01':'2018-12-01'].copy()

longshort_SAE = sae.longshort(returnDNN)

if volscale:
    stat_vol_SAE = sae.performanceStat(sae.SLSstrat(returnDNN, mr_copy))
else:
    stat_plain_SAE = sae.performanceStat(sae.SLSstrat(returnDNN, mr_copy))

#%%############################################################################
# Save data

if volscale:

    vol_dict = {"vol_0": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),0),\
                "vol_1": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),1),\
                "vol_2": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),2),\
                "vol_3": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),3),\
                "pred": returnDNN}
    fp = open("vol_sae.pkl","wb")
    pickle.dump(vol_dict, fp)
    fp.close()

else:

    plain_dict = {"plain_0": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),0),\
                  "plain_1": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),1),\
                  "plain_2": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),2),\
                "plain_3": sae.cumReturn(sae.SLSstrat(returnDNN, mr_copy),3),\
                "pred": returnDNN}
    fp = open("plain_sae.pkl","wb")
    pickle.dump(plain_dict, fp)
    fp.close()
```

### B.6 main_lstm.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PerformanceClass import *
from DataClass import *
import pickle
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
import keras
from math import sqrt

#Seed
np.random.seed(55)

#%%###############################################################################
# Load predicted returns (plain)

fp = open("plain_lstm.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnRNN = pkl.get("pred")

#%%###############################################################################
# Load predicted returns (volatility)

fp = open("vol_lstm.pkl", "rb")
pkl = pickle.load(fp)
fp.close()
returnRNN = pkl.get("pred")

#%%###############################################################################
# Get data

dt = DataClass()
saelstm = Performance()
dr, mr = dt.get_data()

#%%###############################################################################
# Get volatility scaled monthly returns (and predicted volatiliies)
```

```
mr_scaled, sigmaL, sigmaH = dt.get_data_scaled(dt)
volscale = True
if volscale:
    mr = mr_scaled

#%%##############################################################################
# Data preprocessing

trueYL = mr.loc['1954-01-01':'2018-12-01', 'Lo PRIOR'].copy().values.reshape(780,1)

trueYH = mr.loc['1954-01-01':'2018-12-01', 'Hi PRIOR'].copy().values.reshape(780,1)

# Wavelet transform
import pywt
import mad
from statsmodels import robust

def waveletSmooth( x, wavelet="haar", level=2, DecLvl=2):
    # calculate the wavelet coefficients
    coeff = pywt.wavedec( x, wavelet, mode="per", level=DecLvl )
    # calculate a threshold
    sigma = robust.mad(coeff[-level])
    uthresh = sigma * np.sqrt( 2*np.log( len( x ) ) )
    coeff[1:] = ( pywt.threshold( i, value=uthresh, mode="soft" ) for i in coeff[1:] )
    # reconstruct the signal using the thresholded coefficients
    y = pywt.waverec( coeff, wavelet, mode="per" )
    return y

wav_lo = waveletSmooth(mr.iloc[:,0])
wav_lo = wav_lo.reshape(len(wav_lo),1)
wav_hi = waveletSmooth(mr.iloc[:,1])
wav_hi = wav_hi.reshape(len(wav_hi),1)

# prepare data
dummy, dataYL, dummy, trainYL, dummy, testYL, dummy, dummy, YL =
    dt.create_dataset(mr.values, 0, 12, scaler = False)
dummy, dataYH, dummy, trainYH, dummy, testYH, dummy, dummy, YH =
    dt.create_dataset(mr.values, 1, 12, scaler = False)

# prepare data
dataXL, dummy, trainXL, dummy, testXL, dummy, predXL, XL, dummy =
    dt.create_dataset(wav_lo, 0, 12, scaler = False)
dataXH, dummy, trainXH, dummy, testXH, dummy, predXH, XH, dummy =
    dt.create_dataset(wav_hi, 0, 12, scaler = False)

#%%##############################################################################
# SWMLd: RNN-using LSTM-autoencoder selective long/short strategy
```

```
from keras.layers import Input, Dense, BatchNormalization, Dropout, LSTM,
    RepeatVector, TimeDistributed
from keras.models import Model, Sequential
from keras.optimizers import Adam, SGD, Adadelta
from tqdm import tqdm
from keras import backend as K

#Hyperparameters
learning_rate = 0.05
n_epochs = 300
batch_size = 100
n_inputs = 1
n_outputs = 1
n_hidden = 6


def LSTMNetwork(trainX, trainY, testX, testY, trueX):
    #reset memory
    keras.backend.clear_session()

    model = Sequential()

    model.add(LSTM(8, input_shape = (12,1), return_sequences = True))
    model.add(Dropout(0.2))
    model.add(LSTM(8, return_sequences = True))
    model.add(Dropout(0.2))
    model.add(LSTM(8, return_sequences = False))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation = 'linear'))

    model.compile(loss='mse', optimizer = Adam(lr = learning_rate))
    model.fit(trainX, trainY, batch_size = batch_size, epochs = n_epochs,
        validation_data = (testX, testY), shuffle = False, verbose = 0)

    pred_true = model.predict(trueX)

    return (pred_true)


def predictLSTM(trainX, trainY, testX, testY, predX):
    final_pred = np.zeros(shape=(780,1))

    for i in range(final_pred.shape[0]):

        #Reshape data for LSTM cells
        trainX3 = dt.reshape3D(trainX[i])
        testX3 = dt.reshape3D(testX[i])
        predX3 = dt.reshape3D(predX[i])
        final_pred[i] = LSTMNetwork(trainX3, trainY[i], testX3, testY[i], predX3)
        print('Prediction ',i+1,' of ',780)

    return final_pred

pred_lo_LSTM = predictLSTM(trainXL, trainYL, testXL, testYL, predXL)
pred_hi_LSTM = predictLSTM(trainXH, trainYH, testXH, testYH, predXH)

returnRNN = mr['1954-01-01':'2018-12-01'].copy()
returnRNN.loc[:,'Lo PRIOR'] = pred_lo_LSTM
returnRNN.loc[:,'Hi PRIOR'] = pred_hi_LSTM

#%%###########################################################################
# Plot predicted and true returns of full dataset

saelstm.plot2(returnRNN.loc[:,'Lo PRIOR'].values, trueYL)
saelstm.plot2(returnRNN.loc[:,'Hi PRIOR'].values, trueYH)

# Predictive accuracy
mape_lo_LSTM, r2_lo_LSTM = saelstm.pred_accuracy(trueYL, returnRNN.loc[:,'Lo
    PRIOR'].values)
mape_hi_LSTM, r2_hi_LSTM = saelstm.pred_accuracy(trueYH, returnRNN.loc[:,'Hi
    PRIOR'].values)

rmse_lo_LSTM = sqrt(mean_squared_error(returnRNN.loc[:,'Lo PRIOR'].values, trueYL))
rmse_hi_LSTM = sqrt(mean_squared_error(returnRNN.loc[:,'Hi PRIOR'].values, trueYH))
mae_lo_LSTM = mean_absolute_error(returnRNN.loc[:,'Lo PRIOR'].values, trueYL)
mae_hi_LSTM = mean_absolute_error(returnRNN.loc[:,'Hi PRIOR'].values, trueYH)

print('rmse low', rmse_lo_LSTM)
print('rmse high', rmse_hi_LSTM)

#%%###########################################################################
# Implement SLS strategy on RNN-using LSTM-autoencoder based predictions

mr_copy = mr['1954-01-01':'2018-12-01'].copy()

longshort_LSTM = saelstm.longshort(returnRNN)

if volscale:
    stat_vol_LSTM = saelstm.performanceStat(saelstm.SLSstrat(returnRNN, mr_copy))
else:
    stat_plain_LSTM = saelstm.performanceStat(saelstm.SLSstrat(returnRNN, mr_copy))

#%%###########################################################################
# Save data
```

```
if volscale:

    vol_dict = {"vol_0": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),0),\
                "vol_1": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),1),\
                "vol_2": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),2),\
                "vol_3": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),3),\
                "pred": returnRNN}
    fp = open("vol_lstm.pkl","wb")
    pickle.dump(vol_dict, fp)
    fp.close()

else:

    plain_dict = {"plain_0": saelstm.cumReturn(saelstm.SLSstrat(returnRNN,
        mr_copy),0),\
                "plain_1": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),1),\
                "plain_2": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),2),\
                "plain_3": saelstm.cumReturn(saelstm.SLSstrat(returnRNN, mr_copy),3),\
                "pred": returnRNN}
    fp = open("plain_lstm.pkl","wb")
    pickle.dump(plain_dict, fp)
    fp.close()
```

## B.7 plot.py

```
# Plotting cumulative returns

from PerformanceClass import*
from DataClass import*
import matplotlib.pyplot as plt
import pickle

volscale = False

def load(file, arg):
    fp = open(file, "rb")
    pkl = pickle.load(fp)
    fp.close()
    if arg == "vol":
        data = [pkl.get("vol_0"),pkl.get("vol_1"),pkl.get("vol_2"),pkl.get("vol_3")]
    if arg == "plain":
        data =
            [pkl.get("plain_0"),pkl.get("plain_1"),pkl.get("plain_2"),pkl.get("plain_3")]
    return data

def plotCR(wml, wo, lb, svr, sae, lstm, i):
    plt.plot(np.log(wml[i]))
    plt.plot(np.log(wo[i]))
    plt.plot(np.log(lb[i]))
    plt.plot(np.log(svr[i]))
    plt.plot(np.log(sae[i]))
    plt.plot(np.log(lstm[i]))
    plt.xticks(rotation=45)
    plt.ylabel('Log cumulative Returns')
    plt.grid(True)
    plt.legend(['WML','WO','SLSa','SLSb', 'SLSc', 'SLSd'])
    return plt.show()

if volscale:
    vol_LB = load("vol_lb.pkl", "vol")
    vol_WML = load("vol_wml.pkl", "vol")
    vol_WO = load("vol_wo.pkl", "vol")
    vol_SVR = load("vol_svr.pkl", "vol")
    vol_SAE = load("vol_sae.pkl", "vol")
    vol_LSTM = load("vol_lstm.pkl", "vol")

    plotCR(vol_WML, vol_WO, vol_LB, vol_SVR, vol_SAE, vol_LSTM, 0)
    plotCR(vol_WML, vol_WO, vol_LB, vol_SVR, vol_SAE, vol_LSTM, 1)
    plotCR(vol_WML, vol_WO, vol_LB, vol_SVR, vol_SAE, vol_LSTM, 2)
    plotCR(vol_WML, vol_WO, vol_LB, vol_SVR, vol_SAE, vol_LSTM, 3)

else:
    plain_LB = load("plain_lb.pkl", "plain")
    plain_WML = load("plain_wml.pkl", "plain")
    plain_WO = load("plain_wo.pkl", "plain")
    plain_SVR = load("plain_svr.pkl", "plain")
    plain_SAE = load("plain_sae.pkl", "plain")
    plain_LSTM = load("plain_lstm.pkl", "plain")

    plotCR(plain_WML, plain_WO, plain_LB, plain_SVR, plain_SAE, plain_LSTM, 0)
    plotCR(plain_WML, plain_WO, plain_LB, plain_SVR, plain_SAE, plain_LSTM, 1)
    plotCR(plain_WML, plain_WO, plain_LB, plain_SVR, plain_SAE, plain_LSTM, 2)
    plotCR(plain_WML, plain_WO, plain_LB, plain_SVR, plain_SAE, plain_LSTM, 3)
```