Algorithmic Trading with Long Short-Term Memory Recurrent Neural Network and Stationary Time Series

BSc Computer Science

Final Project Report

Author: Piotr Kurek

Supervisor: Dr. Stefanos Leonardos

Student ID: K20055027

May 20, 2024

Abstract

Over the last decade, algorithmic trading enhanced with predictive signals generated by progressively more advanced machine and deep learning models has emerged as one of the financial markets' most prominent trading strategies. This thesis investigates the application of the Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNNs) in algorithmic trading, focusing specifically on the model's effectiveness in predicting signals from stationary time series data. The project begins with a comprehensive review of the existing literature and a detailed mathematical background on the model's architecture. Subsequently, a step-by-step methodology is presented, highlighting data preprocessing, feature engineering, model training process, and evaluation metrics. Empirical experiments for primary methods and baselines use historical stock data to train the models and measure their performance in predicting the stationary time series of logarithmic returns. The results assess the effectiveness and accuracy of the binary classification and regression LSTM models compared to logistic regression and ARIMA models, providing a means for discussing the strengths and weaknesses of deep learning models in capturing long-term patterns and dependencies in financial markets. Next, a backtesting framework for algorithmic trading strategies is discussed to provide a complete understanding of the transition from time series data predictions to a set of predictive trading signals. Trained models and their respective predictions are used to generate trading strategies to compute the cumulative stock portfolio returns and study the suitability and applicability of the models in algorithmic trading. Furthermore, this research project evaluates the possible reasons for the models' behavior, both from a theoretical point of view and from the perspective of applied deep learning in quantitative finance. A thorough final discussion on the financial, social, and professional issues links and contrasts this work to the current state-of-the-art methods used in hedge funds and the challenges that quantitative finance research faces nowadays while balancing the verge of both academia and the professional industry.

Originality Avowal

Piotr Kurek

May 20, 2024

Acknowledgements

I wish to express my gratitude to my supervisor, Dr. Stefanos Leonardos, for his unwavering support and guidance throughout this project and for allowing me to pursue my chosen interests. I am also grateful to Dr. Martin Forde for his valuable insights and stimulating discussions, which were instrumental in making this project possible.

Contents

Chapter 1

Introduction

1.1 Motivation and project scope

This research project investigates the application of Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNNs) in algorithmic trading, with a specific focus on the model's efficiency in predicting signals from stationary time series data. The project aims to comprehensively analyze LSTM's predictive capabilities and potential for generating profitable trading strategies.

The motivation behind this research stems from the growing prominence of algorithmic trading enhanced with predictive signals generated by advanced machine and deep learning models. However, while numerous projects focus on predicting non-stationary stock prices, a limited body of research explores the prediction of stationary returns. According to the mathematical theory behind time series and conditional expectations, predicting stationary returns should be more complicated. This work not only aims to explain the effectiveness of LSTM models in predicting stationary data but also seeks to demonstrate that projects claiming success in predicting non-stationary stock prices may engage in wishful thinking and could be incorrect. The initial assumption for the results of this work is that any model with an accuracy greater than a random guess should result in a profitable trading strategy.

Chapter 1 begins with an outline of the motivation and project scope in Section 1.1. Section 1.2 provides an extensive literature review, critically examining existing relevant research papers and identifying gaps that this study aims to address. Chapter 2 delves into the mathematical foundations of RNN and LSTM, providing detailed theoretical background on the network's architecture and training process in Section 2.1, and LSTM-specific theory in Section

2.2, including key components such as activation functions and ADAM optimization algorithm. Section 2.3 focuses on the theory behind logarithmic returns time series and their stationarity properties. Section 2.4 introduces the baseline models, logistic regression, and ARIMA, while Section 2.5 discusses the evaluation metrics used in the study. Chapter 3 outlines the step-by-step methodology followed in this research. Starting from the description of the tools and dataset used in Sections 3.1 and 3.2, through feature engineering and data analysis in Sections 3.3 and 3.4, to approach used for creating train and test datasets with a sliding window method, data standardization, hyperparameter optimization process and scoring metrics for each method in Sections 3.5, 3.6, 3.7 and 3.8, respectively. Section 3.9 provides a detailed description of the methodology used to implement models. Section 3.10 introduces a backtesting framework for algorithmic trading strategies to demonstrate how time series predictions can be translated into predictive trading signals. Chapter 4 presents the results and compares the accuracy of binary classification and regression LSTM models against baseline models in Section 4.1, facilitated with a discussion on the strengths and weaknesses of deep learning models in capturing long-term patterns and dependencies in financial markets. Section 4.2 focuses on selecting the best model for algorithmic trading based on the generated trading strategies and the cumulative portfolio returns, assessing the models' suitability to algorithmic trading. Section 4.3 evaluates the results, listing the possible reasons for the models' behavior from both a theoretical perspective and the standpoint of applied deep learning in quantitative finance. Chapter 5 discusses the financial, social, and professional issues, linking and contrasting this work to the current state-of-the-art methods used in hedge funds and the challenges faced by quantitative finance research in balancing the needs of academia and the professional industry. Chapter 6 addresses the limitations of the project, while Chapter 7 draws conclusions based on the findings. Finally, Chapter 8 suggests potential future research directions to build upon the insights gained from this project.

In summary, this research project aims to provide a comprehensive investigation into the application of LSTM networks in algorithmic trading, focusing on their ability to predict stationary time series data and generate profitable trading strategies. The findings of this study contribute to the growing body of knowledge at the intersection of deep learning and quantitative finance, offering insights for both academic researchers and industry practitioners.

1.2 Literature review

The existing body of literature on the applications of deep learning models for predicting and generating trading signals that describe techniques resulting in profitable trading strategies is limited. Successful works are often written by industry professionals and quantitative researchers, resulting in restricted access due to confidentiality and intellectual property considerations. While some publicly available studies focus on predicting financial time series using LSTM networks, they often lack coverage of how these predictions can be further utilized in algorithmic trading, or they do not delve into the mathematical foundations influencing the expected value of predictions. This is primarily due to the fact that once a work develops a profitable trading strategy, it is not made publicly available to guard the competitive advantage of the company sponsoring the research. This section discusses some of the relevant research projects along with their strengths and limitations.

Duemig [2019] provides the main inspiration and foundation for the methods described in this project. The study utilizes the LSTM model to perform binary classification on logarithmic returns time series data to predict the direction of S&P 500 index returns. Models are trained using historical index data from 2010 to 2016 and are tested on 2017 data from Yahoo! Finance. The project compares the performance of LSTM models with a logistic regression baseline model using accuracy and AUC score metrics. Input features include index returns, price volatility, trading volume, and their combinations to investigate how different feature sets affect predictions. The implementation employs a sliding window approach, ADAM optimizer, and data standardization. The project implements many great technical approaches, such as using grid search with time series cross-validation to optimize the LSTM network training. Results from each feature set are close to 60% for both LSTM and logistic regression, with both models scoring around 0.50 value for AUC, indicating a marginal improvement over random guess. However, the work lacks an in-depth comparative analysis, theoretical background on time series stationarity, and discussion of the results' implications, which is crucial for understanding how predictive models behave when faced with stationary time series. The short testing period does not capture the full range of market dynamics, and the study does not evaluate the models' performance in generating predictive signals for algorithmic trading. The conclusion states that the results are unsatisfactory, with LSTM not performing better than logistic regression, as both models had the highest AUC score of 0.55 for the feature set composed of stock returns and trading volume. Furthermore, the work implements only a binary classification method, which imposes significant limitations. This issue is prevalent in most works discussed in this section and is one of the main aspects addressed by this research.

Murthy et al. [2022] proposes a similar approach, using an LSTM model for binary classification of S&P 500 index returns with the same features as Duemig [2019] in different combinations. The project uses logistic regression as a baseline model and AUC metric for evaluation. The approach is slightly better, with a training period starting in 1990, resulting in a higher LSTM model accuracy of 70% for some feature sets and a greater ability to distinguish classes, reflected by an AUC score of around 0.60 for both LSTM model and logistic regression. The implementation uses a grid search and a randomized search with time series cross-validation for optimization. However, the project fails to provide any other method, such as regression, to compare with binary classification. It also does not discuss the applications of the LSTM model in algorithmic trading, making it hard to evaluate the results' usefulness for the industry. The conclusions state that the LSTM model did not perform better than the baseline model and failed to achieve higher AUC scores across different feature sets. It suggests that predicting logarithmic returns is complicated and may require more features than those used in the research. However, the results show no improvement when more input features are used. The final scores actually suggest the opposite. Supplying more information to the model might be a possible solution for achieving better predictions, which is discussed later when introducing the conditional expectation of the random variable in time series.

Ma [2020] compares the performance of ARIMA (Autoregressive Integrated Moving Average), ANN (Artificial Neural Network), and LSTM models in stock price prediction, providing a sound theoretical background and utilizing a regression approach with technical indicators as input features, such as trading volume, trend, momentum, and volatility. This approach perceives logarithmic returns as continuous values, which is potentially more useful during backtesting as the predicted signals can be more granular than simple buy or sell orders based on binary class predictions. However, the study focuses on a single stock over a short 10-month testing period, limiting the generalizability of the findings to other stocks or market conditions. It also lacks a quantitative comparison of the models' performance using standard evaluation metrics like Root Mean Squared Error (RMSE), making it hard to verify if the conclusions are backed by data. Like other works, it does not discuss the practical implications of using these models for real-world trading strategies, nor does it compare regression models with binary classification models. Furthermore, instead of predicting the stationary logarithmic returns, it predicts the non-stationary stock prices, claiming successful results without explaining the consequences of the non-stationarity property for those predictions. The lack of explanation of what it means to predict non-stationary time series could easily mislead readers into thinking that the LSTM model is a good and reliable method for trading. However, the work makes an important observation that the LSTM model outperforms the ARIMA model by making more accurate predictions.

Bhandari et al. [2022] uses the regression LSTM model to predict the next-day closing price of the S&P 500 index. The comparison is made between the accuracy of the single-layer and multilayer LSTM models. The work offers detailed insights into the error values for different designs of the LSTM model, evaluating the model's architecture from the perspective of the number of neurons in the layer, optimizer algorithm, learning rate value, and batch size. It concludes that the single-layer LSTM model provides a superior fit and higher prediction accuracy than the multilayer LSTM model.

Chen et al. [2015] proposes an LSTM model for predicting stock returns in the Chinese stock market and defines a training sequence as a collection of daily stock data for a single stock over a fixed period of 30 days, with each day's data containing 10 input features. It labels the performance of sequences based on the earning rate, calculated by comparing the average closing prices in the 3 days after the sequence with the last day of the current sequence. The study develops a multi-class classification method, using the RMSprop optimizer during training and the normalization technique instead of standardization, in contrast to Duemig [2019] and Murthy et al. [2022]. It concludes that the LSTM model performs better than a random guess, increasing the accuracy of stock returns predictions from 14.3% to 27.2%, especially after data normalization, which helps boost precision. An interesting finding is that results can be improved if sequences with extremely low or high returns are excluded. However, it does not apply the results to algorithmic trading.

Zou and Qu [2022] uses vanilla LSTM, stacked LSTM, attention-based LSTM, and the traditional ARIMA model to predict next-day stock prices for the top 10 S&P 500 stocks by market capitalization. Based on the predictions, the work presents two trading strategies (Long-Only and Long-Short) and compares their performance with a benchmark, a 30% annual return of the index. The strength of this work is that it does not focus on a single asset, allowing a comparison between models' Mean Squared Error (MSE) values across different stocks. However, the backtesting part treats the stocks as a single portfolio, making it impossible to observe the annual returns of strategies for each stock. This project uses trained models to generate predictive signals for trading strategies but has limited discussion on the interpretability and explainability of the deep learning models, particularly the attention-based LSTM model. It achieves good returns on both strategies, with a 173% annual return for the Long-Only strategy and 99% annual return for the Long-Short strategy. However, it does so by predicting the stock price, not logarithmic returns, meaning the time series is non-stationary, which, as discussed later, could rely on risky trading assumptions. The relevant conclusions are that the attention-based LSTM outperforms vanilla LSTM and stacked LSTM models, and vanilla LSTM outperforms stacked LSTM due to the latter's potential for overfitting, which is essential for the model choice of this research.

Work by Miao [2020] is an excellent example of the non-stationary time series prediction using the regression LSTM model. The figures in this work represent the actual vs. predicted stock prices and illustrate the earlier point about incorrect underlying assumptions of non-stationary predictions. At first glance, the predictions seem almost perfect, resulting in a positive evaluation of the project results. However, upon closer inspection of figures showing predicted stock prices for Amazon(AMZN), Google(GOOG), and Facebook(FB), a cautious reader will see that the function of the predicted values is a shifted function of actual values by 1 day. The LSTM model knows that the current value is the best estimator of the next value in a non-stationary time series. Hence, the prediction for tomorrow's stock price is today's stock price. Using predicted stock price values solely based on the last known price value cannot be suitable for trading and carries many risks, especially when the stock price process is volatile. This phenomenon is explained further when discussing the stationarity property of the time series to provide a mathematical explanation and contrast non-stationary time series predictions with stationary time series predictions.

LSTM model variations and similar learning methods are also used in predicting financial time series other than stock prices or returns. Wu et al. [2019] uses an LSTM-based agent combined with reinforcement learning. This deep reinforcement learning approach for creating an optimal trading policy gave mixed results for studied stocks. The work shows that incorporating technical indicators can significantly decrease the influence of white noise on the model. Sen et al. [2021] and Zhang et al. [2022] study how LSTM-based predictions of stock price and returns can be used for portfolio optimization using the minimum-variance portfolio optimization approach. Tu [2020] uses the LSTM network to predict the mid-price's next-day direction change in the order book in a high-frequency trading environment. Weng et al. [2017] studies an approach of using multiple disparate data sources, such as technical indicators, Wikipedia traffic, and Google news, to predict next-day Apple (AAPL) stock returns using Support Vector Machine (SVM), ANN, and Decision Tree models. It concludes that the SVM performance is the most accurate and that using various data sources significantly improves models' performance by reducing data dimensions without losing information.

Machine learning is also used in other fields of finance than generating time series predictions for algorithmic trading. For example, AQR Capital Management, a hedge fund known to produce publicly available papers, published a paper by Gu et al. [2018] that uses natural networks and tree models to measure asset risk premia. Apart from showing how diversified the use of AI is in quantitative finance, their project demonstrates that AI is an inseparable part of the industry, where companies usually have their own teams of researchers responsible for finding new trading strategies and alpha signals using cutting-edge technological solutions.

In discussing possible extensions and future work, both Duemig [2019] and Murthy et al. [2022] mention the potential of investigating a regression LSTM-based approach instead of just binary classification. These two works provide a good introduction to the topic of the LSTM network for predicting financial time series and will be the main building blocks for this work. This project extends the work of Duemig [2019] and Murthy et al. [2022] in three main ways. First, it provides a more comprehensive theoretical background on the LSTM network and time series stationarity property to establish mathematical foundations that could serve as theoretical expectations for empirical results. Second, not only is the binary classification method used during the experiment, but also a regression method that utilizes the LSTM and ARIMA models. Lastly, the predictions are evaluated in terms of generating effective predictive signals and their profitability in algorithmic trading, which is not included in the most relevant literature on LSTM model predictions for financial data, making them irrelevant to the industry. The technical approach used by Duemig [2019] and Murthy et al. [2022], such as the choice of software tools, optimization techniques, data source, and the sliding window approach also used by Chen et al. [2015], are employed in this work. However, more techniques are tested during implementation to ensure the best results are obtained. For the LSTM model implementation, the findings of Bhandari et al. [2022] are the central reference point for choosing model parameters. This project uses the findings of Ma [2020] to decide on the baseline model for the regression method - the ARIMA model, a powerful tool proven to be slightly less accurate in making time series predictions according to Ma [2020], hence ideal as a benchmark.

This work addresses the limitations of the relevant literature by not focusing on a single method but by implementing both binary classification and regression methods to allow comparison. It also addresses another limitation of most works described here: the lack of mathematical background behind the stationarity property of time series. It does so by providing insights into what stationarity property means for prediction outcomes depending on whether the predicted series are stationary, like logarithmic returns, or non-stationary, like stock prices. This work's experiment predicts only logarithmic returns. Still, by doing so and referencing the theory of time series, it also explains why predicting non-stationary time series can generate good results, as in Zou and Qu [2022], Miao [2020], and Ma [2020], that might be based on a misleading assumption.

Some works used multiple input features for producing logarithmic return predictions. In contrast, the work of Duemig [2019] and Murthy et al. [2022] shows that the choice and number of features do not significantly impact predictions, as the predictions will always be around 50% accuracy due to the time series conditional expectation under stationarity property. Therefore, the only input feature in this work is the logarithmic returns time series to ensure no other information is supplied to the model, keeping the data truly stationary. The issue with more features is that they all usually come from a similar type of data, which describes technical indicators or features of the traded asset, such as volume or volatility. Hence, they do not bring any new information. For example, volatility can be inferred from the stock price. A possible solution would be to use data from the company's fundamental analysis rather than the stock's technical analysis, as shown by Weng et al. [2017].

The most important extensions of this work to the current literature are providing a comparative analysis between the two methods and applying the results to algorithmic trading, which was not accomplished with LSTM models except in the work of Zou and Qu [2022]. However, the trading in work of Zou and Qu [2022] was done using non-stationary stock price data, resulting in unrealistic returns. Furthermore, while Zou and Qu [2022] states that the attention-based LSTM model is better than vanilla LSTM, this project focuses on the latter for learning purposes, with more advanced types of LSTM to be investigated in the future. This decision is also backed by the conclusion of the Bhandari et al. [2022] that a single-layer vanilla LSTM model performs better when compared with multilayer models.

Throughout this project, especially in Chapters 2 and 3, the works of Duemig [2019], Murthy et al. [2022], Ma [2020], and Zou and Qu [2022] are referenced during implementation and design choices. At the end of this project, the work of Sen et al. [2021], Zhang et al. [2022], and Tu [2020] are referred to in the discussion of possible future extensions.

Chapter 2

Theoretical Background

2.1 Recurrent Neural Networks

2.1.1 Architecture

RNNs are employed for processing sequential data, where features learned across different inputs are shared throughout the network. While there are various types of RNNs, this section focuses on the architecture of the many-to-many vanilla RNNs. In the model shown in Figure 2.1, x^t denotes an input at time step t of the input vector, and y^t represents the corresponding output.[1] The input is initially fed into the hidden layer of the neural network, which attempts to predict an output. When the next input is provided to the network, instead of only predicting an output, it also considers an activation value a^t passed from the previous step. Consequently, the forward propagation operates using conditional probability, predicting the new value conditioned on all past inputs. Typically, the activation value is initialized as a vector of zeros at the first step, resulting in a random prediction for the initial output.

$$a^0 = \overrightarrow{0} \tag{2.1}$$

The neural network scans the input data from left to right, using a shared matrix of parameters denoted by W for each step. The same set of parameters is used to control the input data, activation values, and predictions, represented by W_{ax}, W_{aa}, W_{ya}, respectively. The second index of the subscript indicates which vector is multiplied with the matrix W. The network performs the forward-propagation algorithm, computing the activation value a for each time

[1]See Andrew Ng's notes on RNN architecture.

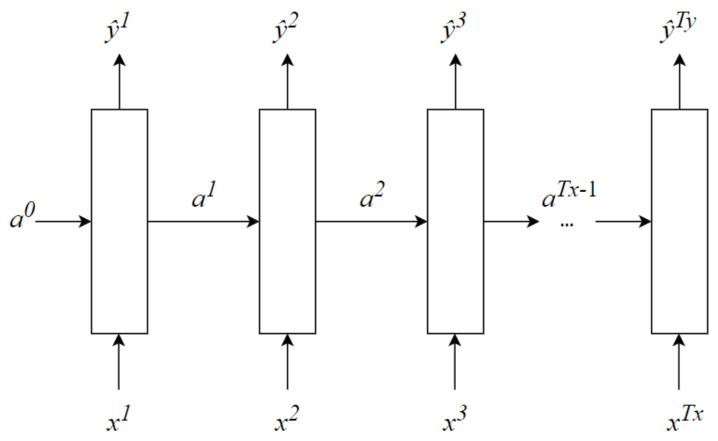


Figure 2.1: Recurrent Neural Network architecture.

step t.

$$a^t = g(W_a[a^{t-1},\ x^t] + b_a]) \tag{2.2}$$

$$\hat{y}^t = g(W_{ya}a^t + b_y) \tag{2.3}$$

In Equations 2.2 and 2.3, b represents the bias term, and g denotes an activation function such as the hyperbolic tangent or softmax function. The activation functions used to calculate the activation and prediction values are usually different.

2.1.2 Training through Backpropagation

The process described above, where the network uses values from previous steps to calculate new values, is known as forward propagation. The RNN learns by calculating gradients in the network using a technique called backpropagation through time. To perform backpropagation, moving through the network from right to left, a loss function is required to fit the network parameters. The goal is to minimize the overall cost function $\mathcal{L}$, which is the average of the individual loss functions $\mathcal{L}^t$, or the error of the output layer over the training dataset. The loss function for the entire training sequence is the sum of the individual loss functions. T denotes the total number of layers in the network.

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}^t(\hat{y}^t,\ y^t) \tag{2.4}$$

The loss function in Equation 2.4 can take various forms, such as the cross-entropy function or mean squared error, which is discussed later. For now, a specific loss function is not required to derive the backpropagation algorithm through which the network learns. Backpropagation efficiently calculates the gradient of the loss for each parameter by traversing the computational graph backward, combining the local gradients of intermediate operations during the forward pass to compute the overall gradient. The backpropagation algorithm finds the cost function gradient using the chain rule of derivatives, which states that the derivative of the composition of two functions can be written as the sum of the multiplication of the individual functions' derivatives.[2] Therefore, the derivative of $\frac{\partial \mathcal{L}}{\partial u_t}$, where $u_t = W_{ya}a^t + b_y$, must be found.

The derivatives with respect to model parameters W_{aa}, W_{ax}, W_{ya}, and b_y can be calculated using the chain rule for a composite function $h(g(f(x)))$. Then the derivative with respect to x is $h^{'}(g(f(x)))g^{'}(f(x))f^{'}(x)$.

$$\frac{\partial \mathcal{L}}{\partial parameter} = \frac{\partial \mathcal{L}}{\partial predicted\ value}\frac{\partial predicted\ value}{\partial parameter} \tag{2.5}$$

The first term on the right side in Equation 2.5 is often referred to as the error term, representing how much the loss function changes with respect to the predicted values. The second term represents how the predicted values change with respect to the model parameters. More details on the derivation of the loss function showed in Equations 2.6-2.10 can be found in the work of Chen [2016].

$$\frac{\partial \mathcal{L}}{\partial W_{aa}} = \sum_{t=1}^{T-1}\sum_{k=1}^{t+1} \frac{\partial \mathcal{L}^{t+1}}{\partial \hat{y}^{t+1}} \frac{\hat{y}^{t+1}}{\partial a^{t+1}} \frac{\partial a^{t+1}}{\partial a^k} \frac{\partial a^k}{\partial W_{aa}} \tag{2.6}$$

$$\frac{\partial \mathcal{L}}{\partial W_{ax}} = \sum_{t=1}^{T-1}\sum_{k=1}^{t+1} \frac{\partial \mathcal{L}^{t+1}}{\partial x^{t+1}} \frac{x^{t+1}}{\partial a^{t+1}} \frac{\partial a^{t+1}}{\partial a^k} \frac{\partial a^k}{\partial W_{ax}} \tag{2.7}$$

$$\frac{\partial \mathcal{L}}{\partial W_{ya}} = \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial \hat{y}^t} \frac{\partial \hat{y}^t}{\partial W_{ya}} \tag{2.8}$$

$$\frac{\partial \mathcal{L}}{\partial b_y} = \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial \hat{y}^t} \frac{\partial \hat{y}^t}{\partial b_y} \tag{2.9}$$

$$\frac{\partial \mathcal{L}}{\partial u_t} = \delta^t a^{t-1} \tag{2.10}$$

[2]See Victor E. Bazterra notes on backpropagation.

After calculating the loss function gradient $\frac{\partial \mathcal{L}}{\partial u_t}$, the goal is to minimize the value of the loss function by using the backpropagation algorithm at each step to calculate the gradients, moving backward through the network. A typical RNN uses stochastic gradient descent to optimize the individual update of the weights for each training example. The idea is to iteratively update the network parameters to minimize the loss function for the training data. This process is often referred to as a Delta Rule.

$$W_{new} \leftarrow W_{old} - \alpha \frac{\partial \mathcal{L}}{\partial u_t} \tag{2.11}$$

In Equation 2.11, α is the learning rate hyperparameter, and $\frac{\partial \mathcal{L}}{\partial u_t}$ is the gradient of the loss function for a specific training example, defined as the difference between the target and actual values measured by a loss function, which replaces the delta term δ^t in Equation 2.10. The gradient is backpropagated through time to capture long-term dependencies in the network. The learning rate controls the degree to which the weights change at each step by determining the step size in the parameter space. Choosing an appropriate learning rate is crucial for defining how much the model parameters change with each update when searching for the minimum value of the optimized function. A learning rate value that is too large might lead to faster convergence but risks overshooting the minimum, while a learning rate value that is too small results in stable convergence but slower computations.

2.1.3 Vanishing Gradient Problem

RNNs often face the issue of the vanishing gradient problem, especially when training the network on very long sequences. This means that the network struggles to capture long-term dependencies. It can be understood by considering the multiplication of many weight matrices in the network, as the same weights are used at every time step. When multiplying fractions, the gradient vanishes as it approaches 0; conversely, when multiplying large numbers, the gradient explodes as it approaches infinity. In the backpropagation algorithm, the new weight for a node is calculated from the old weight and the gradient of a loss function multiplied by the learning rate constant (see Equation 2.10). As seen in the previous subsection, the loss function gradient is derived from the product of gradients using the chain rule of derivatives, which makes the new weights dependent on the gradients of the loss function of previous nodes. During each iteration, network weights are updated with the derivative of the loss function with respect to the current weight. This leads to a decrease in the magnitude of the gradient and a slowing of the training process. In the chain rule, the gradient multiplication used to compute previous layers in the network leads to the gradient decreasing exponentially with t. The vanishing gradient problem occurs frequently, and while there are several ways to deal with it, the most popular and the one used in this work is the LSTM network.

2.2 Long Short Term Memory Model

2.2.1 Architecture

The LSTM network is a variation of the RNN that allows for dealing with the vanishing gradient problem. In this network model, a memory cell vector c is introduced to keep track of long-term dependencies.

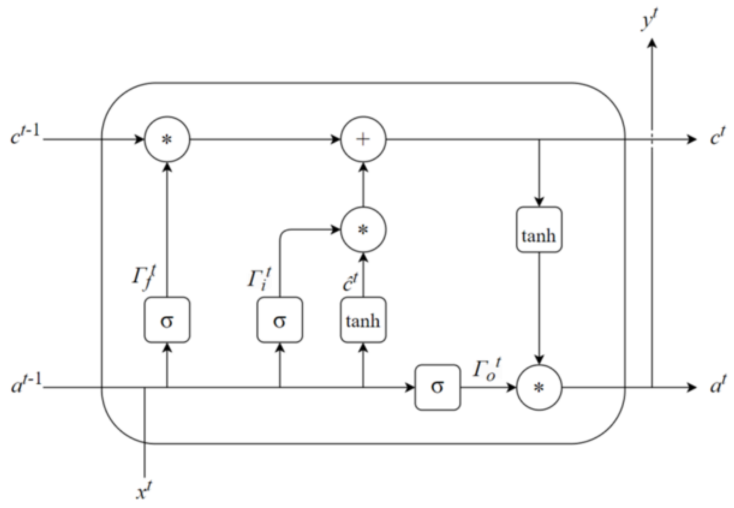


Figure 2.2: Long Short Term Memory cell architecture.

At each time step, a new candidate value is calculated for overwriting the memory cell $\tilde{c}^t$ using the hyperbolic tangent activation function. Gates are also introduced: the input gate Γ_i, forget gate Γ_f, and output gate Γ_o.[3] The input gate controls the candidate value $\tilde{c}^t$, which is a vector of the same dimension as the memory cell and is calculated using the sigmoid function $\sigma(x)$. This results in the gate value being either close to 0 or 1. When the network memory does not need to be updated, the sigmoid function input is a large negative value, and the update gate will have a value close to 0, meaning it does not affect the memory cell where long-term dependencies are stored, solving the vanishing gradient problem. The forget gate controls the value of the memory cell from the previous time step, while the output gate controls the relevance of the new memory cell for the current time step activation value.

[3]See Andrew Ng's notes on LSTM architecture.

The cell state vector, denoted as c^t, is the internal memory of the LSTM cell that stores long-term information and is responsible for selectively retaining or forgetting information over time. The hidden state vector, denoted as a^t, represents the output of the LSTM cell at a particular time step t and encapsulates the relevant information learned from the previous time steps, serving as input to the next time step. The hidden state vector is also used as the output of the LSTM cell and is passed to subsequent layers or used to make predictions. The hidden state vector and cell state vector work together to capture and preserve relevant information over long sequences. The hidden state vector focuses on short-term dependencies and provides the output at each time step, while the cell state vector maintains the long-term memory of the LSTM cell.

$$\tilde{c}^t = \tanh(W_c[a^{t-1},\ x^t] + b_c) \tag{2.12}$$

$$\Gamma_i^t = \sigma(W_i[a^{t-1},\ x^t] + b_i) \tag{2.13}$$

$$\Gamma_f^t = \sigma(W_f[a^{t-1},\ x^t] + b_f) \tag{2.14}$$

$$\Gamma_o^t = \sigma(W_o[a^{t-1},\ x^t] + b_o) \tag{2.15}$$

$$c^t = \Gamma_i^t \circ \tilde{c}^t + \Gamma_f^t \circ c^{t-1} \tag{2.16}$$

$$a^t = \Gamma_o^t \circ \tanh c^t \tag{2.17}$$

In Equations, 2.16 and 2.17, the $\circ$ denotes the Hadamard product of vectors. In the case of the LSTM, the backpropagation algorithm is also used, requiring the calculation of the gradient of the loss function with respect to the input gate, forget gate, and output gate parameters, which is more complicated and is not fully derived here. The exact process is studied in the work of Tu [2020].

2.2.2 Activation Functions

When constructing a neural network, an activation function must be chosen for the hidden layers and output units, and these functions can be different for each layer. The activation function determines the importance of the neuron's input for the network's learning process.

The sigmoid function, Equation 2.18, is a function with a single inflection point at $\sigma(0) = 0.5$, quickly converging to its saturation values of 0 and 1. The downside of this function is that when the input is very small or very large, the gradient becomes very small, which can slow down gradient descent. As its output is 0 or 1, it is used to manage the LSTM memory by either forgetting or remembering information.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.18}$$

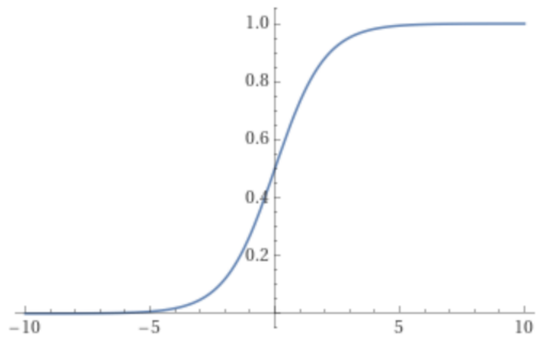


Figure 2.3: Sigmoid function.

The hyperbolic tangent function, Equation 2.19, is a shifted and scaled version of the sigmoid function that has an x-intercept at 0 with extreme values at 1 and -1, where most of the values reside. It usually works better than the sigmoid function as it has a mean of 0, centering the data around 0 and making the learning process easier. It is used to deal with the vanishing gradient problem as its second derivative can perform well over long sequences before approaching 0.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.19}$$

These two functions are part of the LSTM layer itself. However, the output layer in the model also needs an activation function on the hidden states. The choice of the output layer activation function depends on the purpose behind using the LSTM network. The output layer in an LSTM network produces the prediction based on the information processed by the LSTM cells throughout the sequence. The output layer takes the hidden state of the last time step

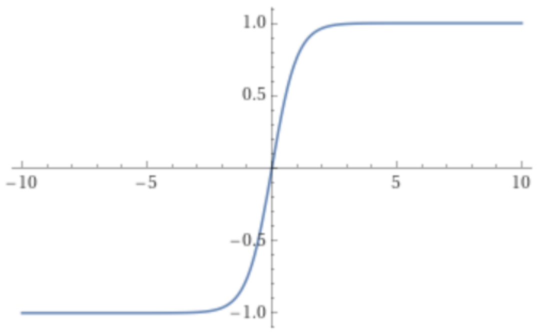


Figure 2.4: Hyperbolic tangent function.

and transforms it into the final prediction, mapping the learned representations in the LSTM hidden states to the desired format for the specific task at hand. For binary classification tasks, the output layer function is a sigmoid function, as its output is a binary class of 0 or 1. Usually, an LSTM model does not have an activation function in the output layer, and the output is a linear combination of the hidden states from the last time step. In this case, the output can take any real value, and therefore, it is used in the regression. The lack of a specific activation function is referred to as a linear activation function.

2.2.3 Binary Classification with LSTM

In Subsection 2.1.2, a specific loss function was not defined. This subsection and the next one address this gap by defining loss functions for binary classification and regression models. For binary classification, the cross-entropy loss formula is used.

$$\mathcal{L}^t(\hat{y}^t,\ y^t) = -[y^t \log \hat{y}^t + (1 - y^t)\log(1 - \hat{y}^t)] \tag{2.20}$$

The loss function for the entire training sequence is the sum of the individual loss functions.

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}^t(\hat{y}^t,\ y^t) = -\frac{1}{T}\sum_{t=1}^{T} y^t \log \hat{y}^t \tag{2.21}$$

With the cross-entropy loss function and the sigmoid function as the activation function for the output layer, the loss function gradient is the difference between the predicted and actual values of the target, where their values are either 0 or 1.[4]

$$\frac{\partial \mathcal{L}}{\partial u_t} = (\hat{Y}^t - Y^t)a^{t-1} \tag{2.22}$$

[4]See Derivation of the loss function for the binary classification.

The update step in gradient descent from Equation 2.10 becomes:

$$W_{new} \leftarrow W_{old} - \alpha(\hat{Y}^t - Y^t)a^{t-1} \tag{2.23}$$

2.2.4 Regression with LSTM

For the regression model, the Mean Squared Error (MSE) function is used:

$$MSE = \frac{1}{T}\sum_{i=1}^{T}(y_i - \hat{y}_i)^2 \tag{2.24}$$

With this loss function, the Equation 2.4 becomes:

$$\mathcal{L}^t(\hat{y}^t,\ y^t) = \frac{1}{T}\sum_{i=1}^{T}(y^t - \hat{y}^t)^2 \tag{2.25}$$

This is then minimized at each step using gradient descent. Therefore, the loss function to be optimized is:

$$\frac{\partial \mathcal{L}}{\partial u_t} = 2(\hat{Y}^t - Y^t)a^{t-1} \tag{2.26}$$

Hence, the update of weights for finding local minima with stochastic gradient descent is:

$$W_{new} \leftarrow W_{old} - \alpha 2(\hat{Y}^t - Y^t)a^{t-1} \tag{2.27}$$

In the implementation of the model in Section 3.7, the loss function is the Root Mean Squared Error (RMSE), as this allows for easier interpretation of the score.

2.2.5 ADAM Optimization Algorithm

So far, stochastic gradient descent has been used in the derivations of the backpropagation algorithm. However, there are many other more efficient optimizers, and the one used to optimize the LSTM network in this work is the Adaptive Momentum Estimation (ADAM) optimizer. This optimizer was used by Duemig [2019] and was found to have the best performance according to Bhandari et al. [2022]. The optimization algorithm determines how the parameters (weights) in the network are updated from the previously calculated gradients of the loss function. It incorporates adaptive learning rates for each parameter and employs moments from the first-order (mean) and second-order (variance) gradients to adaptively adjust the step sizes by changing the previously shown stochastic gradient descent algorithm through the use of a moving average from the first and second moments of the gradient. To implement the ADAM optimizer, an aggregate of gradients and an aggregate of the squared gradients at time t are needed.

First, the algorithm initializes the first and second moment vectors with the moving average of the gradient and the moving average of the squared gradient, respectively. At each time step t, the algorithm uses a backpropagation algorithm to compute the gradient of the loss function with respect to the model parameters weights. Then, the moments are updated with new moving averages by taking the previous value of the moving average and loss function gradient, weighted by the exponential decay rate. The first moment vector stores the short-term memory of the gradients, with more weight given to the latest observations. The second moment vector's purpose is to store the variance estimate of the gradient.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\partial \mathcal{L}}{\partial u_t}\right] \tag{2.28}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial \mathcal{L}}{\partial u_t}\right]^2 \tag{2.29}$$

The next step is to correct these vectors with the decay rate. Due to the initial value of 0 for these vectors, they are biased towards 0.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.30}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.31}$$

The final step is to update the model parameters (weights) to values that optimize the loss function.

$$W_{new} = W_{old} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.32}$$

$\frac{\partial \mathcal{L}}{\partial u_t}$ represents the gradient of the cost function with respect to weights at step t, m_t and v_t are the first and second moment vectors, respectively, β_1 and β_2 are decay rates for the moments, $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected moment estimates, α is the learning rate constant, and ϵ is a small constant to prevent division by zero. The main advantage of the ADAM optimizer is its adaptive learning rates, which are adjusted for each parameter based on how frequently the parameter is updated. This is an important improvement from the stochastic gradient descent, where parameters are updated with a fixed learning rate.

2.3 Logarithmic Returns Time Series

To understand the data used in this study, it is important first to comprehend what logarithmic returns are as a type of time series. As discussed in Chapter 1, it is easier to predict stock prices than logarithmic returns. To explain this phenomenon, the concepts of stationarity and non-stationarity in time series, particularly in stochastic processes, must be explored. By delving into these ideas, the unpredictable nature of logarithmic returns can be better understood.

2.3.1 Stochastic Processes

A stochastic process is defined as a collection of random variables indexed by a time step, where the joint probability of its random variables is specified by the distribution. In the case of stock prices, it is a Gaussian distribution $X_t \sim \mathcal{N}(\mu t, \sigma^2 t)$. A plain stochastic process is denoted by $X_t : t \in T$, where X_t is a random variable representing the value at time t and T is the index set representing time. The stock price time series is defined as an Itô process, which is a stochastic process formally defined by the stochastic differential equation (SDE) of the form:

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t \tag{2.33}$$

In Equation 2.33, X_t is the value of the process at time t, $\mu(t, X_t)$ represents the drift coefficient, $\sigma(t, X_t)$ represents the diffusion process, dt stands for the infinitesimal time increment, and dW_t denotes the differential of the Brownian motion. The Brownian motion is an elementary random walk with multiple steps of unit size, defined as a sum of many random variables z_t that have expectations of 0 and variance equal to 1.

$$B_{1,T} = \sum_{t=t_0+1}^{t_o+T} z_t \tag{2.34}$$

In Equation 2.34, Brownian motion is represented by B, T stands for path length (a path can be understood as a time series that has a final value), and 1 represents a time step. The Brownian motion represents random fluctuations in the stock price model. Therefore, a stock price time series can be defined as an Itô process using the geometric Brownian motion model, with the dynamics of the stock price described by the following SDE:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{2.35}$$

In Equation 2.35, the X_t value of the process at time t is replaced by the S_t stock price value. An equation for the stock price at time T can also be derived by applying Itô's Lemma to the above SDE, resulting in Equation 2.36.

$$S_T = S_0 e^{(\mu - \frac{\sigma^2}{2})dt + \sigma W_t} \tag{2.36}$$

This model is widely used in option pricing; however, as this project focuses on machine learning and time series, the derivation of the stock price model is not discussed in detail. Nevertheless, readers are encouraged to familiarize themselves with stochastic processes in the context of stock returns, described in detail in the work of Kurek [2023] on option pricing methods.

Logarithmic returns time series are modeled as a simple logarithmic transformation of the above stochastic process. The logarithmic return at time t, denoted by $r(t)$, is defined as the natural logarithm of the change between subsequent prices in the time series.

$$r_t = \ln\left(\frac{S_t}{S_{t-1}}\right) \tag{2.37}$$

Therefore, the stock logarithmic returns time series is itself a stochastic process that is assumed to be Gaussian with mean μ and standard deviation σ. The expectations of these stochastic processes and their properties can be examined more closely.

2.3.2 Stationarity and Non-Stationarity of Time Series

Stationarity is a property of a stochastic process where its mean and variance remain constant over time, meaning the time series process defined in terms of the stochastic process exhibits consistent behavior regardless of when the observations are made. The opposite is non-stationarity, where the statistical properties of the process change depending on the time step when the value is observed in the time series. As seen in Equation 2.36, the stock price model follows a Gaussian distribution, and its second moment for a stock price time series is:

$$Var(S_t) = \sigma^2 t \tag{2.38}$$

As shown in Equation 2.38, the variance of the stock price model is time-varying and depends on time step t, violating the stationarity property conditions and indicating non-stationarity. The expected value of the next value in the time series under the conditional expectation is essentially what is computed when predicting the next values with predictive models. More formally, the conditional expectation of the stock price at time $t+1$, given the information available at time t, is the current price itself, making the stock price model a martingale, where the conditional expectation of the future value, given all past values, is equal to the current value. This is what was meant in Chapter 1 when stating that predicting the expected price in the time series is trivial. Now, there is a mathematical explanation.

$$E[S_{t+1}|\mathcal{F}_t] = S_t \tag{2.39}$$

In Equation 2.39, $\mathcal{F}_t$ represents the information available at time t when making a forecast. This proves that the best prediction of the stock price at time $t+1$ is the stock price value at time t. This also explains the results obtained in the work of Miao [2020], discussed in Section 1.2, and why the graph showing the prediction results vs. actual values seems to be shifted by 1 time step.

Logarithmic returns time series, on the other hand, exhibit different behavior. By taking the natural logarithm of the difference between consecutive stock price values in Equation 2.37, the deterministic trend component is effectively canceled. This can be examined by rewriting the equation using the logarithm's property:

$$r_t = \ln(S_t) - \ln(S_{t-1}) \tag{2.40}$$

The stock price model has a deterministic trend component (the price itself) and a random shock component (the Brownian motion). The Equation 2.35 can be rewritten in simple terms to analyze its behavior more closely, as shown in Equation 2.41.

$$S_t = f_t + \epsilon_t \tag{2.41}$$

Where f_t represents the deterministic terms and ϵ_t represents the random term. Hence:

$$\ln(S_t) = \ln(f_t + \epsilon_t) \tag{2.42}$$

If it is assumed that f_t grows linearly, its algorithm $\ln(f_t)$ also grows linearly. Therefore, when subtracting successive stock prices in Equation 2.40, the linear deterministic trend components will cancel each other out, leaving only a random shock ϵ_t. To classify the logarithmic returns time series, the definition of stationarity for the stochastic process must be introduced.

Definition 1 (Strict stationarity) *A stochastic process is strictly stationary if its joint distribution function $F(X_t, X_{t+1}, \ldots, X_{t+k})$ does not depend on t.*

Definition 2 (Weak stationarity) *A stochastic process is weakly stationary if its first moment $E[X_t]$ and second moment $Var[X_t]$ does not depend on t.*

Now that only the random shock component remains in the stock price model, it can be concluded from the definition of the random walk that the first and second moments are as follows:

$$E[r_t] = 0 \tag{2.43}$$

$$Var[r_t] = \sigma_t^2 \tag{2.44}$$

Therefore, the first two moments are constant and do not depend on the time step in the time series, making it weakly stationary according to Theorem 2. However, under the Gaussian distribution, the weak stationarity is equivalent to strict stationarity. Hence, if the next values in the logarithmic returns time series are to be predicted using the conditional expectation, there will be a non-trivial solution. Given the information available at time t, the conditional expectation of the logarithmic return at time $t+1$ is 0, meaning that knowing the current return value does not provide any information about the future return, implying randomness in the returns.

$$E[r_{t+1}|\mathcal{F}_t] = 0 \tag{2.45}$$

This is why working with stationary time series and trying to predict their future expected values might prove complicated for deep learning models and is worth investigating. Furthermore, Equation **??** provides a theoretical expectation for the empirical results of this work. If the expected value of predicted logarithmic returns is 0, then a model has a 50% chance of being correct, just like any random guess. Hence, a model that has greater accuracy than this threshold, meaning it can predict logarithmic return more often than not, should generate trading signals that result in a profitable trading strategy.

2.3.3 Augmented Dickey-Fuller Test and Autocorrelation

In the previous section, it was mathematically proven that, in theory, the logarithmic time series should be strictly stationary. However, as real-world financial data is used in this study, it is crucial also to check this property empirically to ensure that the theoretical assumptions are reflected and valid in the dataset. To do so, the Augmented Dickey-Fuller (ADF) test is employed.

The ADF test is a statistical test used to determine the presence of a unit root in a time series, which indicates non-stationarity. It involves hypothesis testing and assessing the statistical significance of the results. The ADF test is based on regressing the first difference of the time series Δy_t on the lagged values of the time series.

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \sigma_1 \Delta y_{t-1} + \sigma_2 \Delta y_{t-2} + \cdots + \sigma_p \Delta y_{t-p} + \epsilon_t \tag{2.46}$$

In Equation 2.46, α is a constant intercept term, β is the coefficient of the time trend, γ is the coefficient of the lagged level of the time series, $\sigma_1, \sigma_2, \ldots, \sigma_t$ are the coefficients associated with the lagged differences of the time series, and ϵ_t is the white noise. Therefore, y_{t-1} is the lag 1 of the time series and Δy_{t-1} is the first difference of the time series. To assess whether the results of the test are statistically significant, hypothesis testing is used, where the decision to reject or fail to reject the null hypothesis is based on the calculated test statistic and its associated p-value.

The null hypothesis for the ADF test is that there is no stationarity and $\gamma = 1$, implying the presence of a unit root. When a unit root exists, the time series exhibits characteristics such as trends or random walks without a drift, where the series can deviate indefinitely from its mean value. This behavior leads to non-stationarity, making it difficult to model and analyze the time series using traditional statistical methods. The alternative hypothesis corresponds to $\gamma < 0$, implying mean reversion in the time series, a characteristic of stationary processes. When $\gamma < 0$, it implies that the effect of a deviation from the long-run mean of the time series tends to decrease over time, pulling the series back toward its mean value. This behavior is consistent with stationary processes, where fluctuations around the mean tend to be temporary, and the series returns to its long-term average over time. The ADF test statistic is calculated as the t-statistic associated with the coefficient γ divided by its standard error.

$$ADF = \frac{\gamma}{SE(\gamma)} \tag{2.47}$$

Where SE stands for the standard error. The significance of the test is assessed using critical values from the Dickey-Fuller distribution, which depend on the sample size and the specification of the regression equation. If the test statistic is less than the critical values, the null hypothesis is rejected, meaning no unit root is present, and the time series is non-stationary.

Another statistical tool used to check the stationarity properties of the time series data is the Autocorrelation Function (ACF), which measures the autocorrelation between observations in the time series as a function of the time lag between them. The presence of autocorrelation between lags can indicate the presence of serial autocorrelation in the data, which may suggest non-stationarity. The ACF is defined as:

$$ACF(k) = \frac{\sum_{t=k+1}^{T}(y_t - \overline{y})(y_{t-k} - \overline{y})}{\sum_{t=1}^{T}(y_t - \overline{y})^2} \tag{2.48}$$

Here, k is the lag, T is the total number of observations, y_t is the observation at time t, and $\overline{y}$ is the mean value observed in the time series. The dividend here is the covariance between the observations, and the divisor is the variance.

2.4 Baseline Models

This section introduces a theory behind the two models used as baseline models in the experiment - logistic regression and ARIMA. A baseline model serves as a straightforward and relatively simple benchmark against which more sophisticated models can be compared, providing a basic reference point for evaluating the performance of more complex models. Their purpose is to establish a minimum level of accuracy that the investigated models should surpass. If the LSTM model cannot outperform the baseline model, it could indicate that the LSTM models are not a suitable choice for making predictions on stationary financial time series.

2.4.1 Logistic Regression

Logistic regression is a statistical model used for binary classification problems and is often a popular choice for a baseline method. The fundamental concept behind logistic regression is the sigmoid function, often referred to as the logistic function. The logistic function maps any real-valued number to the range of $[0, 1]$, making it suitable for predicting the probabilities of data points belonging to one of the binary classes. This is the main advantage of logistic regression over linear regression, where the latter can only predict 0 or 1, even for examples that are very close to the decision boundary. Logistic regression provides more refined predictions as it softens the threshold function by approximating the hard threshold with its continuous function.

To solve a binary classification problem, logistic regression uses a vector of weights w and a bias term b. Each weight w_i corresponds to a single input feature x_i and describes the importance of the input feature in the classification process. The weight can be positive or negative, depending on whether an input feature results in the positive or negative class. The bias term is the vertical y-axis intercept. After training the regression and finding weights, the classifier makes a classification by summing up and multiplying each feature by its corresponding weight and adding a bias term.

$$x = (\sum_{i=1}^{n} w_i x_i) + b \tag{2.49}$$

Equation 2.49 can be rewritten as the dot product of the features vector x and weights vector w.

$$x = w \cdot x + b \tag{2.50}$$

To transform the input for the logistic function into the log-odds scale, the logit function, which is the inverse of the $\sigma(x)$ function, is used to represent the linear relationship between the input features and the probability.

$$logit(p(x)) = \ln(\frac{p(x)}{1 - p(x)}) = w \cdot x + b \tag{2.51}$$

Equation 2.51 is combined with the logistic function $\sigma(x)$ to obtain a probability.

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{1}{1 + e^{-(w \cdot x + b)}} \tag{2.52}$$

Therefore, the probabilities of x belonging to class y are:

$$P(y = 1) = \sigma(w \cdot x + b) \tag{2.53}$$

$$P(y = 0) = 1 - \sigma(w \cdot x + b) \tag{2.54}$$

To train the regression and obtain the x and w parameters, a loss function that measures the distance between the actual target value y and predicted value $\hat{y}$ is used. For this purpose, the logistic loss, also known as cross-entropy loss for multi-class classification problems, is employed. To calculate this distance, conditional maximum likelihood estimation is used, which involves choosing the parameters x and w that maximize the log probability of correct prediction. The log-loss function for a single i-th point is:

$$\mathcal{L}^t = \begin{cases} -\log(p_i), & \text{if } y_i = 1 \\ -\log(1-p_i), & \text{if } y_i = 0 \end{cases} \tag{2.55}$$

This can be rewritten as a single expression, which is exactly the same loss function used for the LSTM binary classification model in Equation 2.20:

$$\mathcal{L}^t(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{2.56}$$

Where $\hat{y} = \sigma(w \cdot x + b)$. Hence:

$$\mathcal{L}^t(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1-y)\log(1 - \sigma(w \cdot x + b))] \tag{2.57}$$

The loss function of weights and bias that will be minimized with gradient descent is:

$$\mathcal{L}(w, b) = -\frac{1}{T}\sum_{t=1}^{T} \mathcal{L}^t(\hat{y}^t,\ y^t) \tag{2.58}$$

Therefore, the gradient descent update for predicting the values of w and b that minimize the loss function is:

$$W_{new} = W_{old} - \alpha \frac{\partial \mathcal{L}}{\partial W} \tag{2.59}$$

$$B_{new} = B_{old} - \alpha \frac{\partial \mathcal{L}}{\partial B} \tag{2.60}$$

Elastic Net is employed to regularize the logistic regression. The Elastic Net technique addresses the limitations of Lasso (L1 norm) and Ridge regularization (L2 norm) methods by combining them to achieve a balance between feature selection and coefficient shrinkage. With Elastic Net, the objective function 2.57 becomes:

$$\mathcal{L}^t(\hat{y}, y) = -[(y_t \log(\sigma(w^T x_t)) + (1-y_t)\log(1-\sigma(w^T x_t))) + \lambda 1 ||w||_1 + \lambda 2 ||w||_2^2] \tag{2.61}$$

Where $||w||_1$ is the L1 norm (the sum of absolute values) of the weight vector, $||w||_2^2$ is the squared L2 norm of the weight vector, and $\lambda 1$ and $\lambda 2$ are the regularization parameters controlling the strength of L1 and L2 penalties, respectively. The Elastic Net regularization term, $\lambda 1||w||_1 + \lambda 2||w||_2^2$, encourages both sparsity (through L1) and small coefficient values (through L2).

2.4.2 ARIMA

The Autoregressive Integrated Moving Average (ARIMA) model is used for time series forecasting and combines the autoregressive (AR) and moving average (MA) components with the integration (I) component. The theory behind each component is provided, but the practical part of this project only uses the AR component of the ARIMA model. An important feature of this model is that it requires the time series to which it is applied to be stationary.

The $AR(p)$ component takes a value p, which refers to how far back in time the model looks to predict a new value. It is responsible for predicting the future value of the stationary time series based on previous values by capturing the linear dependence of the correct observations on its past values to capture trends in the time series. The model with p lags, meaning p previous values are considered when computing the next value in the time series, is as follows:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-1} + \cdots + \phi_p y_{t-p} + \epsilon_t \tag{2.62}$$

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t \tag{2.63}$$

In Equations 2.62 and 2.63, y_t is the next value being predicted, y_{t-1} is the previous value from the last time step, ϕ_1 is the autoregressive coefficient estimated by the model during the training process, c is a constant term representing the intercept or baseline level of the time series (the average value of the series not explained by the model itself), and ϵ_t is a white noise term representing the random component of the time series, implying that errors in the time series are independent identically distributed random variables with $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$.

The $MA(q)$ component is responsible for capturing long-term dependencies and fluctuations in the time series. It is a linear regression between the current observation and the past white noise error terms. The model takes an order term q, which indicates how many of the most recent white noise errors are considered in the linear combination for prediction at time t. Each of these error terms is multiplied by the moving average coefficient θ, which the model estimates during the training process to capture short-term shocks in the data.

$$y_t = c + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} + \epsilon_t \tag{2.64}$$

$$y_t = c + \sum_{i=1}^{p} \theta_i \epsilon_{t-i} + \epsilon_t \tag{2.65}$$

In the MA component, previous values of the y target variable are not used; instead, only white noise error terms ϵ are employed. ϵ_t is the noise error at the current time step, $\epsilon_{t-1}, \epsilon_{t-2}, \ldots, \epsilon_{t-q}$ are past white noise terms used for estimating the current time series value, and $\theta_1, \theta_2, \ldots, \theta_q$ are the moving average coefficients. The $ARMA(p, q)$ model combines the two models above.

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-1} + \cdots + \phi_p y_{t-p} + \theta_1 \epsilon_{t-1} + \cdots + \theta_2 \epsilon_{t-2} + \theta_q \epsilon_{t-q} + \epsilon_t \tag{2.66}$$

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{p} \theta_i \epsilon_{t-i} + \epsilon_t \tag{2.67}$$

The $ARIMA(p, d, q)$ model is an extended $ARMA(p, q)$ model that introduces the integration component with the new variable d. As the ARMA model works only on stationary time series, the ARIMA model ensures the series is stationary by differencing, taking the d-th difference of the time series until the time series becomes stationary. In this project, the studied time series is already stationary, which is checked using the ADF test before applying the ARIMA model. Therefore, the integration component is not of much use in this case. As discussed later in Chapter 3, only the $AR(p)$ component from the $ARIMA(p, d, q)$ model is used, making the model in question a simple autoregressive model. The p parameter is estimated by looking at the lag cutoff in the PACF (Partial Autocorrelation Function). [5]

2.5 Metrics

2.5.1 Accuracy, Precision, and Recall

When measuring the performance of algorithms that perform binary classification on a dataset, three of the most common metrics are accuracy, precision, and recall. These metrics help

[5] See ACF and PACF in time series.

better understand the models' effectiveness in making binary predictions and serve as a basis for comparing the final results.

The first step in calculating these metrics is to create a confusion matrix that categorizes the predictions into four distinct categories based on the relationship between the actual and predicted values: true positives (the number of correct predictions for the positive class), true negatives (the number of correct predictions for the negative class), false positives (the number of incorrect predictions for the positive class), and false negatives (the number of incorrect predictions for the negative class).

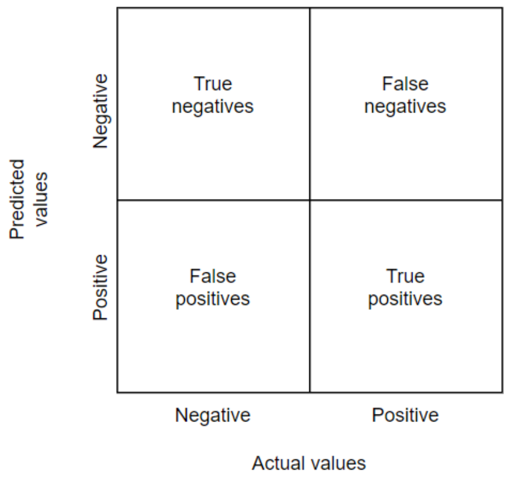


Figure 2.5: Confusion matrix.

The most important metric in this study is accuracy, which represents the proportion of correct predictions and is positively correlated with higher performance. Accuracy is calculated by dividing the sum of true positives and true negatives by the total number of predictions, usually equal to the number of data points in the test set.

$$accuracy = \frac{true\ positives + true\ negatives}{total} \tag{2.68}$$

The other two metrics, precision, and recall, are worth mentioning for a holistic approach but will not be considered in the final scoring metrics of the models, as they focus on positive predicted values and are only useful when both classes bear distinct meanings. In this project, the binary class division categorizes returns into positive or negative, and from the perspective of the trader, there is no difference between them, making both classes equally important. Precision is a measure of the positive predicted value, i.e., how many positive predictions are correct, and is calculated by dividing the true positive value by the sum of true positives and false positives.

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{2.69}$$

Recall, on the other hand, deals with sensitivity or the true positive rate. It indicates how many true positive examples were classified correctly and is calculated by dividing the number of true positives by the sum of true positives and false negatives.

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{2.70}$$

The F_1 score, the harmonic mean of precision and recall, is often popular in the literature. However, due to the aforementioned reason, it will not be the focus of this study, as precision and recall are not ideal metrics when both positive and negative classes are of equal concern.

2.5.2 ROC Curve and AUC Score

Another metric used to assess binary classifiers is the Receiver Operating Characteristic (ROC) curve, which measures the accuracy of binary classifiers with probabilistic output. Instead of predicting a positive or negative class, these classifiers predict the probability of a data point belonging to each class, with a threshold value determining the final classification.

$$class = \begin{cases} positive, & \text{if } p(y_i|x_i, \mathcal{D}) > \text{threshold} \\ negative, & \text{otherwise} \end{cases} \tag{2.71}$$

The ROC curve is obtained by plotting the true positive and false positive pairs at varying threshold values, plotting the true positive rate against the false positive rate. The ROC curve can be interpreted as a cumulative distribution function of the detection probability, and its Area Under the Curve (AUC) value indicates the probability that the classifier will correctly classify a randomly chosen data point. The dotted line represents the performance of a naive classifier, which, on average, predicts exactly half of the data points correctly. This can be thought of as a benchmark, as it represents a 50% chance that the trader always has to be right when placing a trade. Ideally, for a classifier with good performance, the curve should climb steeply near the y-axis so that the AUC value is close to 1.

2.5.3 Root Mean Squared Error

The primary metric for evaluating the performance of regression models is the Root Mean Squared Error (RMSE). This performance indicator measures the average difference between the values predicted by the regression and the actual values, providing an estimation of the model's accuracy for predicting the target value. RMSE is the square root of the error variance and thus can be interpreted as the standard deviation of the error. The error variance, or Mean Squared Error (MSE), is the average squared difference between predicted and actual values and is always positively strict. The final formula for RMSE is given by the square root of the sum of the squared differences between predicted and actual values divided by the number of all observations in the sample.

$$RMSE = \sqrt{\sum_{i=1}^{N} \frac{(\hat{y}_i - y_i)^2}{N}} \tag{2.72}$$

Chapter 3

Design and Implementation

3.1 Tools, libraries, and frameworks

In this research, Python 3.9 is employed as the programming language for conducting the experiment. The JupyterLab integrated development environment (IDE) is chosen for code development, given its acknowledged efficacy in tasks associated with machine learning and data analysis. JupyterLab notebooks possess a distinctive feature wherein cells can be executed individually, which proves advantageous for avoiding the repetition of resource-intensive computations, such as model training, each time an entire experiment is executed.

The data utilized in this experiment is sourced from the Yahoo! Finance library version 0.2.31. For the creation of the LSTM model, the TensorFlow 2.13.0 framework is employed, incorporating the Keras 2.13.1 library, which offers a high-level API for constructing neural networks. Additionally, the SciKeras 0.12.0 library is utilized for the Keras classifier and regressor wrappers, which facilitate grid search and time series cross-validation during model training. The statsmodels 0.14.0 library is used for implementing the ARIMA model and ADF test, while the Scikit-learn 1.2.1 library is harnessed for the logistic regression model, metrics computation, and data standardization.

Furthermore, standard libraries such as Pandas 2.0.3, NumPy 1.23.5, Matplotlib 3.7.1, and Seaborn 0.12.2 are imported for data analysis, manipulation, and visualization purposes. To ensure reproducibility, the seed from the `numpy.random` module is used. Additionally, modules such as `sys` and `os` are utilized for writing and saving results to file, the `time` module monitors execution time, and the `datetime` module is employed to manipulate the time step index column in Pandas DataFrames.

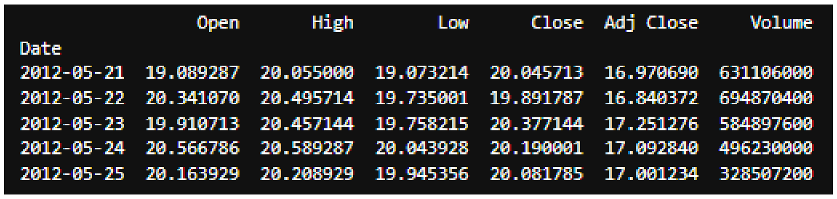
3.2 Dataset

For this project, public financial data from the Yahoo! Finance media provider, accessible via the `yfinance` library, is used. The analysis of the model's performance is carried out with respect to a limited number of stocks within a certain time frame. The 10 companies from the S&P 500 index whose stocks are used for training and testing the models were selected based on the company's stock weight in the SPDR S&P 500 Trust ETF (SPY), which is the oldest exchange-traded fund that tracks the S&P 500 index. The 10 largest stocks and their respective tickers, by weight, are Apple (AAPL), Microsoft (MSFT), Amazon (AMZN), Nvidia (NVDA), Alphabet Class A (GOOGL), Tesla (TSLA), Alphabet Class C (GOOG), Berkshire Hathaway (BRK.B), Meta (META), and UnitedHealth Group (UNH). The entire dataset spans a range of 2920 days, starting on 22.05.2012 and ending on 28.12.2023.

Data for each stock is obtained as a separate Pandas DataFrame using the `get_data` function, shown in Listing 3.1. Each returned DataFrame is indexed by a `Datatime` column, where each row represents a single trading day. The columns `Open`, `High`, `Low`, `Close`, `Adj Close` and `Volume` hold information about the stock's opening price, the highest price of the day, lowest price of the day, closing price, adjusted closing price after adjustments for applicable splits and dividend distributions, and volume traded that day, respectively. Figure 3.1 presents raw data obtained from `yfinance` for the AAPL ticker.

Listing 3.1: Function for getting the stock data.

```python
def get_data(ticker: str, start_date: datetime.datetime, end_date: datetime.datetime)
    -> pd.DataFrame:
    data_df = yf.download(ticker, start=start_date, end=end_date, progress=False)
    return data_df
```



```
                 Open       High        Low      Close  Adj Close     Volume
Date
2012-05-21  19.089287  20.055000  19.073214  20.045713  16.970690  631106000
2012-05-22  20.341070  20.495714  19.735001  19.891787  16.840372  694870400
2012-05-23  19.910713  20.457144  19.758215  20.377144  17.251276  584897600
2012-05-24  20.566786  20.589287  20.043928  20.190001  17.092840  496230000
2012-05-25  20.163929  20.208929  19.945356  20.081785  17.001234  328507200
```

Figure 3.1: Raw data for AAPL.

3.3 Feature Engineering

To analyze and use this data for the experiment, feature engineering must be performed, which involves taking the current DataFrames obtained from downloaded data and transforming them into useful data. More precisely, the input data and target data for binary classification and regression must be calculated. First, let's look at the input data, as it is the same for both model categories. The logarithmic return at time t is used to make predictions about the future time *t+1*. To calculate logarithmic returns, the `Adj Close` column is used, as it is the suitable closing price for any stock, regardless of whether it pays dividends. For future reference and clear naming, its values are copied into a new column named `close_price_t`. Logarithmic returns are calculated instead of simple returns because they are easier to sum up over time when testing the trading strategies.

$$\text{logarithmic return}_t = \ln \frac{\text{close price}_{t+1}}{\text{close price}_t} \tag{3.1}$$

Listing 3.2 shows how logarithmic returns are calculated using `close_price_t`. The implementation uses the NumPy `log` function and takes the difference between consecutive data points using the `diff()` function.

Listing 3.2: Calculate logarithmic returns.

```
data['log_return_t'] = np.log(data['close_price_t']).diff()
```

Next, the target values, relative return classes for binary classification, and stock logarithmic returns for a regression model must be calculated. To obtain target data for regression models, stock logarithmic returns for day $t+1$ are computed so that predictions can be made on day t using the logarithmic returns data from the past x days, up to and including day t. Therefore, the `log_return_t` column is shifted by 1 index upward using `shift(-1)`, and the shifted returns are stored in the `log_return_t+1` column.

Listing 3.3: Calculate logarithmic returns for the next day.

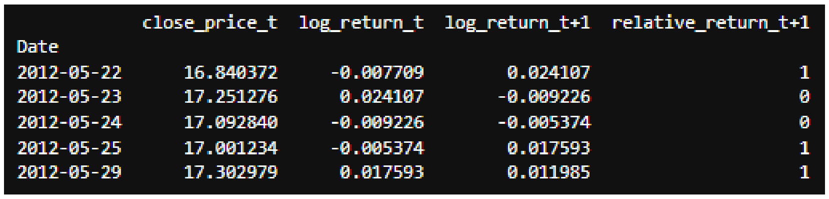
```
data['log_return_t+1'] = data['log_return_t'].shift(-1)
```

Obtaining target data for binary classification is simple; the values from the `log_return_t+1` column are mapped to class 1 if the logarithmic return for a given day is positive and class 0 if it is negative.

Listing 3.4: Calculate relative returns classes.

```
data['relative_return_t+1'] = np.where(data['log_return_t+1'] > 0, 1, 0)
```

After deleting the non-relevant columns using the NumPy `drop` function and dropping rows that contain NaN values with `dropna(axis=0)`, where axis 0 corresponds to the horizontal row axis, the final data, presented in Figure 3.2, is obtained for training and testing the models. This is the data after all operations for the AAPL ticker, which is used in the following sections to demonstrate the process of data analysis and model training.



```
            close_price_t  log_return_t  log_return_t+1  relative_return_t+1
Date
2012-05-22      16.840372     -0.007709        0.024107                    1
2012-05-23      17.251276      0.024107       -0.009226                    0
2012-05-24      17.092840     -0.009226       -0.005374                    0
2012-05-25      17.001234     -0.005374        0.017593                    1
2012-05-29      17.302979      0.017593        0.011985                    1
```

Figure 3.2: Data for AAPL after feature engineering.

3.4 Data Analysis

Due to limited computational resources and the desire to provide a general picture of the process, only the data belonging to the AAPL stock is analyzed in this section and Section 3.7. Therefore, the figures here represent the data for the AAPL stock.

Once the data is preprocessed, the next step is to analyze it. First, it is important to check if the data is balanced, as this might impact how the models learn data patterns later on. The total number of trading days in the dataset during the experiment time period is 2920 days for all stocks, with 90% (2628 days) used for training and the remaining 10% (292 days) used for testing the models and backtesting the strategies The exact process of splitting the data is described in Section 3.5. For the AAPL stock, the data is not perfectly balanced. The training data has 52.4% days with positive returns and 47.6% days with negative returns. The testing data has a similar imbalance, with 53.8% days with positive returns and 46.2% with negative returns. Therefore, the majority class is the class with positive returns. Even though this imbalance is small, it can greatly impact the results.

Figure 3.3 shows the stock logarithmic returns plotted against the stock close price using `matplotlib.pyplot` to visualize the relationship between the stationary and non-stationary time series data. The non-stationary time series of the stock price follows the random walk process described in the Subsection 2.3.1, while the stationary time series of stock returns have much greater volatility and randomness.

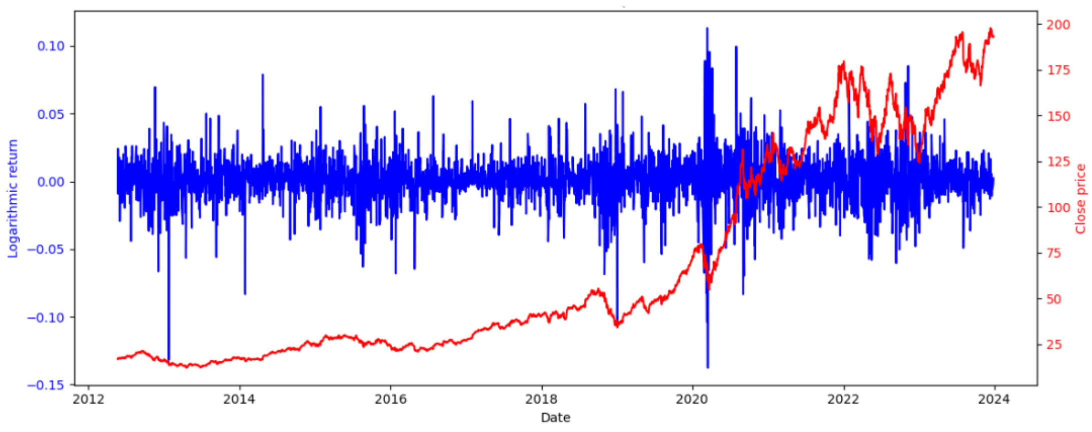


Figure 3.3: Logarithmic returns vs. closing price for AAPL.

Figure 3.4 represents the features correlation matrix created using the Pandas `corr()` function, which calculates the correlation coefficient r using Pearson's Correlation.

$$r = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2 \sum(y_i - \overline{y})^2}} \tag{3.2}$$

Where x_i and y_i are individual data points, and $\overline{x}$ and $\overline{y}$ are the means of x and y, respectively. The resulting correlation coefficient r indicates the following:

$$r \begin{cases} > 0, & \text{if there is a positive relationship} \\ < 0, & \text{if there is a negative relationship} \\ = 0, & \text{if there is no linear relationship} \end{cases} \tag{3.3}$$

The Seaborn `heatmap()` function is used to plot the correlation matrix, with warmer colors representing positive relationships and colder colors indicating negative relationships between features in the dataset.

Time series `log_return_t+1` and `relative_return_t+1` have the only strong positive correlation of 0.7. Any other values are too small to be meaningful; therefore, the matrix shows that there is almost no correlation between the stock price and stock logarithmic returns and no correlation between the subsequent values of stock logarithmic returns, as expected for a stationary time series.

The last step is to perform the ADF test and plot the ACF to check the stationarity properties of the logarithmic return and stock price time series using the `statsmodel` library's `adfuller` and `plot_acf` functions. Listing 3.5 shows ADF test implementation.

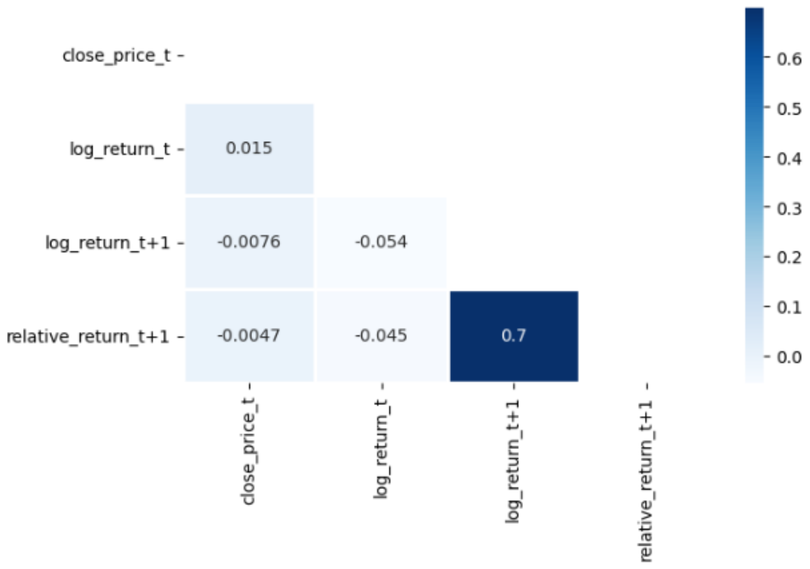


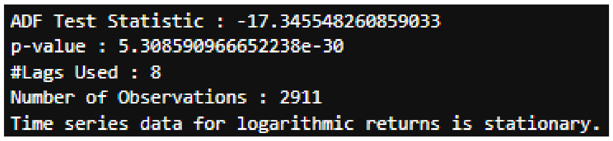
Figure 3.4: Features correlation matrix for AAPL.

Listing 3.5: Function for conducting ADF test.

```
def stationary_test(data: pd.DataFrame, name: str) -> None:
    results = adfuller(data)
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observations']
    for value, label in zip(results ,labels):
        print(label + ' : ' + str(value))
    alpha = 0.05
    if results[1] <= 0.05:
        print('Time series data for ' + name + ' is stationary.')
    else:
        print('Time series data for ' + name + ' is non-stationary')
```

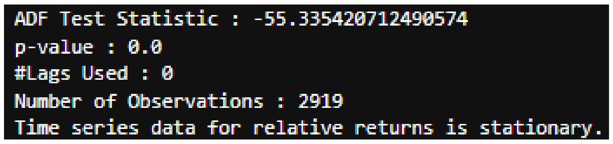
While the ADF test is performed using the library function, hypothesis testing is implemented manually. An alpha of 0.05 is selected as the significance level. Following the theory described in Subsection 2.3.3, if the p-value obtained from the ADF test is less than or equal to the alpha value, the null hypothesis is rejected, and the alternative hypothesis that the data is stationary is accepted. Otherwise, if the p-value is greater than the alpha value, the null hypothesis cannot be rejected, meaning the data is non-stationary. Applying this to the stock data yields the results shown in Figures 3.5, 3.6, and 3.7. The p-value for relative returns is extremely small; hence test displays 0.0.

The empirical experiment's results align well with the expectations based on theory. The



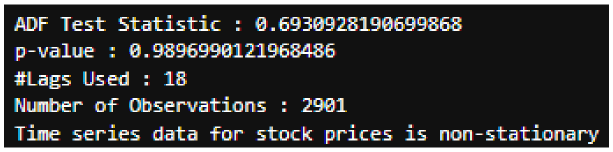
```
ADF Test Statistic : -17.345548260859033
p-value : 5.308590966652238e-30
#Lags Used : 8
Number of Observations : 2911
Time series data for logarithmic returns is stationary.
```

Figure 3.5: ADF test for AAPL's logarithmic returns.



```
ADF Test Statistic : -55.335420712490574
p-value : 0.0
#Lags Used : 0
Number of Observations : 2919
Time series data for relative returns is stationary.
```

Figure 3.6: ADF test for AAPL's relative returns.



```
ADF Test Statistic : 0.6930928190699868
p-value : 0.9896990121968486
#Lags Used : 18
Number of Observations : 2901
Time series data for stock prices is non-stationary
```

Figure 3.7: ADF test for AAPL's price.

stock price time series is non-stationary, making it easy to predict. Meanwhile, the logarithmic and relative returns (mapped logarithmic returns and should have the same test result) are stationary. Figures 3.8, 3.9, and 3.10 show the ACF plots.

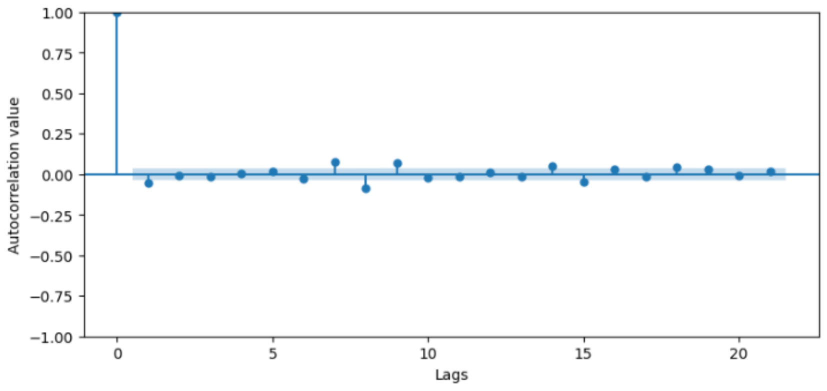


Figure 3.8: ACF plot of AAPL's logarithmic returns.

This visualization confirms the previous results from the ADF test. Generally, a decay in autocorrelation as the lag increases suggests the stationarity of the data, while a constant level of autocorrelation suggests non-stationarity. After the sudden drop in value to almost 0 for the logarithmic and relative returns, the plots show no autocorrelation between subsequent lags. In fact, the value drops right after the first lag, which is the autocorrelation of the current time step to itself and can be completely disregarded. For the non-stationary data, Figure 3.10 shows that autocorrelation is preserved, with negligible decay occurring after 10 lags.

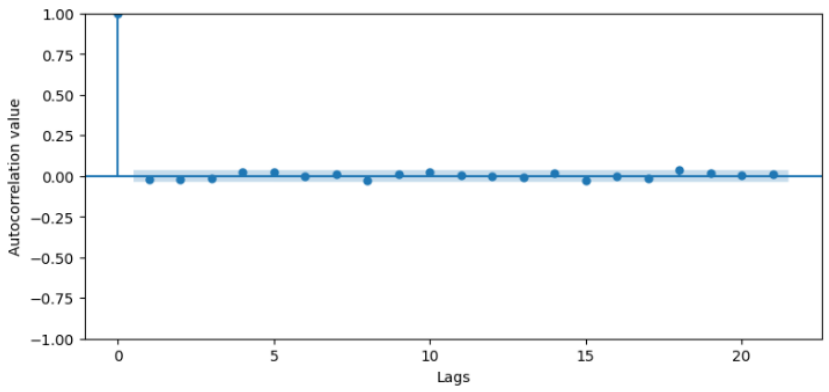


Figure 3.9: ACF plot of AAPL's relative returns.

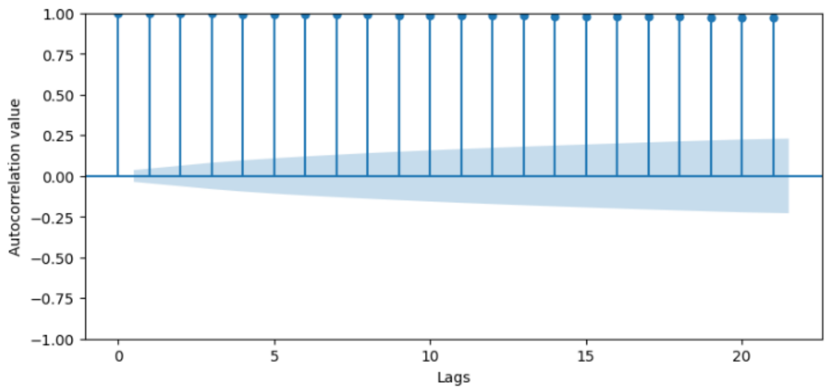


Figure 3.10: ACF plot of AAPL's price.

The last step is to plot the PACF of logarithmic returns using the `plot_pacf` to determine the autoregressive parameter used in the ARIMA model. The difference between ACF and PACF is the inclusion or exclusion of indirect correlations in the calculation.

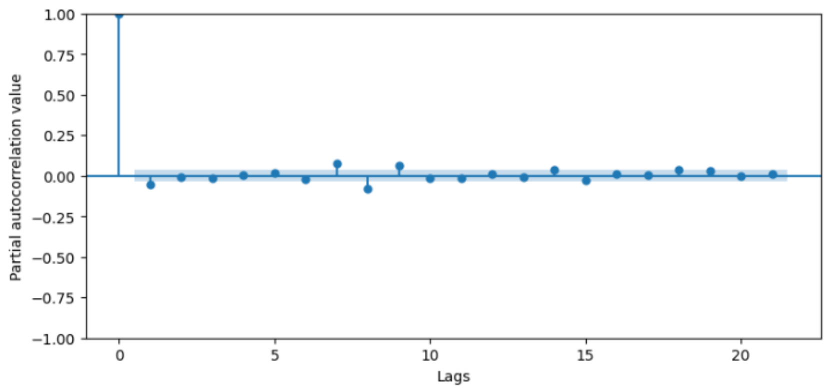


Figure 3.11: PACF plot of AAPL's logarithmic returns.

As seen in Figure 3.11, the lag cut-off happens at lag-1, so the parameter in the ARIMA model will have a value of 1. The PACF plot looks similar to the ACF plot, indicating that the stationary time series of logarithmic returns have very weak autocorrelation that is almost insignificant. This poses a risk that the ARIMA model built on this data could yield poor results, as the autoregressive component may not find relevant autocorrelations to form good predictions.

3.5 Train and Test Datasets with Sliding Window

Before applying the models to the data and generating predictions, it is crucial to divide the data into train and test datasets. While some discussed papers use the `sklearn` library's `train_test_split` function, this project implements the sliding window approach used by Duemig [2019] and Murthy et al. [2022]. Therefore, a custom function is implemented to divide and prepare the data. This function takes input data X, target data Y, and a lookback value for the sliding window as inputs and returns 4 NumPy arrays: input train data, target train data, input test data, and target test data. First, a new Pandas DataFrame is created with two columns for the input and target data passed to the function. The split is done by calculating the index that divides the columns into a 1:9 ratio, with 90% of the data used for training and the remaining 10% for testing. The `iloc` function is used to slice the columns, with each column being a Series object.

Listing 3.6: Split data into train and test datasets.

```
data = pd.DataFrame().assign(input_X=input_data, target_y=target_data)
train_test_ratio = int(len(data)*0.9)
train_set = data.iloc[: train_test_ratio, :]
test_set = data.iloc[train_test_ratio :, :
```

At this point, dividing the data into an additional developer dataset for cross-validation is common. However, in this project, cross-validation is part of the hyperparameter optimization and will be discussed in detail in Section 3.7. After the split, the data is standardized; the detailed description of this process is provided in Section 3.6.

The sliding window mechanism is implemented to control the lookback value when making predictions on the time series. The lookback value represents the number of time steps (in this case, the number of days) from which the model will use the data to make a prediction. It can be thought of as a single input composed of the values of the T previous time steps, which should not be confused with the batch size during training. In simple terms, the lookback value will be the number of previous trading days, up to and including day t, used to make predictions about the values for day $t + 1$. The Listing 3.7 represents the sliding window implementation.

Listing 3.7: Sliding window mechanism.

```python
T = lookback
X_train = []
y_train = []
X_test = []
y_test= []
for i in range(T, len(train_set) - T):
    X_train.append(standarized_train_X_set[i-T : i, :])
    y_train.append(train_set.iloc[i, 1])
for i in range(T, len(test_set)):
    X_test.append(standarized_test_X_set[i-T : i, :])
    y_test.append(test_set.iloc[i, 1])
```

The implementation iterates through the standardized input dataset (both train and test) using a for loop and changes the data dimensions from $2D$ to $3D$. The resulting 3 dimensions are rows (equal to data length), columns (equal to the lookback value representing all days used to make predictions in the corresponding row of the target data), and a final dimension (in this case, always equal to 1). The row index range from $i - T$ to i corresponds to the last T days, up to and including the current day i. It is important to note that the iteration starts from the index equal to the lookback value, as this is the first day in the data preceded by the T number of previous days that can be used for the first predictions. Otherwise, the first T predictions would not use the proper size of the sliding window. Additionally, the dimension of the target data is not changed; the target data columns are simply reassigned from the train and test DataFrames to new arrays.

3.6 Standardization

After splitting the data into train and test datasets, standardization is performed. Standardization is a crucial preprocessing technique that enhances the models' performance by ensuring the input features have a consistent scale. Data standardization involves transforming the features to have a mean of 0 and a standard deviation of 1, using Z-score shown in Equation 3.4.

$$x_{standarized} = \frac{x - \mu}{\sigma} \tag{3.4}$$

Where x is a feature value, μ is the feature mean, and σ is its standard deviation. Another commonly used preprocessing method is normalization, which scales the features to a specific range, often between 0 and 1, using Min-Max scaling.

$$x_{normalized} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3.5}$$

Standardization is more suitable for the data in this project, as it follows a Gaussian distribution, as discussed in Subsection 2.3.1. Moreover, standardization is commonly used with LSTM models, as neural networks work better when the data is close to 0. The `StandardScaler()` function from the `sklearn.preprocessing` performs Z-score standardization on the logarithmic returns.

Listing 3.8: Data standarization process.

```
scaler = StandardScaler()
scaler = scaler.fit(train_X_set)
standarized_train_X_set = scaler.transform(train_X_set)
standarized_test_X_set = scaler.transform(test_X_set)
```

A `StandardScaler` is initialized in the `scaler` variable and fitted to the train input data. The distribution from the fitted train input dataset is then used to transform both the train and test input datasets, as they follow the same Gaussian distribution. Only the input data is standardized for training purposes; the target data is not transformed. Standardizing only the input data ensures that the model learns from consistent feature representations, thereby improving convergence during training. Meanwhile, retaining the original scale of the target variable allows for straightforward interpretation of model predictions in the context of the original problem domain. Many different techniques and combinations were tested throughout the implementation, and the best results were achieved when only the input data was standardized. Standardizing the target data for regression and then re-scaling it back after training did not yield satisfactory results. It is also claimed that linear transformation for the target variables is not necessary for regression tasks. Standardizing the target data for binary classification, either 0 or 1, simply does not make sense, as these values are arbitrary.

3.7 Hyperparameter Optimization

Before conducting the final experiment using the LSTM network, binary classification and regression LSTM models must be created, and hyperparameter optimization must be performed. This section focuses on the optimization process and obtaining the best parameter values for the exemplary AAPL stock data without delving into the details of the underlying LSTM models, which are discussed in Section 3.9. Due to limited computational resources, the optimization process is performed only for a single stock, assuming that the parameter values obtained for the AAPL binary classification and regression LSTM models are representative of the best possible values for all other stocks. This necessary simplification is further discussed in Chapter 6.

The first step is to wrap the Keras estimator model using `KerasClassifier`/`KerasRegressor` from the `scikeras.wrappers` module, which implements the Scikit-Learn classifier/regressor interface for binary classification/regression. The `build_fn` argument passed to the wrapper function is the callable function of the previously defined LSTM model, which the wrapper will construct, compile, and return as a Keras estimator model. Model parameters, such as the number of features used in model training and the lookback value, are also passed.

Once the Keras wrapper for the estimators is created, Grid Search is employed. This hyperparameter optimization technique is used to systematically search through a specified grid of hyperparameters to identify the combination that yields the best model performance. This process involves evaluating the model's performance for each combination of hyperparameters using cross-validation, which is computationally expensive. The specific cross-validation technique used is time series cross-validation. Unlike traditional cross-validation methods, such as k-fold, which randomly shuffles the data before splitting it into training and testing sets, time series cross-validation respects the temporal order of the data. This is crucial in time series analysis, as the predictive performance of a model often depends on its ability to capture temporal patterns and trends. Time series cross-validation, shown in Figure 3.12, uses a walk-forward approach, where a rolling window of a fixed size moves forward through time with each iteration, using a subset of the training dataset to train the model and the validation set, the subsequent period in the training dataset, to evaluate its performance.

In Python, the Grid Search with time series cross-validation is performed using `GridSearchCV` from the `sklearn.model_selection` module. First, the `GridSearchCV` object is instantiated by passing in the wrapped Keras model with a dictionary of grid parameter values. The Listing 3.9 shows the parameters and their values passed as a dictionary to construct a parameter grid. The values selected during optimization are the same for both LSTM models, except for

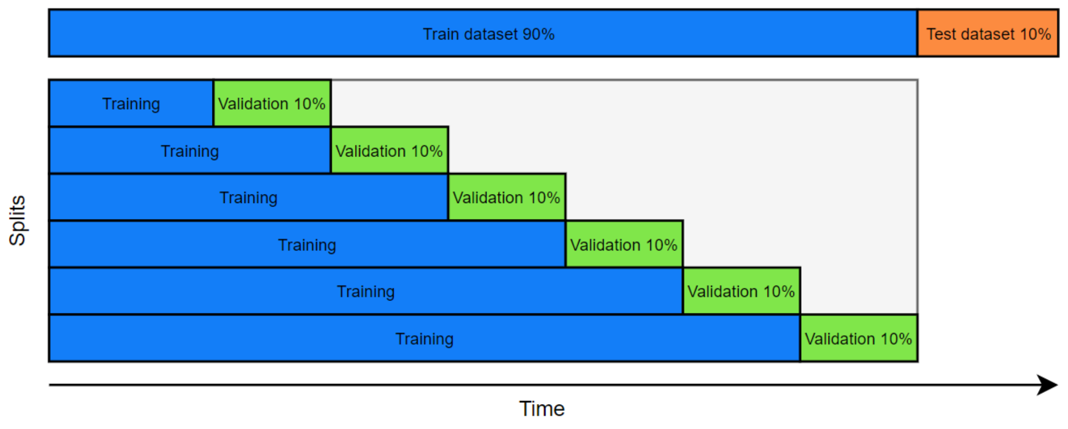


Figure 3.12: Time series cross-validation.

the recurrent dropout parameter and activation function. In the source code in Appendix C, this dictionary is commented out, as the Grid Search implementation tests one hyperparameter value at a time, since due to the limited computational resources, the search for all hyperparameters at the same time is too computationally expensive. Once an optimal value for a hyperparameter is found, it is then used during a search for other hyperparameters.

Listing 3.9: Dictionary with hyperparameter values used in GridSearchCV.

```
hyperparam_values = {'batch_size':[1],
                     'epochs':[1, 10],
                     'model__num_units':[10, 50, 100, 200],
                     'model__learning_rate':[0.01, 0.001, 0.0001, 0.00001],
                     'model__dropout_rate':[0.1, 0.2, 0.3, 0.4, 0.5],
                     'model__recurrent_dropout':[0.1, 0.2, 0.3, 0.4, 0.5],
                     'model__activation_function': ['relu', 'tanh', 'linear',
                         'sigmoid', 'softmax'],
                     'model__kernel_initializer': ['uniform', 'normal', 'he_normal',
                         'he_uniform']}
```

The batch size is the number of data points used in each training iteration. A batch size of 1 ensures that the stochastic gradient descent (SGD) within the ADAM optimizer is used to optimize the model's loss function, and the weights in the SGD are updated based on this single data point. The batch size is different from the lookback, which refers to the number of past time steps used as input for predicting the next time step in the time series. In other words, lookback defines the context the model considers when making predictions. This context is equal to a single batch.

The number of epochs defines the number of times the entire dataset is passed forward and backward through the neural network during the training process. One epoch is completed when every data sample has been used once to update the model's parameters. If the number of epochs is too small, the model may not have sufficient opportunity to learn the underlying patterns in the data, leading to underfitting. Conversely, if the number of epochs is too large, the model may overfit the training data, capturing noise and memorizing specific examples instead of learning generalizable patterns. The numbers of epochs tested are 1 and 10, with early stopping used to determine the optimal number of epochs and prevent overfitting. The most optimal value found by the grid search is 1.

The number of units in the LSTM model refers to the size of the hidden state vectors within each LSTM cell. The values tested are 10, 50, 100 and 200, with 200 units selected during the process.

The learning rate determines the step size in the ADAM optimizer. The values tested are 0.01, 0.001, 0.0001, 0.00001, with the grid search selecting the learning rate of 0.0001.

Recurrent dropout is a regularization technique applied to the recurrent connections between hidden states across time steps in the LSTM layers to prevent overfitting during training. It works by randomly setting a fraction of the hidden state values to 0 during each training iteration, effectively dropping out some of the connections to regularize the network by reducing the reliance on specific hidden state paths and encouraging the network to learn more robust and generalizable representations. The values tested are 0.1, 0.2, 0.3, 0.4, and 0.5, with the grid search selecting a recurrent dropout rate of 0.4 for the binary classification model and 0.2 for the regression model.

The dropout rate is a similar technique used in the dense layer of the model. The values tested are the same as in the recurrent dropout parameter, with 0.2 being the most optimal value.

The activation function parameter was discussed in detail in Subsection 2.2.2, where linear and sigmoid functions were addressed. More functions are supplied to the parameter grid, but their descriptions are omitted, as the Grid Search ultimately selected a sigmoid function for binary classification and a linear function for regression.

Lastly, the kernel initializer parameter defines the distribution used to initialize the weights of the network's kernels in the dense layer. The selected initializer is `he_uniform`, which draws samples from a uniform distribution within $[-limit, limit]$, where:

$$limit = \sqrt{6/in\_units} \tag{3.6}$$

The *in_units* is the number of input units in the weight vector of the LSTM layer cells.[1]

Cross-validation is defined by passing in `TimeSeriesSplit`, which provides the train and validation datasets indices for splitting the time series data, with the number of splits equal to 10. The process is set to run on all available processors by passing `n_jobs=1`, and the Grid Search is asked to retrain the model using the best hyperparameters found during the grid search on the entire dataset by setting `refit=True`. This means that the `best_estimator_` attribute of the `GridSearchCV` object will contain the model retrained on the full dataset. Finally, appropriate scoring is set, which is used to evaluate the model's performance. For binary classification, scoring is set to `accuracy`, which computes the accuracy of predicted labels, and for regression, `neg_root_mean_squared_error` is used, which stands for RMSE score in Scikit-Learn.

The final step is to invoke the `fit()` function on the `GridSearchCV` object, which will train the network iteratively with all sets of hyperparameters. In addition to passing in the train input data and train target data, `shuffle=False` is set, as the data order should be maintained. Callback functions are also passed to optimize the training process further. The first callback function is `EarlyStopping`, imported from `keras.callbacks`, which calculates and monitors the loss metrics (accuracy or RMSE) on the current validation set after each epoch. Once the number of consecutive epochs with no improvement exceeds the specified patience value, set as a default to 0, the callback triggers early stopping to save computational resources. As Grid Search finds an optimal number of epochs to be 1, it means that more epochs do not contribute to further minimization of the loss function, and `EarlyStopping` callback is triggered after checking the second epoch when running a Grid Search with 10 epochs. The second callback is a custom `ResetStates` function that resets the internal states of the LSTM layers after each epoch during training. This is useful for stateful LSTM models, where the internal states of the network do not reset after each iteration by default. The purpose of the `ResetStates` callback is to provide more control over the training process and ensure that the internal states of the recurrent layers are reset at specific intervals (in this case, after each epoch), preventing the accumulation of state information across batches or epochs.

After training is completed, the mean cross-validated score of the best model and the set of parameters that led to it can be accessed with the `best_score_` and `best_params_` attributes.

[1]See Keras 3 API documentation on layer weight initializers.

This full parameter optimization is performed twice: once for the binary classification LSTM model and once for the regression LSTM model for the exemplary AAPL stock. Once a list of parameters that work best for both tasks is obtained, the structure of the Keras wrappers and `GridSearchCV` is used to perform each single experiment for each stock. However, this time, a directory of all parameter values is not passed to the Grid Search, as it would take too much time and resources to perform individual optimization for each stock. Instead, only the list of parameter values that were initially found to work best for the AAPL stock is passed. The only parameter not passed is the kernel initializer, as it is set manually in the LSTM model due to technical difficulties with the string containing the initializer name when using seeding to minimize randomization between experiments.

3.8 Scoring

Different scoring methods are used to measure the models' performance, with all imported functions in this section coming from the `sklearn.metrics` module. For binary classification methods, the accuracy metric is used. The function `calc_metrics` shown in Listing 3.10 takes in the predicted binary results, and the actual test target values, and accuracy is calculated with the imported `accuracy_score` function. For completeness, precision, and recall of the binary predictions are also calculated.

Listing 3.10: Function for computing binary classififcation metrics.

```python
def calc_metrics(y_test: np.array , y_pred_binary: np.array, model_name: str) -> float:
    accuracy = accuracy_score(y_test, y_pred_binary)
    precision = precision_score(y_test, y_pred_binary)
    recall = recall_score(y_test, y_pred_binary)
    print(model_name + ' test data analysis:')
    print("Accuracy: ", accuracy)
    print("Precision: ", precision)
    print("Recall: ", recall)
    print('-----------------------------------------')
    return accuracy
```

To visualize the results, the `create_confusion_matrix` function shown in Listing 3.11 is implemented, which takes the same set of parameters and uses the `confusion_matrix` function to plot the confusion matrix using `matplotlib.pyplot` and `seaborn` libraries to create a `heatmap` representing the actual versus predicted values for both positive and negative classes.

Listing 3.11: Function for creating a confusion matrix.

```python
def create_confusion_matrix(y_test: np.array, y_pred_binary: np.array, model_name:
     str) -> None:
    conf_matrix = confusion_matrix(y_test, y_pred_binary)
    fig, ax1 = plt.subplots(figsize=(9, 6))
    plt.title('Confusion matrix - ' + model_name)
    sns.heatmap(conf_matrix, linewidths=1, cmap="Blues", annot=True, fmt='g')
    ax1.set_xlabel('Actual')
    ax1.set_ylabel('Predicted')
    ax1.set_xticklabels(['Negative', 'Positive'])
    ax1.set_yticklabels(['Negative', 'Positive'])
```

The second score for methods tasked with binary classification is the AUC score, represented by the plotted ROC curve, which implementation is shown in Listing 3.12. For the AUC score, the `roc_auc_score` function is used, which takes the predicted probabilities of the two classes and the actual test values. The ROC curve rates and threshold values are calculated using the `roc_curve` function, which also takes the probabilities of the predicted classes as input rather than their binary values.

Listing 3.12: Function for plotting the ROC curve.

```python
def create_roc_curve(y_test: np.array, y_pred_prob: np.array, model_name: str) -> None:
    fp_rate, tp_rate, thresholds = roc_curve(y_test, y_pred_prob)
    auc_score = roc_auc_score(y_test, y_pred_prob)
    fig, ax1 = plt.subplots(figsize=(9, 6))
    plt.plot([0, 1], [0, 1], linestyle= '--', color='black')
    plt.plot(fp_rate, tp_rate, label='AUC = {:.3f}'.format(auc_score), color='blue')
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC Curve - ' + model_name)
    plt.legend(loc='lower right')
    plt.show()
```

For regression methods, the focus is on two scoring criteria: RMSE and accuracy. The latter is possible by mapping the regression outcome to a binary representation, allowing for a comparison between binary classification and regression methods. The choice of RMSE over MSE is made because RMSE is measured in the same units as the target variable, allowing for easier interpretation of results, while MSE results are measured in squared units of the target variable. To calculate RMSE, the `mean_squared_error` function is used with the `squared` parameter set to false. It takes the predicted regression values and actual test target values as input.

Listing 3.13: Calculate RMSE.

```
test_rmse = mean_squared_error(y_test, y_pred, squared=False)
```

A mapping trick is used to calculate the accuracy of regression models, where the lambda `map` function is employed to map the predicted logarithmic returns into their binary representation.

Listing 3.14: Map continuous predicitons into binary classes.

```
y_pred_binary = list(map(lambda x: 0 if x<0 else 1, y_pred))
```

After classifying continuous values as binary classes, the same functions used for binary classification scoring metrics can be employed to calculate the accuracy of regression predictions and plot the confusion matrix.

3.9 Methodology

The main objective of this project is to analyze the effectiveness of LSTM models in predicting the logarithmic returns of the stock market using stationary logarithmic returns time series data and to utilize these predictions to construct signals for a profitable algorithmic trading strategy. Two methods are implemented to predict the logarithmic returns: binary classification and regression. The binary classification method employs stock logarithmic returns to predict the binary classes of the `relative_return_t+1` data, while the regression method uses the same input data to predict continuous values of `log_return_t+1`. However, the ARIMA model in the regression method uses the input and target data that belong to the same time series, `log_return_t+1`.

Each method has two models: an LSTM model and a baseline model. If the baseline model results in better predictions than the more complex LSTM model, there is no point in using the latter in future research. In total, there are two methods and four models. The binary classification method consists of the binary classification LSTM model and the logistic regression baseline model. The accuracy and AUC are used as scoring criteria for this method. The choice of baseline model and scoring criteria in this method is motivated by the work and findings of Duemig [2019] and Murthy et al. [2022]. The regression method includes a regression LSTM model and an ARIMA model as the baseline. The choice of the ARIMA model is motivated by the work of Ma [2020], which suggests that the LSTM model has higher accuracy than the ARIMA model. The scoring criteria between the models in the regression method will be RMSE and accuracy.

Since the accuracy score is present in both methods, it will be used to compare the results between the methods. This allows for the comparison of models within each method and the comparison of the methods themselves. For each of the 10 selected stocks, four experiments are run (one for each model), and the scoring data is collected in dictionaries where the key is the stock ticker name. The data stored throughout the experiment for each stock includes the model's predictions on the test dataset for later signal generation, training set accuracy for binary classification models, test set accuracy for binary classification models and regression models, AUC for binary models, train RMSE for the LSTM regression model, and test RMSE for regression models. All results are saved in the 'Results' directory, with data for each stock under its respective name.

For all experiments, the same lookback value of 21 trading days is used for the sliding window, as this is the average number of trading days in a month. Different lookback values ranging from 1 to 100 were experimented with, but the differences were negligible, with the best average scores occurring with a lookback of 21 days. Section 3.2 includes details on the experiment timeline.

After collecting data from all experiments, the predictions are mapped into buy and sell signals. Backtesting is performed using predicted logarithmic returns from the experiments rather than new data because a detailed analysis of those predictions is already available and can be used to understand the strategies' behavior better.

Subsections from 3.9.1 to 3.9.4 describe the implementation of each method and model in detail. Subsection 3.10 provides a description of the backtesting system implementation and individual strategies.

3.9.1 Binary Classification LSTM Model

The first model in the binary classification method is the LSTM model, with the entire implementation contained within the `run_binary_LSTM_experiment` function that takes the stock ticker string and a file, to which it writes the results, as inputs. The process begins with downloading the stock data, performing feature engineering, and specifying the input and target data. The input data is `log_return_t`, and the target data is `relative_return_t+1`. The data is then split into training and test datasets. The next step involves using the `KerasClassifier` object to construct a wrapper on the LSTM model, which is then passed to a `GridSearchCV` for training. The parameter grid uses the values obtained during the hyperparameter optimization process for the binary classification LSTM model on the AAPL stock, with the only difference being the kernel initializer, which is now set manually. The steps performed so far should be familiar after reading the previous sections, where the technicalities were described in detail. This process is iterated through in Subsection 3.9.2, 3.9.3, and 3.9.4 to ensure consistency and to point out any differences in implementation, but the general picture should be similar.

The code for binary classification LSTM model that is passed to the `KerasClassifier` wrapper can be found in the `create_experiment_binary_lstm_model` function, which is similar to the function used during hyperparameter optimization, except for the kernel initializer that has to be set manually due to technical challenges faced with seeding the model to make it more stable across experiments. This function takes the grid parameters as input and returns the `model` object.

First, the ADAM optimizer is initialized using the algorithm from the `keras.optimizers` module, which takes the learning rate as a parameter.

Listing 3.15: Initialize the ADAM optimization algorithm.

```
optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
```

The kernel initializer uses the `he_uniform` distribution from the `keras.initializers` module. Additionally, the initializer is seeded with a random integer.

Listing 3.16: Initialize kernel initializer with seeding.

```
kernel_initializer=keras.initializers.he_uniform(seed=7)
```

The LSTM model is created with the Keras `Sequential` class, which is responsible for creating a linear stack of layers inside the `model` object.

Listing 3.17: Initialize the model.

```
model = keras.Sequential()
```

Layers are added to the model, starting with the LSTM layers using the Keras `LSTM` class.

The `LSTM` object constructor specifies the number of hidden units, the input shape composed of the lookback value, and the number of features (which is 1 for all models, as only stock logarithmic returns time series data is used as input). The `stateful` argument is set to true so that the internal states of the LSTM cells are preserved between batches during training and prediction. This means that each batch's final state becomes the next batch's initial state, allowing the network to maintain memory of previous sequences and capture long-term dependencies more effectively. Lastly, the recurrent dropout value is passed.

Listing 3.18: Add LSTM layers to the model.

```
model.add(layers.LSTM(num_units, batch_size=batch_size, input_shape=(lookback,
    num_features), stateful=True, recurrent_dropout=recurrent_dropout))
```

Next is a `Dropout` class layer, responsible for regularizing the network and preventing overfitting by applying dropout to the input using the dropout rate value.

Listing 3.19: Add Dropout layer to the model.

```
model.add(layers.Dropout(dropout_rate))
```

The last layer is a single unit of the `Dense` class, which is a densely connected neural network layer that calculates the dot product between kernel weights (initialized according to the kernel initializer value) and input and passes the results of the operation to the element-wise sigmoid activation function. This crucial layer defines the prediction output; applying the sigmoid activation function will make the output binary.

Listing 3.20: Add Dense layer to the model.

```
model.add(layers.Dense(1, activation=activation_function,
    kernel_initializer=kernel_initializer))
```

The next step is to invoke the `compile` function on the `model` object to bind everything together. At this step, the loss function is specified as `binary_crossentropy`, the ADAM optimizer is passed as a parameter, and the metrics are set to accuracy.

Listing 3.21: Compile the model.

```
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

The model is passed to the Keras wrapper and is trained using the `fit` function of the `GridSearchCV` object. Once training is completed, the predictions are made using the `predict` and `predict_proba` functions of the estimator model to obtain binary and probabilistic predictions of the classes.

Listing 3.22: Make predicitons on test dataset.

```
y_pred_binary = trained_lstm_model.predict(X_test)
y_pred_prob = trained_lstm_model.predict_proba(X_test)
```

The last step is to calculate and store all the scoring metrics discussed in Section 3.8, including accuracy and AUC scores, as well as visualizing the experiment outcomes with the confusion matrix and ROC curve.

3.9.2 Regression LSTM Model

The regression LSTM model is the primary model for the second method. It is defined in the `run_regression_LSTM_experiment` function, and its implementation is very similar to the binary classification LSTM model. The process starts with preparing the data, splitting it into train and test datasets, and standardizing it. The input for the regression model is `log_return_t`, and the target is `log_return_t+1`. Next, the `KerasRegressor` and `GridSearchCV` are instantiated. The model passed to the Keras wrapper function is created inside of the `create_experiment_regression_lstm_model` function, which takes the same parameters as the function for the binary classification LSTM model. There are two differences between these two models. First, a different set of values obtained during the hyperparameter optimization process is passed, most notably a different activation function. For regression, the activation function is linear, so the final Dense layer output is continuous. The second difference is the loss function passed when compiling the model; this time, the RMSE function is chosen, which is appropriate for regression tasks.

By default, the RMSE score gives a negative value; this is due to Keras's unified scoring API. When the loss function is minimized, the Grid Search looks for the lowest possible value, and the scorer function flips the sign. Hence, this value is interpreted as positive and has its sign filled manually.

The rest of the experiment remains the same. After training the model and making predictions on the `X_test` dataset, the predictions are stored, and scoring metrics are calculated. The continuous values obtained from the regression are mapped to binary format using a lambda expression, where any negative continuous value belongs to class 0 and any positive values belong to class 1. Binary classification metrics are calculated for this model by comparing the mapped binary values with the actual relative return labels.

3.9.3 Logistic Regression Model

Logistic regression is used as a baseline model in the first method. The implementation can be found in the `run_logistic_regression_experiment` function, which follows the same process as the one described in Subsection 3.9.1. The model itself is run by calling `run_logistic_reg` function. This function takes the input train, input test, and target train datasets as inputs.

Inside the `run_logistic_reg` function, the input data is first reshaped from a $3D$ NumPy array to $2D$, as logistic regression does not work with the lookback feature. After reshaping the data, the model is instantiated using the `SDGClassifier` from the `sklearn.linear_model`. The `SDGClassifier` handles many different linear classifiers with stochastic gradient descent training. To control which model is fitted, the `loss` parameter is set to `log_loss`, resulting in the logistic regression classifier. A regularization term is also specified by passing `elasticnet` to the `penalty` parameter. The model is fitted to the reshaped train input data, and the training accuracy is calculated using the `score()` function.

Listing 3.23: Fit the logistic regression and calculate accuracy score on train dataset.

```
logistic_reg = SGDClassifier(loss='log_loss', penalty='elasticnet', shuffle=False)
trained_logistic_reg = logistic_reg.fit(X_train_2D, y_train)
train_accuracy = trained_logistic_reg.score(X_train_2D, y_train)
```

The trained model predicts the binary classes of the logarithmic returns using the input test data and the probability of classes using the `predict_proba` function so that these probabilities can later be used to create an ROC curve.

Listing 3.24: Make predicitons with logistic regression.

```
y_pred_prob = trained_logistic_reg.predict_proba(X_test_2D)
y_pred_binary = trained_logistic_reg.predict(X_test_2D)
```

The training accuracy, binary predictions, and probabilistic predictions are returned to the main experiment function, where the well-known procedure of computing metrics scores is followed.

3.9.4 ARIMA Model

The last is an ARIMA model, the baseline model for the second method. The implementation differs from the other models due to the specific nature of the autoregressive model. The ARIMA model can be used to predict the next values in the time series, but it requires previous values from the same time series as input. Therefore, even though the input data and target data are defined as `log_return_1` and `log_return_t+1`, only the train target and test target data (`y_train` and `y_test`) are of interest. These two time series are used to run the model inside the `run_arima()` function, which returns the predicted continuous values.

Listing 3.25: Function for running ARIMA model.

```python
def run_arima(train: np.array, test: np.array) -> np.array:
    sequence = [x for x in train]
    y_pred = []
    for i in range(len(test)):
        arima_model = ARIMA(sequence, order=(1, 0, 0))
        trained_arima_model = arima_model.fit()
        forecast = trained_arima_model.forecast()
        y = forecast[0]
        y_pred.append(y)
        observed = test[i]
        sequence.append(observed)
    return y_pred
```

The function starts by initializing a `sequence` variable to hold all the values from the training data and an empty prediction list. The training process for the ARIMA model is implemented using a for loop that iterates through the indices of the array storing the test data. Inside the loop, the model is initialized each time using the `ARIMA` class from the `statsmodels.tsa.arima.model` module. The object is initialized with the `sequence` variable, holding all past observations, and the `order` variable, representing the three components discussed in Subsection 2.4.2. Only the autoregressive component is initialized with values 1 dictated by the PACF plot. The other two components, integration and moving average, are not used; hence, they have values of 0. After creating a new model instance in each iteration, the model is trained using the `fit()` function and predicts the next values in the time series with the `forecast()` function. From the predicted values, the first value (the next time step in the time series) is taken and added to the predictions list. Finally, the actual value from the test data at the current time step is added to the `sequence` and treated as an observed value.

After `run_arima()` returns the predictions, the experiment follows the procedure for regression methods. First, the RMSE is calculated, then the continuous values are mapped to binary classes to obtain scoring metrics for binary classification, and finally, the results are plotted and saved to a file.

3.10 Backtesting Trading Strategies

The implementation of the trading strategies in this project is simplified and relatively straightforward. The trading algorithm based on the predicted signals is as follows: if the signal is equal to 1, the algorithm takes a long position, and if the signal is -1, the algorithm takes a short position. The backtesting does not take into account any additional costs associated with taking a position, such as commissions. Each market order is equal to an arbitrary size of 1 that reflects the daily return of the stock. The position opens at the beginning of the day and closes at the end of the day. The backtesting period is equal to the 292 days for which the predicted values were previously saved.

The trading algorithm is implemented inside the `run_trading_strategy_backtesting()` function. The first step is to map the predicted logarithmic returns into signals. The predictions stored inside the respective dictionaries are loaded into variables and are mapped into signals using the NumPy `where()` function. The mapping criteria are based on whether the predicted logarithmic return is greater or less than 0. If it is greater than 0, the signal is 1, and -1 otherwise.

The returns from each strategy, where a strategy is associated with the model used for generating signals, are then calculated. For each strategy, the signals are multiplied by the actual logarithmic return values that occurred for the day of the predicted signal, giving the strategy return for that day. If the signal is 1 (buy) and the actual return is positive, the strategy return is positive. However, if the actual return is negative, the strategy has a negative return for that day. Analogously, for a negative signal -1 (sell), the strategy results in a positive return if the return is negative. If the return is positive, the strategy has a negative return.

A simple buy-and-hold trading strategy is used to benchmark the models' strategies. A benchmark strategy is implemented using the logarithmic returns of the stock, with a signal for this strategy always equal to 1, meaning a long position is taken every day during the backtest. This is a very popular choice for a benchmark, equivalent to the approach used by Zou and Qu [2022], where the benchmark was an index return. It shows what would happen to the position if someone bought a stock on the first day of backtesting and did not change the position throughout the entire process.

In total, there are four strategies that originate from the four models' predictions and one additional benchmark strategy. To obtain the final values for each strategy's returns at the end of backtesting, cumulative returns are calculated using the Pandas `cumsum()` function. The result is a time series of returns accumulated at each time step; therefore, the final strategy returns can be accessed by specifying the last index of the series.

At the end of the backtesting, the results are saved to the appropriate file and plotted for visualization.

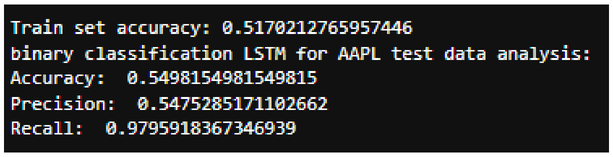
Chapter 4

Results

This chapter discusses the results obtained from the two previously described methods. The analysis begins with an examination of the output for the exemplary AAPL stock to understand the outcomes produced by the implemented methods. Subsequently, a detailed evaluation procedure is presented. The analysis proceeds to a low-level examination of the results, comparing outcomes between the primary and benchmark models and across the methods. The focus then shifts towards selecting the most optimal model for algorithmic trading based on the returns generated by the winning strategy. Finally, the results are evaluated, key findings summarized, outcomes compared to initial requirements and relative literature, and potential reasons behind the observed results are discussed. All values are rounded up to 3 significant figures.

4.1 Results Comparison

Following the execution of the experiment for the AAPL stock, the code yielded results alongside their graphical representations for both methods. The initial focus is on analyzing the output of the first method, which encompasses two models - binary classification LSTM and logistic regression. The output for each binary model includes a run summary, a confusion matrix, and an ROC curve graph with the AUC score.



```
Train set accuracy: 0.5170212765957446
binary classification LSTM for AAPL test data analysis:
Accuracy:  0.5498154981549815
Precision:  0.5475285171102662
Recall:  0.9795918367346939
```

Figure 4.1: Binary classification LSTM experiment summary for AAPL.

Figure 4.1 shows the training set accuracy of the binary classification LSTM model with a value of 0.517. The test set accuracy, precision, and recall are 0.550, 0.548, and 0.980, respectively. Figure 4.2 represents the predicted binary classes on the confusion matrix, revealing that the model predicted almost all classes to be positive, potentially indicating that the model struggles to learn data patterns and instead sticks to the best possible guess based on the data imbalance and majority class, which in this case is a positive class.

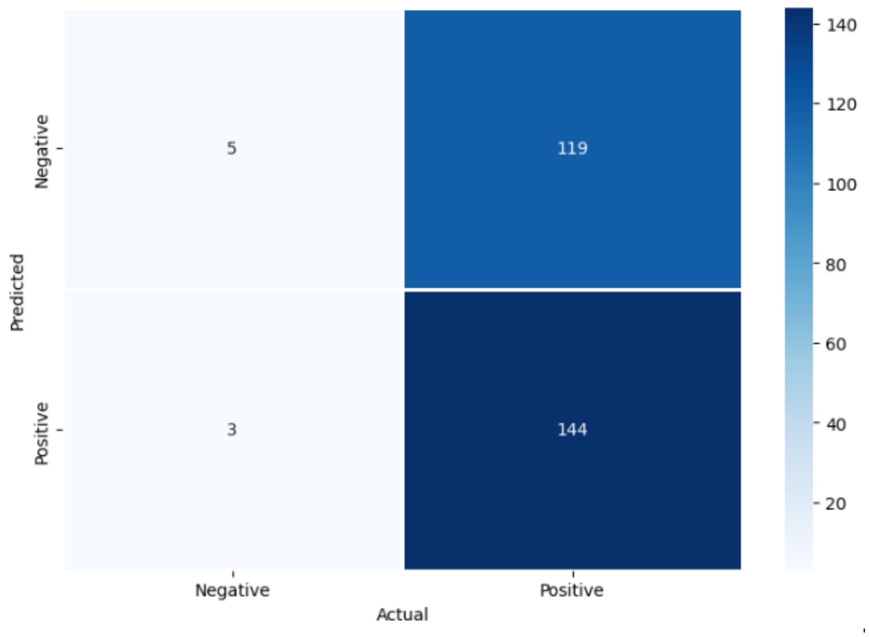


Figure 4.2: Binary classification LSTM confusion matrix for AAPL.

Figure 4.3 shows the ROC curve with an AUC score of 0.473, suggesting that the model's predictive abilities are worse than that of a naive guess. The output for the logistic regression model follows a similar structure. Figure 4.4 shows the model test run summary, with a test accuracy of 0.458, lower than the primary model. This could be a sign that the baseline model is less accurate and, therefore, less suitable for making binary predictions on stationary time series data. However, Figure 4.5 demonstrates that the logistic regression model has more diverse predictions and does not classify all data points as the same class. The model tries to capture the underlying patterns and dependencies in the data instead of always making the best guess. Figure 4.6 displays the ROC curve for the logistic regression with an AUC score of 0.465, indicating that the baseline model performed worse than the primary model in the second metric as well.

The second part of the experiment focuses on the regression method. The output for the regression LSTM model includes a model run summary with a train RMSE and binary metrics,

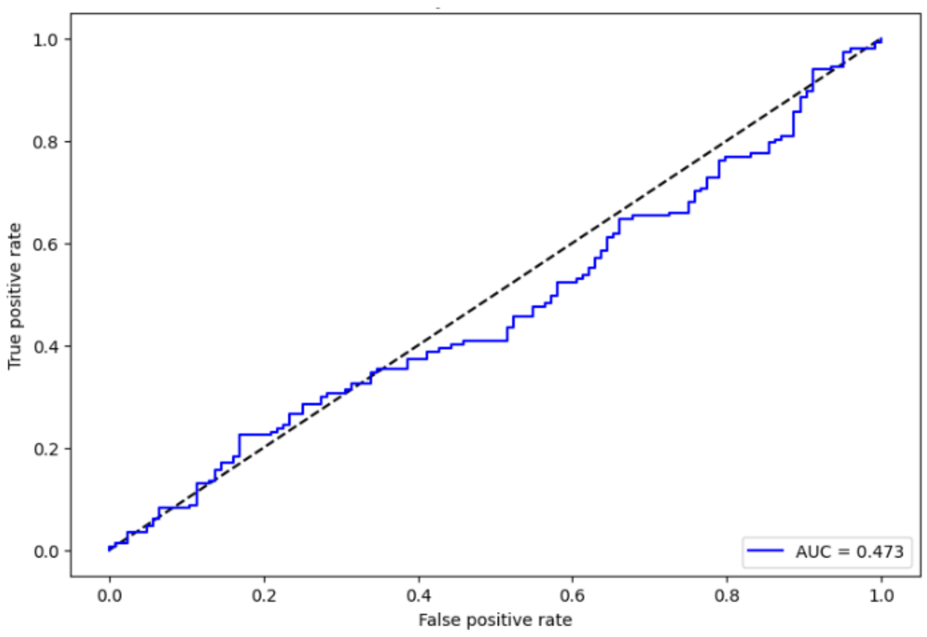
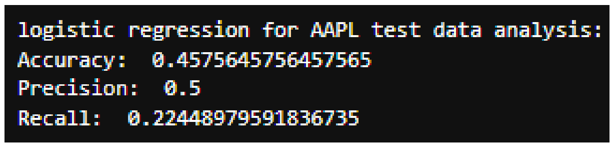


Figure 4.3: Binary classification LSTM ROC curve for AAPL.



```
logistic regression for AAPL test data analysis:
Accuracy:  0.4575645756457565
Precision:  0.5
Recall:  0.22448979591836735
```

Figure 4.4: Logistic regression experiment summary for AAPL.

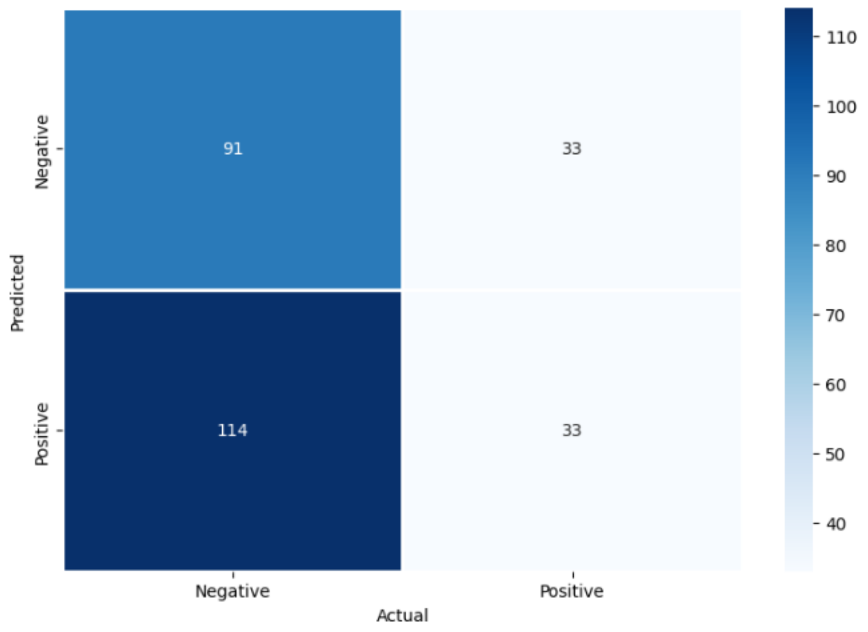


Figure 4.5: Logistic regression confusion matrix for AAPL.

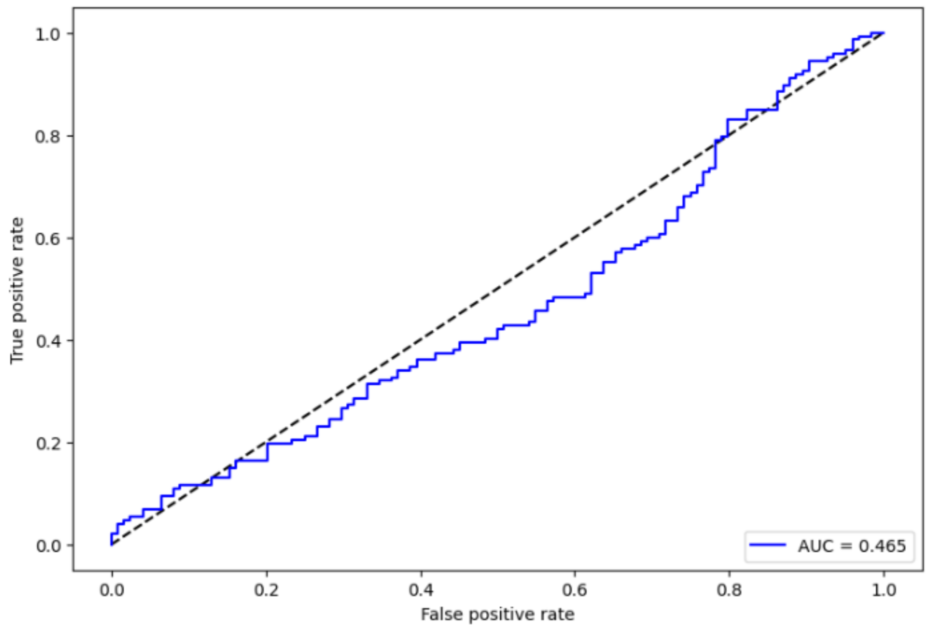
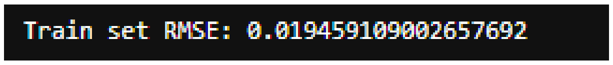


Figure 4.6: Logistic regression ROC curve for AAPL.

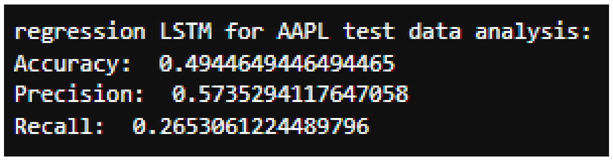
a confusion matrix showing the binary values mapped from continuous regression values, and a graph comparing the regression predictions to the actual time series values. Figure 4.7 shows the train RMSE for the regression LSTM model with a value of 0.019. A very low value of the RMSE indicates that the regression predictions performed well, yet in the case where the compared values are logarithmic returns that have inherently low values (see the y-axis values in Figure 4.10), this value is not satisfactory at all.



```
Train set RMSE: 0.019459109002657692
```

Figure 4.7: Regression LSTM RMSE for AAPL.

Figure 4.8 presents the summary of binary classification metrics, with the accuracy value of 0.494. Again, the LSTM model seems to be resulting in accuracy values very close to the 0.5 value of a naive guess, this time scoring lower than the threshold, making the regression accuracy worse than that obtained by the LSTM model in the first method.



```
regression LSTM for AAPL test data analysis:
Accuracy:  0.4944649446494465
Precision:  0.5735294117647058
Recall:  0.2653061224489796
```

Figure 4.8: Regression LSTM experiment summary for AAPL.

Figure 4.9 indicates a similar distribution of the predicted classes to logistic regression.

The model does not classify data based on the majority class but rather the opposite, as most predicted classes belong to the minority class.

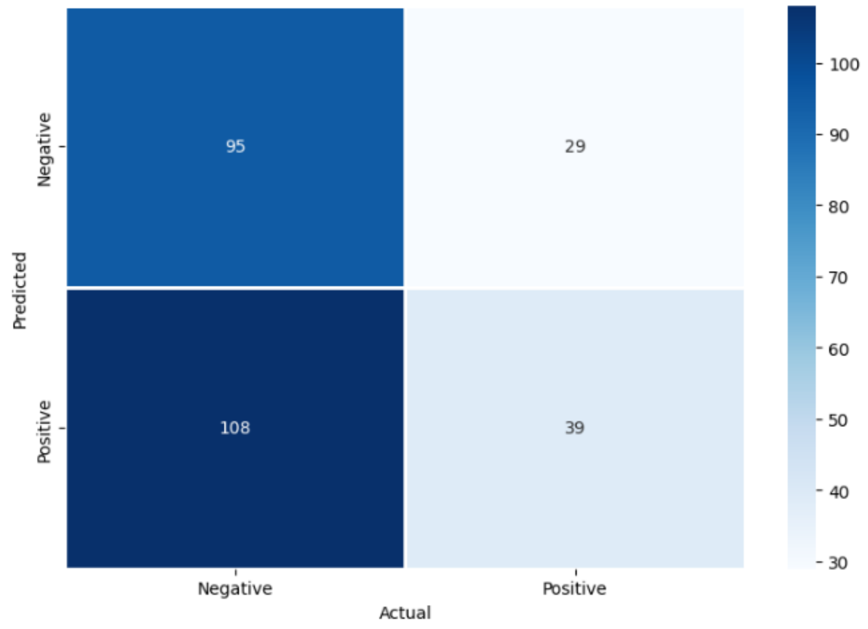


Figure 4.9: Regression LSTM confusion matrix for AAPL.

Figure 4.10 shows a plotted graph of the actual versus predicted values of the logarithmic return time series for the regression LSTM model with a test RMSE of 0.014, which again is not satisfactory. However, the graph shows that the predicted values are trying to follow the general trend of the actual values. Following the trend is very promising for the later use of the predictions as trading signals.

The final set of results is reserved for the ARIMA model predictions. Figure 4.11 shows the run summary with an accuracy value of 0.542, the highest accuracy score for the AAPL stock. This could mean that the ARIMA model performs better than the primary model.

The confusion matrix in 4.12 indicates that the ARIMA model's behavior for predicting time series is similar to the one of the binary classification LSTM model, with most predictions ending up in the positive class. These models could, therefore, have a similar sensitivity level to the data imbalance.

Figure 4.13 displays the actual and predicted values of the ARIMA model, with a test RMSE value of 0.013, which is marginally lower than the regression LSTM prediction. This makes the ARIMA model perform better overall in both scoring metrics, accuracy, and RMSE. However, the red line showing predicted values does not capture the time series trends well, which could negatively impact the effectiveness of these predictions when generating trading signals.

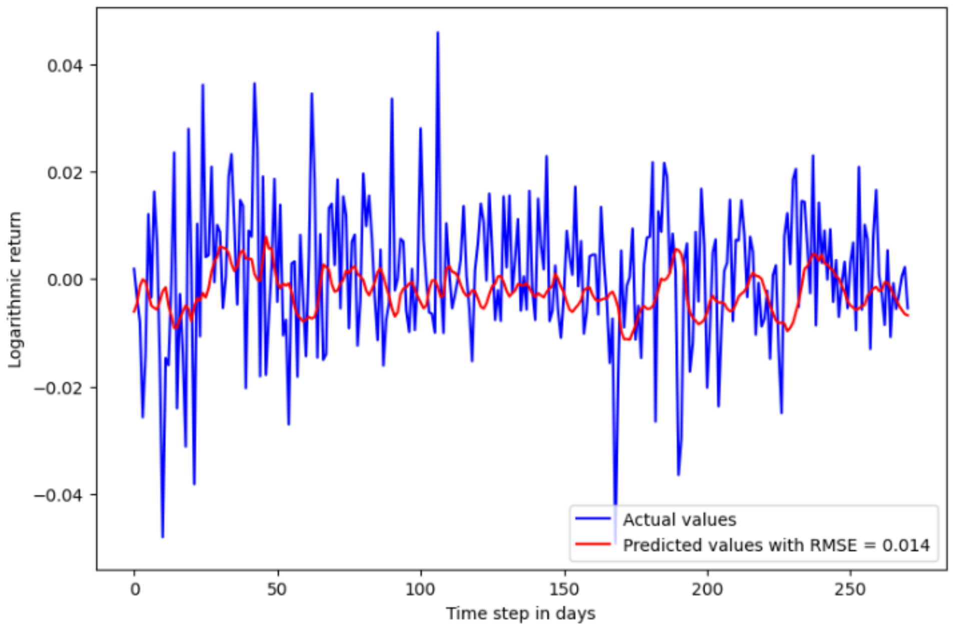
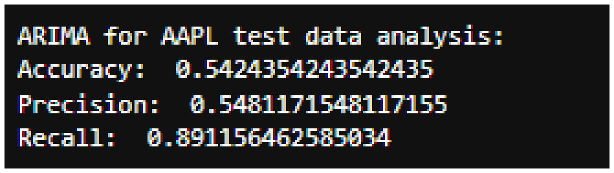


Figure 4.10: Regression LSTM predicted values vs. actual values for AAPL.



```
ARIMA for AAPL test data analysis:
Accuracy:  0.5424354243542435
Precision:  0.5481171548117155
Recall:  0.891156462585034
```

Figure 4.11: ARIMA experiment summary for AAPL.

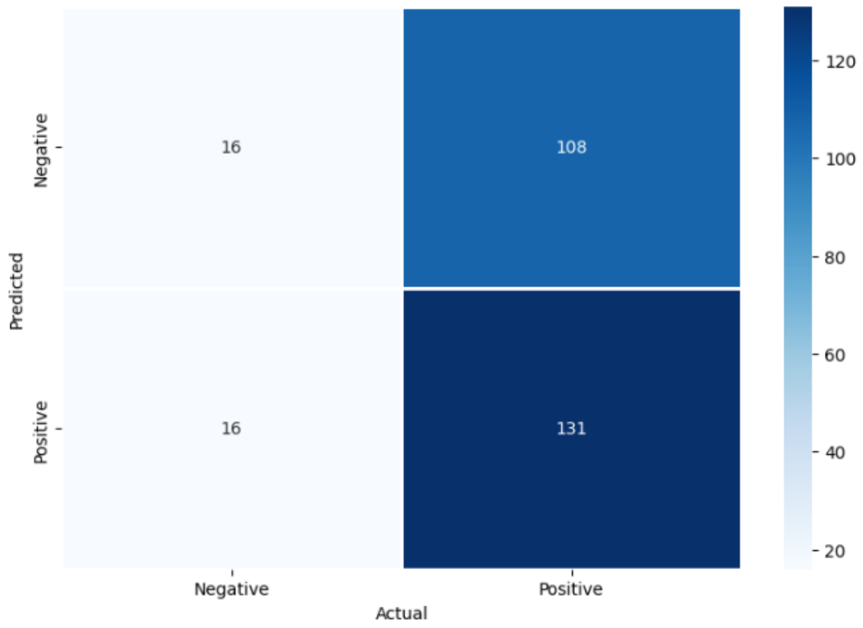


Figure 4.12: ARIMA confusion matrix for AAPL.

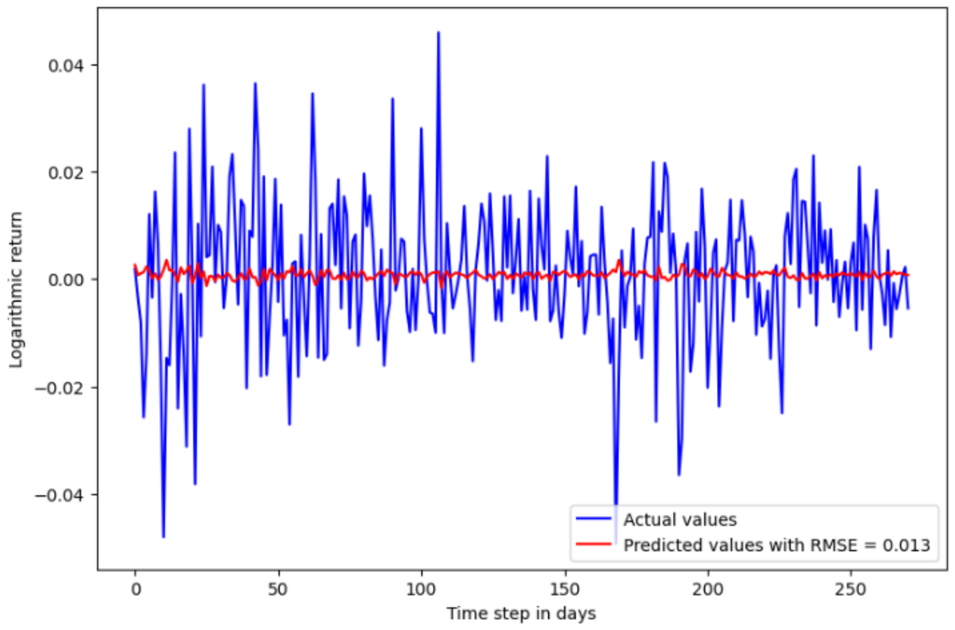


Figure 4.13: ARIMA predicted values vs. actual values for AAPL.

Now that the exemplary experiment output has been explained, the focus shifts to the general results of the stock portfolio. To analyze both methods, the mean score value of each model is used as a point of reference for the overall model's performance. To assess the performance of the models in the binary classification method, the accuracy metric and AUC score are used, with the AUC score bearing greater weight. The regression models are compared using the RMSE values. The comparison between the methods is made using the accuracy score of the best-performing model from each method. Table 4.1 summarizes the results of the first method for all stocks, with the mean value and standard deviation in the last row. Furthermore, the raw accuracy data is converted to percentages during analysis for discussion purposes. The code used to generate the summary tables can be found in the source code in Appendix C.

The LSTM model achieved the same accuracy of 52.3% during training and testing, outperforming the logistic regression model accuracy of 50.4% for training and 50.7% for testing. However, the AUC score favors the baseline model, with 0.525 for the LSTM model and 0.536 for the logistic regression model. Subsection 2.5.2 shows that the AUC score measures the models' overall performance in distinguishing between two classes. This is well depicted in Figure 4.2, where the LSTM model struggles to separate the two classes and classifies almost all data points as the majority class. Following both metrics, the conclusion for the first method is that the binary classification LSTM model is more accurate in predicting the values of the stationary time series, making it better for generating trading signals, yet the logistic regression model

Table 4.1: Binary classification method results.

	Accuracy					
	Training		Testing		AUC	
Stock ticker	LSTM	Logistic Reg.	LSTM	Logistic Reg.	LSTM	Logistic Reg.
AAPL	0.517	0.490	0.550	0.458	0.473	0.465
MSFT	0.522	0.485	0.535	0.498	0.532	0.557
AMZN	0.535	0.498	0.528	0.535	0.536	0.518
NVDA	0.530	0.510	0.539	0.524	0.489	0.532
GOOGL	0.525	0.493	0.535	0.535	0.558	0.582
TSLA	0.514	0.524	0.506	0.502	0.575	0.476
META	0.506	0.478	0.476	0.458	0.513	0.530
GOOG	0.527	0.505	0.542	0.520	0.572	0.615
BRK-B	0.512	0.520	0.509	0.520	0.462	0.542
UNH	0.539	0.537	0.509	0.517	0.538	0.544
$\bar{X}$	$0.523 \pm .015$	$0.504 \pm .019$	$0.523 \pm .022$	$0.507 \pm .028$	$0.525 \pm .040$	$0.536 \pm .045$

is better when it comes to separating the binary classes. However, the higher accuracy of the primary model can be explained by the fact that the model always chooses the majority class for most of its predictions (except for META, see raw data in Appendix A), which automatically translates to higher accuracy, but it does not mean it would perform better if the data were perfectly balanced. Table 4.2 provides a summary of the second method results.

Table 4.2: Regression method results.

	RMSE				
	Training	Testing		Testing accuracy	
Stock ticker	LSTM	LSTM	ARIMA	LSTM	ARIMA
AAPL	0.020	0.014	0.014	0.494	0.542
MSFT	0.020	0.018	0.016	0.491	0.520
AMZN	0.022	0.023	0.021	0.458	0.524
NVDA	0.029	0.032	0.030	0.465	0.513
GOOGL	0.018	0.024	0.019	0.480	0.542
TSLA	0.036	0.036	0.036	0.506	0.528
META	0.024	0.030	0.025	0.458	0.539
GOOG	0.020	0.023	0.019	0.551	0.542
BRK-B	0.014	0.010	0.009	0.465	0.524
UNH	0.018	0.013	0.013	0.568	0.483
$\bar{X}$	$0.022 \pm .006$	$0.022 \pm .009$	$0.020 \pm .008$	$0.494 \pm .039$	$0.526 \pm .018$

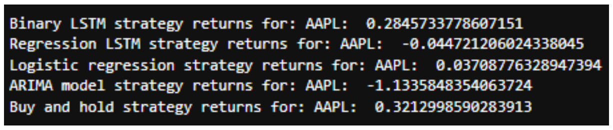
The first column presents the RMSE values of the LSTM model during training. Since there is no data for the equivalent values of the ARIMA model, it cannot be used in comparison, but its value of 0.022 is the same as during testing, indicating that the LSTM model is not overfitting. The most important columns for comparing the models' performance are columns 3 and 4, showing the test RMSE for both models. The errors seem very small; however, in reality, they are not, as the calculation of the error from the equation uses logarithmic return values that are already small, resulting in their squared difference being even smaller. The LSTM model has an RMSE value of 0.022, while the ARIMA model has 0.020. Nevertheless, the lower error for the ARIMA model does not necessarily mean it is better at capturing the data series trends, as can be seen in Figure 4.13, where the ARIMA model's predicted values do not capture the magnitude of the trend changes. Yet, what matters is not the magnitude but the direction, as the regression values are mapped to binary classes based on it. The last two columns show the accuracy of those mapped regression values, and they are indeed better for the ARIMA model, which has a 52.6% accuracy, outperforming the primary model that has an accuracy worse than a naive guess at 49.4%. Concluding the second method results, the ARIMA model performed better than the regression LSTM model in both metrics, RMSE, and accuracy, making it more suitable for predicting the stationary time series of logarithmic returns.

Now, it is time to compare the results between the two methods to understand which approach is more effective. This is done by comparing the classification accuracy values from each method. In the first method, the best mean accuracy is 52.3%, achieved by the binary classification LSTM model, with the highest overall score of 55.0% for the AAPL stock. In the second method, the best performance belongs to the ARIMA model, where the mean accuracy is 52.6%, and the highest overall accuracy score is 54.2% for the GOOG and AAPL stocks. The results show that the best model for predicting the logarithmic returns time series is the ARIMA model, which is not surprising, as this model is extremely powerful in time series forecasting. The second best model, with only a difference of 0.03%, is the binary classification LSTM. The next section uses these predictions to generate predictive signals to analyze the relation between model performance and model suitability for algorithmic trading.

4.2 Choosing the Best Model for Algorithmic Trading

The second part of the experiment involves generating predictive returns and constructing a backtesting framework to compute the returns of each trading strategy. The most profitable strategy is chosen based on the mean value of strategy returns over the stock portfolio. Figure 4.14 shows the exemplary final returns over the testing period for AAPL, and Figure 4.15 provides their visual representation.

For the AAPL stock, the strategies' returns, expressed as percentages, are as follows: binary classification LSTM strategy (blue line) 28.5%, regression LSTM strategy (orange line) -0.04%,



```
Binary LSTM strategy returns for: AAPL:  0.2845733778607151
Regression LSTM strategy returns for: AAPL:  -0.044721206024338045
Logistic regression strategy returns for: AAPL:  0.03708776328947394
ARIMA model strategy returns for: AAPL:  -1.1335848354063724
Buy and hold strategy returns for: AAPL:  0.3212998590283913
```

Figure 4.14: Strategies returns for AAPL.

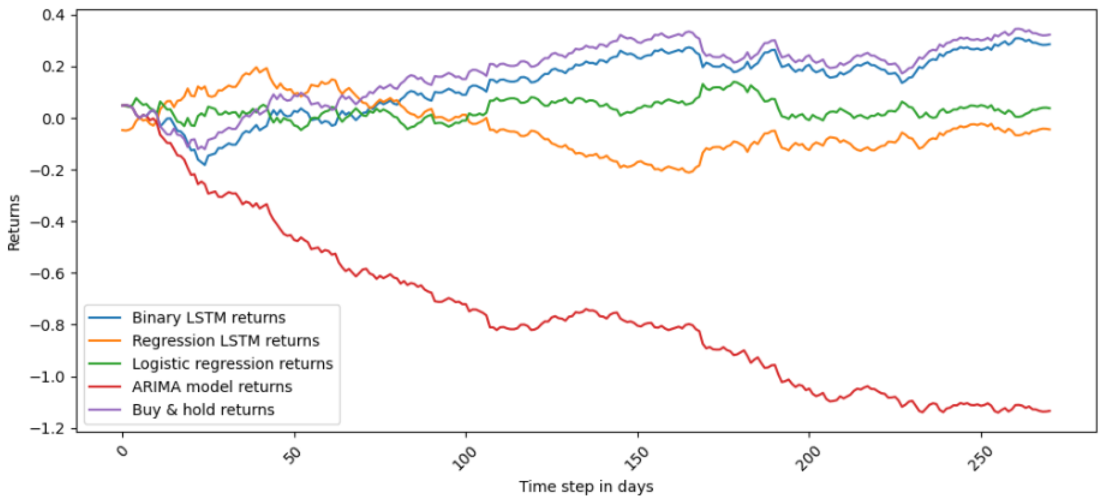


Figure 4.15: Visual representation of the strategies cumulative returns for AAPL.

logistic regression strategy (green line) 3.7%, ARIMA strategy (red line) −113.4%, and buy-and-hold strategy (purple line) 32.1%. Based on Section 4.1 and model accuracy, the expected best-performing strategies should be based on ARIMA and binary classification LSTM. However, only the binary classification LSTM strategy performed well, almost outperforming the benchmark. Surprisingly, the ARIMA-based strategy performed the worst despite having the best accuracy. Table 4.3 provides the strategies' results for all stocks in the portfolio, allowing for a better understanding of their performance by examining individual stocks and the mean values of final cumulative return. The code used to generate this summary can be found in the source code in Appendix C.

The buy-and-hold strategy's final return of 47.9% is the benchmark that should be beaten by any strategy in order to be considered successful. Unfortunately, all strategies' cumulative mean returns over the 292 testing days are lower than this benchmark. The most profitable model-based strategy is the strategy based on the binary classification LSTM model, with a mean return of 41.6%, while the least profitable is the ARIMA-based strategy, with a negative return of −173.3%. The logistic regression and regression LSTM strategies also performed poorly, with negative returns of −20.0% and −22.2%, respectively. However, although a mean return of binary classification LSTM model strategy is lower than a benchmark strategy, the LSTM-based strategy outperformed the benchmark for stock MSFT, AMZN, GOOGL, TSLA,

Table 4.3: Algorithmic trading strategies returns.

Stock ticker	Buy and Hold	Binary LSTM	Log reg.	Regression LSTM	ARIMA
AAPL	0.321	0.285	0.037	−0.045	−1.134
MSFT	0.454	0.526	−0.439	−0.181	−2.807
AMZN	0.507	0.542	0.108	−0.714	−0.392
NVDA	1.153	0.887	−0.188	−0.610	−3.575
GOOGL	0.387	0.524	−0.271	−0.111	−2.535
TSLA	0.337	0.588	−0.312	0.057	0.337
META	1.186	0.302	−1.186	−1.186	−2.298
GOOG	0.311	0.414	0.341	−0.332	−1.848
BRK-B	0.126	0.075	−0.152	−0.126	−1.583
UNH	0.012	0.018	0.386	0.398	−1.496
$\bar{X}$	$0.479 \pm .392$	$0.416 \pm .257$	$-0.200 \pm .402$	$-0.222 \pm .487$	-1.733 ± 1.157

GOOG, and UNH, which represent 6 out of 10 stock in the portfolio. The mean is lower largely due to the strong performance of the benchmark strategy for NVDA and META stocks. The only other cases where the model-based strategy outperformed the benchmark are the logistic regression model strategy for GOOG and UNH stock, regression LSTM strategy for UNH stock, and ARIMA-based strategy managed to obtain the same return as the benchmark strategy for TSLA stock. The relatively high returns of the binary classification LSTM strategy are largely due to the limitations and simplicity of the implemented backtesting framework, which does not account for trading costs that could greatly reduce those values. Nonetheless, the simplified backtesting framework results are still good indicators of the strategies' performance. If additional considerations, such as trading costs, were added to the framework, the returns for all strategies would decrease uniformly as all strategies make the same number of trades with the same position size.

The last row of the table includes the standard deviation of the mean returns. For each model-based strategy, the returns are negatively correlated with the standard deviation value; the higher the final return, the lower the standard deviation. The most stable strategy, with a standard deviation of 25.7%, belongs to the binary classification LSTM-based strategy. Yet, this is stability across the portfolio, not in terms of a single stock prediction. Hence, the actual stability of the model's predictions cannot be deduced from this data. To do that, the experiment would have to be simulated multiple times.

Surprisingly, the ARIMA-based strategy follows the same negative return pattern for all stocks, as seen in the exemplary AAPL stock. There are three possible reasons behind these very negative returns. First, while the ARIMA model had the best accuracy among all models, the predictions did not translate into good signals. It could be that most of the days for which the ARIMA predictions were accurate had very small daily returns. As seen in Figure 4.13, the model captured return directions well but failed to recognize the magnitude of returns in the time series, so outliers with great magnitude could have greatly impacted the ARIMA-based strategy's profitability. These directional predictions can be explained by the linear assumptions of the ARIMA model. Second, the logarithmic returns time series may lack an autocorrelation between time steps, which is needed for the autoregressive component to make predictions. This was mentioned in Section 3.4 when plotting ACF and PACF. Third, the experiment's time frame may be too short. An alternative approach of extending the data period, starting from 01.01.1980, was tested during the methodology development. It resulted in a testing period of over 1000 days, during which the ARIMA model consistently proved to generate the most profitable strategy, with a final return of 519% for the AAPL stock. However, this approach could not be implemented because each stock in the portfolio should have the same training and testing periods.

In conclusion, the research results are far from satisfactory and prove that the LSTM model's accuracy is only marginally better than a random guess and does not generate predictive market signals that result in strong strategy performance. However, if any model should be chosen for further investigation, it should be the binary classification LSTM model, which had the second-highest accuracy for predicting stationary time series and generated the most profitable model-based, which managed to outperform the benchmark strategy for the majority of stocks.

4.3 Evaluation

This section builds on the quantitative assessment of Sections 4.1 and 4.2 to evaluate the results with respect to initial requirements, theoretical expectations, and relative literature. This section draws justified conclusions from the rationale behind model performance.

The experiment results match the initial theoretical assumptions. Equation 2.45 in Subsection 2.3.2 states that the expected value of the random variable in the stationary time series is 0. Therefore, the binary classification has a 50% chance of being correct. According to the law of large numbers, after many repetitions of making that guess, the average of the model's results should converge to the true value of 50%. This is indeed true and proven by the results in Table 4.1 and Table 4.2, where the models' accuracy after making 292 predictions for each stock converges to a mean value within a close range to the true value of 50%. The same explanation is used for the AUC scores. If these values were different, it could suggest that the time series data is not stationary or the models can predict the random walk process. However, the models' performance can still be compared and assessed, as some were ”luckier” in guessing than others, indicating that certain models may be more appropriate for predicting stationary data.

The results show that the best model in terms of accuracy is the binary classification LSTM model, outperforming the logistic regression baseline model yet performing worse in terms of AUC score. In the work of Duemig [2019], the findings were the opposite, with the baseline model outperforming the LSTM model in accuracy metric and the LSTM and logistic regression models achieving AUC scores of 0.52 and 0.54, respectively, for the logarithmic returns input feature, which is very close to scores achieved by the models in this research. The accuracy scores were 65.01% for LSTM and 66.07% for logistic regression, considerably higher with a difference of around 13%-16%, compared to this work's results. This difference could be mostly explained by the fact that the work of Duemig [2019] was predicting logarithmic returns of a different dataset, namely the S&P 500, not individual stocks. Nevertheless, their conclusion is the same as this research: the accuracy metric is biased due to data imbalance, and while their data was more imbalanced than the data used here, it shows that LSTM models are very sensitive to data imbalance and tend to choose the majority class when making predictions on stationary time series. Therefore, the AUC score is more reflective of model performance, as it is not sensitive to class imbalance. The overall conclusion for the binary method is the same in both works: the models' results in all used metrics are close to 0.50, proving that LSTM models and logistic regression cannot efficiently predict time series with the expected value of 0 and are hardly better than a naive guess.

The regression models in the second method also gave results as expected according to the theory. The regression LSTM and ARIMA model predictions are close to 50% accuracy, meaning they could not properly learn the data patterns, and their predictions were rather random, resulting in values similar to a naive guess. The overall poor results of the regression LSTM could be a result of using a linear function in the output layer, which assumes a linear relationship between the learned representations from the LSTM layers and the target variable. Logarithmic returns data is not linear, so adding non-linear layers or using a different activation function in the output layer could have resulted in better predictions. Regression models results can also be explained by the law of large numbers and the expected value of a random variable in the stationary time series. Comparing the RMSE of the models with relevant literature is difficult, as most relevant works apply regression methods to predict stock prices rather than logarithmic returns, resulting in a different scale of errors due to the large difference in values between price data and logarithmic returns data. For example, the work of Bhandari et al. [2022] uses the regression LSTM model to predict stock close prices and obtains the mean RMSE value of 50.00, which has the order of magnitude of 10^1, while the RMSE values obtained here are of the order 10^{-2}.

The backtesting results discussed in Section 4.2 are as expected, but only for the binary classification LSTM model. According to initial assumptions, models with an accuracy greater than 50% were expected to be profitable. In the long run, when trading on financial markets, predictive signals with an accuracy marginally better than a naive guess should result in positive returns, as the strategy will be accurate more often than not. This is an important finding, proving that predictive signals do not have to be perfect to make money but only need to beat a random prediction. However, the results show this is true only for the binary classification LSTM model, pointing to another significant finding: not only do the signals need an accuracy greater than 50%, but they also need to be accurate on days representing the biggest returns. Otherwise, if the model predicts anything north of 50% but does so on days with negligible returns, the returns on days with accurate predictions will be overthrown by days where the prediction was not accurate but which exhibits a greater impact on overall return. This conclusion is further backed by the unexpected results of the ARIMA model, which had the best accuracy across the models but failed to map those successful predictions to the days that had the biggest impact on cumulative return. This leads to the conclusion that the linearity of the ARIMA model is not appropriate for predicting logarithmic return fluctuations, which is why the binary classification LSTM model performs better, as its non-linear nature is more suitable for predicting complex logarithmic return patterns.

Comparing the results with returns obtained by Zou and Qu [2022], the returns presented here are worse. However, this is because the predictive signals used by Zou and Qu [2022] were based on stock price time series, where predictions of the next day's price were the current day's price, resulting in good returns in a stable market (top 10 S&P 500 stocks by market capitalization) but imposing significant risk in a more volatile market with greater price fluctuations. Moreover, the obtained final returns are very volatile when compared across different stocks in the portfolio. Even for the most stable strategy, based on the binary classification LSTM model, returns range from 1.80% for UNH stock to 88.7% for NVDA stock. This indicates that the strategy is not suitable for live trading, as such great volatility comes with financial risk. It could be suggested that future research should focus on stocks for which the strategy performed well, such as NVDA, and build a portfolio based on selected stocks. This would indeed result in better overall returns for the portfolio as a whole, but not because of the model's ability to generate good predictive signals but rather due to the dataset imbalance. As mentioned, the accuracy of the binary classification LSTM model largely depends on data imbalance, and in the case of NVDA stock, the imbalance is significant. The company had very prosperous years, resulting in high returns on equity due to the demand for its product and high prices. Therefore, the dataset has a majority of days with positive returns, which was picked up by the LSTM model, resulting in a prediction of the positive class for most days. Due to the company's success, these days had very high positive returns for a long position.

Another reason that could explain the poor results, apart from the mathematical standpoint on the expected value of stationary time series, is economic theory. According to economic theory, if stock returns were predictable and people were rational agents acting on the stock market with unlimited credit ability, any opportunity to earn additional money compared to the whole market would be exploited in seconds, and the markets would close. If the financial time series were predictable, people responsible for quantitative research and trading at hedge funds would have already exploited this opportunity. If this was achieved at any time, the market would close, leading to the conclusion that achieving an accuracy much higher than a naive guess, resulting in better predictive signals and, hence, strategy returns, could be impossible.

As seen in the confusion matrices for AAPL stock in Section 4.1, the way models make predictions differ from each other. Some tend to choose the majority class, while others, like logistic regression, try to predict both classes more uniformly, with all models ending up close to the expected 50% accuracy. A significant limitation of this research approach is the use of data-driven deep learning models, which cannot be explained. Therefore, the evaluation cannot explain the reasoning of the LSTM models; LSTM models can only be interpreted based on their outputs, from which the general rationale can be inferred. This limitation is, of course, expected, as the goal was to study RNN models. Nevertheless, this has significant implications for using deep learning models in industry. This concept is discussed in more detail in Chapter 5.

Lastly, an alternative approach that uses more input features could be implemented to predict the logarithmic returns time series. This would change the expected value of equation 2.45 from 0 to a value that depends on values of additional features used to make predictions at time step t. Finding extra data to improve the prediction of the time series values could mean achieving better scores. However, this is not necessarily true. Work of Duemig [2019] used a mix of 3 feature sets for predicting logarithmic returns direction: price volatility, logarithmic returns, and trading volume. The AUC scores of the LSTM model that used all 3 features as input data achieved similar results as in this project, with a higher AUC score of 0.53 for the LSTM model and a lower score of 0.50 for the logistic regression model, compared to AUC score of models from this project where the only input was logarithmic returns. The accuracy data is not relevant, as the test data in both projects is different, and as mentioned, the models are very sensitive to data imbalance and noise. Therefore, increasing the number of input features as an alternative approach does not constitute better results unless the data comes from multiple alternative sources, as proposed by Weng et al. [2017].

In summary, after evaluating the results with respect to theoretical expectations and contrasting them with relevant literature, it can be concluded that while the results are not satisfactory, they are well-aligned with theory and match the observed values from other research. This allows for a clear explanation of the obtained values with regard to mathematical concepts of expected values in stationary time series. The alternative approaches, such as extending the experiment timeline and adding more input features, were also discussed, giving a comprehensive view of this project's findings. Any other ideas for different approaches and designs, as well as a discussion of the applicability of the work's findings to the financial industry, will be further discussed in subsequent chapters.

Chapter 5

Financial, Legal, Social and Professional Issues

In quantitative finance research, it is imperative to clarify the research project's purpose. It is essential to highlight that the findings and insights derived from this study are not intended to serve as financial advice. This research aims to explore the application of deep learning techniques in predicting financial markets and contribute to the broader discussion on algorithmic trading. This disclaimer is crucial for public well-being, as anyone who studies this research should understand that the outcomes are for educational and research purposes only and should not be interpreted as financial advice.

As discussed in Section 4.3, the reason behind unsatisfactory prediction results can be explained with economic theory. However, many online tutorials still show how to predict stock prices using LSTM networks, which, as is now known, is widely misunderstood. Subsection 2.3.2 shows that the best estimation of the next day's stock price is the current stock price, which is why this work focused on predicting stock returns rather than stock prices. With unsatisfactory results across the portfolio, it is important to address the social implications of relying on predictive models for investment decisions. Simply put, if Brownian motion, the underlying random factor of most financial time series, could be predicted, companies would have already exploited it, and the market would have eliminated this inefficiency before the public could take advantage of it. Of course, markets can be predicted to some extent using simple models with good variables, and according to many hedge fund portfolio managers, finding the right variables for the simplest regression is the trick. Nevertheless, someone who is not an expert in this field should definitely not rely on predictive models when making financial decisions, as this carries a huge risk that most people are unaware of. Any market participant interested in using AI for trading should be conscious of additional risks and conduct thorough research on the model they want to use.

Deep learning and machine learning models are common in the financial industry, especially hedge funds, which rely on cutting-edge technological solutions for creating and enhancing trading strategies and generating alphas. However, while these companies leverage advanced methodologies for predictive analytics, the complexity of these models poses significant challenges. Deep learning models are data-driven models characterized as black boxes that often lack interpretability, making the rationale behind their decision-making process obscure. Due to concerns about the explainability and transparency of deep learning models, hedge funds tend to choose simpler models, such as regression or autoregressive models, which were used as baseline models in this project. These machine learning models, while still powerful, allow quantitative traders to understand the underlying processes and make it easier to fix the model if needed. Another reason for using explainable AI and software that can be trusted is the presence of stakeholders who have the right to ask the asset manager to explain the decision process of the trading algorithm, which would not be possible with RNNs. Apart from stakeholders, understanding the logic behind the trading algorithm is important for risk management and regulatory compliance. Therefore, despite the general popularity of deep learning, it is not necessarily widely used in quantitative trading; instead, companies prefer understandable and reliable software that does not carry additional risk. Deep learning models are employed during the data gathering process, where Large Language Models (LLMs) and Convectional Neural Networks (CNNs) are employed to extract data from text and images. Hence, from an industrial perspective, this research does not carry great value due to the model's unexplained decision-making process. A hedge fund would probably not be interested in a predictive trading strategy based on LSTM networks but rather on one based on logistic regression and ARIMA models. A good example of this preference is one of the latest papers by the Oxford-Man Institute for Quantitative Finance, carried out in collaboration with anonymous industry partners. A work by Cartea et al. [2023] on statistical predictions of trading strategies uses basic machine learning models like logistic regression and random forest to predict and classify trading algorithms and their properties; this way, the model's behavior is explainable and can be applied in industry.

Another important legal issue is the role of intellectual property in the quantitative finance industry, particularly in relation to safeguarding trading strategies and alpha models. In financial markets, proprietary algorithms and data are critical assets, offering companies a competitive advantage. However, there are ethical considerations surrounding the ownership of such intellectual property. On one hand, protecting proprietary methodologies increases innovation by incentivizing companies and individuals to invest in research and development. On the other hand, an overly restrictive approach to intellectual property may negatively affect collaboration and knowledge sharing within the industry, slowing collective progress. The ethical dilemma of finding a balance between fostering innovation and maintaining fair competition is a very important topic that should be debated, especially now when the number of intellectual property infringements is on the rise and news headlines provide more examples of this every day. As a student and aspiring researcher in financial mathematics and machine learning, this is also my concern. This research and smaller projects completed this year are the first step in building my intellectual portfolio of trading strategies. However, securing intellectual property is not straightforward. The goal of having a portfolio is to showcase the ability to produce high-quality research and bring value to a company. It is usually used to secure a full-time position after a research degree. This means that projects, research papers, and code repositories should be publicly accessible online first to secure PhD funding or an internship. The downside, however, is that many hedge funds take advantage of students and often invite them for an interview to copy their algorithms and proprietary ideas. Many fellow researchers would likely agree that this is a pressing problem in the legal landscape and should be studied closely during research degrees to prepare students to keep their intellectual property secure.

Lastly, considerations for data privacy, transparency, and professional integrity, as outlined in the British Computer Society's Code of Conduct, should be discussed. The data used in this research originates from Yahoo! Finance, a publicly accessible source. Hence, there was no need to ensure the protection of individuals' sensitive financial information or data anonymization. Nonetheless, it is crucial to remember about privacy concerns and prevent unauthorized access or misuse of data throughout the research process. This is especially important when working on more advanced projects during a research degree, where industrial partners often supply data.

This research also emphasizes transparency of code implementation, data usage, and results. The raw experiment data is included in the Appendix A. Additionally, I intend to make the research publicly available on GitHub and submit the paper to a scientific journal for publication, facilitating the reproduction of methodologies and enhancing trust in the research findings. Sharing research materials and adhering to transparency principles contributes to knowledge advancement and fosters collaboration within academic and professional communities.

Moreover, the professional integrity of this work was demonstrated by adhering to ethical standards and academic policy throughout the research process. Any knowledge from external sources was referenced appropriately, ensuring credit was given to the original authors of ideas used for this project.

Chapter 6

Limitations

The main limitation of this research are limited computational resources. As discussed in Chapter 3, some implementation choices were primarily influenced by the lack of necessary computational power required for more complex processes and calculations. This impacted the capabilities of the Grid Search algorithm, where instead of applying hyperparameter optimization to all stocks used to generate results, the focus was on a single exemplary AAPL stock, with the optimization outcomes applied uniformly across all other assets. This could significantly impact the LSTM models' performance in a broader market. Additionally, the range of values per feature used to generate the parameters grid was limited, preventing the analysis of a wider spectrum of values.

Despite the complexity of even a simple LSTM model, it struggled to capture and exploit patterns in the provided data to the point where predictions could result in a strategy that outperformed the benchmark. More sophisticated LSTM models, like stacked or attention-based LSTM, often result in better performance, although not always as suggested by Zou and Qu [2022]. This project should have included these other models and compared their performance for completeness. However, this limitation is minor as this project primarily served as a learning opportunity for applying a simple deep learning model to financial data.

The project may have also suffered from the wrong choice of baseline model for the regression method - ARIMA. The poor results in Section 4.2 could be due to the low autocorrelation between time steps in the logarithmic returns time series, making it hard for the autoregressive component to capture data trends. As pointed out by Ma [2020], an ARIMA-GARCH model may have been more suitable.

Another limitation is the simplification of the backtesting process. The buy-and-hold strategy used as a benchmark is very basic. While it performs well as a benchmark, allowing comparison of strategies with market beta, it is naive to say that a strategy beating this benchmark is fit for live trading, especially if the strategy exhibits great volatility and unpredictability in returns. There are more advanced strategies that, while still simple, are harder to beat, such as momentum or mean reversion strategies. These, along with index trackers, could provide more valuable comparative insights. Additionally, the research overlooked trading costs associated with every buy or sell order, such as bid-ask spread and commissions. These would generate extra costs, potentially decreasing each strategy's profitability considerably, as the strategies' design forced them to execute one order per day. Therefore, the backtesting results may appear more favorable than in a live trading scenario. Additionally, the data quality in this research and its granularity is far from that of hedge funds, limiting the amount of insights that could potentially be extracted from data and applied to trading strategy.

Nevertheless, the above limitations, especially the strategies' simplification, stem from the initial project goal - exploring the applications of the LSTM network in algorithmic trading by analyzing and measuring prediction accuracy for signal generation rather than using the signals to create state-of-the-art trading strategies. Chapter 8 discusses how to improve upon this work's limitations in terms of future research directions.

Chapter 7

Conclusions

In conclusion, this research yielded several noteworthy findings regarding the effectiveness of RNNs, particularly LSTM networks, in predicting financial stationary time series and their applications to algorithmic trading. The accuracy of the models aligns with theoretical expectations and results presented in relevant literature. The LSTM model's performance is not satisfactory, as its accuracy is marginally better than a naive guess, which has a 50% chance of being correct. However, as demonstrated during strategy backtesting, these few percentage points can make a difference in the long run; being right more often than wrong in the stock market can be considered a success. Nevertheless, a better backtesting framework that imitates real-world trading conditions would be necessary to determine if this marginal difference between a naive guess and LSTM models leads to long-term profitability. Only a few strategies for selected stocks, mostly the ones based on the binary classification LSTM model, outperformed the benchmark strategy, leading to the overall poor performance of other models. Generally, the best-performing model in algorithmic trading was the binary classification LSTM model, which generated a profitable strategy for every stock. Yet, the AUC metric proves it was worse at classifying logarithmic returns than the logistic regression model, showing the model's sensitivity to data bias. The other primary model, the regression LSTM, failed to generate a profitable strategy and could not outperform the ARIMA model in terms of accuracy metrics. Another observation worth noting is the behavior of the ARIMA model, which performed very well in terms of accuracy metrics but failed to transform those predictions into profitable signals. Suggesting that the good accuracy of the predictive model does not imply the model's suitability for algorithmic trading.

Section 4.3 provided an interesting perspective on a possible reason for the prediction results being far from perfect - economic theory suggests that if markets were predictable, rational agents would quickly exploit this opportunity to predict markets with RNNs. The entire market would close in a matter of seconds. This argument explains the results of this work and suggests why obtaining perfect prediction results for any financial time series could prove impossible. If achieving good predictions with AI were obtainable, hedge funds and banks would have already been trying to achieve this for the past two decades; instead, they focus on simpler models. This argument also proves why perfect results for predicting non-stationary stock price time series, discussed in Section 1.2, are based on a completely misguided perception and an illusionary belief. If they were truly meaningful and not just a shift in predictions by one day, as presented by Miao [2020], we would see an immediate exploitation of the markets. Hence, this project has debunked the myth of perfect predictions of financial data with LSTM models by investigating predictions for data that exhibit stationarity and explaining why those predictions cannot be perfect in the real world, as well as by providing a mathematical proof that shows what predictions for non-stationary data really are. One could, however, start to think that if perfect predictions are not obtainable, what would classify a prediction as good; perhaps a prediction marginally better than a random guess is not so bad after all.

Finally, the consequences of applying the models from this project to a live trading environment were discussed. Using predictive models based on LSTM networks could result in catastrophic results for an investment portfolio. Although the binary classification LSTM model generated a profitable strategy, the final returns are positive partially due to the simplified assumptions made during backtesting and data imbalance. In live trading, the omitted trading costs could overpower any strategy profits, not to mention that the strategy struggled to outperform the very simple benchmark strategy. The volatile nature of the models and their unexplainability make them unfit for use in the financial industry, as they carry too much risk associated with the unpredictable nature of deep learning models. To use a neural network for trading, one would need better data, simpler models, and more realistic trading conditions to test the predictions. Looking at the proof-of-concept results, which this research was all about, the future of any research in this field does not look promising. Nonetheless, I believe that predictive signals based on AI are the future of algorithmic trading but generated with much simpler models that are a port of explainable AI.

Chapter 8

Future Work

This chapter discusses the possible prospects for future research, highlighting opportunities that could build upon the findings of this project. Apart from addressing the existing limitations described in Chapter 6, future research should utilize continuous data for trading instead of mapping regression values to binary. Continuous data could enhance the depth and utility of results, potentially improving the trading strategy performance. A strategy then would not only be composed of the binary signal to buy or sell. It would be accompanied by the exact predicted value of the logarithmic return, making it possible to adjust the market order size and entry conditions more granularly.

During my Master's degree and subsequent PhD studies, I want to focus on using machine learning models in a different time frame. Ideally, my future work will apply the predictive models to high-frequency trading to analyze whether the network's training updates are fast, meaningful, and accurate enough to predict the mid-price in the order book. Those predictive models could then be applied to stochastic optimal control and trade execution, where the predicted value could replace the Brownian motion component of the value function. This idea would be a great extension of the work by Tu [2020], and necessary arrangements have already been made to enable me to work on this idea.

A promising research direction is presented in the work of Sen et al. [2021] on optimizing the stock portfolio using LSTM-based price predictions and the work done by Zhang et al. [2022] on portfolio optimization with LSTM-based return and risk information. Both research use a minimum variance portfolio optimization method for implementing a portfolio algorithmic strategy. This approach results in less volatile results, higher strategy profitability, and a more promising outlook for real-world applications.

The above research ideas are far more advanced than what was covered in this project and are suitable for further studies. Nevertheless, anyone working on them should keep in mind one of the main conclusions arrived at in this project: complex deep learning models are not particularly suitable for signal generation in the industry. Therefore, all future research endeavors should focus on the promising new field of explainable AI and causal machine learning that can provide reasoning behind the choices made by the model, presenting a wide range of opportunities in the financial industry.

References

H. N. Bhandari, B. Rimal, N. R. Pokhrel, R. Rimal, K. R. Dahal, and R. K. Khatri. Predicting stock market index using lstm. *Machine Learning with Applications*, 9:100320, Sep 2022. doi: 10.1016/j.mlwa.2022.100320.

A. Cartea, S. Cohen, R. Graumans, S. Labyad, L. Sánchez-Betancourt, and L. van Veldhuijzen. Statistical predictions of trading strategies in electronic markets. *SSRN Electronic Journal*, 2023. doi: 10.2139/ssrn.4442770.

G. Chen. A gentle tutorial of recurrent neural network with error backpropagation. *CoRR*, 2016. doi: 10.48550/arXiv.1610.02583.

K. Chen, Y. Zhou, and F. Dai. A lstm-based method for stock returns prediction: A case study of china stock market. *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015. doi: 10.1109/bigdata.2015.7364089.

D. Duemig. Predicting stock prices with lstm networks, 2019. URL `http://cs230.stanford.edu/projects_winter_2019/reports/15624789.pdf`. Accessed on 29.03.2024.

S. Gu, B. Kelly, and D. Xiu. *Empirical asset pricing via machine learning*, Dec 2018. doi: 10.3386/w25398.

P. Kurek. Methods for option pricing, Oct 2023. URL `https://www.researchgate.net/publication/378936798_Methods_for_Option_Pricing`.

Q. Ma. Comparison of arima, ann and lstm for stock price prediction. *E3S Web of Conferences*, 218:01026, 2020. doi: 10.1051/e3sconf/202021801026.

Y. Miao. A deep learning approach for stock market prediction, 2020. URL `https://cs230.stanford.edu/projects_fall_2020/reports/55614857.pdf`. Accessed on 29.03.2024.

A. Murthy, N. Balaji, B. R. Puneeth, N. Megha, P. Sunil Kumar, and A. Shikah Rai. Predicting stock price with lstm networks. *2022 IEEE 2nd International Conference on Mobile Networks and Wireless Communications (ICMNWC)*, Dec 2022. doi: 10.1109/icmnwc56175.2022.10031935.

J. Sen, A. Dutta, and S. Mehtab. Stock portfolio optimization using a deep learning lstm model. *2021 IEEE Mysore Sub Section International Conference (MysuruCon)*, Oct 2021. doi: 10.1109/mysurucon52639.2021.9641662.

Y. Tu. Predicting high-frequency stock market by neural networks, 2020. URL `https://www.imperial.ac.uk/media/imperial-college/faculty-of-natural-sciences/department-of-mathematics/math-finance/Tu_Yuchen_01219050.pdf`.

B. Weng, M. A. Ahmed, and F. M. Megahed. Stock market one-day ahead movement prediction using disparate data sources. *Expert Systems with Applications*, 79:153–163, Aug 2017. doi: 10.1016/j.eswa.2017.02.041.

J. Wu, C. Wang, L. Xiong, and H. Sun. Quantitative trading on stock market based on deep reinforcement learning. *2019 International Joint Conference on Neural Networks (IJCNN)*, Jul 2019. doi: 10.1109/ijcnn.2019.8851831.

Y. Zhang, Y. Su, W. Liu, and X. Yang. *Portfolio optimization with LSTM-based return and risk information*, Sep 2022. doi: 10.2139/ssrn.4215299.

Z. Zou and Z. Qu. Using lstm in stock prediction and quantitative trading, 2022. URL `https://cs230.stanford.edu/projects_winter_2020/reports/32066186.pdf`.

Appendix A

Raw experiment data

The experiment's results for each stock are presented below. Each listing represents a separate text file generated by running the code.

Listing A.1: AAPL results.

```
Binary classification LSTM results:
Train set accuracy: 0.509
Test set accuracy: 0.542
AUC score: 0.488
------------------------------
Logistic regression results:
Train set accuracy: 0.490
Test set accuracy: 0.458
AUC score: 0.465
------------------------------
Regression LSTM results:
Train set RMSE: 0.0195
Test set RMSE: 0.0142
Test set accuracy: 0.494
------------------------------
ARIMA results:
Test set RMSE: 0.0135
Test set accuracy: 0.542
------------------------------
Trading strategy backtesting results:
```

```
Binary LSTM strategy returns: 0.285
Regression LSTM strategy returns: -0.045
Logistic regression strategy returns: 0.037
ARIMA model strategy returns: -1.134
Buy and hold strategy returns: 0.321
```

Listing A.2: AMZN results.

```
Binary classification LSTM results:
Train set accuracy: 0.535
Test set accuracy: 0.528
AUC score: 0.536
-----------------------------
Logistic regression results:
Train set accuracy: 0.498
Test set accuracy: 0.535
AUC score: 0.518
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0215
Test set RMSE: 0.0227
Test set accuracy: 0.458
-----------------------------
ARIMA results:
Test set RMSE: 0.0208
Test set accuracy: 0.524
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.542
Regression LSTM strategy returns: -0.714
Logistic regression strategy returns: 0.108
ARIMA model strategy returns: -0.392
Buy and hold strategy returns: 0.507
```

Listing A.3: BRK-B results.

```
Binary classification LSTM results:
```

```
Train set accuracy: 0.512
Test set accuracy: 0.509
AUC score: 0.462
-----------------------------
Logistic regression results:
Train set accuracy: 0.520
Test set accuracy: 0.520
AUC score: 0.542
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0137
Test set RMSE: 0.0099
Test set accuracy: 0.465
-----------------------------
ARIMA results:
Test set RMSE: 0.0089
Test set accuracy: 0.524
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.075
Regression LSTM strategy returns: -0.152
Logistic regression strategy returns: 0.081
ARIMA model strategy returns: -1.583
Buy and hold strategy returns: 0.126
```

Listing A.4: GOOG results.

```
Binary classification LSTM results:
Train set accuracy: 0.527
Test set accuracy: 0.542
AUC score: 0.572
-----------------------------
Logistic regression results:
Train set accuracy: 0.505
Test set accuracy: 0.520
AUC score: 0.615
-----------------------------
```

```
Regression LSTM results:
Train set RMSE: 0.0204
Test set RMSE: 0.0227
Test set accuracy: 0.551
-----------------------------
ARIMA results:
Test set RMSE: 0.0185
Test set accuracy: 0.542
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.414
Regression LSTM strategy returns: 0.341
Logistic regression strategy returns: 0.171
ARIMA model strategy returns: -1.848
Buy and hold strategy returns: 0.311
```

Listing A.5: GOOGL results.

```
Binary classification LSTM results:
Train set accuracy: 0.525
Test set accuracy: 0.535
AUC score: 0.558
-----------------------------
Logistic regression results:
Train set accuracy: 0.493
Test set accuracy: 0.535
AUC score: 0.582
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0175
Test set RMSE: 0.0241
Test set accuracy: 0.480
-----------------------------
ARIMA results:
Test set RMSE: 0.0192
Test set accuracy: 0.542
-----------------------------
```

```
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.524
Regression LSTM strategy returns: -0.111
Logistic regression strategy returns: -0.271
ARIMA model strategy returns: -2.535
Buy and hold strategy returns: 0.387
```

Listing A.6: META results.

```
Binary classification LSTM results:
Train set accuracy: 0.506
Test set accuracy: 0.476
AUC score: 0.513
-----------------------------
Logistic regression results:
Train set accuracy: 0.478
Test set accuracy: 0.458
AUC score: 0.530
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0239
Test set RMSE: 0.0303
Test set accuracy: 0.458
-----------------------------
ARIMA results:
Test set RMSE: 0.0246
Test set accuracy: 0.539
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.302
Regression LSTM strategy returns: -1.186
Logistic regression strategy returns: -1.186
ARIMA model strategy returns: -2.298
Buy and hold strategy returns: 1.186
```

Listing A.7: MSFT results.

```
Binary classification LSTM results:
Train set accuracy: 0.522
Test set accuracy: 0.535
AUC score: 0.532
-----------------------------
Logistic regression results:
Train set accuracy: 0.485
Test set accuracy: 0.498
AUC score: 0.557
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0197
Test set RMSE: 0.0176
Test set accuracy: 0.491
-----------------------------
ARIMA results:
Test set RMSE: 0.0159
Test set accuracy: 0.520
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.526
Regression LSTM strategy returns: -0.181
Logistic regression strategy returns: -0.439
ARIMA model strategy returns: -2.807
Buy and hold strategy returns: 0.454
```

Listing A.8: NVDA results.

```
Binary classification LSTM results:
Train set accuracy: 0.530
Test set accuracy: 0.539
AUC score: 0.489
-----------------------------
Logistic regression results:
Train set accuracy: 0.510
Test set accuracy: 0.524
AUC score: 0.532
```

```
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0290
Test set RMSE: 0.0323
Test set accuracy: 0.465
-----------------------------
ARIMA results:
Test set RMSE: 0.0302
Test set accuracy: 0.513
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.887
Regression LSTM strategy returns: -0.610
Logistic regression strategy returns: -0.188
ARIMA model strategy returns: -3.575
Buy and hold strategy returns: 1.153
```

Listing A.9: TSLA results.

```
Binary classification LSTM results:
Train set accuracy: 0.514
Test set accuracy: 0.506
AUC score: 0.575
-----------------------------
Logistic regression results:
Train set accuracy: 0.524
Test set accuracy: 0.502
AUC score: 0.476
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0360
Test set RMSE: 0.0363
Test set accuracy: 0.506
-----------------------------
ARIMA results:
Test set RMSE: 0.0356
Test set accuracy: 0.528
```

```
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.588
Regression LSTM strategy returns: 0.057
Logistic regression strategy returns: -0.312
ARIMA model strategy returns: 0.337
Buy and hold strategy returns: 0.337
```

Listing A.10: UNH results.

```
Binary classification LSTM results:
Train set accuracy: 0.539
Test set accuracy: 0.509
AUC score: 0.538
-----------------------------
Logistic regression results:
Train set accuracy: 0.537
Test set accuracy: 0.517
AUC score: 0.544
-----------------------------
Regression LSTM results:
Train set RMSE: 0.0176
Test set RMSE: 0.0134
Test set accuracy: 0.568
-----------------------------
ARIMA results:
Test set RMSE: 0.0132
Test set accuracy: 0.483
-----------------------------
Trading strategy backtesting results:
Binary LSTM strategy returns: 0.018
Regression LSTM strategy returns: 0.386
Logistic regression strategy returns: -0.001
ARIMA model strategy returns: -1.496
Buy and hold strategy returns: 0.012
```

Appendix B

User Guide

To run this project, you need to have Python 3.9 installed. The following libraries are required:

- yfinance (version 0.2.31)
- tensorflow (version 2.13.0)
- keras (version 2.13.1)
- scikeras (version 0.12.0)
- statsmodels (version 0.14.0)
- sklearn (version 1.2.1)
- pandas (version 2.0.3)
- numpy (version 1.23.5)
- keras (version 2.13.1)
- matplotlib (version 3.7.1)
- tensorflow (version 2.13.0)
- seaborn (version 0.12.2)

You can install these libraries using pip3:

```
pip3 install yfinance==0.2.31 tensorflow==2.13.0 keras==2.13.1 scikeras==0.12.0
    statsmodels==0.14.0 scikit-learn==1.2.1 pandas==2.0.3 numpy==1.23.5
    matplotlib==3.7.1 seaborn==0.12.2
```

To reproduce the results, follow these steps:

1. Download project_code.ipynb file.
2. Navigate to the directory with the project's code.
3. Install the required libraries.
4. In Jupyter Notebook, run project_code.ipynb cells with the 'run all cells' command to reproduce the experiments and generate results.

The results for each experiment are stored within the new 'Results' directory in a text file named after the stock's ticker name. The experiment summary and strategy backtesting results are saved in CSV files, scores_summary.csv, and strategies_summary.csv, in the same directory where the project's code is located.

Appendix C

Source Code

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Piotr Kurek 08.04.2024

The order of code listings below follows the same structure as the Jupyter Notebook code. Each listing represents a separate cell. For reference, please see List of Listings.

Listing C.1: Import packages.

```
# Yahoo! Finance
import yfinance as yf

# sklearn
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, confusion_matrix, accuracy_score,
    precision_score, recall_score, roc_curve, auc, roc_auc_score
from sklearn.linear_model import SGDClassifier
from sklearn import preprocessing

# scikeras wrapper for keras models
from scikeras.wrappers import KerasClassifier, KerasRegressor

# tensorflow/keras
import tensorflow
from tensorflow import keras
from keras import layers
from keras import initializers
from keras.callbacks import EarlyStopping

# statsmodels
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# data analysis/plotting
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns

# time/date
import time
import datetime

# writing to file
import sys
import os
from io import TextIOWrapper

# warnings
import warnings
warnings.filterwarnings("ignore")

# seed
from numpy.random import seed
seed(1)
tensorflow.random.set_seed(2)
```

Listing C.2: Define all functions.

```
# -------------------------------------------
# DATA ANALYSIS AND PREPROCESSING
```

```python
# ------------------------------------------

def get_data(ticker: str, start_date: datetime.datetime, end_date: datetime.datetime)
    -> pd.DataFrame:

    """
    Downloads data from Yahoo! Finance.

    Args:
        ticker: stock name
        start_date: start date
        end_date: end date

    Returns:
        The downloaded stock data.
    """

    data_df = yf.download(ticker, start=start_date, end=end_date, progress=False)

    return data_df

def feature_engineering(data: pd.DataFrame) -> pd.DataFrame:

    """
    Calculates logarithmic returns and relative returns.

    Args:
        data: stock data

    Returns:
        DataFrame with new features.
    """

    data['close_price_t'] = data['Adj Close'] # use adjusted close prices in case
        stock pays dividends
    data['log_return_t'] = np.log(data['close_price_t']).diff() # use logarithmic
```

```
        returns, they are better for calculations during strategy testing

    # calculate stock logarithmic returns for day t+1 - target for a regression
        prediction on day t using logarithmic returns data of x past days up to and
        including day t,
    # shift the target column by 1 index upward
    data['log_return_t+1'] = data['log_return_t'].shift(-1)

    # relative returns (for binary classification) for day t+1
    data['relative_return_t+1'] = np.where(data['log_return_t+1'] > 0, 1, 0)

    data = data.drop(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'],
        axis=1).dropna(axis=0) # drop columns, drop rows with NaN

    return data

def stationary_test(data: pd.DataFrame, name: str) -> None:

    """
    Performs Augmented Dickey-Fuller test for stationarity.

    Args:
        data: time series data
        name: time series name
    """

    results = adfuller(data)
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observations']
    for value, label in zip(results ,labels):
        print(label + ' : ' + str(value))

    # significance testing
    alpha = 0.05
    if results[1] <= 0.05:
        print('Time series data for ' + name + ' is stationary.')
    else:
        print('Time series data for ' + name + ' is non-stationary')

def split_and_standarize_data(input_data: pd.Series, target_data: pd.Series, lookback:
    int) -> [np.array, np.array, np.array, np.array]:

    """
    Prepares data by splitting it into training and testing datasets, standardizing
        inputs, and transforming inputs into sequences.

    Args:
        input_data: input time series data
        target_data: target time series data
        lookback: lookback value

    Returns:
        NumPy arrays: input train, input test, target train, target test.
    """

    data = pd.DataFrame().assign(input_X=input_data, target_y=target_data)

    # split data into train and test sets
    train_test_ratio = int(len(data)*0.9)
    train_set = data.iloc[: train_test_ratio, :]
    test_set = data.iloc[train_test_ratio :, :]

    # reshape inputs from train and test sets before standarization
    train_X_set = train_set['input_X'].values.reshape(-1, 1)
    test_X_set = test_set['input_X'].values.reshape(-1, 1)

    # standarize input data
    scaler = StandardScaler()
    scaler = scaler.fit(train_X_set)
    standarized_train_X_set = scaler.transform(train_X_set)
    standarized_test_X_set = scaler.transform(test_X_set) # use the same distribution
        for test data

    # sequence length of previous trading days to use for prediction
    T = lookback

    X_train = []
    y_train = []
    X_test = []
    y_test= []

    # transform input data into sequences
    for i in range(T, len(train_set) - T):
        X_train.append(standarized_train_X_set[i-T : i, :])
        y_train.append(train_set.iloc[i, 1])

    for i in range(T, len(test_set)):
        X_test.append(standarized_test_X_set[i-T : i, :])
        y_test.append(test_set.iloc[i, 1])

    X_train, X_test, y_train, y_test = np.array(X_train), np.array(X_test),
        np.array(y_train), np.array(y_test)

    print('------------------------------------------')
    print('Data shape:')
    print("X_train:", np.array(X_train).shape)
    print("y_train:", np.array(y_train).shape)
    print("X_test:", np.array(X_test).shape)
    print("y_test:", np.array(y_test).shape)
    print('------------------------------------------')

    return X_train, X_test, y_train, y_test

# ------------------------------------------
# HYPERPARAMETER OPTIMIZATION
# ------------------------------------------

def create_binary_lstm_model(num_units: int,
                             learning_rate: float,
                          dropout_rate: float,
                          recurrent_dropout: float,
                          activation_function: str,
                          kernel_initializer: str,
                          lookback: int,
                          num_features=1,
                          batch_size=1,
                          verbose=0) -> keras.Model:

    """
    Creates binary classification LSTM model for hyperparameter optimization.

    Args:
        num_units: number of LSTm units
        learning_rate: learning rate for optimizer update
        dropout_rate: dropout rate value
        recurrent_dropout: recurrent dropout value
        activation_function: activation function for output layer
        kernel_initializer: kernel initializer for network's kernels weights
        lookback: lookback value
        num_features: number of input features
        batch_size: bacth size during training
        verbose: verbose flag

    Returns:
        A Keras LSTM model.
    """

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model = keras.Sequential()
    model.add(layers.LSTM(num_units, batch_size=batch_size, input_shape=(lookback,
        num_features), stateful=True, recurrent_dropout=recurrent_dropout))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Dense(1, activation=activation_function,
        kernel_initializer=kernel_initializer))
    model.compile(loss='binary_crossentropy', optimizer=optimizer,
        metrics=['accuracy'])

    return model

def create_regression_lstm_model(num_units: int,
                              learning_rate: float,
                              dropout_rate: float,
                              recurrent_dropout: float,
                              activation_function: str,
                              kernel_initializer: str,
                              lookback: int,
                              num_features=1,
                              batch_size=1,
                              verbose=0) -> keras.Model:

    """
    Creates regression LSTM model for hyperparameter optimization.

    Args:
        num_units: number of LSTm units
        learning_rate: learning rate for optimizer update
        dropout_rate: dropout rate value
        recurrent_dropout: recurrent dropout value
        activation_function: activation function for output layer
        kernel_initializer: kernel initializer for network's kernels weights
        lookback: lookback value
        num_features: number of input features
        batch_size: bacth size during training
        verbose: verbose flag

    Returns:
        A Keras LSTM model.
    """

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model = keras.Sequential()
    model.add(layers.LSTM(num_units, batch_size=batch_size, input_shape=(lookback,
        num_features), stateful=True, recurrent_dropout=recurrent_dropout))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Dense(1, activation=activation_function,
        kernel_initializer=kernel_initializer))
    model.compile(loss='mean_squared_error', optimizer=optimizer)

    return model

# --------------------------------------------
# SCORING METRICS
# --------------------------------------------

def calc_metrics(y_test: np.array , y_pred_binary: np.array, model_name: str) -> float:

    """
    Calculates binary classification scoring metrics: accuracy, precision, and recall.

    Args:
        y_test: actual binary values
        y_pred_binary: predicted binary values
        model_name: name of the model

    Returns:
        Accuracy value.
    """

    accuracy = accuracy_score(y_test, y_pred_binary)
    precision = precision_score(y_test, y_pred_binary)
    recall = recall_score(y_test, y_pred_binary)
    print(model_name + ' test data analysis:')
    print("Accuracy: ", accuracy)
    print("Precision: ", precision)
    print("Recall: ", recall)
    print('------------------------------------------')

    return accuracy

def create_confusion_matrix(y_test: np.array, y_pred_binary: np.array, model_name:
     str) -> None:

    """
    Plots confusion matrix.

    Args:
        y_test: actual binary values
        y_pred_binary: predicted binary values
        model_name: name of the model
    """

    conf_matrix = confusion_matrix(y_test, y_pred_binary)
    fig, ax1 = plt.subplots(figsize=(9, 6))
    plt.title('Confusion matrix - ' + model_name)
    sns.heatmap(conf_matrix, linewidths=1, cmap="Blues", annot=True, fmt='g')
    ax1.set_xlabel('Actual')
    ax1.set_ylabel('Predicted')
    ax1.set_xticklabels(['Negative', 'Positive'])
    ax1.set_yticklabels(['Negative', 'Positive'])

def create_roc_curve(y_test: np.array, y_pred_prob: np.array, model_name: str) -> None:

    """
    Plots ROC curve.

    Args:
        y_test: actual binary values
        y_pred_binary: predicted probabilities of binary classes
        model_name: name of the model
    """

    # calculate false positive and true positive rates
    fp_rate, tp_rate, thresholds = roc_curve(y_test, y_pred_prob)
    auc_score = roc_auc_score(y_test, y_pred_prob)
    fig, ax1 = plt.subplots(figsize=(9, 6))
    plt.plot([0, 1], [0, 1], linestyle= '--', color='black')
    plt.plot(fp_rate, tp_rate, label='AUC = {:.3f}'.format(auc_score), color='blue')
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC Curve - ' + model_name)
    plt.legend(loc='lower right')
    plt.show()

def plot_regression_values(y_pred: np.array, y_test: np.array, rmse: float,
     name_stock: str) -> None:

    """
    Plots regression results.

    Args:
        y_pred: predicted regression values
        y_test: actual values
        rmse: RMSE value
        name_stock: stock ticker
    """

    fig, ax1 = plt.subplots(figsize=(9, 6))
    plt.plot(list(y_test), color='blue', label='Actual values')
    plt.plot(list(y_pred), label='Predicted values with RMSE = {:.3f}'.format(rmse),
         color='red')
    plt.xlabel('Time step in days')
    plt.ylabel('Logarithmic return')
    plt.title('Actual vs. predicted return for ' + name_stock)
    plt.legend(loc='lower right')
    plt.show()

# -------------------------------------------
# LSTM MODELS
# -------------------------------------------
# (the only difference between the models used during hyperparameter optimization and
    these, is the manual setting of kernel initializer)

def create_experiment_binary_lstm_model(num_units: int,
                                        learning_rate: float,
                                        dropout_rate: float,
                                        recurrent_dropout: float,
                                        activation_function: str,
                                        lookback: int,
                                        batch_size=1,
                                        num_features=1,
                                        verbose=0) -> keras.Model:

    """
    Creates binary classification LSTM model for the experiment.

    Args:
        num_units: number of LSTm units
        learning_rate: learning rate for optimizer update
        dropout_rate: dropout rate value
        recurrent_dropout: recurrent dropout value
        activation_function: activation function for output layer
        lookback: lookback value
        num_features: number of input features
        batch_size: bacth size during training
        verbose: verbose flag

    Returns:
        A Keras LSTM model.
    """

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    kernel_initializer=keras.initializers.he_uniform(seed=7)
    model = keras.Sequential()
    model.add(layers.LSTM(num_units, batch_size=batch_size, input_shape=(lookback,
        num_features), stateful=True, recurrent_dropout=recurrent_dropout))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Dense(1, activation=activation_function,
        kernel_initializer=kernel_initializer))
    model.compile(loss='binary_crossentropy', optimizer=optimizer,
        metrics=['accuracy'])

    return model

def create_experiment_regression_lstm_model(num_units: int,
                                            learning_rate: float,
                                            dropout_rate: float,
                                            recurrent_dropout: float,
                                            activation_function: str,
                                            lookback: int,
                                            batch_size=1,
                                            num_features=1,
                                            verbose=0) -> keras.Model:

    """
    Creates regression LSTM model for the experiment.

    Args:
        num_units: number of LSTm units
        learning_rate: learning rate for optimizer update
        dropout_rate: dropout rate value
        recurrent_dropout: recurrent dropout value
        activation_function: activation function for output layer
        lookback: lookback value
        num_features: number of input features
        batch_size: bacth size during training
        verbose: verbose flag

    Returns:
        A Keras LSTM model.
    """

```

```python
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    kernel_initializer=keras.initializers.he_uniform(seed=7)
    model = keras.Sequential()
    model.add(layers.LSTM(num_units, batch_size=batch_size, input_shape=(lookback,
        num_features), stateful=True, recurrent_dropout=recurrent_dropout))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Dense(1, activation=activation_function,
        kernel_initializer=kernel_initializer))
    model.compile(loss='mean_squared_error', optimizer=optimizer)

    return model

# -------------------------------------------
# BASELINE MODELS
# -------------------------------------------

def run_logistic_reg(X_train: np.array, X_test: np.array, y_train: np.array,
    model_stock_name: str) -> [float, np.array, np.array]:

    """
    Runs logistic regression.

    Args:
        X_train: train input
        X_test: test input
        y_train: train target
        name_stock: stock ticker

    Returns:
        Train dataset accuracy, predicted probabilities of binary classes, and
            predicted binary values.
    """

    # reshape the 3D array of inputs to 2D array
    X_train_2D = X_train.reshape((X_train.shape[0], -1))
    X_test_2D = X_test.reshape((X_test.shape[0], -1))

    # train model
    logistic_reg = SGDClassifier(loss='log_loss', penalty='elasticnet', shuffle=False)
    trained_logistic_reg = logistic_reg.fit(X_train_2D, y_train)
    train_accuracy = trained_logistic_reg.score(X_train_2D, y_train)

    # test model
    y_pred_prob = trained_logistic_reg.predict_proba(X_test_2D)
    y_pred_binary = trained_logistic_reg.predict(X_test_2D)

    return train_accuracy, y_pred_prob, y_pred_binary

def run_arima(train: np.array, test: np.array) -> np.array:

    """
    Runs ARIMA model.

    Args:
        train: train sequence
        test: test sequence

    Returns:
        Predicted values.
    """

    sequence = [x for x in train]
    y_pred = []
    for i in range(len(test)):
        arima_model = ARIMA(sequence, order=(1, 0, 0)) # instead of using a lookback
            of 21 days, use a value given by the PACF function
        trained_arima_model = arima_model.fit()
        forecast = trained_arima_model.forecast()
        y = forecast[0]
        y_pred.append(y)
        observed = test[i]
        sequence.append(observed)
```

```python

    return y_pred

# ------------------------------------------
# EXPERIMENT FUNCTIONS
# ------------------------------------------

def run_binary_LSTM_experiment(stock: str, file: TextIOWrapper) -> None:

    """
    Runs method 1 primary model: binary classification LSTM

    Args:
        stock: ticker name
        file: file for saving results
    """

    model_stock_name = 'binary classification LSTM for ' + stock

    # prepare data
    stock_df = get_data(stock, start_date=start_date, end_date=end_date)
    data_df = feature_engineering(stock_df)
    input_data = data_df['log_return_t']
    target_data = data_df['relative_return_t+1']
    X_train, X_test, y_train, y_test = split_and_standarize_data(input_data,
        target_data, lookback)
    num_features = X_train.shape[2]

    # best hyperparameters after optimization
    hyperparam_values = {'batch_size':[1],
                        'epochs':[1],
                        'model__num_units':[200],
                        'model__learning_rate':[0.0001],
                        'model__dropout_rate':[0.2],
                        'model__recurrent_dropout':[0.4],
                        'model__activation_function': ['sigmoid']}
```

```
                  #'model__kernel_initializer': ['he_uniform']} for experiment use
                      seeding, so string name will not work

    # keras classifier
    lstm_model = KerasClassifier(build_fn=create_experiment_binary_lstm_model,
                                 num_features=num_features,
                                 lookback=lookback,
                                 verbose=0)

    # grid search and time series cross-validation
    gridsearch_lstm_model= GridSearchCV(estimator=lstm_model,
                                  param_grid=hyperparam_values,
                                  n_jobs=-1,
                                  cv=TimeSeriesSplit(n_splits=10),
                                  refit=True,
                                  scoring='accuracy')

    # train model
    trained_lstm_model = gridsearch_lstm_model.fit(X_train, y_train, shuffle=False,
        callbacks=[early_stopping, reset_states])
    train_accuracy = trained_lstm_model.best_score_
    print('Train set accuracy:', train_accuracy)
    stock_lstm_b_train_accuracy[stock] = "{:.3f}".format(train_accuracy)
    file.write('\nTrain set accuracy: %0.3f\n' % (train_accuracy))

    # make prediction
    y_pred_binary = trained_lstm_model.predict(X_test) # binary classification
    y_pred_prob = trained_lstm_model.predict_proba(X_test) # probabilities of binary
        classes

    # store predictions
    b_predictions[stock] = y_pred_binary

    # metrics
    test_accuracy = calc_metrics(y_test, y_pred_binary, model_stock_name)
    stock_lstm_b_test_accuracy[stock] = "{:.3f}".format(test_accuracy)
    file.write('\nTest set accuracy: %0.3f\n' % (test_accuracy))

    # confusion matrix
    create_confusion_matrix(y_test, y_pred_binary, model_stock_name)

    # ROC curve
    create_roc_curve(y_test, y_pred_prob[:, 1], model_stock_name)

    # AUC score
    auc_score = roc_auc_score(y_test, y_pred_prob[:, 1])
    stock_lstm_b_auc_score[stock] = "{:.3f}".format(auc_score)
    file.write('\nAUC score: %0.3f\n' % (auc_score))

def run_regression_LSTM_experiment(stock: str, file: TextIOWrapper) -> None:

    """
    Runs method 2 primary model: regression LSTM

    Args:
        stock: ticker name
        file: file for saving results
    """

    model_stock_name = 'regression LSTM for ' + stock

    # prepare data
    stock_df = get_data(stock, start_date=start_date, end_date=end_date)
    data_df = feature_engineering(stock_df)
    input_data = data_df['log_return_t']
    target_data = data_df['log_return_t+1']
    X_train, X_test, y_train, y_test = split_and_standarize_data(input_data,
        target_data, lookback)
    num_features = X_train.shape[2]

    # best hyperparameters after optimization
    hyperparam_values = {'batch_size':[1],
                    'epochs':[1],
                    'model__num_units':[200],
                    'model__learning_rate':[0.0001],
                    'model__dropout_rate':[0.2],
                    'model__recurrent_dropout':[0.2],
                    'model__activation_function': ['linear']}
                    #'model__kernel_initializer': ['he_uniform']} for experiment use
                        seeding, so string name will not work

    # keras regressor
    lstm_model = KerasRegressor(build_fn=create_experiment_regression_lstm_model,
                                num_features=num_features,
                                lookback=lookback,
                                verbose=0)

    # grid search and time series cross-validation
    gridsearch_lstm_model= GridSearchCV(estimator=lstm_model,
                                  param_grid=hyperparam_values,
                                  n_jobs=-1,
                                  cv=TimeSeriesSplit(n_splits=10),
                                  refit=True,
                                  scoring='neg_root_mean_squared_error')

    # train model
    trained_lstm_model = gridsearch_lstm_model.fit(X_train, y_train, shuffle=False,
        callbacks=[early_stopping, reset_states])
    train_rmse = -1*trained_lstm_model.best_score_ # change sign
    print("Train set RMSE:", train_rmse)
    stock_lstm_r_train_rmse[stock] = "{:.4f}".format(train_rmse)
    file.write('\nTrain set RMSE: %0.4f\n' % (train_rmse))

    # make prediction
    y_pred = trained_lstm_model.predict(X_test)

    # store predictions
    r_predictions[stock] = y_pred

    # calculate rmse
    test_rmse = mean_squared_error(y_test, y_pred, squared=False)
    stock_lstm_r_test_rmse[stock] = "{:.4f}".format(test_rmse)
    file.write('\nTest set RMSE: %0.4f\n' % (test_rmse))

    # map continuous values to binary values
    input_data_binary = data_df['close_price_t']
    target_data_binary = data_df['relative_return_t+1']
    X_train_b, X_test_b, y_train_b, y_test_b =
        split_and_standarize_data(input_data_binary, target_data_binary, lookback) #
        only interested in y_test_b for comparison
    y_pred_binary = list(map(lambda x: 0 if x<0 else 1, y_pred))

    # metrics
    test_accuracy = calc_metrics(y_test_b, y_pred_binary, model_stock_name)
    stock_lstm_r_test_accuracy[stock] = "{:.3f}".format(test_accuracy)
    file.write('\nTest set accuracy: %0.3f\n' % (test_accuracy))

    # confusion matrix
    create_confusion_matrix(y_test_b, y_pred_binary, model_stock_name)

    # plot predicted vs. actual values
    plot_regression_values(y_pred, y_test, test_rmse, model_stock_name)

def run_logistic_regression_experiment(stock: str, file: TextIOWrapper) -> None:

    """
    Runs method 1 baseline model: logistic regression

    Args:
        stock: ticker name
        file: file for saving results
    """

    model_stock_name = 'logistic regression for ' + stock

    # prepare data
    stock_df = get_data(stock, start_date=start_date, end_date=end_date)
    data_df = feature_engineering(stock_df)
    input_data = data_df['log_return_t']
    target_data = data_df['relative_return_t+1']
    X_train, X_test, y_train, y_test = split_and_standarize_data(input_data,
        target_data, lookback)

    # run regression
    train_accuracy, y_pred_prob, y_pred_binary = run_logistic_reg(X_train, X_test,
        y_train, model_stock_name)
    stock_log_reg_train_accuracy[stock] = float("{:.3f}".format(train_accuracy))
    file.write('\nTrain set accuracy: %0.3f\n' % (train_accuracy))

    # store predictions
    log_reg_predictions[stock] = y_pred_binary

    # metrics
    test_accuracy = calc_metrics(y_test, y_pred_binary, model_stock_name)
    stock_log_reg_test_accuracy[stock] = float("{:.3f}".format(test_accuracy))
    file.write('\nTest set accuracy: %0.3f\n' % (test_accuracy))

    # confusion matrix
    create_confusion_matrix(y_test, y_pred_binary, model_stock_name)

    # ROC curve
    create_roc_curve(y_test, y_pred_prob[:, 1], model_stock_name)

    # AUC score
    auc_score = roc_auc_score(y_test, y_pred_prob[:, 1])
    stock_log_reg_auc_score[stock] = float("{:.3f}".format(auc_score))
    file.write('\nAUC score: %0.3f\n' % (auc_score))

def run_ARIMA_experiment(stock: str, file: TextIOWrapper) -> None:

    """
    Runs method 2 baseline model: ARIMA

    Args:
        stock: ticker name
        file: file for saving results
    """

    model_stock_name = 'ARIMA for ' + stock

    stock_df = get_data(stock, start_date=start_date, end_date=end_date)
    data_df = feature_engineering(stock_df)

    input_data = data_df['log_return_t']
    target_data = data_df['log_return_t+1']

    # ARIMA model uses only one time series data for making predictions, use y_train
        sequence to train the model
    # and then make walk-forward predictions with 1-lag and use new observed values to
        update the training sequence
    X_train, X_test, y_train, y_test = split_and_standarize_data(input_data,
        target_data, lookback)

    # run ARIMA
    y_pred = run_arima(y_train, y_test)

    # store predictions
    arima_predictions[stock] = y_pred

    # calculate rmse
    test_rmse = mean_squared_error(y_test, y_pred, squared=False)
    stock_arima_test_rmse[stock] = "{:.4f}".format(test_rmse)
    file.write('\nTest set RMSE: %0.4f\n' % (test_rmse))

    # map continuous values to binary values
    input_data_binary = data_df['log_return_t']
    target_data_binary = data_df['relative_return_t+1']
    X_train_b, X_test_b, y_train_b, y_test_b =
        split_and_standarize_data(input_data_binary, target_data_binary, lookback) #
        only interested in y_test_b
    y_pred_binary = list(map(lambda x: 0 if x<0 else 1, y_pred) )

    # metrics
    test_accuracy = calc_metrics(y_test_b, y_pred_binary, model_stock_name)
    stock_arima_test_accuracy[stock] = "{:.3f}".format(test_accuracy)
    file.write('\nTest set accuracy: %0.3f\n' % (test_accuracy))

    # confusion matrix
    create_confusion_matrix(y_test_b, y_pred_binary, model_stock_name)

    # plot predicted vs. actual values
    plot_regression_values(y_pred, y_test, test_rmse, model_stock_name)

def run_trading_strategy_backtesting(stock: str, file: TextIOWrapper) -> None:

    """
    Runs strategy backtesting.

    Args:
        stock: ticker name
        file: file for saving results
    """

    strategy_df = pd.DataFrame()

    # get predictions
    returns_pred_b_lstm = b_predictions[stock]
    returns_pred_r_lstm = r_predictions[stock]
    returns_pred_log_reg = log_reg_predictions[stock]
    returns_pred_arima = arima_predictions[stock]

    # map predictions to signals
    strategy_df['signal_b_lstm'] = np.where(returns_pred_b_lstm > 0, 1, -1)
    strategy_df['signal_r_lstm'] = np.where(returns_pred_r_lstm > 0, 1, -1)
    strategy_df['signal_log_reg'] = np.where(returns_pred_log_reg > 0, 1, -1)
    strategy_df['pred_returns_arima'] = returns_pred_arima # change type from list to series
    strategy_df['signal_arima'] = np.where(strategy_df['pred_returns_arima'] > 0, 1, -1)

    # calculate strategies returns - product of signal for day t and actual returns for the same day
    # therefore need to get log_return_t data for test period
    stock_df = get_data(stock, start_date=start_date, end_date=end_date)
    data_df = feature_engineering(stock_df)
    input_data = data_df['log_return_t'] # the choice of input data does not matter
    target_data = data_df['log_return_t']
    X_train, X_test, y_train, y_test = split_and_standarize_data(input_data, target_data, lookback)
    log_return_t = y_test
    strategy_df['b_lstm_strategy_returns'] = strategy_df['signal_b_lstm'] * log_return_t
    strategy_df['r_lstm_strategy_returns'] = strategy_df['signal_r_lstm'] * log_return_t
    strategy_df['log_reg_strategy_returns'] = strategy_df['signal_log_reg'] * log_return_t
    strategy_df['arima_strategy_returns'] = strategy_df['signal_arima'] * log_return_t

    # benchmark buy and hold strategy returns are just plain returns
    strategy_df['buy_and_hold_strategy_returns'] = log_return_t

    # get final returns
    b_lstm_strategy_returns = strategy_df['b_lstm_strategy_returns'].cumsum().iloc[-1]
    r_lstm_strategy_returns = strategy_df['r_lstm_strategy_returns'].cumsum().iloc[-1]
    log_reg_strategy_returns = strategy_df['log_reg_strategy_returns'].cumsum().iloc[-1]
    arima_strategy_returns = strategy_df['arima_strategy_returns'].cumsum().iloc[-1]
    buy_and_hold_strategy_returns =
        strategy_df['buy_and_hold_strategy_returns'].cumsum().iloc[-1]

    # summary of final returns
    print('------------------------------------')
    print('Binary LSTM strategy returns for: ' + stock + ': ', b_lstm_strategy_returns)
    print('Regression LSTM strategy returns for: ' + stock + ': ',
        r_lstm_strategy_returns)
    print('Logistic regression strategy returns for: ' + stock + ': ',
        log_reg_strategy_returns)
    print('ARIMA model strategy returns for: ' + stock + ': ', arima_strategy_returns)
    print('Buy and hold strategy returns for: ' + stock + ': ',
        buy_and_hold_strategy_returns)
    print('------------------------------------')

    # save strategy results
    b_lstm_returns[stock] = float("{:.3f}".format(b_lstm_strategy_returns))
    r_lstm_returns[stock] = float("{:.3f}".format(r_lstm_strategy_returns))
    log_reg_returns[stock] = float("{:.3f}".format(log_reg_strategy_returns))
    arima_returns[stock] = float("{:.3f}".format(arima_strategy_returns))
    buy_and_hold_returns[stock] = float("{:.3f}".format(buy_and_hold_strategy_returns))
    file.write('\nBinary LSTM strategy returns: %0.3f\n' % (b_lstm_strategy_returns))
    file.write('\nRegression LSTM strategy returns: %0.3f\n' %
        (r_lstm_strategy_returns))
    file.write('\nLogistic regression strategy returns: %0.3f\n' %
        (log_reg_strategy_returns))
    file.write('\nARIMA model strategy returns: %0.3f\n' % (arima_strategy_returns))
    file.write('\nBuy and hold strategy returns: %0.3f\n' %
        (buy_and_hold_strategy_returns))

    # plot cumulative returns
    plt.figure(figsize=(12,5))
    plt.plot(strategy_df.index, strategy_df['b_lstm_strategy_returns'].cumsum(), label
        = 'Binary LSTM returns')
    plt.plot(strategy_df.index, strategy_df['r_lstm_strategy_returns'].cumsum(), label
        = 'Regression LSTM returns')
    plt.plot(strategy_df.index, strategy_df['log_reg_strategy_returns'].cumsum(),
        label = 'Logistic regression returns')
    plt.plot(strategy_df.index, strategy_df['arima_strategy_returns'].cumsum(), label
        = 'ARIMA model returns')
    plt.plot(strategy_df.index, strategy_df['buy_and_hold_strategy_returns'].cumsum(),
        label = 'Buy & hold returns')
    plt.title('Strategy comparison')
    plt.xlabel('Time step in days')
    plt.ylabel('Returns')
    plt.xticks(rotation=45)
    plt.legend()
    plt.show()

def run_all_experiments(stock: str, directory: str) -> None:

    """
    Runs all experiment methods and strategy backtesting.

    Args:
        stock: ticker name
        directory: directory for saving results
    """

    command = './' + directory + '/Results_' + stock + '.txt'
    file = open(command, "w")

    print('Running binary classification LSTM for: ' + stock)
    file.write("Binary classification LSTM results:\n")
    run_binary_LSTM_experiment(stock, file)
    print('----------------------------------')
    time.sleep(5)
    print('Running logistic regression for: ' + stock)
    file.write("-----------------------------\n")
    file.write("Logistic regression results:\n")
    run_logistic_regression_experiment(stock, file)
    print('----------------------------------')
    time.sleep(5)
    print('Running regression LSTM for: ' + stock)
    file.write("-----------------------------\n")
    file.write("Regression LSTM results:\n")
    run_regression_LSTM_experiment(stock, file)
    print('----------------------------------')
    time.sleep(5)
    print('Running ARIMA for: ' + stock)
    file.write("-----------------------------\n")
    file.write("ARIMA results:\n")
    run_ARIMA_experiment(stock, file)
    print('----------------------------------')
    time.sleep(5)
    print('Running trading startegy backtesting for: ' + stock)
    file.write("-----------------------------\n")
    file.write("Trading strategy backtesting results:\n")
    run_trading_strategy_backtesting(stock, file)
    print('----------------------------------')
```

Listing C.3: Define all global variables.

```
# set start date and end date for the timeline of the experiment, these dates result
    in timeline from 22.05.2012 to 28.12.2023
start_date = datetime.datetime(2012, 5, 19)
end_date = datetime.datetime(2023, 12, 31)

# set the lookback length - number of trading days up to and including day t, to
    predict day t+1
lookback = 21

# top 10 stocks in SP&500 by index weight
stocks_list = ['AAPL', 'MSFT', 'AMZN', 'NVDA', 'GOOGL', 'TSLA', 'META', 'GOOG',
    'BRK-B', 'UNH']

# binary classification LSTM predictions for strategy testing
b_predictions = {}
# regression LSTM predictions for strategy testing
r_predictions = {}
```

```
# baseline logistic regression predictions for strategy testing
log_reg_predictions = {}
# baseline ARIMA predictions for strategy testing
arima_predictions = {}

# train set accuracy of binary classification LSTM models
stock_lstm_b_train_accuracy = {}
# test set accuracy of binary classification LSTM models
stock_lstm_b_test_accuracy = {}

# test set accuracy of regression LSTM models
stock_lstm_r_test_accuracy = {}

# train set accuracy of logistic regression models
stock_log_reg_train_accuracy = {}
# test set accuracy of logistic regression models
stock_log_reg_test_accuracy = {}

# test set accuracy of ARIMA models
stock_arima_test_accuracy = {}

# AUC scores of binary classification LSTM models
stock_lstm_b_auc_score = {}
# AUC scores of logistic regression models
stock_log_reg_auc_score = {}

# train set RMSE of regression LSTM models
stock_lstm_r_train_rmse = {}
# test set RMSE of regression LSTM models
stock_lstm_r_test_rmse = {}
# test set RMSE of ARIMA models
stock_arima_test_rmse = {}

# trading strategy results
b_lstm_returns = {}
r_lstm_returns = {}
```

```
log_reg_returns = {}
arima_returns = {}
buy_and_hold_returns = {}
```

Listing C.4: Download data, perform feature engineering, and data analysis.

```
# download data
stock_df = get_data('AAPL', start_date=start_date, end_date=end_date)

# view raw data
print('DataFrame head:')
print(stock_df.head())

# prepare data
data_df = feature_engineering(stock_df)

# calculate time range
train_test_ratio = int(len(data_df)*0.9)
train_data = data_df.iloc[: train_test_ratio, :]
test_data = data_df.iloc[train_test_ratio :, :]

# calculate experiment timeline
print('---------------------------')
print('Number of days used for training: ' + str(len(train_data)) + ', starts on: ' +
    str(train_data.index[0]) + ', ends on: ' + str(train_data.index[-1]))
print('Number of days used for testing: ' + str(len(test_data)) + ', starts on: ' +
    str(test_data.index[0]) + ', ends on: ' + str(test_data.index[-1]))
print('---------------------------')

# calculate data balance
print('Train data - days with positive return: ' +
    str(train_data["relative_return_t+1"].value_counts()[1]) + ', days with negative
    returns: ' + str(train_data["relative_return_t+1"].value_counts()[0]))
print('Test data - days with positive return: ' +
    str(test_data["relative_return_t+1"].value_counts()[1]) + ', days with negative
    returns: ' + str(test_data["relative_return_t+1"].value_counts()[0]))
print('---------------------------')
```

```

# view data
print('DataFrame shape')
print(data_df.shape)
print('---------------------------')
print('DataFrame information:')
print(data_df.info())
print('---------------------------')
print('DataFrame head:')
print(data_df.head())
print('---------------------------')
print('DataFrame tail:')
print(data_df.tail())
print('---------------------------')
print('DataFrame statistics')
print(data_df.describe())

# plot data
fig, ax1 = plt.subplots(figsize=(12, 5))
ax2 = ax1.twinx()
plt.title('Stock return vs. close price')
color1 = 'blue'
ax1.set_xlabel('Date')
ax1.set_ylabel('Logarithmic return', color=color1)
ax1.plot(data_df['log_return_t'], color=color1)
ax1.tick_params(axis='y', labelcolor=color1)
color2='red'
ax2.set_ylabel('Close price', color=color2)
ax2.plot(data_df['close_price_t'], color=color2)
ax2.tick_params(axis='y', labelcolor=color2)
fig.tight_layout()
plt.show()
```

Listing C.5: Create correlation matrix.

```
# correlation matrix
corr_matrix = data_df.corr()
```

```
mask = np.zeros_like(corr_matrix)
mask[np.triu_indices_from(mask)] = True
fig, ax1 = plt.subplots(figsize=(7, 4))
fig.suptitle('Features correlation matrix', fontsize=12)
heat_map = sns.heatmap(corr_matrix, mask=mask, linewidths=1, cmap="Blues", annot=True)
```

Listing C.6: Check time series stationarity with ADF test and plot ACF and PACF.

```
# test whether time series data is stationary and plot autocorrelation functions
stationary_test(data_df['log_return_t+1'], 'logarithmic returns')
fig, ax1 = plt.subplots(figsize=(9, 5))
plot_acf(data_df['log_return_t+1'].dropna(), lags=21, ax=ax1, title='Autocorrelation
    function of logarithmic returns')
ax1.set_xlabel('Lags')
ax1.set_ylabel('Autocorrelation value')
plt.show()
print('------------------------------------------')
stationary_test(data_df['relative_return_t+1'], 'relative returns')
fig, ax1 = plt.subplots(figsize=(9, 5))
plot_acf(data_df['relative_return_t+1'].dropna(), lags=21, ax=ax1,
    title='Autocorrelation function of relative returns')
ax1.set_xlabel('Lags')
ax1.set_ylabel('Autocorrelation value')
plt.show()
print('------------------------------------------')
stationary_test(data_df['close_price_t'], 'stock prices')
fig, ax1 = plt.subplots(figsize=(9, 5))
plot_acf(data_df['close_price_t'].dropna(), lags=21, ax=ax1, title='Autocorrelation
    function of stock prices')
ax1.set_xlabel('Lags')
ax1.set_ylabel('Autocorrelation value')
plt.show()

# plot PACF of logarithmic returns time series to determine AR coefficient in ARIMA
    model
print('------------------------------------------')
fig, ax1 = plt.subplots(figsize=(9, 5))
```

```
plot_pacf(data_df['log_return_t+1'].dropna(), lags=21, ax=ax1, title='Partial
    autocorrelation function of logarithmic returns')
ax1.set_xlabel('Lags')
ax1.set_ylabel('Partial autocorrelation value')
plt.show()
```

Listing C.7: Define callbacks for LSTM models.

```
# define callbacks for LSTM models

# early stopping with loss function monitoring
early_stopping = EarlyStopping(monitor='val_loss')

# reset states after each epoch
class ResetStates(keras.callbacks.Callback):
    def on_epoch_end(self):
        self.model.reset_states()
reset_states = ResetStates()
```

Listing C.8: Configure binary classification LSTM model.

```
# configure binary classification LSTM model
model_name_b = 'binary classification LSTM'
input_data_b = data_df['log_return_t']
target_data_b = data_df['relative_return_t+1']

X_train_b, X_test_b, y_train_b, y_test_b = split_and_standarize_data(input_data_b,
    target_data_b, lookback)

num_features_b = X_train_b.shape[2]
```

Listing C.9: Hyperparamter optimization of binary classification LSTM model.

```
# hyperparam_values = {'batch_size':[1],
#                     'epochs':[1, 10], #1
#                     'model__num_units':[10, 50, 100, 200], #200
#                     'model__learning_rate':[0.01, 0.001, 0.0001, 0.00001], #0.0001
#                     'model__dropout_rate':[0.1, 0.2, 0.3, 0.4, 0.5], #0.2
#                     'model__recurrent_dropout':[0.1, 0.2, 0.3, 0.4, 0.5], #0.4
#                     'model__activation_function': ['relu', 'tanh', 'linear',
    'sigmoid', 'softmax'], #sigmoid
#                     'model__kernel_initializer': ['uniform', 'normal', 'he_normal',
    'he_uniform']} #he_uniform

# optimize hyperparameters with Grid Search and time series cross-validation
hyperparam_values_b = {'batch_size':[1],
                   'epochs':[1],
                   'model__num_units':[200],
                   'model__learning_rate':[0.0001],
                   'model__dropout_rate':[0.2],
                   'model__recurrent_dropout':[0.4],
                   'model__activation_function': ['sigmoid'],
                   'model__kernel_initializer': ['he_uniform']}

# keras classifier wrapper
binary_lstm_model = KerasClassifier(build_fn=create_binary_lstm_model,
                                num_features=num_features_b,
                                lookback=lookback,
                                verbose=0)

# grid search and time series cross-validation
gridsearch_binary_lstm_model= GridSearchCV(estimator=binary_lstm_model,
                                  param_grid=hyperparam_values_b,
                                  n_jobs=-1,
                                  cv=TimeSeriesSplit(n_splits=10),
                                  refit=True,
                                  scoring='accuracy')

print('Performing Grid Search with time series cross-validation for binary LSTM
    model...')

start = time.time()
tuned_binary_lstm_model = gridsearch_binary_lstm_model.fit(X_train_b, y_train_b,
    shuffle=False, callbacks=[early_stopping, reset_states])
```

```
end = time.time()

print("Best accuracy score: %f achieved with parameters:" %
    tuned_binary_lstm_model.best_score_)

for param_name, param_value in tuned_binary_lstm_model.best_params_.items():
   print(param_name + ': ' + str(param_value))

print('------------------------------------------')
print("Training time:", end - start)
```

Listing C.10: Configure regression LSTM model.

```
# configure regression LSTM model
model_name_r = 'regression LSTM'
input_data_r = data_df['log_return_t']
target_data_r = data_df['log_return_t+1']

X_train_r, X_test_r, y_train_r, y_test_r = split_and_standarize_data(input_data_r,
    target_data_r, lookback)

num_features_r = X_train_r.shape[2]
```

Listing C.11: Hyperparamter optimization of regression LSTM model.

```
# hyperparam_values = {'batch_size':[1],
#                     'epochs':[1, 10], #1
#                     'model__num_units':[10, 50, 100, 200], #200
#                     'model__learning_rate':[0.01, 0.001, 0.0001, 0.00001], #0.0001
#                     'model__dropout_rate':[0.1, 0.2, 0.3, 0.4, 0.5], #0.2
#                     'model__recurrent_dropout':[0.1, 0.2, 0.3, 0.4, 0.5], #0.2
#                     'model__activation_function': ['relu', 'tanh', 'linear',
    'sigmoid', 'softmax'], #linear
#                     'model__kernel_initializer': ['uniform', 'normal', 'he_normal',
    'he_uniform']} #he_uniform

# tune hyperparameters with Grid Search and time series cross-validation
```

```
hyperparam_values_r = {'batch_size':[1],
                       'epochs':[1],
                       'model__num_units':[200],
                       'model__learning_rate':[0.0001],
                       'model__dropout_rate':[0.2],
                       'model__recurrent_dropout':[0.2],
                       'model__activation_function':['linear'],
                       'model__kernel_initializer':['he_uniform']}

# keras regressor wrapper
regression_lstm_model = KerasRegressor(build_fn=create_regression_lstm_model,
                                       num_features=num_features_r,
                                       lookback=lookback,
                                       verbose=0)

# grid search and time series cross-validation
gridsearch_regression_lstm_model= GridSearchCV(estimator=regression_lstm_model,
                                         param_grid=hyperparam_values_r,
                                         n_jobs=-1,
                                         cv=TimeSeriesSplit(n_splits=10),
                                         refit=True,
                                         scoring='neg_root_mean_squared_error')

print('Performing Grid Search with time series cross-validation for regression LSTM
    model...')

start = time.time()
tuned_regression_lstm_model = gridsearch_regression_lstm_model.fit(X_train_r,
    y_train_r, shuffle=False)
end = time.time()
train_rmse = -1*tuned_regression_lstm_model.best_score_ # change sign

print("Best RMSE score: %f achieved with parameters" % train_rmse)

for param_name, param_value in tuned_regression_lstm_model.best_params_.items():
    print(param_name + ': ' + str(param_value))

print('------------------------------------------')
print("Training time:", end - start)
print('------------------------------------------')
```

Listing C.12: Create directory to save files.

```
# create directory to save files
directory = 'Results'
command = 'mkdir ' + directory
os.system(command)
```

Listing C.13: Run experiment for Apple.

```
# experiment for Apple
run_all_experiments('AAPL', directory)
```

Listing C.14: Run experiment for Microsoft.

```
# experiment for Microsoft
run_all_experiments('MSFT', directory)
```

Listing C.15: Run experiment for Amazon.

```
# experiment for Amazon
run_all_experiments('AMZN', directory)
```

Listing C.16: Run experiment for NVIDIA.

```
# experiment for NVIDIA
run_all_experiments('NVDA', directory)
```

Listing C.17: Run experiment for Alphabet Class A.

```
# experiment for Alphabet Class A
run_all_experiments('GOOGL', directory)
```

Listing C.18: Run experiment for Tesla.

```
# experiment for Tesla
run_all_experiments('TSLA', directory)
```

Listing C.19: Run experiment for Meta.

```
# experiment for Meta
run_all_experiments('META', directory)
```

Listing C.20: Run experiment for Alphabet Class C.

```
# experiment for Alphabet Class C
run_all_experiments('GOOG', directory)
```

Listing C.21: Run experiment for Berkshire Hathaway.

```
# experiment for Berkshire Hathaway
run_all_experiments('BRK-B', directory)
```

Listing C.22: Run experiment for UnitedHealth Group.

```
# experiment for UnitedHealth Group
run_all_experiments('UNH', directory)
```

Listing C.23: Print out and save experiment scores summary.

```
scores_summary = pd.DataFrame(columns=['Train accuracy binary LSTM', 'Test accuracy
    binary LSTM',
                                   'Train accuracy logistic regression', 'Test
                                       accuracy logistic regression',
                                   'Test accuracy regression LSTM',
                                   'Test accuracy ARIMA',
                                   'Train RMSE regression LSTM', 'Test RMSE regression
                                       LSTM',
                                   'Test RMSE ARIMA',
                                   'AUC binary LSTM',
                                   'AUC logistic regression'], index=stocks_list)

for value, stock in stock_lstm_b_train_accuracy.items():
    scores_summary.loc[value, 'Train accuracy binary LSTM'] = stock
```

```

for value, stock in stock_lstm_b_test_accuracy.items():
    scores_summary.loc[value, 'Test accuracy binary LSTM'] = stock

for value, stock in stock_log_reg_train_accuracy.items():
    scores_summary.loc[value, 'Train accuracy logistic regression'] = stock

for value, stock in stock_log_reg_test_accuracy.items():
    scores_summary.loc[value, 'Test accuracy logistic regression'] = stock

for value, stock in stock_lstm_r_test_accuracy.items():
    scores_summary.loc[value, 'Test accuracy regression LSTM'] = stock

for value, stock in stock_arima_test_accuracy.items():
    scores_summary.loc[value, 'Test accuracy ARIMA'] = stock

for value, stock in stock_lstm_r_train_rmse.items():
    scores_summary.loc[value, 'Train RMSE regression LSTM'] = stock

for value, stock in stock_lstm_r_test_rmse.items():
    scores_summary.loc[value, 'Test RMSE regression LSTM'] = stock

for value, stock in stock_arima_test_rmse.items():
    scores_summary.loc[value, 'Test RMSE ARIMA'] = stock

for value, stock in stock_lstm_b_auc_score.items():
    scores_summary.loc[value, 'AUC binary LSTM'] = stock

for value, stock in stock_log_reg_auc_score.items():
    scores_summary.loc[value, 'AUC logistic regression'] = stock

command = './' + directory + '/scores_summary,csv'
scores_summary.to_csv('scores_summary.csv', index=True)

scores_summary
```

Listing C.24: Print out and save strategy returns summary.

```
strategies_summary = pd.DataFrame(columns=['Returns binary LSTM', 'Returns regression
    LSTM', 'Returns logistic regression', 'Returns ARIMA', 'Returns buy and hold'],
    index=stocks_list)

for value, stock in b_lstm_returns.items():
   strategies_summary.loc[value, 'Returns binary LSTM'] = stock

for value, stock in r_lstm_returns.items():
   strategies_summary.loc[value, 'Returns regression LSTM'] = stock

for value, stock in log_reg_returns.items():
   strategies_summary.loc[value, 'Returns logistic regression'] = stock

for value, stock in arima_returns.items():
   strategies_summary.loc[value, 'Returns ARIMA'] = stock

for value, stock in buy_and_hold_returns.items():
   strategies_summary.loc[value, 'Returns buy and hold'] = stock

command = './' + directory + '/strategies_summary.csv'
strategies_summary.to_csv('strategies_summary.csv', index=True)

strategies_summary
```

List of Figures

List of Tables

List of Listings