University of Economics in Prague

Faculty of Finance and Accounting

Department of Banking and Insurance

MASTER THESIS

Forecasting Stock Returns Using Artificial Intelligence

Author: Bc. Šimon Seman

Supervisor: Ing. Milan Fičura, Ph.D.

Prague, June 2020

Declaration of Authorship

I hereby declare that I compiled this thesis independently using only the listed resources and literature.

In Prague, 25.5.2020 ....

Student's signature

Acknowledgements

I would like to thank my supervisor Milan Fičura for his valuable comments, ideas and advice during the preparation of the thesis.

Abstract

Neural networks have been increasingly utilized for different tasks in the financial sector. The main goal of this master thesis is to compare neural networks' short-term forecasting ability against a conventional time series forecasting model. A portion of the work addresses the theoretical foundations of neural networks and the associated machine learning concepts. In the empirical part, we applied two types of LSTM networks and a GARCH model on five daily stock returns data sets and compared their out-of-sample prediction accuracy using RMSE and Diebold-Mariano test. Additionally, we re-stated the task into a classification problem by transforming the models' forecasts into binary trading signals and applied them in a backtest, where the performance of each strategy was evaluated by its profitability.

Keywords

neural networks, time series forecasting, GARCH, machine learning

Abstrakt

Neuronové sítě jsou stále více využívány ve finančním sektoru. Hlavním cílem této diplomové práce je porovnání krátkodobé predikční schopnosti neuronových sítí vůči konvenčnímu statistickému modelu časových řad. Část práce přibližuje teoretické základy neuronových sítí a relevantních konceptů strojového učení. V empirické části jsme aplikovali dva druhy LSTM sítí a GARCH model na pět datových sad denních výnosů akciových indexů a porovnali přesnost jejich predikce na testovacích datech pomocí RMSE a Diebold-Mariano testu. Následně byl problém přeformulován na klasifikační a predikované výnosy transformované do podoby binarizovaných obchodních signálů byly využity v backtestu, kterým byly jednotlivé strategie evaluovány na základě jejich výnosnosti.

Klíčová slova

neuronové sítě, modelování časových řad, GARCH, strojové učení

Contents

List of Figures

List of Tables

Introduction

During the past decade, Artificial Intelligence or AI for short has become a buzzword that has been widely used across markets from the automotive industry to the financial sector. Artificial Neural Networks (ANNs), a subset of the AI domain, represent a concept that has been theoretically laid out more than half a century ago but started to gain popularity only recently thanks to the higher availability of fast computing power. In the financial sector, a variety of different neural networks have been developed and applied to solve various tasks. For financial time-series data and forecasting, Long short-term memory (LSTM) networks have been predominantly utilized, because of their ability to handle sequential data well.

The primary goal of this thesis is to develop a functional neural network capable of predicting future stock returns. We apply two kinds of LSTM networks - a Univariate LSTM model whose input consists solely of past returns and a Multivariate LSTM model which takes in other exogenous variables, as well, and compare their predictive power with a GARCH, a benchmark time-series forecasting statistical model on time series of returns of five components of the S&P 500 index. Additionally, we perform a backtest on the forecasts and compare them with a baseline Buy and Hold model using various portfolio performance metrics.

The thesis contains 6 separate chapters and is outlined as follows. The first chapter introduces artificial neural networks from a theoretical perspective, introduces several variations of the networks, and explains the mathematical background of the associated concepts like back-propagation, activation functions, etc. The second chapter discusses the theory of model selection process, performance metrics, validation methods, and different model optimization methods such as regularization or hyperparameter tuning. The third chapter elaborates on the topic of time series analysis and the traditional time series forecasting methods (ARIMA, GARCH). The fourth chapter provides a brief literature review of several recent works of similar focus. The fifth chapter describes the methodology of the empirical part of the thesis - the data and its transformations, utilized performance metrics and statistical tests, the process of model construction, and the methodology of the backtest. The final chapter presents the performance of the models and the results of the comparative analysis and the backtest.

1. Artificial Neural Networks

1.1 Perceptron

The basis of today's neural networks stems from an artificial neuron called perceptron developed by Frank Rosenblatt in the mid 20th century. It was designed to be used for classification into two classes, under the assumption that these classes were linearly separable. Although, today's neural networks are much more complicated, their core principles were initially laid out in perceptron.

Perceptron works in the following way; it takes several binary inputs $x_1, x_2,...,x_n$ and outputs a single binary variable. A weight parameter $\boldsymbol{w}$ is added to each input variable, which signifies its importance. The binary value of the output would, then depend on the weighed sum of inputs $\sum_i w_i x_i$, more specifically, whether this sum was above or below a given threshold T. If we take the vector of input values $\boldsymbol{x}$ as given, then by changing the threshold or any of the weights we adjust the output correspondingly. Algebraically notated:

$$\text{output} = \begin{cases} 0 & \text{if: } \sum_i w_i x_i \leq \text{ T} \\ 1 & \text{if: } \sum_i w_i x_i > \text{ T} \end{cases} \tag{1.1}$$

To work with a commonly used notation in today's neural networks, we move threshold T to the left side of the inequality and change its notation to b for *Bias*. The higher the value of the bias, the more likely would the perceptron fire, i.e. outputs 1 instead of 0.

$$\text{output} = \begin{cases} 0 & \text{if: } \sum_i w_i x_i - b \leq 0 \\ 1 & \text{if: } \sum_i w_i x_i - b > 0 \end{cases} \tag{1.2}$$

By stacking multiple perceptrons into a layer and stacking multiple layers next to each other, we would create a more complex network of perceptrons, also called a *Multi-layer Perceptron (MLP)*. In such network, every perceptron works in the same fashion as described above. It takes every possible value from the previous layer as input and gives out one output, which is fed into every neuron in the next layer. The following chapter deals with the architecture of neural networks more in depth.[1]

1.2 Feedforward neural network

(Deep) Feedforward neural network, also called Multi-layer perceptron is a model that approximates a function f. Feedforward means, that the information flows from the input forward, and no output information is inserted again in the model as input (such networks are called *Recurrent neural networks* and are discussed in subsequent chapters).

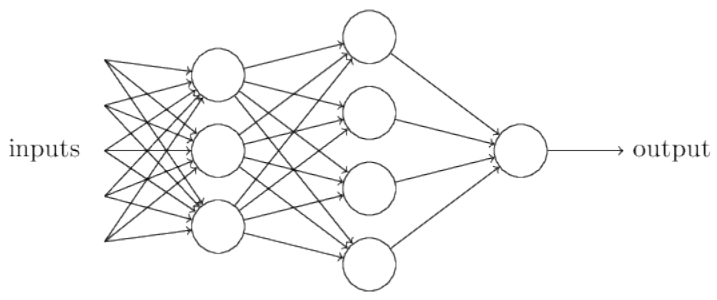

Figure 1.1: Example of a MLP

Source: neuralnetworksanddeeplearning.com

In order to understand the workings of neural networks, it is necessary to get accustomed with their structure and associated terminology. As stated before, every neural network consists of an input layer (on the left side), and arbitrary (but usually relatively small) number of hidden layers and an output layer.

Each layer is made of nodes, called neurons. Neuron is a computational unit, which collects all input values and multiplies it with corresponding weights. Weight is a parameter which signifies the strength of a connection between two neurons; its value can be any real number. To be more precise; every neuron in any given layer is connected to every other neuron in the neighboring layer. These connections are also called *synapses*. However, nodes in the same layer do not share any connection.

After the neuron sums the products of all weights and inputs, a bias can be added to the resulting value (each neuron has its own bias), which servers as an input for an activation function (see Activation function -).

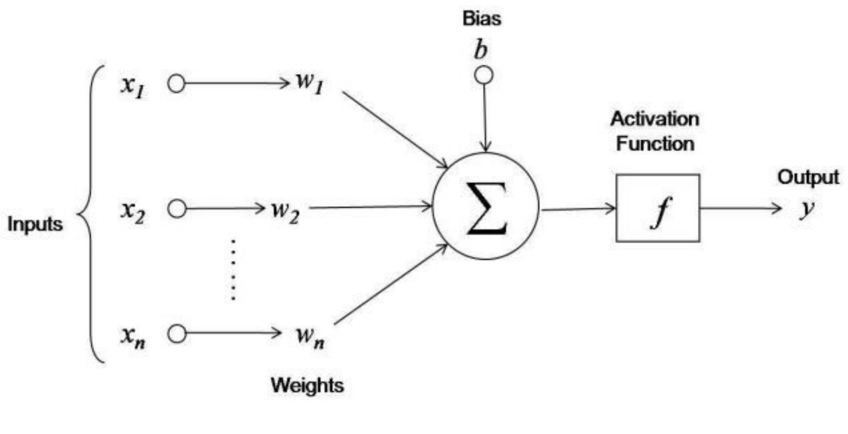

Figure 1.2: Neuron Construction

Source: deeplearningbook.org

It is relatively simple to set the number of neurons in both input and output layer. If we wanted to build a network capable of recognizing objects in pictures, such as digits[1], the number of input neurons would correspond to the number of pixels in one picture and their value would probably be a value of the RGB color space normalized in some way. As for the number of output neurons, the most intuitive number that comes up to mind would be 10, each neuron for one possible digit. Although, there are other layouts possible, e.g. 4 output neurons, where the output would be in binary numbers ($2^4 = 16$ possible outputs). The hidden layer layout cannot be determined in the same straightforward way as input and output layers. However, some approaches were developed and are discussed in the subsequent parts of the thesis.[3]

1.2.1 Forward pass

Forward pass, also known as forward propagation, is a process of pushing input variables through the network's layers all the way to the output layer. We will illustrate this process on a simple network consisting of 2 input neurons, a hidden layer with 3 neurons and 1 output neuron.

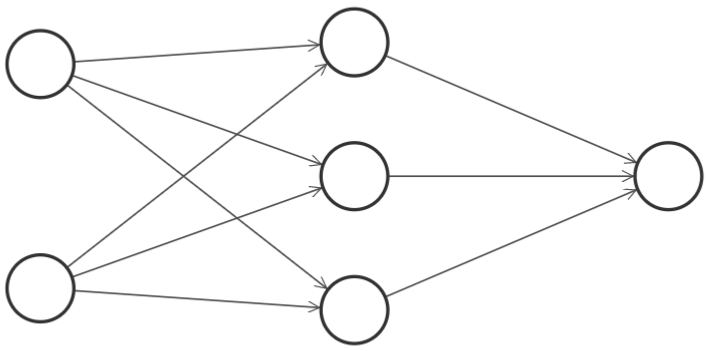

Figure 1.3: Sample Neural Network

Source: neuralnetworksanddeeplearning.com

For the sake of simplicity, let us consider a data set consisting of only one observation. Having 2 input neurons means that we expect the observation to be characterized by exactly 2 variables (the value of the variable x_1 would correspond to the first node, while the variable x_2 to the second node). The value from each input neuron is then transferred to every neuron in the hidden layer via corresponding synapse and multiplied by the weight $w_j^{(1)}$ of the respective synapse.

It is a common practise to mark layer numbers as a superscript and corresponding neuron as a subscript. As for the weights, since they connect 2 neurons from different layers, we mark them with the number of the layer to which they enter as input. Then, the first number (j)

[1]MNIST hand written digits database is often used as an example in the literature[2]

in the subscript points to the neuron in that layer and the second number (k) points to the neuron from which the weight originated. An activation function in the l-th layer of the j-th neuron would be notated in the following way;

$$a_j^l = f\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \tag{1.3}$$

where f is some activation function.

We already described what happens inside the neuron; all products of input and weight are summed, bias is added and that serves as an input $z^{(2)}$ for the activation function $a^{(2)}$. Since we are working with 3 hidden neurons, we get 3 functions $a_1^{(2)}, a_2^{(2)}$ and $a_3^{(2)}$. Each serves as an input for the output neuron, where analogous process happens; the activation function is applied for the input $z^{(3)}$, which is a sum of products of activation functions $a_j^{(2)}$ and respective weights $w_j^{(2)}$. The output of the activation function is the network's estimate $\hat{y}$.

It can become quite cumbersome to describe this process in scalar units if we used a network with more layers, nodes and observations. Instead, a more optimal way is to switch to vector notation, which enables us to get rid of all subscripts. So instead of noting down every weight in the l-th layer, we use just one vector w^l of all weights in that layer. The activation function of the l-th layer can be rewritten as:

$$a^l = f(w^l a^{l-1} + b^l). \tag{1.4}$$

We need to keep in mind that by vectorizing a function f, we apply the function to every element of the input vector x.

To sum up, we need just 4 equations to describe the forward propagation process in our sample network;

1. vector of inputs in the second layer: $z^{(2)} = xw^{(1)} + b^{(1)}$,
2. vector of activation functions of the second layer: $a^{(2)} = f(z^{(2)})$,
3. vector of inputs in the third (output) layer: $z^{(3)} = a^{(2)}w^{(2)} + b^{(2)}$,
4. vector of network outputs (just a scalar value in out case): $\hat{y} = f(z^{(3)})$.

[1]

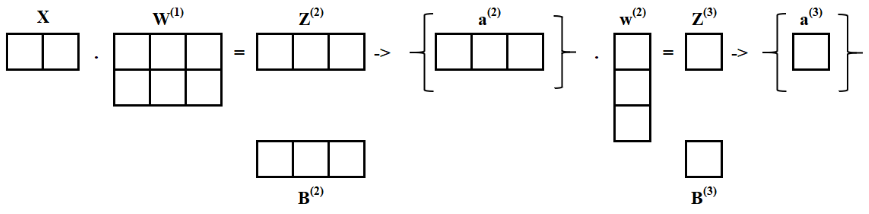

Figure 1.4: Matrix notation visualised

Source: Author's own work

1.2.2 Activation function

The way a perceptron deals with the input data is just one representation of an activation function. In general, it is a mathematical function which determines the output of a neuron (and the whole neural network) by calculating a weighted sum of its inputs and adding bias. The range of possible outputs is what sets different activation functions apart.

Binary step function is the simplest form of activation function. It is, in essence a perceptron, which either fires, when the weighted input exceeds the threshold, or it does not. Its simplicity reflects in the following drawbacks. First, because it can output only two values it can be used only for binary classification. Second, its gradient is always zero, which makes it impossible to effectively train such network with back-propagation.

Linear activation function takes the inputs multiplied by weights, and outputs a signal proportional to the input values. In comparison to the binary step function, it allows multiple other outputs, besides 0 and 1.

$$A = \sum_i w_i x_i \tag{1.5}$$

However, one limitation has to be taken into consideration, when choosing this function. By combining any number of linear functions in a linear manner, we always get just another linear function.Therefore, by stacking multiple layers of neurons, whose activation functions are linear, the activation function of the output layer will always be linear and all the previous layers can be omitted.

Among non-linear activation functions, *Sigmoid function* (also called Logistic function) is one of the most commonly used in practise. Similarly to Binary step function, sigmoid function takes any real value as input and outputs a value between 0 and 1. Being non-linear, one of its properties is differentiability, which makes it usable for back-propagation.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{1.6}$$

$$\sigma'(z) \equiv \sigma(z) \cdot (1 - \sigma(z)) \tag{1.7}$$

When plotted, it visually resembles a smoothed out version of a Binary Step function; their output values are equal for substantially high or low input values - if we assume $z = w \cdot x + b$, we get 1 if z has a large positive value, and 0 if z is significantly smaller than 0. However, because of its smoothness, for inputs close enough to 0, it outputs real values other than just 0 or 1. This gives the opportunity for small changes in weights w or bias b to project themselves into the output.

One fallback that using sigmoid brings is low responsiveness to changes in input Δz towards both ends of the sigmoid function. Algebraically, this means that significant changes in input do not reflect into a considerable change in output. In other words, the gradient in those regions is too small. Having low, if any response at all, from changing input parameters makes it either very slow, or even impossible for the network to learn.

Tanh Function is just a scaled version of sigmoid function with very similar use cases. Its output range, in comparison to sigmoid, can also take on negative values (between -1 and 1) and its derivatives with respect to input value are steeper. Moreover, Tanh function suffers from the same negative properties as the sigmoid.

$$\tanh(z) = \frac{2}{1+e^{-2z}} - 1 \tag{1.8}$$

ReLu, short for Rectified Linear Units, is a more recent, but heavily used activation function. It can take on values from 0 to infinity. In the positive axis, it acts as a linear function. In the negative axis, it outputs 0. However, as a whole it has non-linear property, which makes it differentiable.

One of its characteristics is sparse activation of the neurons; when working with sigmoid or tanh function, all neurons fire irrespecitve of their input value, which takes a lot of computational power and time (also known as *dense activation*). On the other hand, with ReLu, when we initially set the weights randomly, only approximately half of the neurons would fire, which makes it practical and cost-effective to use with large neural networks.

$$R(z) = max(0, z) \tag{1.9}$$

There is also a negative aspect of sparsely activated neurons that needs to be counted with. Having gradient equal to 0 means that those neurons stop responding to any variations in input. Collectively, these "dead" neurons will make a portion of the network passive. It is commonly referred to as the "Dying ReLu" problem. There is a variation called *Leaky ReLu*, which addresses that problem with a gradient slightly larger than 0 in the negative axis.

1.3 Neural network training

1.3.1 Cost function

After observations pass forward through the network, we need to estimate the performance of the network with some function. For that purpose we use a *Cost function C* (also known as Loss function or Objective function) which generally takes the predicted output $\hat{y}$ and the target value of that observation (i.e. the expected value from the labeled training data) as parameters and outputs a single real number.

Although, there are many cost functions to choose from, we can use a relatively simple *Mean squared error* as an illustrative example. A squared error is just a squared difference between

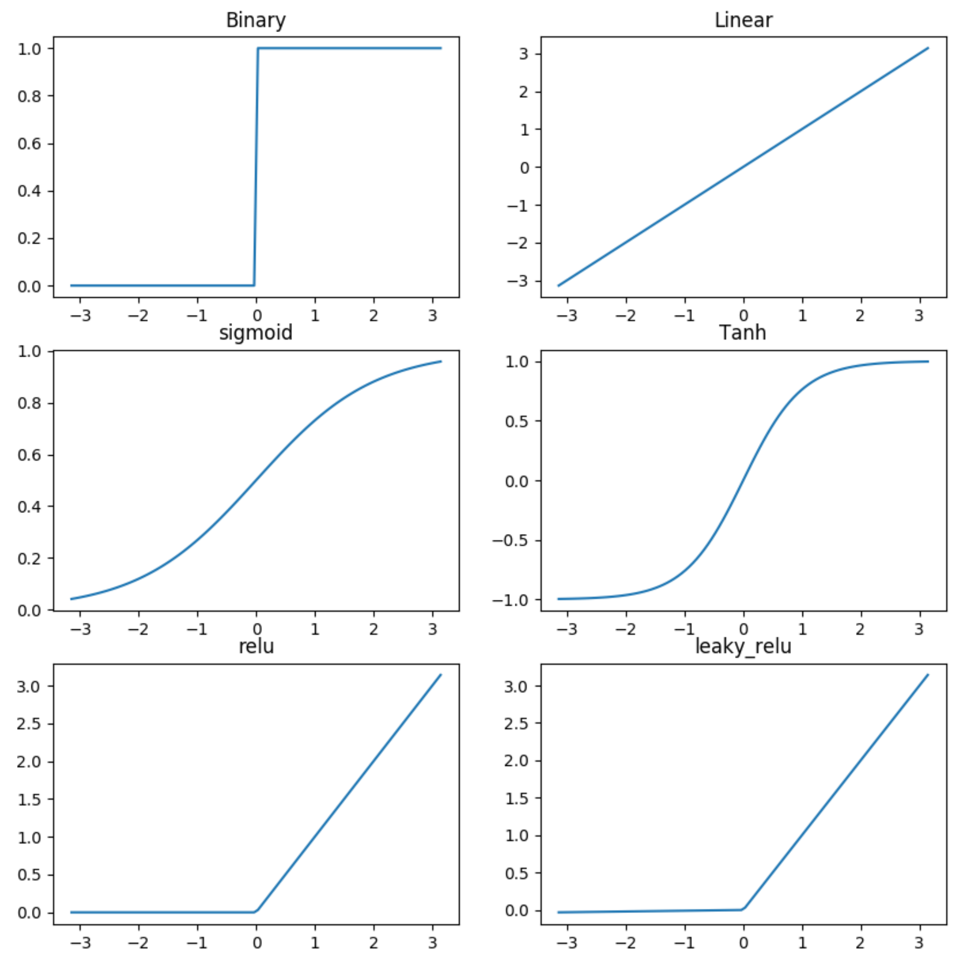

Figure 1.5: Plots of activation functions

Source: Author's own work

a predicted value $\hat{y}$ and an expected value y. If our sample data set contained more than just one observation we would calculate the mean of squared errors of all observations.

$$C = \sum \frac{1}{2}(y - \hat{y})^2 \tag{1.10}$$

Cost function for neural networks is principally the same as for other parametric models but since our goal is to find its minimum, it has to be differentiable. For that we will use a method called *gradient descend*.

1.3.2 Gradient descend

If we look closer at the cost function C, we can see that its value can change either by changing y or $\hat{y}$. We take the vector of labeled training data y as a given parameter, so we have to minimize C by changing the output $\hat{y}$. We recall that the output of the network is a function of weight and bias. So our goal is to find such combination of weights and biases that gets us the lowest possible cost.

While tackling this problem analytically (finding extremes of a function by computing derivatives of a function) seems like a logical step, it would work only on a small network. In practise, it is not feasible to find minimum of a complicated function containing thousands of variables, which is the case of today's networks with even millions of weights and biases.

A more practical approach is to use gradient descent. Although this procedure is capable of approximating minimums of complex functions with many variables, it is useful to picture a function of just one variable with a clearly formed minimum, such as a parabola. With this approach, our aim is to start at a random position and find in which direction the slope (gradient ∇C) of the function is the highest and move a bit in exactly the opposite direction. The step size (or learning rate) η is a parameter which changes by how much we "move" in each iteration. We keep repeating this process until we reach the local minimum of C. We should aim to set the size of η not too big otherwise we would "jump over" the minimum, but also not too small because that slow down the descent unnecessarily.

In more mathematical terms it means to have randomly set initial vectors of weights w and biases b and to obtain partial derivatives of the cost function with respect to given vectors $\partial C/\partial w$ and $\partial C/\partial b$. We should eventually end up in the minimum of the cost function C if we update the vector of weights to w' and biases to b' in every iteration;

$$w \quad \rightarrow \quad w' = w - \eta \frac{\partial C}{\partial w}, \tag{1.11}$$

$$b \quad \rightarrow \quad b' = b - \eta \frac{\partial C}{\partial b}. \tag{1.12}$$

One impracticality that we need to address relates to the size of the training data. If our data set of size n is too large, it would be too time consuming to apply gradient descent to every observation separately and then average them; $\nabla C = \frac{1}{n}\sum_x \nabla C$. Instead, we can utilize an approach called *stochastic gradient descend*; we can take a random sample of the training inputs $X_1, X_2, \ldots, X_m$ of size m on which we estimate the gradient ∇C. Provided the sample is large enough, the following statement will hold;

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C. \tag{1.13}$$

1.3.3 Back-propagation

Back-propagation is the backbone of a neural network training process. It enables us to compute the gradient of the network's cost function C and effectively set all weights and biases to a state, in which cost function would be minimized.

To practically illustrate this process, let us consider a setting where we sent one observation of our training data into the input layer, which was then fed forward through all layers of the network (i.e. we computed $z^l = w^l a^{l-1} + b^l$ and $a^l = f(z^l)$ for every layer l of the network) and afterward, we calculate the cost function C (e.g. a mean squared error, or just a squared error since we have only one observation).

At this point, we now have to introduce a new intermediate metric; the *error of a single neuron* δ^l_j, in this case the error of the *j-th* neuron in the *l-th* layer. This error term is calculated as a partial derivative of the cost function with respect to the given neuron's input z;

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}. \tag{1.14}$$

Analogically, we can compute the error term of the whole output layer δ^L. With the help of chain rule, we get the following equation (where f is the activation function);

$$\delta^L = \frac{\partial C}{\partial a^L_j} f'(z^L_j). \tag{1.15}$$

We can rewrite the previous equation into matrix-based form and get

$$\delta^L = (a^L - y) \odot f'(z^L), \tag{1.16}$$

where $(a^L - y)$ is just $\partial C/\partial a^L$. The symbol $\odot$ refers to the so called Hadamard product (also equivalently Schur product) and it denotes an elementwise product of two vectors or matrices of the same dimensions.

After the error of the output layer δ^L is calculated, we progressively backpropagate the error into all layers, starting from the second-to-last layer $l = L - 1$, until the second layer $l = 2$. Again, by means of the chain rule, we calculate the error term δ^l for all of these layers. We will always compute the error term only for the layer that is one step behind, so given we want to calculate δ^l, we already have to have calculated δ^{l+1};

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l), \tag{1.17}$$

where $(w^{l+1})^T$ is transposed matrix of weights of the $l+1$ layer.

Finally, we can compute the gradient of the cost function, which is given by the rate of change of the cost with respect to any weight and with respect to any bias in the network.

The rate of change of the cost function with respect to the weight w_j^l is a partial derivative of C with respect to w_j^l expressed in terms of the j-th error term in the l-th layer and the activation function of the k-th neuron in the previous layer

$$\frac{\partial C}{\partial w_j^l} = \delta_j^l. \tag{1.18}$$

The rate of change of C with respect to the bias b_j^l, is just a partial derivative of C with respect to b_j^l expressed by the error term of the same neuron

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{1.19}$$

1.4 Recurrent neural network

Recurrent neural networks (*RNNs* for short) present a more complex form of neural networks in the sense that they are not static (or stateless) as their simpler counterparts. While traditional Feed-forward networks always work with a fixed-sized vector as input and output, and assume independence of individual input vectors from each other (e.g. image recognition task - every image in the sample is independent), RNNs are capable of capturing stepwise information from the input sequence, memorizing it and utilizing it in the next iterations of the sequence. This property makes them suitable for dealing with data of sequential nature, such as text and speech (natural language processing), video (essentially, an image recognition task with an added time dimension), and time-series. [4]

In order to look closer into the recurrent nature of the network, we have to "unfold" (or unroll) it into steps and introduce a new set of variables (steps notated with a lower index t). The number of steps would equal the length of the input sequence we are working with. For example, in case of natural language processing, the steps would equal the number of words in the text. It should be noted that the following is just a simplified schematic example of a plain vanilla RNN, whose main purpose is to describe the novel features that standard feed-forward networks do not posses.

We already worked with the input variable X, now with a subscript $_t$, which adds a new dimension - time step t. Next, we need to introduce the *hidden state* (alternatively *cell state*) of the network c_t (i.e. the information that the network carries from the previous time steps onwards; practically a cell's memory). The hidden state at the time step t is composed of

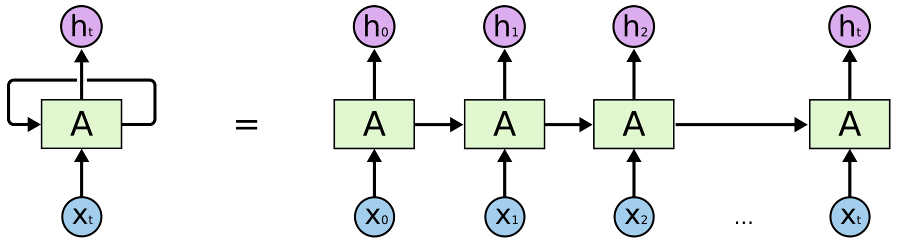

Figure 1.6: RNN unrolled in time steps

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

previous hidden state c_{t-1} and the input from the current time step x_t and is wrapped in some non-linear function f, such as sigmoid, tanh or ReLu. By this logic, hidden state c_t contains information from all previous hidden states, hence the recurrence in the name of this network

$$c_t = f(Ux_t + Wc_{t-1}). \tag{1.20}$$

Finally, the hidden state c_t, which can also be wrapped in some function appropriate to the given task gives us h_t - output at the time step t.

RNNs do not necessarily have to have the same "many to many" architecture as the one we illustrated (i.e. both input and output in form of a sequence). Depending on the task we try to solve, we could have "many to one" (e.g. having a sequence of words as input and outputting a sentiment of that text) or alternatively "one to many" architecture.

An additional difference in RNN's characteristics is that the parameters (matrices) U,V and W remain the same for all time steps. The unfolded RNN could then be compared to a very deep feed-forward neural network (i.e. with many hidden layers) with constant weights across all layers.

RNNs can also be trained with a back-propagation algorithm called *bacpropagation through time (BPTT)*. However, plain vanilla RNNs trained with BPTT suffer from a limitation called *exploding/vanishing gradient*, which plainly put, means that the back-propagated error either exponentially blows up or vanishes. As a consequence of this, it becomes very hard for any RNN to learn to hold long-term dependencies in its memory. *Long-short term memory networks* and their variations were developed directly to address this drawback. [2] [4] [5]

1.4.1 LSTM Networks

Long-short term memory (more commonly LSTM) networks' defining feature is their ability to effectively learn to remember information for long periods of time (i.e. many time steps apart). Similarly to unfolded RNNs, LSTM networks also have a chain-like structure, but the

[2]Vanishing gradient also occur in standard Feed-forward networks with non linear activation functions, whose derivative equals to zero at both extremes (e.g. sigmoid, tanh). Using more appropriate activation function and correct weight initialization helps mitigate this issue.

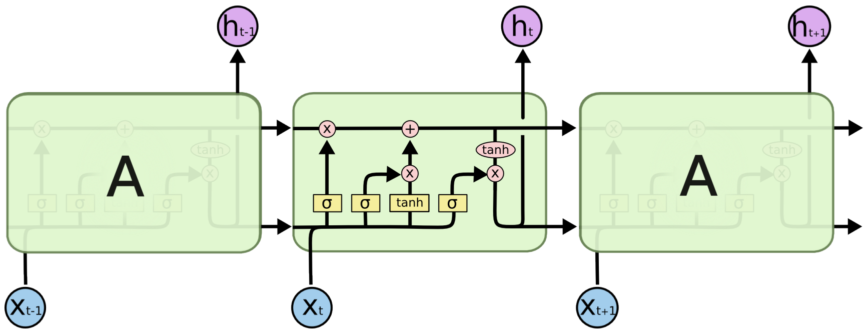

Figure 1.7: LSTM network's cell

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

structure of the recurrent unit (cell) A contains 4 gated non-linear functions, opposed to just one in RNN's recurrent unit. Those functions could also be viewed as the unit's layers and they interact not only with input x_t and previous cell's state c_{t-1}, but also with its output h_{t-1}.

Cell state c_t is an information flow which runs through all cells. Each cell can add or remove information from the flow via its gates (i.e. at every time step). We now look into the workings of these 4 gated layers in more detail.[3]

Forget gate layer f_t, as the name suggests, handles what information from the cell state is going to be removed. It uses previous cell's output h_{t-1} and current cell's input x_t as input for the sigmoid function which outputs a value between 0 and 1 for each value in the previous cell's state matrix c_{t-1}.[4]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1.21}$$

Analogically, *input gate layer* i_t uses the same parameters as the forget gate layer and produces a vector of values (forget gate activation vector), which signifies whether the element in the cell state matrix should be updated or not. Additionally, a *tanh layer* computes new candidates which should be added into the cell state in form of a vector $\tilde{C}_t$.[5]

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \tag{1.22}$$

$$\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c). \tag{1.23}$$

Current cell state C_t then consists of all of the information above; on one side the cell forgets all information from the previous cell state in the range determined by the output vector f_t,

[3]The comparison to gates is partially accurate because these layers have an option to stop or let new information to update the cell state. Since these gates use sigmoid functions whose output ranges from 0 to 1 in a continuous manner, they can be partially shut or open.

[4]Every gate multiplies its input by a learnt parameter W and adds a bias b before running it through a non-linear function like sigmoid or tanh

[5]*tanh* function is used in order to squish the values between -1 and 1.

on the other it adds new information $\tilde{C}_t$ in scaled by the vector i_t

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t. \tag{1.24}$$

Finally, in order to calculate the current cell's output h_t, we first need the output gate activation vector o_t, which is determined the same way as f_t and i_t, just with its own weight matrix W_o. o_t is used for an element wise multiplication of the cell state C_t wrapped in a *tanh* function

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \tag{1.25}$$

$$h_t = o_t \odot tanh(C_t). \tag{1.26}$$

We can mention several popular alternative versions of LSTM, which are often used both practically and in academic papers. First variant is an LSTM with peepholes, which enables some or all gates to know the cell's state. For example, if we added the peephole to the forget gate, computation of its activation vector would be

$$f_t = \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f). \tag{1.27}$$

Another variation is to merge the process of forgetting and adding new information to the cell state in a so called coupled gates. Specifically, forgetting information takes place only under the condition that new information is available to fill in the slot after it

$$C_t = f_t \odot C_{t-1} + (1 - f_t) \odot \tilde{C}_t. \tag{1.28}$$

A more differing deviation from classic LSTM presents the *Gated Recurrent Unit*, (GRU). Most notably, it uses just a single update gate instead of separate forget and input gates and merges hidden state with cell state. Academic literature has been increasingly paying more attention to GRUs recently.[6]

2. Model Selection and Evaluation

2.1 Model Performance

Generally, when building any machine learning model, our goal is to end up with a model which performs well not only on the data it has been trained on, but also on the previously unseen data. This quality is also known as *generalization*.

If we want to set the model to its optimal capacity, we first need to split the data set with labeled target variable into training and testing set, then choose an error metric and compute it on the training set (*training error*) and test set (*generalization error*). The capacity of a model is a rather broad term which describes the model's complexity. Simply put, depending on what kind of model we have in mind, this could be the degree of polynomial in a linear regression, or number of hidden units in neural network. The generalization error, as a function of the model's capacity follows a U-shaped curve, while the training error does not stop decreasing with every increment of the capacity.

Sub-optimal performance of a model can be two-fold. When the model suffers from *underfitting*, its training error is higher than it needs to be and it has not learnt enough from the training data, i.e. the model does not capture the underlying patterns with enough complexity. On the contrary, if the model captures excess information, (such as noise) from the training set, it would lack generalization power and perform badly on the previously unseen data. Such model operates in the *overfitting regime.*

Optimal capacity of a model can also be expressed with a so-called *bias-variance trade-off. Bias* (here referring to the statistical bias, not the parameter of neural networks) is a measure of the deviation between true values from the data and the output from our model. We say the model has high bias if it has low capacity and fails to capture patterns with high informative value. *Variance* measures model's ability of having consistent outputs with different data sets. If a model has high variance, it has picked not only meaningful information, but also unnecessary noise during learning and suffers from overfitting.

A model in the underfitting zone could be then interpreted as one with high bias and low variance, while an overfitted model on the opposite side of a spectrum would have high variance and low bias. If we manage to tune the model to the optimal complexity, it should have the lowest possible value for the combination of bias and variance and its generalization error would then contain only the irreducible error, which just noise in the data.[7]

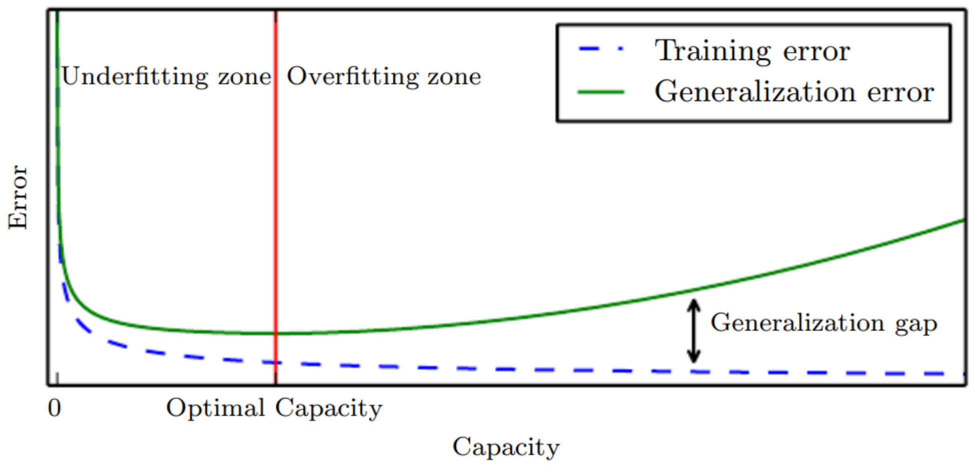

Figure 2.1: Relationship of training error and generalization error

Source: deeplearningbook.org

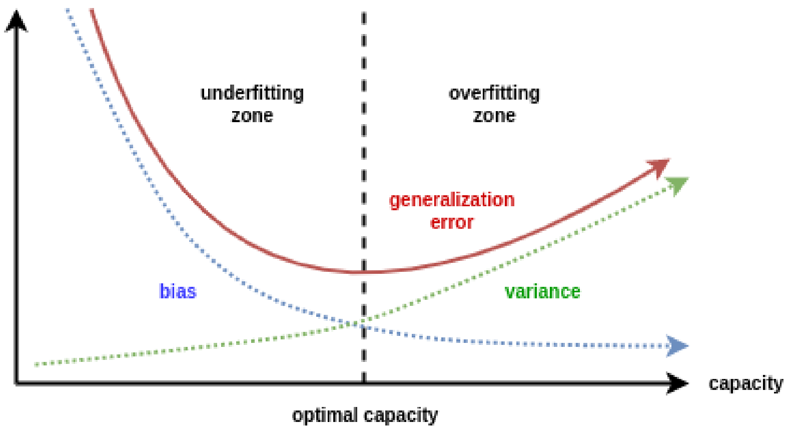

Figure 2.2: Relationship of bias, variance and generalization error

Source: deeplearningbook.org

2.2 Model Validation

2.2.1 Holdout Validation

The most primitive way to organize a data set in supervised learning is a *holdout method.* It separates the labeled data set into a training set, that is used as an input for the model. Another part of the data is withheld and does not run through the model (i.e. model is not trained on these records). This part is called the test set and a fully trained model is applied on this data set to evaluate its generalization power (alternatively, it can be called an out of sample evaluation). If we trained and evaluated the model on the same data set, we would get a very optimistic bias, due to overfitting. A common split ratio is around 2/3 in favor of the training set but can be closer to 1 with a large enough data set.

We generally work with an assumption, that our data sample's characteristics are representative of the population - the records are *independent, and identically distributed* (i.i.d. for short). This means that all observations come from the same probability distribution and are statistically independent of each other (i.i.d. does not hold with time-series data, though). Analogically, when splitting the data, it is also crucial to make sure that both sets of data display similar statistical characteristics.

An example of Two-way Holdout Validation used in a classification task can be described with the following steps:

- Randomly divide the data into a training set and a test set with an appropriate split ratio
- Choose a learning algorithm with optimally set up hyperparameters
- Fit the model onto the previously unseen data from the test set and predict class labels
- Compare predicted class labels with the "real" correct labels and calculate an error and/or generalization accuracy metrics

It is apparent that this validation method is ill-suited for a learning algorithm with externally adjustable parameters, called hyperparameters, which the model cannot learn during the training (e.g. parameter k in a KNN classifier). Their adjustment largely influences the quality of the output from the model.

Therefore, an additional block of data called the *Validation set* has to be divided and would serve as a place/ground for hyperparameter tuning. Such a method is called a Three-way Holdout Validation, or three-way split. This approach introduces another optimization task - hyperparameter optimization which typically optimizes some kind of a performance metric (e.g. accuracy). This process sits on top of the learning algorithm which tries to optimize the objective function on the training set.

The whole Three-way validation can be described as follows:

- Randomly split the data into a training set, a validation set for model selection and a

test set for the final evaluation
- Choose an array of hyperparameter settings which would be used by the learning algorithm
- Evaluate the performance of the models on the validation set and pick a model with those hyperparameters, which are associated with the best performance (the number of models to evaluate equals to the length of the array of hyperparameter settings from the previous step)
- Use an independent test set and estimate the generalization power of the chosen model

2.2.2 k-fold Cross Validation

The most common approach for both model evaluation and model selection in recent years has become the *k-fold Cross-Validation.* The main distinctive feature of this technique is its ability to use each section of the data set for both training and testing (alternatively training set and validation set, when used for model selection). k stands for the number cross-validation iterations (or folds).

First, we illustrate this method used for model validation, with $k = 5$, which gets us 5 models being tested and evaluated. This process would look like this:

- Split the data into k equally sized sections
- Leave one section out for testing purposes and fit the remaining k-1 as a training data on the model
- Fit the model on the testing set and calculate desired performance and/or error metrics
- Repeat 2nd and 3rd step $(k - 1)$ more times, each time leaving out 1 unique section for testing
- Calculate arithmetic mean of the performance metrics

Analogically to the Holdout Validation, the same assumptions about the independence and identical distribution hold (we also consider hyperparameters to be fixed). Using this evaluation method deems practical especially when working with a limited data sample. This way, we do not have to resort to reserving a substantial portion of data for evaluation purposes, leaving us with a small number of observations for the training set, which enhances pessimistic bias.

When it comes to setting up the number of folds k, there, unfortunately, is not a single universal rule applicable to all distributions, that always get us a model in the minimum of the bias-variance trade-off. A common choice in practice is usually a number between 5 and 10.

If we set $k = n$, where n is the number of observations in our data, we get a special case of the k-fold Cross-Validation - *Leave-One-Out Cross-Validation.* Although computationally demanding, this technique may come in handy with exceptionally small data sets, where every next observation withheld from the training may substantially increase the pessimistic bias.

Finally, we illustrate the application of k-fold Cross-Validation on a model selection (for clarity, we use a simple Holdout method for model evaluation):

- Split the data set into a training set and a test set
- Apply preferred hyperparameter settings to the training set (alternatively, we can use any of the hyperparameter tuning algorithms, such as grid search or random search).
- Apply Cross-Validation for each hyperparameter configuration and calculate the performance of each model
- Take the hyperparameter setting of the best performing model and train a model on the whole training set
- Evaluate the model on the test set

2.3 Regularization

Although, the most straightforward way to prevent overfitting is to provide more observations for the training data, in some cases it might be either costly or difficult in some other way to do so (medical trials, for example). At that point, we can deploy one of the various regularization techniques that are at the disposal. In general, regularization penalises some of the model's non-zero parameters in order to lower the complexity of the model.

The simplest and most widely used regularization techniques are *L1 and L2 regularization. Lasso regression* (least absolute shrinkage and selection operator) is a version of any standard cost function C augmented by an added regularization term, also called the *L1 penalty:*

$$C_{new} = C_{old} + \lambda \left\| w \right\|_1 , \tag{2.1}$$

where the L1 norm $\left\| w \right\|_1$ is can be rewritten as the sum over all absolute weight values of a weight matrix; $\sum_i \sum_j |w_{ij}|$. This regularization term is weighted by the hyperparameter λ (regularization rate), which determines how much we regularize our network. Depending on the size of λ, some weights can be zeroed under L1 regularization. Therefore, L1 is a suitable regularization technique in those cases, when we aim to compress our model, since it can push some of the weights down to zero and effectively shut off some nodes in the network.

Similarly, *Ridge regression*, also *weight decay*, adds an *L2 penalty term* to the cost function;

$$C_{new} = C_{old} + \frac{\lambda}{2} \left\| w \right\|_2 , \tag{2.2}$$

where the regularization term (L2 norm) is an Euclidian distance of the weight matrices, i.e. the sum of the squared weight values of the weight matrix; $\sum_i \sum_j w_{ij}^2$. This term is, again, multiplied by the scalar value of the λ. Using squared values will decrease the weight values proportionally to the value of λ, but will not make them vanish completely, as opposed to the L1 norm.

The third option is to combine both aforementioned penalties into one and thus use a so called *elastic net.* This regularization term contains a second hyperparameter α, which controls the balance between the L1 and L2 term. This approach is suitable, when model's features outnumber the amount of observations in our data, which is often the case with complex neural networks.

$$C_{new} = C_{old} + \alpha \frac{\lambda}{2} \|w\|_2 + (1 - \alpha)\lambda \|w\|_1 . \tag{2.3}$$

Using *dropout method* is another very straightforward approach to regularization. At every iteration during the back-propagation learning of the network, it shuts off some randomly selected nodes with their ingoing and outgoing synapses. The probability P of choosing and shutting off a given node is the an externally set hyperparameter. Since the network performs with a different set of nodes and produces different output at each iteration, this regularization can be seen as an ensamble technique. Generally, ensamble models tend to perform better than a single model, because they are capable of capturing more randomness from the data. [8]

Early stopping, as the name suggests, prevents the network from overtraining, by stopping the training when the performance on the validation set starts to worsen. The validation set is a subset of the training data on which the model's validation error is measured. Strictness of this regularization method can be adjusted with a hyperparameter called *patience* - an integer value that states how many training epochs (after the validation error starts increasing) we are willing to wait before the back-propagation learning is terminated. Setting the patience to a right level is not as trivial as it might seem; if the patience value is relatively low, we could end up terminating the learning process prematurely in a local minimum of the validation error. [9] [10]

2.4 Setting up hyperparameters

Hyperparameter optimization is an intricate subject without a universally agreed strategy to follow. However, there is a heuristic approach which should ensure relatively consistent results regardless of the data set we are working with.

Generally speaking, our starting point should be a simple network and a limited data set to allow for a speedy network training. In each iteration, we should identify which regime (underfitting vs. overfitting) the model operates in, so that we when we adjust the value of one of the hyperparameters by some margin and keep others constant, we can attribute the shift in model's capacity to the specific change of that hyperparameter. This iterative process can be automatised or done manually.

Our objective of selecting hyperparameters which achieve lowest possible generalisation error can be expressed as an optimization function with run-time and memory budget being constraints of that function. *Manual approach* requires a good understanding of the data and also hyperparameters and their real relationship to model's generalization error. Some guidelines when selecting commonly used hyperparameters are:

- A higher number of hidden units in the network increases capacity of the model, together with network's requirements on the computational power
- An optimally tuned learning rate increases capacity of the model (due to its U-shaped relationship with training error)
- A higher value of regularization term λ (e.g. weight decay) decreases the capacity, because of its shrinking effect on the values of weights
- A higher probability of the dropout decreases model's capacity

Some hyperparameters do not necessarily have to be manually determined; the number of learning epochs can be automatized with the *early stopping* regularization algorithm, while *adaptive learning rate algorithms* can help set up the learning rate. Although, these algorithms come with their own set of hyperparameters to be set.

Specifically, approaches of choosing a proper value for the learning rate are the most discussed ones in the literature, because of the significance this hyperparameter has on the model performance. Rather than setting up a global learning rate, each of the model's parameters could have its own learning rate which would be updated through the learning process. This technique is utilized by some of the adaptive learning rate algorithms such as Adam, RMSProp, or AdaGrad.

The other solution at hand is to optimize *hyperparameters automatically.* Since hyperparameter tuning can be seen as an optimization of some objective function describing model's capacity or accuracy (e.g. generalization error), we can develop an algorithm which would wrap the machine learning model and effectively hide hyperparameters from the user. This sometimes leaves the user with a decision of choosing "secondary" hyperparameters, which optimize the automatized algorithm. The two most common algorithms are *grid search* and *random search.*

Grid search algorithm sets a finite number of possible values for each of the hyperparameters and develops a model for every combination of the possible values, thus (when visualized using just 2 hyperparameters for clarity) creating something resembling a grid. The algorithm's computational requirements rise on an exponential scale when adding hyperparameters, therefore it is an appropriate candidate to consider, if we need to tune just a moderate number of them; if we wanted to select the best of 10 possible values for 5 hyperparameters, the algorithm would need to put together 10000 (10^5) models and pick one with the best score according to our criteria.

Prior to running this algorithm, we need to pick an array of possible values for each hyperparameter. First, we choose starting value and considerably distant ending value of the array, so that there is a high probability that the optimal value lies within the range. Then, we choose an appropriate spacing between the values (typically, values on a logarithmic scale).

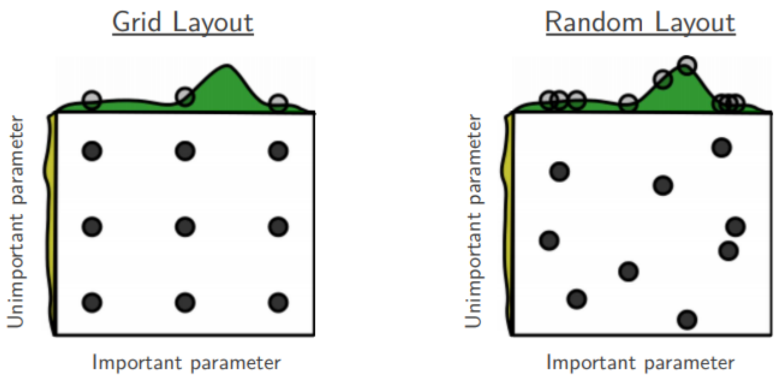

Figure 2.3: Grid search and Random search comparison

Source: deeplearningbook.org

Running a grid search repeatedly can bring better results for one of the two reasons. If we get the best performance with the value lying on the edge of the hyperparameter's range, we should re-run the search with the range shifted to that side. Otherwise, if the best performing model had hyperparameter's value from within the range, we should use a narrower range in the next iteration.

Random search is an alternative algorithm, which according to literature, achieves better results in a faster manner compared to the grid search. Prior to running the algorithm, we need to define marginal distributions fer every hyperparameter (e.g. Bernoulli for binary variables and a log-scale uniform distribution for positive real values) on an interval (a,b) analogically to grid search. From each distribution, a random value is picked and combined with randomly picked values for every other hyperparameter. This is done n times, where n is a desired number of joint hyperparameter configurations. Same as with grid search, multiple runs of the algorithm are required for a more refined results.

We can illustrate not only higher effectivity, but also better results achieved using random search compared to the grid search on the following example. In this scenario, we optimize just 2 hyperparameters; one having a large impact on the model's valuation error and one having a negligible effect. With grid search, if we tried just a combination of 3 values for each hyperparameter (i.e. 9 in total), we would end up performing 2/3 of them pointlessly, because just a change in the important hyperparameter would reflect in model's performance. However, given the nature of the random search, no 2 hyperparameter value combinations would with high probability have the same values for the given hyperparameter, thus with a comparable computational complexity, random search would test not 3, but 9 unique values for the important hyperparameter (figure 2.3). [11]

3. Time series analysis

Time series is a sequence of data indexed in time order. Mathematically, it is usually defined as a vector (or a set of vectors) $X_t, t = 0,1,2...,n$, where t is the time elapsed. If the observations of the vector X_t are exactly determined by a mathematical formula, we consider it to be a deterministic series. If future values can be described only by their probability distribution, the series is said to be a stochastic process. If the successive data points are equally spaced in time, it is a sequence of discrete time data. A time series is uni-variate if it contains records of a single variable, or multi-variate if it takes more than one variable. [12]

Four major elements can be identified in a time series through additive or multiplicative decomposition:

$$Y_t = S_t + C_t + T_t + \epsilon_t, \tag{3.1}$$

$$Y_t = S_t \times C_t \times T_t \times \epsilon_t, \tag{3.2}$$

where S_t = Seasonal component, C_t = Cyclical component, T = Trend component, and ϵ_t = Stochastic (unexplained) component. The additive decomposition is the most appropriate if the variation around the trend-cycle does not vary with the level of the time series, whereas multiplicative decomposition is more appropriate when variation appears proportional to the level of the series.
Time series can be considered *stationary* or *non-stationary.* In stationary time series we further distinguish between *Weak stationarity* and *Strict stationarity.* A time series is said to be weakly stationary (or covariance-stationary) if the the first moment (mean) is *time invariant*, and if autocorrelation depends only on the interval k, not on time itself. The second moment (variance) is finite for all times.

$$E(Y_t) = E[Y_{t-1}] = \mu \qquad \forall t, \tag{3.3}$$

$$Var(Y_t) = \gamma_0 < \infty \qquad \forall t, \tag{3.4}$$

$$Cov(Y_t,Y_{t-k}) = \gamma_k \qquad \forall t, \forall k. \tag{3.5}$$

Time series x_t is stationary in the strict sense if the statistical distribution of the series remains unchanged for any shift in time (i.e. $x_{t_1}, \ldots ,x_{t_n}$ and $x_{t_1+m}, \ldots ,x_{t_n+m}$ have the same statistical distribution for every t_i, m).

Stationary time series have a mean reverting property and are desired when fitting the data or forecasting new ones. However, real world data, especially of financial nature tend to not materialize this characteristic on their own. Then, it is necessary to make the series stationary usually by differencing or de-trending the time series.[13]

3.1 Autoregressive (AR) Models

In *Autoregressive models*, we forecast the variable by a linear combination of its own past values. It can be viewed as a modified version of a multiple regression, where exogenous predictors are substituted by lagged values of the explained variable.

We say that an *AR(p) model*, is an autoregressive model of order p and is formulated as:

$$\begin{aligned} y_t &= c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t && (3.6) \\ &= c + \sum_{i=1}^{p} \phi_i y_{t-i} + \varepsilon_t && (3.7) \end{aligned}$$

where c is a constant, $\phi_1, \ldots, \phi_p$ are the parameters of the model ($\phi_i \in \mathbb{R}$) and ε_t is white noise ($\varepsilon_t \sim N(0,\sigma^2)$).

By introducing a *Backward Shift Operator*[1] $\mathbf{B}$ the AR(p) process can be notated as a function θ of $\mathbf{B}$:

$$\begin{aligned} y_t &= c + (1 - \phi_1\mathbf{B} - \phi_2\mathbf{B}^2 - \ldots - \phi_p\mathbf{B^p})y_t + \varepsilon_t, && (3.8) \\ \theta_p(\mathbf{B})y_t &= c + \varepsilon_t && (3.9) \end{aligned}$$

The order p of the autoregression materializes in the number of lagged values of the series that are used in the model.

3.1.1 AR(1) process

AR(1) - first-order autoregressive model - is a process regressed only on the most recent lagged value y_{t-1}. It is defined as

$$y_t = c + \phi y_{t-1} + \varepsilon_t. \qquad (3.10)$$

AR(1) process shows different statistical properties depending on the value of the parameter ϕ and c in the given process and thus the underlying time series y_t can be characterized as:

- white noise - if $\phi = 0$ and $c = 0$ (essentially AR(0)),
- a random walk - if $\phi = 1$ and $c = 0$,
- a random walk with a drift - if $\phi = 1$ and $c \neq 0$.

This process can be characterized as weakly stationary if the following constraint is met - $|\phi| < 1$. For an AR(2) process to be weakly stationary, the following must hold - $|\phi_2| < 1$, $\phi_1 + \phi_2 < 1$, $\phi_2 - \phi_1 < 1$. However, conditions for stationarity in higher-order AR models cannot be as easily formulated.

[1]The backshift notation $\mathbf{B}$ is a way of converting a variable at time t to the same variable lagged (i.e. backshifted) one time unit; $\mathbf{B}y_t = y_{t-1}$. Variable y repeatedly backshifted n times is defined as: $\mathbf{B}^n y_t = y_{t-n}$.

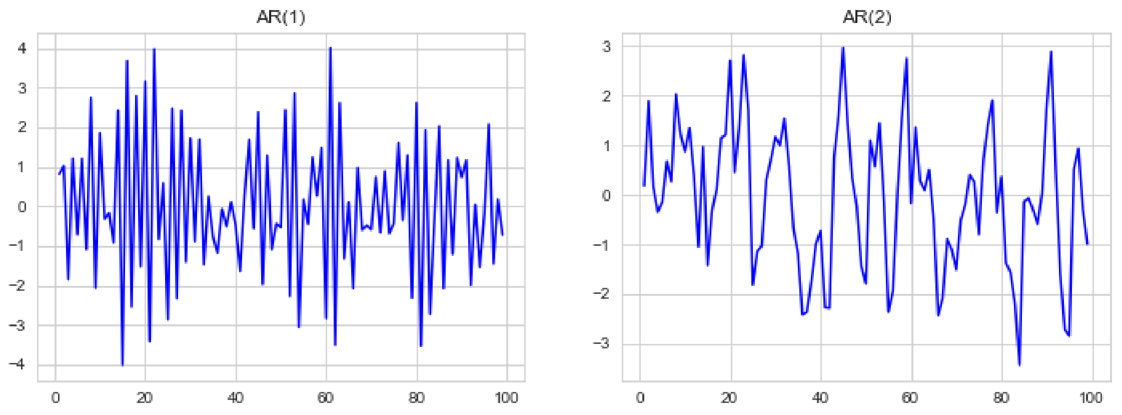

Figure 3.1: An illustrative AR(1) and AR(2) process[2]

Source: Author's own work

3.2 Moving Average (MA) Models

Moving Average (MA) models, in parallel with the AR models, are kind of a multiple regression with the main difference being the utilization of past forecast errors as predictors. Put in a different perspective, MA models incorporate the past white noise shocks ε directly in them, whereas in AR model, they are referred to only indirectly in form of previous terms of the series y_t. Because of this distinction, a particular MA(q) model takes into account only the last q shocks, while AR models (regardless of their order) include all previous shocks. The formula for MA(q) - moving average of order q can be written as:

$$y_t \quad = \quad c + \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + \cdots + \theta_q\varepsilon_{t-q}, \tag{3.11}$$

where c is a constant, $\theta_1, \ldots, \theta_p$ are the parameters of the model ($\theta_i \in \mathbb{R}$) and ε_t is white noise ($\varepsilon_t \sim N(0,\sigma^2)$). Expressed through a Backward Shift Operator, MA(q) can be rewritten as

$$y_t = c + (1 + \theta_1\mathbf{B} + \theta_2\mathbf{B}^2 + \ldots + \theta_q\mathbf{B}^q)\varepsilon_t \tag{3.12}$$

Any stationary AR(p) model can be rewritten as a MA of infinity-order by recursively substituting lagged observation y_{t-1} with $\phi_1 y_{t-2} + \varepsilon_{t-1}$, and y_{t-2} with $\phi_1 y_{t-3} + \varepsilon_{t-2}$, and so on.

$$\begin{aligned} y_t &= \phi_1 y_{t-1} + \varepsilon_t = \phi_1(\phi_1 y_{t-2} + \varepsilon_{t-1}) + \varepsilon_t & (3.13)\\ &= \phi_1^2 y_{t-2} + \phi_1\varepsilon_{t-1} + \varepsilon_t & (3.14) \end{aligned}$$

We recall that an AR(1) process is stationary if $|\phi| < 1$, so with increasing value of k the value of ϕ_1^k would lessen, which eventually would lead to a MA(∞):

$$y_t = \varepsilon_t + \phi_1\varepsilon_{t-1} + \phi_1^2\varepsilon_{t-2} + \phi_1^3\varepsilon_{t-3} + \cdots . \tag{3.15}$$

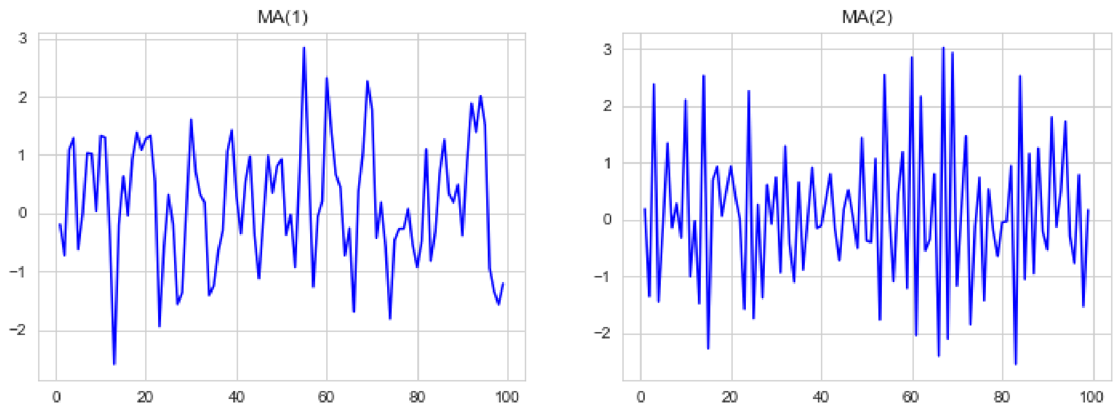

Figure 3.2: An illustrative MA(1) and MA(2) process [3]

Source: Author's own work

3.3 Autoregressive Integrated Moving Average (ARIMA) Models

A non-seasonal *ARIMA* model a generalized version of *Autoregressive Moving Average Model* (ARMA), which is a combination of the polynomials representing both Autoregressive and Moving Average models. A time series model is an ARMA of order p,q if:

$$y_t = c + \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t, \tag{3.16}$$

where c is a constant, $\theta_1, \ldots, \theta_p$ and $\phi_1, \ldots, \phi_p$ are the parameters of the model ($\theta_i, \phi_i \in \mathbb{R}$) and ε_t is white noise ($\varepsilon_t \sim N(0,\sigma^2)$).

A generalization of ARIMA is hidden in the acronym "I" (*integrated*), which enables these models to be applied onto non-stationary time series, as well. Using an integrated time series y_t of order d ($I(d)$) means to take d^{th} difference of the series. Generally, second-order differencing is sufficient for stationarizing any time series. A differenced series y_t is calculated as change between consecutive observations of the series, for all but first observation ($T-1$):

$$y'_t = y_t - y_{t-1}. \tag{3.17}$$

Thus, an *ARIMA* (p,d,q), where p is the order of AR, d is the degree of differencing and q is the order of MA part, can be written as:

$$y'_t = c + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t. \tag{3.18}$$

Converted to the backshift notation we get:

$$\phi_p(\mathbf{B})(1-\mathbf{B})^d y_t = c + \theta_q(\mathbf{B})\varepsilon_t, \tag{3.19}$$

where:

$$\text{AR(p)} = \phi_p(\mathbf{B}) = (1 - \phi_1 B - \cdots - \phi_p B^p), \tag{3.20}$$

$$\text{I(d)} = (1-\mathbf{B})^d y_t, \tag{3.21}$$

$$\text{MA(q)} = \theta_q(\mathbf{B})\varepsilon_t = (1 + \theta_1 B + \cdots + \theta_q B^q)\varepsilon_t. \tag{3.22}$$

3.4 GARCH models

Generalised Autogressive Conditional Heteroskedastic (GARCH) models and its variants are suitable for time series which display signs of *volatility clustering* - an exhibition of volatility that is not constant in time and is dependent (conditional) on previous time periods of elevated volatility[4], since ARIMA class of models do not directly account for this statistical property. This phenomenon is especially present in financial time series, such as stock returns. [14] [15]

Autoregressive Conditional Heteroskedastic (ARCH) Model can be seen as an application of an autoregressive process on the variance. Its application is eligible mainly on residuals of a time series on which an appropriate model (e.g. ARIMA) was fitted beforehand and left the residuals which can be characterized as discrete white noise.

Its simplest variant ARCH(1) - Autoregressive Conditional Heteroskedastic Model of order unity can be represented as

$$\epsilon_t = z_t\sqrt{\alpha_0 + \alpha_1\epsilon_{t-1}^2}, \tag{3.23}$$

where ϵ_t is a series of residuals (i.e. error terms) with respect to a mean process and can be further decomposed into a stochastic piece z_t, which is a white noise with zero mean and variance equal to 1, and a time dependent standard deviation σ_t:

$$\epsilon_t = \sigma_t z_t. \tag{3.24}$$

σ_t^2 is given by:

$$\sigma_t^2 = \alpha_0 + \alpha_1\epsilon_{t-1}^2, \tag{3.25}$$

where α_0, α_1 are parameters of the model.

Extending this model to an arbitrary (q) number of lag squared residual errors to be included in the model, an ARCH(q) takes the following form:

$$\epsilon_t = z_t\sqrt{\alpha_0 + \sum_{i=1}^{q}\alpha_i\epsilon_{t-i}^2}, \tag{3.26}$$

where parameters $a_0 > 0$ and $a_i \geq 0$, $i > 0$.

An extension of ARCH model is *Generalized Autoregressive Conditional Heteroskedasticity* which adds a moving average component to the autoregressive compponent of the ARCH model. GARCH (p,q) model, where p is the order of the GARCH terms σ^2 and q is the order of the ARCH terms ϵ^2:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q}\alpha_i\epsilon_{t-i}^2 + \sum_{j=1}^{p}\beta_j\sigma_{t-j}^2. \tag{3.27}$$

[4]A statistical term that characterizes this behaviour is *conditional heteroskedasticity.*

3.5 Box-Jenkins Methodology

Box-Jenkins Analysis is a systematic iterative process for correctly identifying, fitting and validating ARIMA models developed by George Box and Gwilym Jenkins in 1976. It consists of the following 3 steps:

1. *Identification* - determining the order of the ARIMA model (values of p,d, and q).
2. *Estimation and Selection* - estimating the parameters of the model specified in the previous step.
3. *Diagnostic Checking* - determining whether the selected model is adequate.[16]

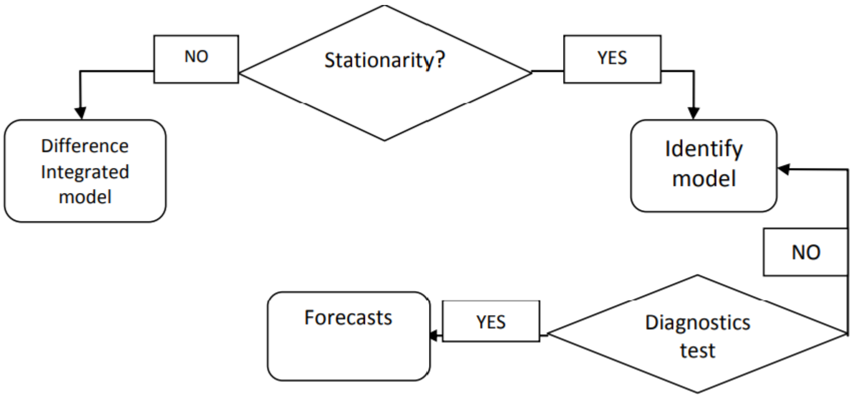

Figure 3.3: Logical Scheme of Box-Jenkins Methodology

Source: Din, Marilena (2015)[17]

3.5.1 Identification

First step involves the following objectives; determining the stationarity of the time series (d), identifying the presence of a seasonal component and the ARMA order (p,q).

Stationarity can be recognized visually in two ways. First, a broad estimate of stationarity can be obtained from a run-sequence plot of the time series. Second way involves plotting autocorrelation function (ACF) and partial autocorrelation function (PACF) over the data.[5] Non-stationarity can be confirmed if the ACF decreases to zero slowly or does not decrease at all with increasing lags. Then, it is suggested either to remove the trend component or to difference the series, which should remove the number of large autocorrelations. However, if the differenced series still does not appear stationary a second-difference ($d = 2$) needs to be used.

[5]Autocorrelation is correlation for time series observations with the previous time steps, called *lags*. Autocorrelation Function (ACF) displays the autocorrelation of a time series by lags. Partial Autocorrelation Function (PACF) gives, broadly speaking, a correlation of a time series with its own lagged values, regressed the values of the time series at all shorter lags.

Table 3.1: Characteristics of ARMA models using ACF and PACF

Source: Florian Pelgrin, Box-Jenkins methodology

	AR(p)	**MA(q)**	**ARMA(p,q)**
ACF	Tails off	Cuts off after q	Tails off
PACF	Cuts off after p	Tails off	Tails off

Unit root tests are another method to demonstrate stationarity. *Unit root* is a feature of some stochastic processes that makes them non-stationary. On a simple AR(1) model

$$y_t = \gamma y_{t-1} + \mu_t, \tag{3.28}$$

where y_t is the variable of interest at the time t, and μ_t the error term. If the coefficient $\gamma = 1$, the model would be considered non-stationary - with a unit root.

Identifying the order of AR and MA (p, q) can be determined by examining the ACF and PACF graphical plots. If the ACF exponentially decays to zero, we have an indication of an AR model. Alternatively, alternating positive and negative significant autocorrelations signal the same thing. Next we use a PACF to determine the order p of the AR, which would be equal to the number of lags above the confidence band.

If the ACF shows one or more significant spikes with the subsequent lags equal to zero (or within the confidence band), it showcases an MA model. The order q depends on the number of lags after which the autocorrelation vanishes. A rather slow decay of autocorrelation with added lags is a sign of a mixed autoregressive and moving average (ARMA) model.

If the valus of autocorrelation repeatedly spike up after a set number of lags, the time series contains seasonal autoregressive terms. Other two extreme cases are white noise (no significant autocorrelations) and non-stationarity (practically no decay in autoregression).[18]

3.5.2 Estimation and Selection

This step's objective is to estimate and evaluate the parameters of the model we identified in the previous step. It can be done using several different methods, such as *Linear least squares, Maximum likelihood, Generalized method of moments*, etc.

These methods differ in terms of assumptions, objective functions and other characteristic (i.e. Linear least squares method aims at minimizing the sum, while Maximum likelihood estimation aims at maximizing the likelihood function, etc). However, any more detailed explanation of the estimation methods is beyond the scope of this work.

Measuring *Information criteria* is a way of assessing the *Goodness-of-fit* of the model, which essentially means measuring the variance of the residuals:

$$\hat{\sigma}(k) = \frac{1}{T}\sum_{t=1}^{T}\hat{\epsilon}_t^2, \tag{3.29}$$

where k is the total number of estimated parameters, T is the number of observations, $\hat{\epsilon}_t$ is the adjusted residual at time t.

Akaike information criterion (AIC) developed by Hirotugu Akaike is a relative measure of the fitting quality obtained by a statistical model. Specifically, AIC estimates the relative amount of information lost by a given model on the forecasted (out of sample) data. Its mathematical formula is

$$AIC(k) = -2\log(L) + 2k \tag{3.30}$$

where k is the number of parameters and L is the maximized value of the likelihood function. AIC is based on the normality assumption and may return more than one minimum (because of using a single parameter k instead of p,q), but is generally considered to be efficient.

(Schwarz) Bayesian information criterion (BIC, or also SBIC, SIC), an information criterion closely related to AIC, is formulated as

$$BIC(k) = -2\log(L) + k\log(n) \tag{3.31}$$

where k is the number of parameters estimated. It is apparent from the definition that BIC applies a stricter penalty compared to AIC, therefore it tends to favor models of lower order. It is also more consistent in selecting p and q with data sample size approaching infinity.

When selecting out of a finite number of models with either one of these ICs, a model with the lowest value is preferred. Both BIC and AIC have a penalty term, which rises proportionally to the complexity of the model (by complexity we refer to the number of parameters in the model). Put the other way around, these ICs deal with the trade-off between goodness-of-fit and the simplicity of the model (i.e. under and over-fitting). [19]

3.5.3 Diagnostic Checking

The final step before generating forecasts is the diagnostic checking of the model; its main objective is to check the adequacy of the model selected in the previous steps. This is done by examining the statistical properties of the residuals. Particularly, the normality assumption should be met and no systematic pattern should be observable.

Normality of residuals can be examined with *Q-Q plots*, which help us assess if a set of data plausibly came from some theoretical distribution. A Q-Q plot is a kind of scatterplot which is created by plotting two sets of quantiles against one another (theoretical and sample quantiles). The premise behind them is the following; if both sets of quantiles come from the same distribution, the scatterplot should form a straight diagonal line.

The latter can be also visually checked by studying the autocorrelation plots of the residuals and checking, whether any further autocorrelation can be found. The values of p and q need to be adjusted, unless all autocorrelations and partial autocorrelations are within the bounds of the confidence interval (i.e. contain no additional structure). [20] [12]

4. Literature review

A significant amount of effort has been dedicated to cover the prediction of asset price movement using machine learning in the past decades. Using more complex and computationally more advanced methods in the form of deep learning algorithms is a relatively novel approach. Nevertheless, academic papers regarding this approach are being published regularly.

The literature can be characterized and segmented by different qualitative metrics, such as:

- type of asset (i.e. stocks, index, commodities, forex pairs, cryptocurrency, etc.),
- deep learning algorithms (i.e. LSTM, GRU, Convolutional Neural Networks, etc.),
- type of prediction (i.e. price or trend prediction),
- horizon of prediction and data frequency (i.e. intra-day, end-of-day, monthly data, etc.)
- kind of data (i.e. only price data, or price data supplemented with firm characteristics and/or macroeconomic indicators, etc.),
- other factors (time frame in scope, performance criteria, technical environment, etc).

[21]

Lee, Sang and Yoo, Seong (2018) propose a data-driven portfolio construction based on future returns predicted by different DL models.

The data set consisted of the top 10 performing stocks on Standard Poor's 500 index (SP500) from 1997 to 2016. Monthly normalized price data (open price, high price, low price, adjust close price, and volume) were used as features for forecasting month-ahead returns. Outputs from 3 different variations of RNNs (RNN, LSTM, GRU) that were constructed, were then evaluated by 2 performance measures (hit ratio and monthly returns). Predictions from LSTM, which outperformed the other two DL models were used for constructing Threshold-based Portfolios (TBPs). The final LSTM was trained by back-propagation algorithm minimizing the quadratic loss L and its hyperparameters were optimized via grid search on a validation set. The topology of the network consisted of an input layer, a hidden recurrent layer of 36 neurons, a 50% dropout layer and an output layer with linear activation function.

TBPs proved promising in portfolio management construction, as their construction is purely data-driven and requires no prior financial knowledge or expertise.[22]

Chong, Eunsuk and Han, Chulwoo and Park, Frank (2017) use 3 different data representation methods (PCA, autoencoder and restricted Boltzmann machines) on lagged returns of 38 different stocks from Korean Stock market between years 2010 and 2014 and get 7 different sets of features.

Each data set consisting of selected features was used as input for a three-layer deep neural network (DNN) and its performance compared to a benchmark model (linear AR). Authors considered 4 different performance metrics - normalized mean squared error (NMSE), root mean squared error (RMSE), mean absolute error (MAE), and mutual information (MI).

Constructed DNNs outperformed the benchmark only on training set, however, DNNs managed enhance the performance capacity when applied on residuals of the AR model.[23]

Working paper by John Albergis and Zachary C. Lipton (2017) is based on a premise that in the long run, the company's intrinsic value must be reflected in the stock price. More concretely, on an approach called value investing, which states that the best predictors of the intrinsic value are currently known fundamental data of the company (e.g. assets, income, book-to-market, etc.).

However, rather than using currently published company fundamentals, the authors constructed several deep learning models that predicted future values of these indicators which then served as an input for stock price prediction. The authors gathered monthly data from 1970 to 2017 of 11815 stocks of US-based companies listed on major US stock exchanges whose market capitalization exceeded \$100M. A total of 20 input features were considered for each stock (11 balance sheet features from the most recent quarter, 5 income statement features calculated as trailing 12 months and 4 stock price movement features of different lengths). A multilayer perceptron (MLP) and variations of RNN (both GRU and LSTM) were fitted onto the training sample (1970-1999). Out-of-sample validation (2000-2017) using MSE as a performance measure resulted in the best score for MLP.

Predicted values of the fundamentals were consequently used in a trading simulation and scored better than traditional factor models utilizing only currently known information.[24]

Yumo Xu, Shay B. Cohen (2018) choose a less conservative approach that has been gaining more and more attention in recent years. They submit a model called StockNet which predicts future stock movement based on historical prices and sentiment analysis from relevant social media posts (tweets).

StockNet is an RNN based model and can be decomposed into 3 separate components (Market Information Encoder, Variational Movement Decoder, and Attentive Temporal Auxiliary). The authors ran the model on 88 stocks from 9 industries dated from the start of 2014 until the last day of 2015. The model used a 5-day lag window to construct the data set and output a binary value which indicates the future price movement. Its performance was determined by Accuracy and Matthews Correlation Coefficient (MCC).[1]

Besides the fully equipped model, authors created 4 inferior instances of the StockNet (e.g. using only historical prices or using only tweet information, etc.) and additionally compared the performance against 5 other baseline models (a random naive movement predictor, Autore-

[1]MCC is a measure of the quality of a binary classification that is computed from the confusion matrix.

gressive Integrated Moving Average, a discriminative Random Forest classifier, a generative topic model jointly learning topics and sentiment, a discriminative deep neural network with hierarchical attention). The full-fledged StockNet model achieved the highest scores in both performance metrics.[25]

Working paper of Fischer, T., Krauss, C. (2018) predicts stock price movement with an LSTM network and compares it to 3 other models (random forest, logistic regression, and a deep neural network) on the stocks listed on the SP 500 index.

The time range of the data spans between the years 1992 and 2015. Daily price development is transformed into a standardized returns, which is used both. Data is separated into 23 rolling blocks consisting of 1000 entries. Each block is further split into a 750-day long (approximately 3 years) training and a 250-day long validation block. The blocks are composed in a way that no two validation blocks overlap.

The LSTM uses a look-back input vector of 240 previous entries, has one hidden LSTM layer with 25 neurons and an output layer with two neurons and softmax activation function, which forecasts a single value for the next entry. Both dropout layer and an early stopping algorithm are included as regularization measures. All models forecast for each entry the probability that a given stock will out/under-perform the cross-sectional median of returns in the given entry. At each time (entry), all stocks are ranked in descending order. The higher the stock in the ranking, the more undervalued it is. This information is subsequently used in building portfolios of $2k$ stocks ($k \in \{10, 50, 100, 150, 200\}$). k top stocks are longed and k worst stocks are shorted. Sharpe ratio and average daily returns were selected as performance metrics.

The LSTM achieved positive average daily returns before subtracting transaction costs at 0.46% for $k = 10$. Other models, besides the random forest, obtained substantially lower daily returns. In terms of the Shape ratio, LSTM achieved the highest scores for $k \leq 100$. Random forest managed to perform slightly better with a higher number of assets.[26]

5. Methodology

5.1 Hypothesis

The following chapters describe the process of model construction, fitting on the selected stock returns data, comparison and back-testing. The scope of the thesis consists of building 3 forecasting models in total - an ARIMA-GARCH model, a univariate LSTM network and a multivariate LSTM network. The main aim is to determine whether any of the proposed LSTM networks can outperform the baseline ARIMA-GARCH model in terms of its predictive ability.

The models are constructed to forecast the next-day's return. These are binarized and interpreted as trading signals. Each model's short-term forecasting power is back-tested on the trading period (i.e. test set) and compared in terms of cumulative returns and other performance metrics against the Buy and Hold baseline strategy and against each other.[1]

5.2 Data and software

The data chosen for this project consist of stock price developments of the 5 highest ranked components of the Standard & Poors 500 index in the Market Capitalisation metric (as of April 20, 2020) that have been publicly traded on January 1, 2011 or earlier.[2] Such well-known companies are picked on purpose, since the additional company data (fundamentals) are also easily accessible.

Table 5.1: Components of S&P 500 with the highest Market Cap

Company	**Ticker**	**Sector**
Microsoft Corporation	MSFT	Technology Services
Apple Inc.	APPL	Electronic Technology
Amazon Com Inc.	AMZN	Retail Trade
Alphabet (Class C)	GOOG	Technology Services
Johnson & Johnson	JNJ	Healthcare

We use Adjusted Close Price of the selected stocks as the primary input data. The data were limited to a time frame starting January 1, 2011 and ending December 31, 2018, which makes a total of 2011 daily observations (approximately 250 trading days each year). First 7 years (from January 1, 2011 to December 31, 2017) are used as an in-sample set and the remaining year (251 observations) as an out-of-sample set, i.e. trading period on which the

[1]From this chapter onward, we speak of Log Returns when mentioning Returns.

[2]A given ticker could be chosen only if the fundamentals for it were available in the data source used in this work for the whole time period in scope.

back-testing is performed. In LSTM models we further split the in-sample set into a 5 years long train set and a 2 years long validation set.

The data are obtained through a python API called *SimFin*, which provides data sets of daily share prices of US market companies, their Income Statements and also Balance Sheets. The free version is limited to the data with a 12 month delay, which presents no significant issue for the tasks in our scope.

The practical part of the thesis is carried out exclusively on *Python 3.7* running on Anaconda environment. *Pandas and NumPy* are utilized for data pre-processing, time-series forecasting ARIMA-GARCH model is built on statistical libraries *statsmodels, arch and scipy.* For LSTM networks construction, *TensorFlow* and a high-level API for neural network construction *Keras* are primarily used. A CPU with 16 GB of RAM was was sufficiently powerful for processing the data and training the networks, although the overall training time was of the order of several hours. Finally, *Matplolib* and *seaborn* are used for plotting the data. Additionally, the Diebold-Mariano test was carried out in *R* language, because of the lack of any functional with the test in python environment.

5.2.1 Additional data sets

Besides daily prices, we utilize information from two other data sets in the multivariate LSTM model, one being the Balance sheet of a given company and the other its Income sheet. Concretely, we create a multitude of derived technical, valuation, volume and growth indicators. For both sets, quarterly reports, being the most granular available, are utilized. All financial statements are provided with a report date, but since the reports are made available to the public after some time, we add a 30 day lag to the report date to account for that.

5.3 Data transformations

5.3.1 Stock returns

Returns (also *Rate of return*) present a unit-less measure of the investment performance of the investment asset. *Gross returns* R_t between two time periods $t, t-1$ can be calculated as:

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1, \tag{5.1}$$

where P_t is the asset price at time t.

Log returns r_t over the same time period are defined as:

$$r_t = \log \frac{P_t}{P_{t-1}} = \log P_t - \log P_{t-1}, \tag{5.2}$$

Table 5.2: Sample of the firm characteristics considered in multivariate LSTM model

Indicator	**Definition**
Current Ratio	$\frac{Current\ assets}{Current\ liabilities}$
Gross Profit Margin	$\frac{Net\ sales - Cost\ of\ Goods\ Sold}{Net\ sales}$
Net Profit Margin	$\frac{Net\ income}{Revenue}$
Quick Ratio	$\frac{Current\ assets - Inventories - Prepaid\ expenses}{Current\ liabilities}$
Return on Assets	$\frac{Net\ income}{Total\ assets}$
Return on Equity	$\frac{Net\ income}{Avg\ Shareholders'\ equity}$
Return on Research Capital	$\frac{Current\ Year\ Gross\ Profit}{Prev.\ Year\ R\&D\ Expenditures}$
Market Cap	$Total\ shares * Market\ Value\ Per\ Share$
Price-To-Book Ratio	$\frac{Market\ Value\ Per\ Share}{Book\ Value\ Per\ Share}$
Price-To-Earnings Ratio	$\frac{Market\ Value\ Per\ Share}{Earnings\ Per\ Share}$
Price-To-Free Cash Flow Ratio	$\frac{Market\ Value\ Per\ Share}{Free\ Cash\ Flow\ Per\ Share}$
Price-to-Sales Ratio	$\frac{Market\ Value\ Per\ Share}{Sales\ Per\ Share}$

where $\log P_t$ is a natural logarithm of the price P at time t.

There are several reasons for preferring log returns before gross returns. First, if we assume that prices are log-normally distributed, then log returns would be of a normal distribution. This supports one of the main premises of finance that price dynamics of stocks follow a geometric Brownian motion. Second, for small returns (e.g. trades with short holding duration) we can approximate that log returns are close to its raw values - *approximate raw-log equality*:

$$log(1 + r) \approx r, r \ll 1. \tag{5.3}$$

Third, we benefit from the *time-additivity* property of the log returns. When calculating the compounding return from the raw returns, we must create a product of the returns over the given period of trades. The product of normally distributed variables is, however, no longer normal per the probability theory.

$$log(1 + r_1) + log(1 + r_2) + ... + log(1 + r_n) = \prod_i (1 + r_i). \tag{5.4}$$

When compounding log returns, we perform a summation which remains normally distributed, given that returns are independent. The following equation displays that the compound return over n periods can be reformulated simply as the difference between logs of the first and last period in scope. This also reduces the algorithmic complexity of the task.[27] [28]

$$\sum_i log(1 + r_i) = log(1 + r_1) + log(1 + r_2) + ... + log(1 + r_n) = log(p_n) - log(p_0). \tag{5.5}$$

5.3.2 Feature Scaling

Feature scaling is one of data pre-processing methods used for modifying the range of values for a certain variable. It is a crucial condition for optimal functioning of almost all machine learning algorithms like k-nearest neighbors, SVMs, Principal Component Analysis, etc. [3] Feature scaling is a necessary pre-processing step when building neural networks, mainly because it prevents the neurons from saturating too early during the parameter learning process. The two most common approaches are *Min-Max scaling* and *Standardization (Z-score normalization).*

Standardization is an approach, which results in a variable with the following statistical properties:

$$\mu = 0,\ \sigma = 1, \tag{5.6}$$

where μ is the mean and σ the standard deviation from the mean. The standard scores are calculated with the following formula:

$$z = \frac{x - \mu}{\sigma} \tag{5.7}$$

Min-Max scaling is a method which scales the variable to a desired range - most common are intervals from 0 to 1, or from -1 to 1. Both intervals are suitable for neural networks and the choice can be based upon the activation function that is used in the neurons. A potential downside is that having bounded range results in a suppressed standard deviation, which can diminish the effects of outliers in the data.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{5.8}$$

Regardless of the chosen standardization method, we should pay attention to limiting data from which the standardization scaler is created on only the insample data, otherwise information from out-of-sample data would leak into the model.[29]

[3]Decision trees are a notable exception that can usually handle raw input variables.

5.3.3 Feature selection

Feature selection is a process of removing irrelevant features from the input data set. It can help reduce overfitting, shorten the training time and heighten the interpretability and accuracy of the model. There are three main principles that can be applied in feature selection - *Filter method, Wrapper method* and *Embedded method.*

For the purpose of this work, we use *Univariate regression tests* only on the multivariate LSTM model, since it does not make sense to apply it in the univariate models. For each variable we construct a linear model having the given variable as a regressor and the predicted variable as dependent variable - *regressand.* We estimate the overall significance of the model by an *F-test.* F-test estimates whether the model with a given regressor provides a better fit to the model than a model with no independent variable. When performing F-test we check the validity of two hypotheses:

- H_0: an intercept-only model (no independent variable) fits the data equally well.
- H_1: model with the given independent variable fits the data better than the intercept-only model.

Python's *scikit-learn* package has an implementation of such univariate regression which outputs the test statistics[4] and a corresponding p-value. After we complete all iterations and compute the scores and p-values for each variables, we filter out those variables where we rejected the null hypothesis on our confidence interval.

5.4 Metrics and tests

5.4.1 Stationarity tests

The importance of stationarity testing in time series was already referenced to in previous chapters. Several tests for checking the presence of *unit root* in the time series have been developed.

Dickey-Fuller test is one of such tests. Its null hypothesis states that a unit root is present in a time series. The alternative hypothesis is usually stationarity, therefore we want to reject the null hypothesis, so the p-value should be lower than 0.05 (or less, depending on the confidence interval). On a test equation for a unit root with drift and deterministic time trend

$$\Delta y_t = y_t - y_{t-1} = \alpha + \beta t + \gamma y_{t-1} + e_t \tag{5.9}$$

the unit root is present if $\gamma = 0$.

[4]F-score, a fraction of explained and unexplained variance

Augmented Dickey-Fuller test (ADF) is a generalized version of the Dickey-Fuller test, which can accommodate the AR processes of higher orders by including Δy_{t-p} in the test equation (with unchanged null hypothesis)

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + \dots \tag{5.10}$$

Phillips-Perron test (PP) builds on the Dickey-Fuller test and accounts for any serial correlation and heteroskedasticity in error terms. The null hypothesis is that the series contains unit root, the alternative hypothesis states that the process is stationary.[30]

We check the presence of Unit root by means of both *Augmented Dickey-Fuller* and *Phillips-Perron* test which are implemented in python's *Arch* package.

5.4.2 Diagnostic tests

Diagnostic tests are performed to validate whether our autoregressive time series model is correctly specified. Namely, we check for autocorrelation in residuals with *Durbin-Watson test (DW)* and the heteroskedasticity with *Breusch-Pagan test (BP)*.

Durbin-Watson test is used to detect autocorrelation in lag 1 of the residuals. If the residual at time t is given as $\epsilon_t = \rho e_{t-1} + \nu_t$, then the null hypothesis states $\rho = 0$ and the alternative $\rho \neq 0$. The test statistic d can gain any real numeric value between 0 and 4 and is computed as:

$$d = \frac{\sum_{t=2}^{T} (\epsilon_t - \epsilon_{t-1})^2}{\sum_{t=1}^{T} \epsilon_t^2}, \tag{5.11}$$

where T is the number of observations in the series. A value of $d = 2$ translates into no autocorrelation, $d < 2$ is interpreted as positive autocorrelation, $d > 2$ as negatively autocorrelated data. Although it is possible to test d for significance, checking the proximity of d to number 2 suffices our purposes.

One of the most common tests for heteroskedasticity is Breusch-Pagan test. It is conducted by regressing the estimated squared residuals $\hat{\epsilon}^2$ on the independent variables x_t of the model

$$\hat{\epsilon}^2 = \delta_0 + \delta_1 x_1 + \delta_2 x_2 + \dots + e. \tag{5.12}$$

Its null hypothesis is homoskedasticity of residuals, for which the following holds: $\delta_1 = \delta_2 = \dots = \delta_k$. From the regression a goodness-of-fit measure $R^2_{\hat{\epsilon}^2}$ is computed and inserted into the F statistic formula

$$F = \frac{\frac{R^2_{\hat{\epsilon}^2}}{k}}{\frac{1 - R^2_{\hat{\epsilon}^2}}{n-(k+1)}}, \tag{5.13}$$

which is approximately distributed as $F_{k,n-(k+1)}$ under the null hypothesis.

5.4.3 Prediction evaluation and portfolio perfomance metrics

We opt for calculating *Root Mean Square Error (RMSE)* as the prediction error metric for both the benchmark model and LSTM networks. In addition, the LSTMs use *Mean Square Error (MSE)* as the loss function for the training process.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{5.14}$$

Additionally, in the backtesting part, we compute an *Accuracy score* for the forecasted returns binarized into trading signals. Plainly put, it is a fraction of the of the correct predictions from the model. The formula for accuracy score can be written as

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} 1(\hat{y}_i = y_i), \tag{5.15}$$

where n is the number of observations in the sample and $1(X)$ is an *indicator function* which indicates the membership of an element X in a subset A, returning the value 1 for every X, for which $X \in A$, and 0 for every $X \notin A$.

We also perform a head-to-head comparison of out-of-sample forecasts with *Diebold-Mariano test (DM).* Originally proposed by Diebold and Mariano (1995), this model-free test compares forecast errors $e_{i1} \ldots, e_{iT}$ and $e_{j1} \ldots, e_{jT}$ from two alternative predictions. The test considers a series of loss differentials $\{d_t\}_{t=1}^{T}$. Under the assumption that each forecast is evaluated by some loss function g, for the loss differential d_t holds $d_t = g(e_{it}) - g(_{jt})$.

The null hypothesis presents equal predictive accuracy $E[d_t] = 0$ for all t, i.e. the forecast errors from the two forecasts produce the same loss. The alternative hypothesis of the two sided test is different levels of accuracy for the two forecasts. The DM test-statistic used for calculating associated p-values is computed as:

$$DM = \frac{\bar{d}}{\sqrt{\frac{\hat{\omega}}{T}}}, \tag{5.16}$$

where $\bar{d}$ is the sample mean of the loss differential series $\{d_t\}_{t=1}^{T}$ and $\hat{\omega}$ is a consistent estimator of the asymptotic variance of $\sqrt{T}\bar{d}$. Under fairly weak conditions is the DM test-statistic asymptotically $N(0,1)$ under the null hypothesis.
[31]

In the backtesting part, we evaluate performance of each model by *Sharpe ratio*, which is defined as a portfolio's mean return in excess of the riskless return divided by the portfolio's standard deviation. In other words, it represents a metric of the portfolio's risk-adjusted (excess) return.

$$\text{Sharpe ratio} = \frac{R_p - R_f}{\sigma_p}, \tag{5.17}$$

where R_p is return of the portfolio, R_f is the risk-free rate, and σ_p standard deviation of the portfolio's excess return.

5.5 Model Construction

5.5.1 ARIMA-GARCH models

This section deals with the implementation of the ARIMA-GARCH model, which is often used as a benchmark forecasting model for financial time series data and serves the same purpose in this instance, as well. We follow the process listed below when constructing the ARIMA model:

1. **Data load, transformation and split.** We load the price time series for the given ticker from the *SimFin* library and create Log Return column from the *Adjusted Closing Price.* We limit the scope of the time series to a range starting with year 2011 and ending with 2018, inclusive, and split it into in-sample and out-of-sample data sets. We do not perform any sort of further normalization.
2. **Stationarity tests.** On each in-sample time series, we check stationarity with *Augmented Dickey–Fuller* and *Phillips-Perron* Unit-Root tests with 3 alternatives of trend component included in the test - no trend component, a constant, constant and a linear trend.
3. **ARIMA model fitting and order estimation.** In this step, we perform a grid search of hyperparameters p,q. We limit the search space of both hyperparameters to a maximum order 4, inclusive. After each iteration of model fitting, we measure the goodness-of-fit, save the value of *Akaike Information Criterion* and select the model with the lowest value. We omit several steps from the Box-Jenkins methodology so that the whole process can be performed without any manual user input.
4. **Diagnostic tests.** As the final step, we perform diagnostic tests on the selected best fitting ARIMA model. We test the residuals for serial autocorrelation with *Durbin-Watson test*, and for homoskedasticity with *Breusch-Pagan test.*

The process of construction and selection of the best-fitting GARCH model assumes we already performed ARIMA model selection, i.e. the data is already split into train and test set and stationarity is confirmed on the train set. The process proceeds as follows:

1. **GARCH model fitting and order estimation.** Analogically to the ARIMA model fitting, we perform a grid search of hyperparameters p, q and lag order of the AR mean of the GARCH process. We use a limited search space[5] and save the value of *Akaike Information Criterion.* We select the model with lowest value of AIC.
2. **Out-of-sample prediction.** We use the best fitting model for the next observation mean prediction on test data set, plot the predictions on a line chart and calculate RMSE.

[5]Maximum value for lag=4, P=3, Q=3.

5.5.2 Univariate LSTM model

In this section we break down the step-wise process of constructing the Univariate LSTM forecasting model. The whole structure also applies to the Multivariate alternative.

1. **Data load, transformation and split.** We carry out the identical method of loading the data set, creating target column and limiting the scope of the data as in ARIMA-GARCH model. In this instance, however, we split the data into train, validation and test sets, as opposed to a train and test set.
2. **Standardization.** The target Log Return column is put through a standardization function. We utilize a Min-max scaling method with feature range from -1 to 1. This should ensure optimal learning performance of the network, since the LSTM cells contain a *tanh* activation layer.
3. **Network construction.** We build a model by creating an instance of a *Sequential* class of the Keras package with one hidden LSTM layer and a dense output layer. We use *Adam* optimizer and Mean Squared Error as a validation loss function.[6]
4. **Network training and hyperparameters optimization.** For each ticker, we transform the train and validation sets into *tensors*[7] and use them as inputs into a model building method on top of which operates a hyperparameter tuner. The tuner performs a *Random Search* over a three-dimensional search space aiming to find the best performing combination of these hyperparameters:
   - *Number of cells in the LSTM layer* - with a lower boundary = 16, upper boundary = 48 and step size = 16,
   - *Initial learning rate of the Adam optmizer* - an array of 4 possible values - $[0.1, 0.01, 0.001, 0.0001]$,
   - *Dropout rate* of the dropout layer that follows the LSTM layer - with a lower boundary = 0.01, upper boundary = 0.46 and step size = 0.15.

   When predicting the value of the next observation's target variable, the model considers a window of past 20 observations. Each training has a fix-sized upper limit of 500 training epochs with an *Early Stopping* implemented, which triggers after 50 consecutive epochs of no improvement in validation loss. This model does not make use of a batch learning because of the considerably small data sample size.
5. **Out-of-sample prediction.** A trained network with best performing set of hyperparameters is saved and used for out-of-sample prediction, plotting on a line chart, calculating and saving RMSE.

[6]Adam (adaptive moment estimation) is an optimizer which calculates the individual adaptive learning rate for each parameter from estimates of the first and second moments of the gradients, which speeds up the learning process. Adam optimizer has been widely used in the recent years mainly because of its fast learning pace and performance consistency. However, any further elaboration on learning optimizers and their comparison would deserve a separate research and thus is beyond the scope of this work.

[7]In this context, Tensor is an object of TensorFlow library which represents a rectangular array of data of an arbitrary shape and datatype.

5.5.3 Multivariate LSTM model

Construction of Multivariate LSTM model proceeds in similar manner, with an added step of *feature selection* which precedes the Network construction step. In feature selection, we perform Univariate regression tests on in-sample standardized variables and keep those whose p-value < 0.05.[8] Besides standardization, we do not perform any additional data transformations on these variables.

5.6 Backtest

After forecasting log returns with all 3 models on the out-of-sample data, we compare the predicted outputs using a simple backtest. We convert the numerical values of predicted next day's log returns to a binary variable, which would serve as a trading signal. Concretely, we convert the predicted return to a buy signal (1) if $r_t \geq 0$, and to a sell signal (-1) otherwise. On each day of the test period, we update the position according to the signal of the given day.

At the end of the trading period, on each stock, we compare the performance of each model with the others and with the *Buy and hold* strategy using metrics such as *Accuracy score, Total return, Average daily return, Sharpe ratio.* Finally, we compute aggregate statistics across all stocks to compare the overall performance of each strategy(i.e. model).[9]

[8]In those cases where no variable meets the condition, we pick two variables with the lowest p-value.

[9]The whole backtesting process assumes zero transaction costs.

6. Results

The results are presented in three stages. The first stage contains forecast results for separate models - GARCH, Univariate LSTM, and Multivariate LSTM in sections 6.1, 6.2 and 6.3. Section 6.4 compares out-of-sample forecast accuracy of these models and the final section 6.5 presents results of the backtest.

6.1 ARIMA-GARCH models

This section presents outputs of the ARIMA-GARCH model construction process described in section 5.5.1. After step 1 - data loading, creating Log Return variable and splitting into in and out-of-sample data sets we examine the Log Returns and their statistical properties on the whole data set (table 6.1).[1]

Table 6.1: Statistical properties of the in-sample time series of Returns

	MSFT	**AAPL**	**AMZN**	**GOOG**	**JNJ**
observations	2011	2011	2011	2011	2011
mean	0.00	0.00	0.00	0.00	0.00
standard dev.	0.01	0.02	0.02	0.02	0.01
min	-0.12	-0.13	-0.14	-0.09	-0.11
max	0.10	0.08	0.15	0.15	0.05
kurtosis	7.25	4.85	8.16	12.10	10.15
skewness	-0.10	-0.30	0.18	0.67	-0.80

We check the stationarity of the in-sample time series by means of Augmented Dickey-Fuller (ADF) and Phillips-Perron (PP) tests. In both tests, we unanimously reject the null hypothesis of a Unit-root present in the series for no trend component $\emptyset$, a constant (c) and a constant and a linear trend (c and t). Thus, we regard every time series to be stationary on a significance level = 0.01. Table 6.2 summarizes the results for all time series.

In the next step we perform a grid search over p,q parameters of ARIMA model, fit the best model with the lowest goodness-of-fit AIC and perform diagnostic tests on the residuals - Durbin-Watson (DW) and Breusch-Pagan (BP) test. Table 6.3 contains results of diagnostic tests. The DW statistic is approximately = 2, hence none of the time series indicates the presence of serial autocorrelation at lag 1. However, with the exception of MSFT ticker, we reject the null hypothesis of homoscedastic residuals. Because ARIMA models failed to explain the conditional variance of returns, we move to GARCH models.[2]

[1]This is a purely descriptive table, therefore we do not consider it to be a look-ahead bias situation.

[2]Although all models are integrated of order 0, we continue using ARIMA instead of ARMA denotation for consistency.

Finally, we proceed to GARCH model construction with autoregressive mean. We deploy a grid search over the parameters lag, p, q and pick the one with lowest AIC value to forecast the out-of-sample returns. The RMSE of the predicted returns and specifications of the best fitting model for each ticker are displayed in table 6.4.

Table 6.2: Test statistics and p-values of ADF and PP Stationarity tests

Ticker	**Trend comp.**	**ADF**	**ADF p-value**	**PP**	**PP p-value**
MSFT	$\emptyset$	-15.57	0.00	-46.88	0.00
	c	-15.87	0.00	-47.42	0.00
	c and t	-17.66	0.00	-47.49	0.00
AAPL	$\emptyset$	-13.84	0.00	-43.50	0.00
	c	-13.97	0.00	-43.53	0.00
	c and t	-13.98	0.00	-43.52	0.00
AMZN	$\emptyset$	-44.94	0.00	-45.10	0.00
	c	-45.05	0.00	-45.43	0.00
	c and t	-45.05	0.00	-45.44	0.00
GOOG	$\emptyset$	-44.06	0.00	-44.21	0.00
	c	-44.12	0.00	-44.38	0.00
	c and t	-44.11	0.00	-44.37	0.00
JNJ	$\emptyset$	-22.86	0.00	-46.28	0.00
	c	-23.01	0.00	-46.55	0.00
	c and t	-23.02	0.00	-46.56	0.00

Table 6.3: Durbin-Watson statistic of Breusch-Pagan statistic and p-values for the best performing ARIMA models

Ticker	**p**	**q**	**DW**	**BP**	**BP p-value**
MSFT	3	3	1.98	1.38	0.24
AAPL	2	2	1.95	18.61	0.00
AMZN	1	2	2.00	13.45	0.00
GOOG	1	3	2.00	94.49	0.00
JNJ	1	0	2.00	16.57	0.00

6.2 Univariate LSTM models

This section displays the results of the Univariate LSTM networks. The section 5.5.2 describes the process of loading and transforming the data, building and training the network and carrying out predictions.

Step 4 of the process includes hyperparameter tuning - a Random search over Number of Units in the LSTM cell, the Dropout rate and the initial learning rate for the optimizer. The

Table 6.4: Order of the best performing GARCH models and the associated AIC and out-of-sample RMSE

Ticker	**Lag**	**p**	**q**	**AIC**	**RMSE**
MSFT	3	2	2	6133	0.0177
AAPL	3	1	2	6493	0.0176
AMZN	3	1	2	7237	0.0224
GOOG	3	1	2	6208	0.0173
JNJ	3	2	2	4312	0.0144

table 6.5 shows the best performing hyperparameters for each ticker. On our modestly sized sample of 5 tickers, there is no visible pattern in hyperparameter combination that would be chosen on more than 1 data set.

Table 6.5: Univariate LSTM - Best set of hyperparameters

Ticker	**No. Units**	**Dropout rate**	**Learning rate**
MSFT	16	0.16	0.0001
AAPL	32	0.31	0.1000
AMZN	32	0.31	0.0001
GOOG	16	0.01	0.0010
JNJ	32	0.01	0.0100

The table 6.6 summarizes the rounded RMSE on train, validation and test set. Two patterns are observable - the RMSE on validation sets is considerably lower than on train sets, while test sets score the highest RMSE overall. Lower generalization power of the network might signal overfitting.

Table 6.6: Univariate LSTM - RMSE on train, validation and test set

Ticker	**RMSE Train**	**RMSE Validation**	**RMSE Test**
MSFT	0.0147	0.0114	0.0182
AAPL	0.0167	0.0121	0.0188
AMZN	0.0205	0.0145	0.0233
GOOG	0.0158	0.0107	0.0183
JNJ	0.0089	0.0073	0.0146

The plots of return forecasts (figures A.6 - A.10) display the actual (green) and predicted (blue) log returns on train, validation and test set. There is a noticeable heterogeneity in variance of the predictions among the tickers (e.g. AAPL vs AMZN).[3]

[3]The gap in predicted values after validation and test split corresponds to the window of 20 historical observations that are needed as input for the network.

6.3 Multivariate LSTM models

This section delivers results of the Multivariate LSTM networks. The process of their construction is identical to the Univariate variant, with the exception of added feature selection segment before the network construction step. Table 6.7 lists all variables selected into the model. The exogenous variables selected for GOOG ticker have p-value exceeding the significance level, nevertheless, they were included in order to have a multivariate LSTM for every ticker.

Table 6.7: Test statistic and the associated p-value for columns selected by univariate regression

Ticker	**Column**	**F-score**	**p-value**
MSFT	P/Sales	8.5174	0.0036
	Dividend Yield	6.2665	0.0124
	Volume	4.8388	0.0280
	P/Book	4.6577	0.0311
	P/FCF	4.1748	0.0412
	FCF Yield	4.1198	0.0426
AAPL	Earnings Yield	9.9129	0.0017
	FCF Yield	9.6590	0.0019
	P/E	9.1680	0.0025
	P/FCF	8.2490	0.0041
	Volume	7.7994	0.0053
	Relative Volume	7.0719	0.0079
	P/Sales	6.9915	0.0083
	Inventory Turnover	4.8290	0.0282
	P/NCAV	4.1545	0.0417
	P/Book	3.8557	0.0498
AMZN	P/Sales	9.4673	0.0021
	P/Book	8.4364	0.0037
	Adj. Close	4.3304	0.0376
	Market-Cap	4.1338	0.0422
	Relative Volume	4.0127	0.0454
GOOG	Adj. Close*	2.4898	0.1148
	Earnings Yield*	1.0737	0.3003
JNJ	Relative Volume	4.5493	0.0331

An analogous patter to that of Univariate LSTM can be observed in table 6.9 - a potential sign of overfitting, because of a consistent better performance on the validation set.

From plots on figures A.11 - A.15, one can observe the highest variance of predicted returns among all three forecast models we discussed. Additionally, some out-of-sample predictions

Table 6.8: Multivariate LSTM - Best set of hyperparameters

Ticker	**No. Units**	**Dropout rate**	**Learning rate**
MSFT	48	0.16	0.0010
AAPL	48	0.31	0.1000
AMZN	48	0.01	0.0001
GOOG	16	0.31	0.0100
JNJ	32	0.01	0.0100

Table 6.9: Multivariate LSTM - RMSE on train, validation and test set

Ticker	**RMSE Train**	**RMSE Validation**	**RMSE Test**
MSFT	0.0146	0.0115	0.0189
AAPL	0.0165	0.0120	0.0262
AMZN	0.0205	0.0146	0.0246
GOOG	0.0158	0.0108	0.0183
JNJ	0.0086	0.0073	0.0149

(namely AAPL and AMZN tickers) show signs of extreme overfitting. This can probably be attributed to the fact, that the network falsely assigned unproportionally high weights to some exogenous variables. Some of these variables, intuitively, should not meet stationarity criteria, which could possibly hinder the correct performance of the network[4].

[4]Practically none of the exogenous variables should be stationary by default, possibly with the exception of variables Volume and Relative Volume.

6.4 Forecast comparison

In this part we evaluate the predictive accuracy of the three models against each other by means of Diebold-Mariano test (DM) on each ticker separately. We deploy the DM test with the null hypothesis that forecast from *i-th* and *j-th* model have the same forecast accuracy, and the alternative that the forecast from *i-th* model achieves lower accuracy. The $i \times j$ matrix contains p-value of the DM test. The last column contains RMSE for the *i-th* model. The test and RMSE have been calculated on out-of-sample data. In case of the GARCH predictions, first 20 observations of the out-of-sample data have been cut out, so that the data length matches that of the LSTM prediction data.[5]

In terms of RMSE, there is a visible pattern of underperformance by the Multivariate LSTM on all samples. The error by GARCH and Univariate LSTM is roughly comparable - slightly lower for GARCH in tickers AAPL, AMZN and GOOG, higher in MSFT, and equal in JNJ ticker.

This is generally supported by the results of the DM tests that can be seen in tables 6.10 - 6.14. We reject the null hypothesis at 5% significance level that the multivariate LSTM's prediction accuracy is equal to that of either Univariate LSTM or GARCH in the samples for MSFT, AAPL and AMZN tickers. We do not reject the null on DM tests between Univariate LSTM and GARCH on all samples with the exception of GOOG ticker, where Univariate LSTM has inferior accuracy.

Table 6.10: **MSFT ticker** - DM test and RMSE on out-of-sample data

i j=	GARCH	Univar LSTM	MultiVar LSTM	RMSE
GARCH	-	0.3873	0.9993	0.0183
Univar LSTM	0.6127	-	0.9996	0.0182
MultiVar LSTM	0.0007	0.0004	-	0.0189

Table 6.11: **AAPL ticker** - DM test and RMSE on out-of-sample data

i j=	GARCH	Univar LSTM	MultiVar LSTM	RMSE
GARCH	-	0.9981	1.0000	0.0181
Univar LSTM	0.0019	-	1.0000	0.0188
MultiVar LSTM	0.0000	0.0000	-	0.0262

[5]All subsequent computations are performed on the out-of-sample data of equal length.

Table 6.12: **AMZN ticker** - DM test and RMSE on out-of-sample data

i j=	GARCH	Univar LSTM	MultiVar LSTM	RMSE
GARCH	-	0.9984	0.9998	0.0231
Univar LSTM	0.0016	-	0.9985	0.0233
MultiVar LSTM	0.0002	0.0015	-	0.0246

Table 6.13: **GOOG ticker** - DM test and RMSE on out-of-sample data

i j=	GARCH	Univar LSTM	MultiVar LSTM	RMSE
GARCH	-	1.0000	1.0000	0.0179
Univar LSTM	0.0000	-	0.1850	0.0183
MultiVar LSTM	0.0000	0.8150	-	0.0183

Table 6.14: **JNJ ticker** - DM test and RMSE on out-of-sample data

i j=	GARCH	Univar LSTM	MultiVar LSTM	RMSE
GARCH	-	0.5509	0.7542	0.0146
Univar LSTM	0.4491	-	0.7352	0.0146
MultiVar LSTM	0.2458	0.2648	-	0.0149

6.5 Backtest

The final part of this work presents results of a simple backtest outlined in section 5.6. Prior to carrying out the backtesting, we binarized the forecasted returns and used them as trading signals to decide whether to go long or short on a given stock in the next trading day. Since we shifted from a regression to a classification problem, we are able to compute different set of performance metrics - Accuracy, Total Returns, Average Daily Return and Sharpe Ratio. All metrics are computed on the out-of-sample data (i.e. the trading period) of 231 trading days.

The table 6.15 compares the strategies of our three models against each other and against Buy and Hold strategy which serves as a benchmark. The data in the table is broken down to a ticker granularity. The table 6.16 then contains aggregate average performance metrics for each strategy on an equally weighted portfolio of the five tickers.

Table 6.15: Performance metrics for a given model (strategy) and a ticker

Ticker	**Model**	**Accuracy**	**Total Return**	**Average Daily Return**	**Annualized Sharpe Ratio**
MSFT	Buy and Hold	0.5541	0.1081	0.0005	0.4059
	GARCH	0.5584	0.1239	0.0005	0.4650
	Univariate LSTM	0.5671	0.1930	0.0008	0.7250
	Multivariate LSTM	0.5281	-0.2039	-0.0009	-0.7657
AAPL	Buy and Hold	0.5195	-0.0420	-0.0002	-0.1547
	GARCH	0.6017	1.0914	0.0047	4.1548
	Univariate LSTM	0.5281	0.1688	0.0007	0.6221
	Multivariate LSTM	0.5152	0.4151	0.0018	1.5359
AMZN	Buy and Hold	0.5628	0.0436	0.0002	0.1282
	GARCH	0.5801	0.3873	0.0017	1.1402
	Univariate LSTM	0.5584	0.0071	0.0000	0.0208
	Multivariate LSTM	0.4372	0.0243	0.0001	0.0715
GOOG	Buy and Hold	0.5108	-0.1166	-0.0005	-0.4372
	GARCH	0.6147	1.3464	0.0058	5.3233
	Univariate LSTM	0.5152	-0.1375	-0.0006	-0.5156
	Multivariate LSTM	0.5022	0.1718	0.0007	0.6444
JNJ	Buy and Hold	0.5411	-0.0719	-0.0003	-0.3494
	GARCH	0.4589	-0.5641	-0.0024	-2.7815
	Univariate LSTM	0.5325	-0.0945	-0.0004	-0.4589
	Multivariate LSTM	0.5195	-0.2940	-0.0013	-1.4335

On MSFT ticker, GARCH model copied the development of the stock, i.e. the Buy and Hold strategy except for the last few trading days. All strategies managed to achieve accuracy rate higher than 50%. However, the Multivariate LSTM ended with a negative total and also average daily return. Univariate LSTM model slightly outperformed the other two profitable strategies in every measure.

GARCH model significantly outperformed its competitors on AAPL, AMZN and GOOG tickers, where it recorded 60%, 58%, and 61.5% accuracy, and the cumulative return of 1.0914, 0.3873, and 1.3464, respectively. The benchmark strategy finished with negative total returns of -0.0420 on AAPL ticker, and -0.1166 on GOOG ticker. The Multivariate LSTM achieved the lowest accuracy across all strategies and data sets on AMZN ticker of around 43%, but noticeably, resulted with positive total returns at the end of the trading period.

None of the strategies managed to become profitable on the JNJ ticker. However the benchmark strategy and both LSTM networks reached an acuracy higher than 50%. The JNJ ticker was also the sole one, on which the GARCH flopped with accuracy rate of 46% negative total returns and Sharpe ratio, as well.

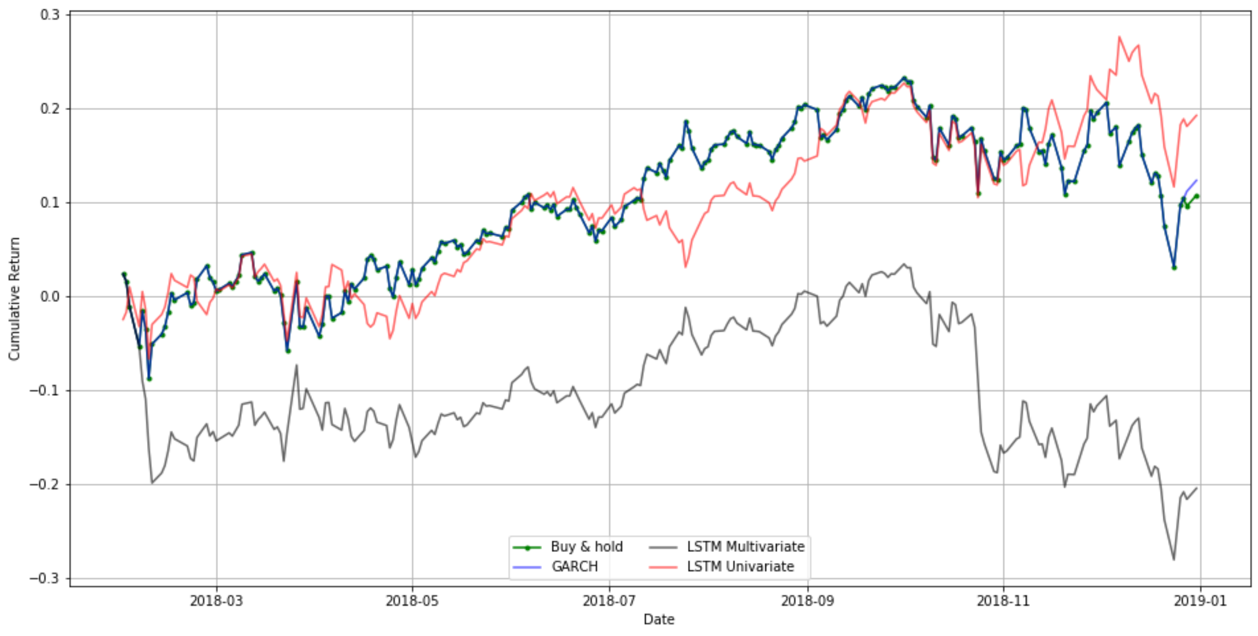

Figure 6.1: **MSFT ticker** - Cumulative Log returns for each model

Table 6.16: Average performance metrics for each model (strategy)

Model	Accuracy	Total Return	Average Daily Return	Annualized Sharpe ratio
Buy and Hold	0.5377	-0.0157	-0.0001	-0.0815
GARCH	0.5628	0.4770	0.0021	1.6604
Univariate LSTM	0.5403	0.0274	0.0001	0.0787
Multivariate LSTM	0.5004	0.0227	0.0001	0.0105

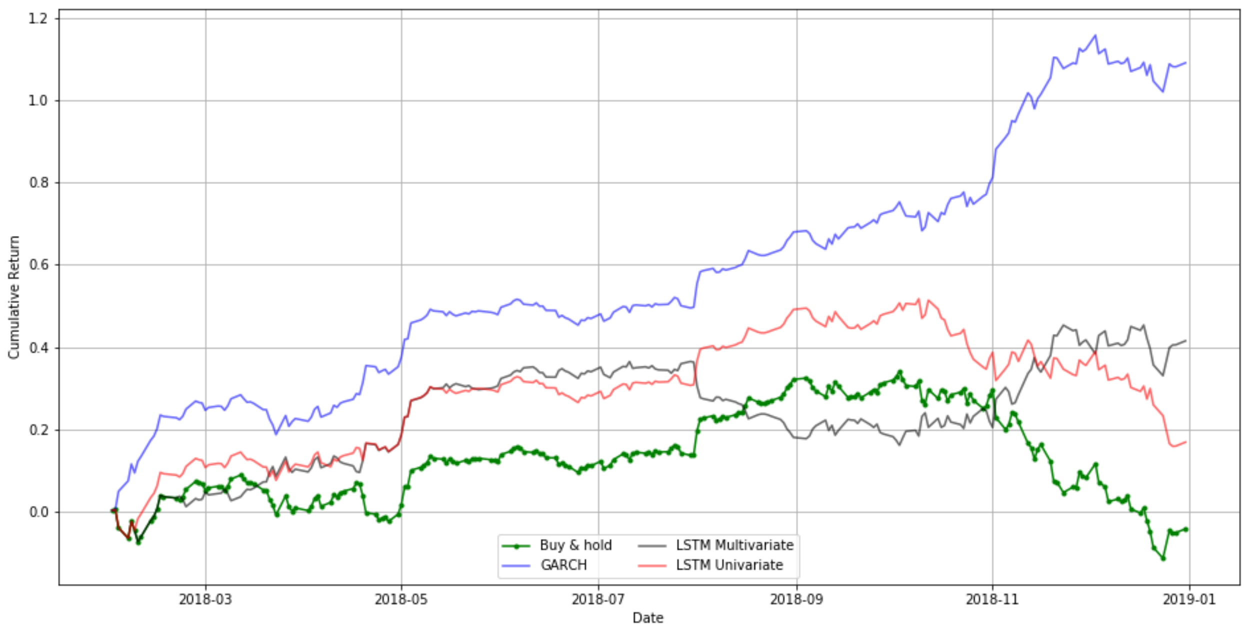

Figure 6.2: **AAPL ticker** - Cumulative Log returns for each model

In terms of accuracy, on aggregate level, all strategies scored higher than 50%, although the Multivariate LSTM only negligibly. GARCH managed to score significantly higher than a random chance, with the exception of JNJ returns where it considerably underperformed all other models. GARCH also scored almost the same as Buy and Hold on MSFT ticker, which can be explained by the forecast plot A.2 - all forecasted return values lie above 0, thus the strategy was never to sell the stock. In aggregate, GARCH achieved accuracy of more than 56%, leaving Univariate LSTM second, Buy and Hold third with around 54% and Multivariate LSTM last with 50% accuracy score.

Regarding Total Returns, three of the stocks finished in negative cumulative returns at the end of the trading sample. This reflected on the Buy and Hold strategy which, in aggregate, ended in negative Average Total Returns and also Average Daily Returns, as the only one.

Since all three models ended on average with positive total returns, they also managed to score a positive Sharpe ratio. Only the benchmark strategy deviated from the rest and recorded negative values in Total Return, Average Return and Sharpe Ratio.

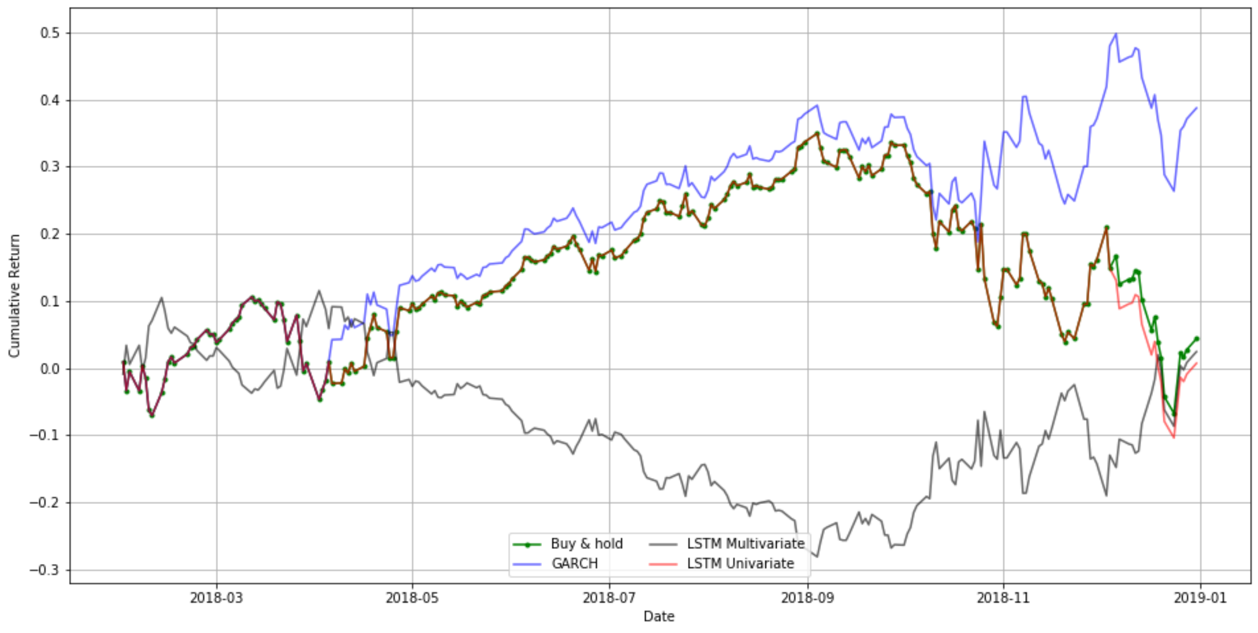

Figure 6.3: **AMZN ticker** - Cumulative Log returns for each model

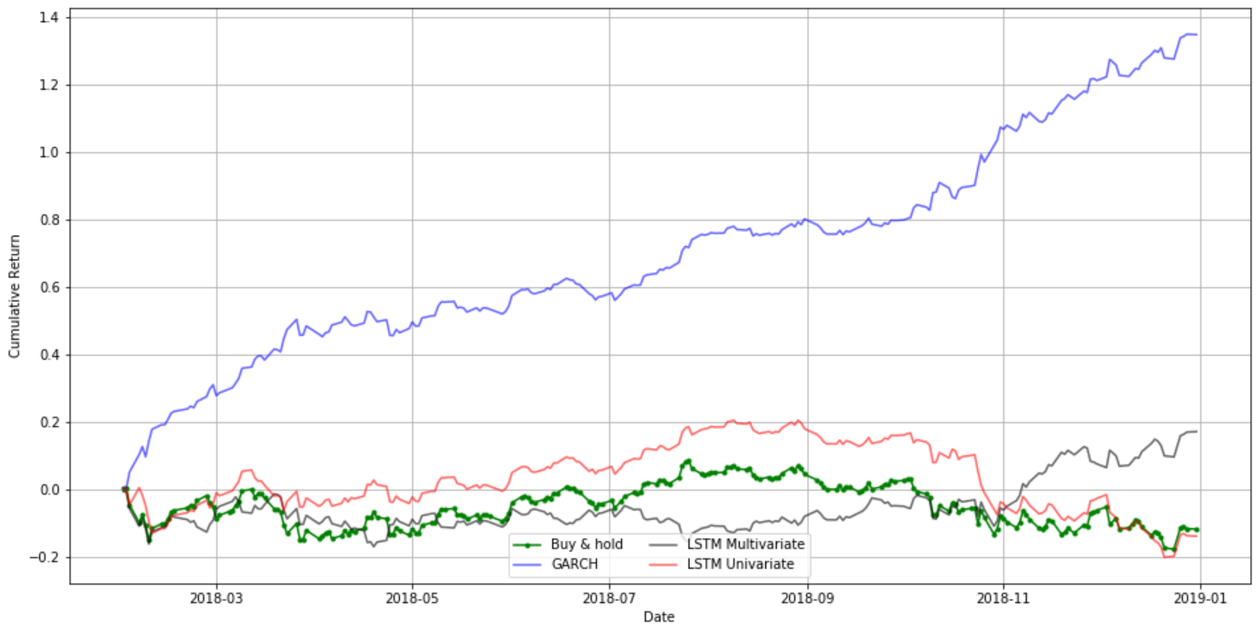

Figure 6.4: **GOOG ticker** - Cumulative Log returns for each model

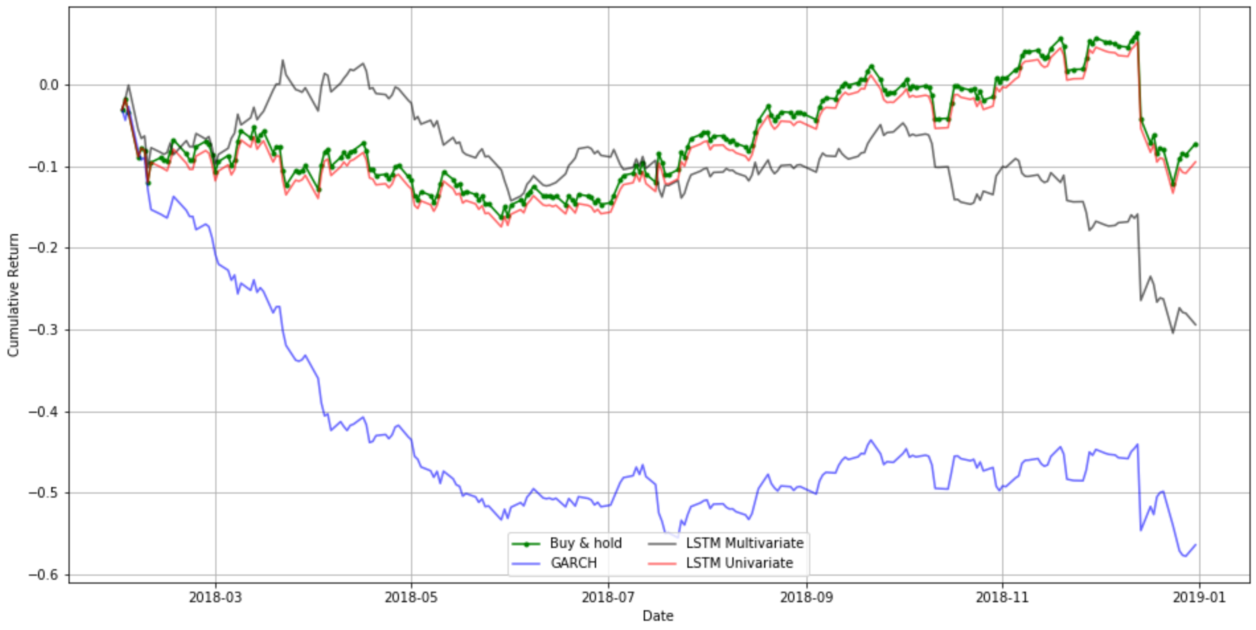

Figure 6.5: **JNJ ticker** - Cumulative Log returns for each model

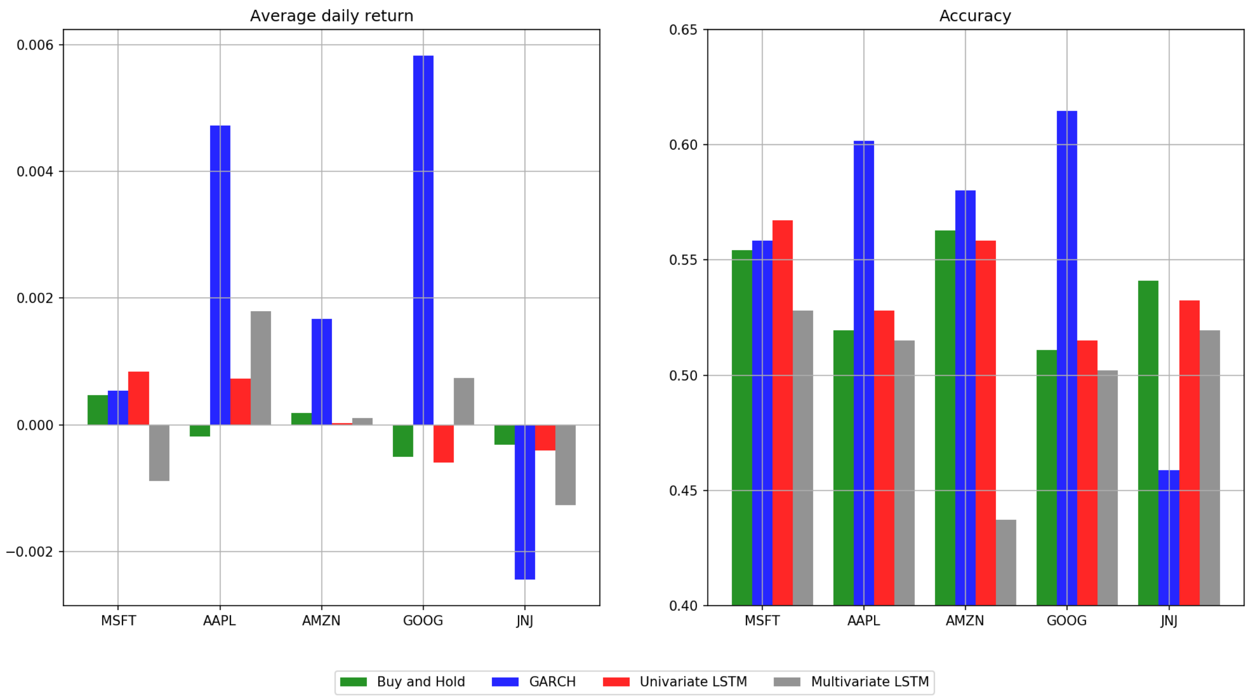

Figure 6.6: Performance measures on each ticker

Conclusion

This thesis aimed to compare the short-term predictive capability of artificial neural networks represented by two variants of Long Short-Term Memory (LSTM) networks with a more traditional time series forecasting approach represented by Generalized autoregressive conditional heteroskedasticity (GARCH) models.

Theoretical sections of the thesis focused on the construction and optimization of the neural networks. In the first part, we covered the construction of the most common neural networks together with the associated aspects, such as the mathematical background of the neural network training, activation and cost functions, etc. Another portion of the work zoomed in on different auxiliary machine learning topics, such as model selection process, performance metrics, validation methods, regularization, and hyperparameter tuning. Additionally, we laid out the theoretical background of the established time series forecasting models (ARIMA, GARCH) that served as a benchmark in the comparison analysis.

We carried out two kinds of comparison tasks on three constructed models - the benchmark GARCH model, the Univariate LSTM network whose input consisted of a vector of past returns, and the Multivariate LSTM network which additionally took vectors of several other exogenous variables as input.

First, the forecasting models were used for prediction of the next observation's value of the target variable - Log returns - on a sample consisting of five tickers selected from the S&P 500 index. The predictive accuracy was evaluated in terms of root mean squared error (RMSE) and compared employing the Diebold-Mariano test. In three of five trading samples, the test confirmed a superior accuracy for the benchmark GARCH model. The Univariate LSTM managed to achieve comparable results in the remaining two samples. The Multivariate LSTM model did not manage to outperform the other proposed models on any one data set.

Second, the task was re-framed from regression to a classification problem, and the models' predicted outputs were transformed into binary trading signals, which were subsequently utilized in the backtesting scenario comprising 231 trading days of the out-sample data. Although neither of the LSTM networks managed to consistently outperform the GARCH model in terms of overall Total Returns and Accuracy score, the Univariate LSTM model achieved a promising overall Accuracy score of 54% that resulted into positive Total Returns. For the sake of simplicity, we omitted all transaction costs associated with trading on a stock market. This decision definitely helped creating overly optimistic results.

The Multivariate LSTM network achieved results inferior to its competitors in both of the aforementioned tasks. Several factors might have contributed to its relatively poor performance. First and foremost, LSTM networks, and any neural network in general, needs a substantial amount of data in the training sample to learn the intricacies of the data. Our data sets of a length slightly surpassing 2000 observations can hardly be considered comprehensive enough. Second, the inappropriate use of exogenous variables, namely the implementation of feature selection accompanied by a lacking data transformation step might have also reflected in the lower quality of the forecasts. Finally, we assume that re-stating the task to a classification problem from the beginning could potentially help improve the training of the network.

Although the LSTM networks achieved lower scores in the performance metrics than the benchmark model, the thesis succeeded in terms of creating a framework for scalable and reproducible neural networks with a potential to be improved by addressing the issues we have outlined before.

List of references

1. NIELSEN, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com`.
2. *THE MNIST DATABASE of handwritten digits*. Available also from: `http://yann.lecun.com/exdb/mnist/`.
3. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.
4. BRITZ, Denny. *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs*. Available also from: `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`.
5. GERS, Felix. *Long Short-Term Memory in Recurrent Neural Networks*. 2001. Master's thesis. Lausanne, EPFL.
6. OLAH, Christopher. *Understanding LSTM*. Available also from: `http://colah.github.io/posts/2015-08-Understanding-LSTMs`.
7. RASCHKA, Sebastian. Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning. *CoRR*. 2018, vol. abs/1811.12808. Available from arXiv: `1811.12808`.
8. ZOU, Hui; HASTIE, Trevor. Regularization and variable selection via the elastic net. *J. R. Statist. Soc. B*. 2005, pp. 301–320. Available also from: `http://web.stanford.edu/~hastie/Papers/B67.2%20%282005%29%20301-320%20Zou%20&%20Hastie.pdf`.
9. MICELI BARONE, Antonio Valerio; HADDOW, Barry; GERMANN, Ulrich; SENNRICH, Rico. Regularization techniques for fine-tuning in neural machine translation. 2017, pp. 1489–1494.
10. SRIVASTAVA, Nitish; HINTON, Geoffrey; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, vol. 15, pp. 1929–1958.
11. BERGSTRA, James; BENGIO, Yoshua. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*. 2012, vol. 13, no. 10, pp. 281–305. Available also from: `http://jmlr.org/papers/v13/bergstra12a.html`.
12. HIRA, Farhan Islam; MARUF, Mazharul Ferdous; HOSSAIN, Afzal. *Stock market prediction using time series analysis*. 2018. PhD thesis. BRAC University.
13. SHUMWAY, R.H.; STOFFER, D.S. *Time Series Analysis and Its Applications*. Springer, 2014. ISBN 9781475732627. Available also from: `https://books.google.cz/books?id=N-EYswEACAAJ`.
14. RUPPERT, David; MATTESON, David S. GARCH Models. In: *Statistics and Data Analysis for Financial Engineering*. Springer, 2015, pp. 405–452.

15. BALAKRISHNAN, N.; COLTON, T.; EVERITT, B.; PIEGORSCH, W.W.; RUGGERI, F.; TEUGELS, J.L. *Wiley StatsRef: Statistics Reference Online*. John Wiley & Sons, Incorporated,., 2014. Wiley Online Library: Books. ISBN 9781118445112. Available also from: `https://books.google.cz/books?id=j321oQEACAAJ`.

16. BOX, G.E.P.; JENKINS, G.M.; REINSEL, G.C.; LJUNG, G.M. *Time Series Analysis: Forecasting and Control*. Wiley, 2015. Wiley Series in Probability and Statistics. ISBN 9781118674918. Available also from: `https://books.google.cz/books?id=EjXHCQAAQBAJ`.

17. DIN, Marilena. ARIMA by Box Jenkins Methodology for Estimation and Forecasting Models in Higher Education. In: 2015. Available from DOI: `10.13140/RG.2.1.1259.6888`.

18. DIN, Marilena. ARIMA by Box Jenkins Methodology for Estimation and Forecasting Models in Higher Education. In: 2015. Available from DOI: `10.13140/RG.2.1.1259.6888`.

19. KONISHI, Sadanori; KITAGAWA, Genshiro. *Information Criteria and Statistical Modeling*. Springer, 2008. Springer series in statistics. ISBN 9780387718873. Available also from: `https://books.google.cz/books?id=x%5C_7angEACAAJ`.

20. FORD, Clay. *Understanding Q-Q Plots*. University of Virginia Library, 2015. Available also from: `https://data.library.virginia.edu/understanding-q-q-plots/`.

21. SEZER, Omer; GUDELEK, Ugur; OZBAYOGLU, Murat. Financial Time Series Forecasting with Deep Learning : A Systematic Literature Review: 2005-2019. 2019.

22. LEE, Sang; YOO, Seong. Threshold-based portfolio: the role of the threshold and its applications. *The Journal of Supercomputing*. 2018. Available from DOI: `10.1007/s11227-018-2577-1`.

23. CHONG, Eunsuk; HAN, Chulwoo; PARK, Frank. Deep Learning Networks for Stock Market Analysis and Prediction: Methodology, Data Representations, and Case Studies. *Expert Systems with Applications*. 2017, vol. 83. Available from DOI: `10.1016/j.eswa.2017.04.030`.

24. ALBERG, John; LIPTON, Zachary C. Improving factor-based quantitative investing by forecasting company fundamentals. *arXiv preprint arXiv:1711.04837*. 2017.

25. XU, Yumo; COHEN, Shay B. Stock movement prediction from tweets and historical prices. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 1970–1979.

26. FISCHER, Thomas; KRAUSS, Christopher. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*. 2018, vol. 270, no. 2, pp. 654–669.

27. HUDSON, Robert S; GREGORIOU, Andros. Calculating and comparing security returns is harder than you think: A comparison between logarithmic and simple returns. *International Review of Financial Analysis*. 2015, vol. 38, pp. 151–162.

28. *Why Log Returns*. 2011. Available also from: `https://quantivity.wordpress.com/2011/02/21/why-log-returns/`.
29. RASCHKA, Sebastian. About Feature Scaling and Normalization (and the effect of standardization for Machine Learning algorithms). *Polar Political Legal Anthropology Rev.* 2014, vol. 30, no. 1, pp. 67–89.
30. DICKEY, D.; FULLER, Wayne. Distribution of the Estimators for Autoregressive Time Series With a Unit Root. *JASA. Journal of the American Statistical Association*. 1979, vol. 74. Available from DOI: `10.2307/2286348`.
31. DIEBOLD, Francis X; MARIANO, Robert S. Comparing predictive accuracy. *Journal of Business & economic statistics*. 2002, vol. 20, no. 1, pp. 134–144.

Part I

Appendix

A. Plots

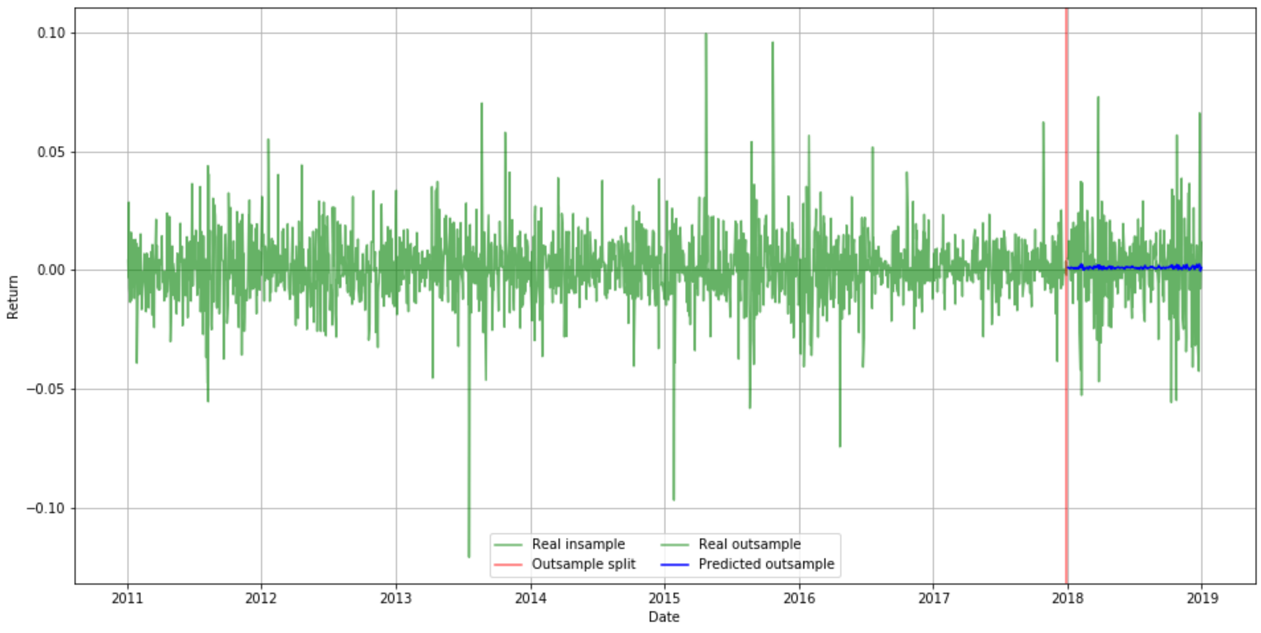

Figure A.1: **MSFT ticker** - Actual and Predicted Log Returns by GARCH model

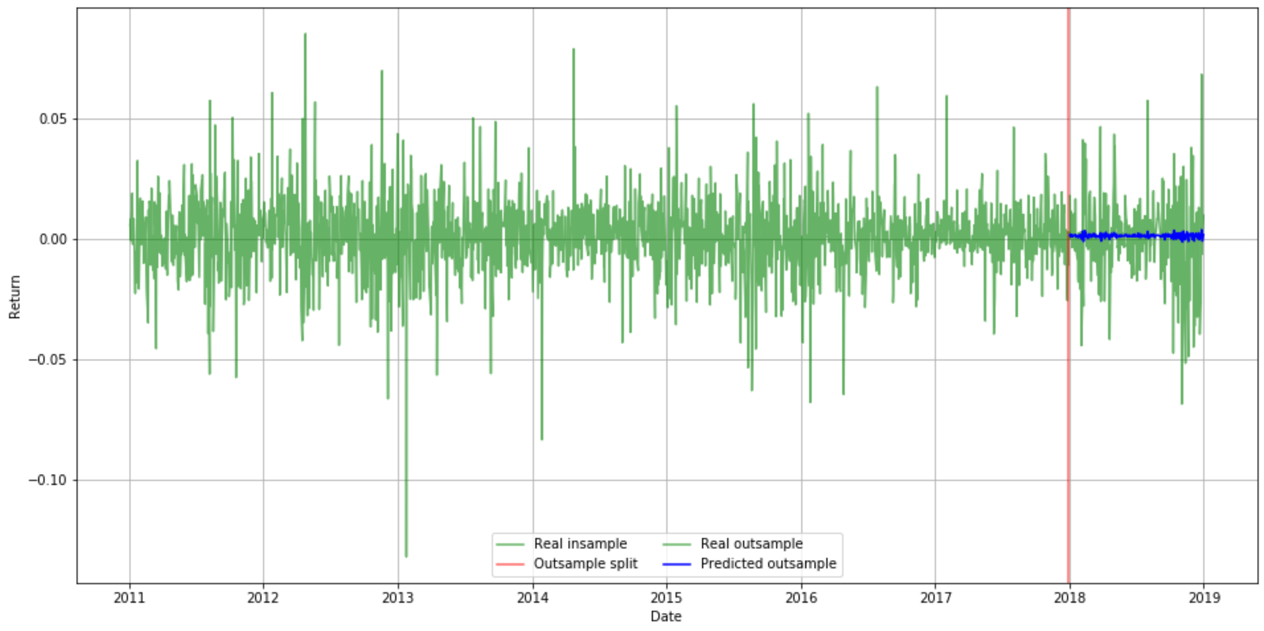

Figure A.2: **AAPL ticker** - Actual and Predicted Log Returns by GARCH model

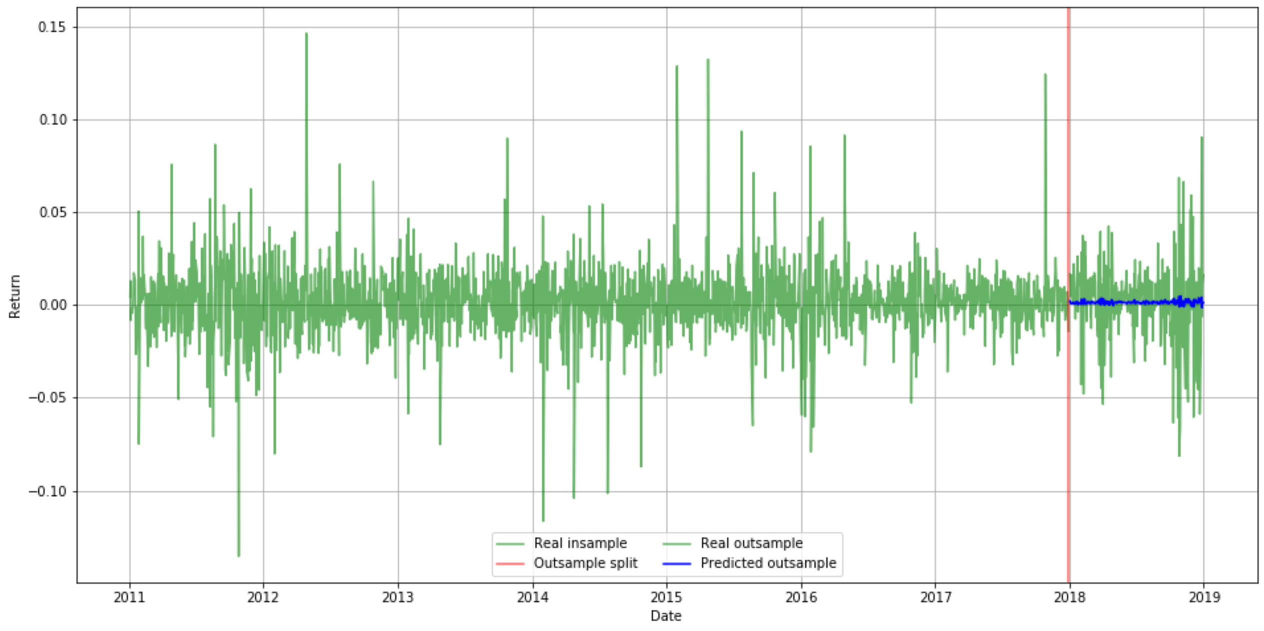

Figure A.3: **AMZN ticker** - Actual and Predicted Log Returns by GARCH model

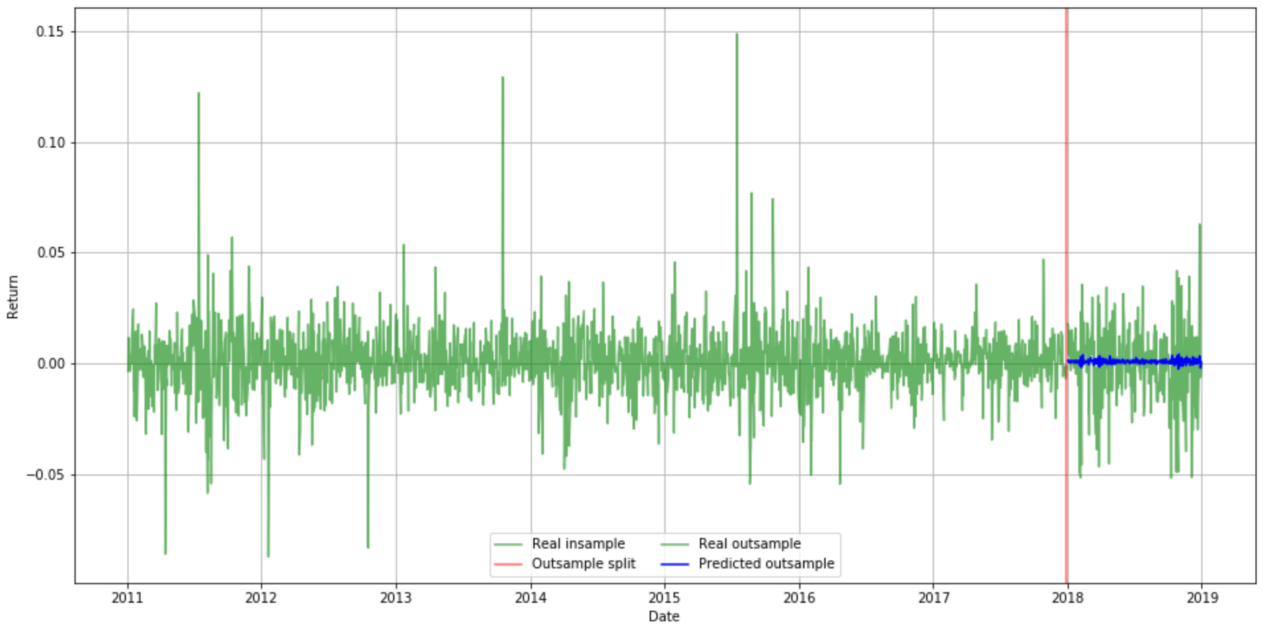

Figure A.4: **GOOG ticker** - Actual and Predicted Log Returns by GARCH model

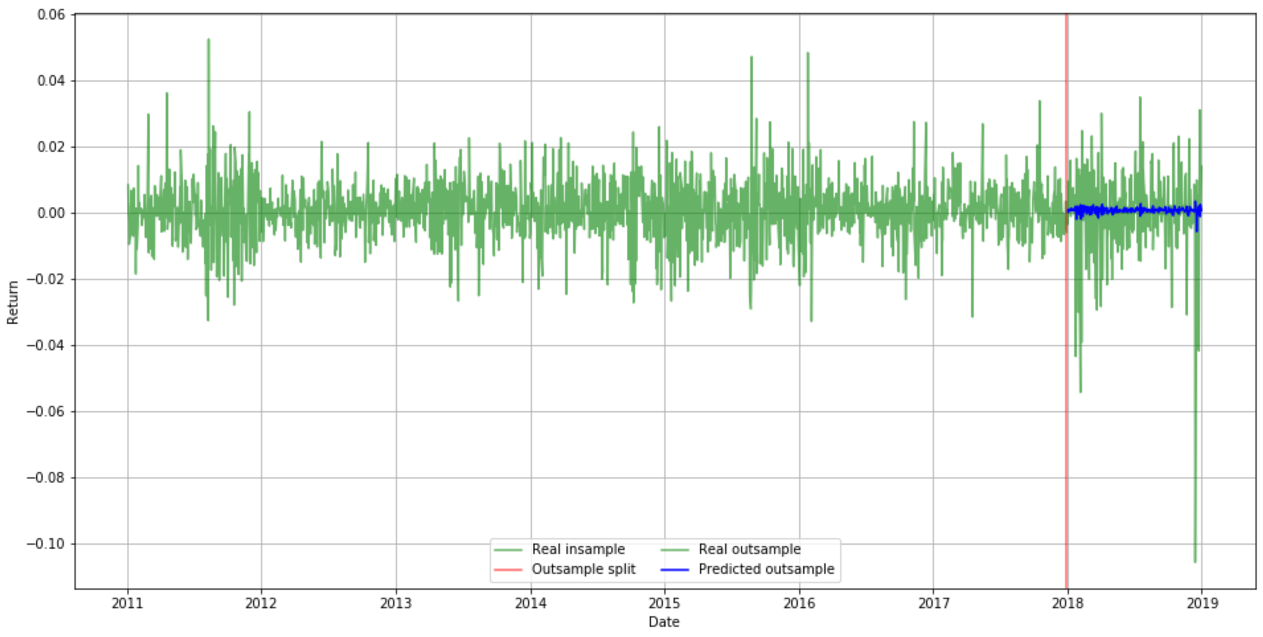

Figure A.5: **JNJ ticker** - Actual and Predicted Log Returns by GARCH model

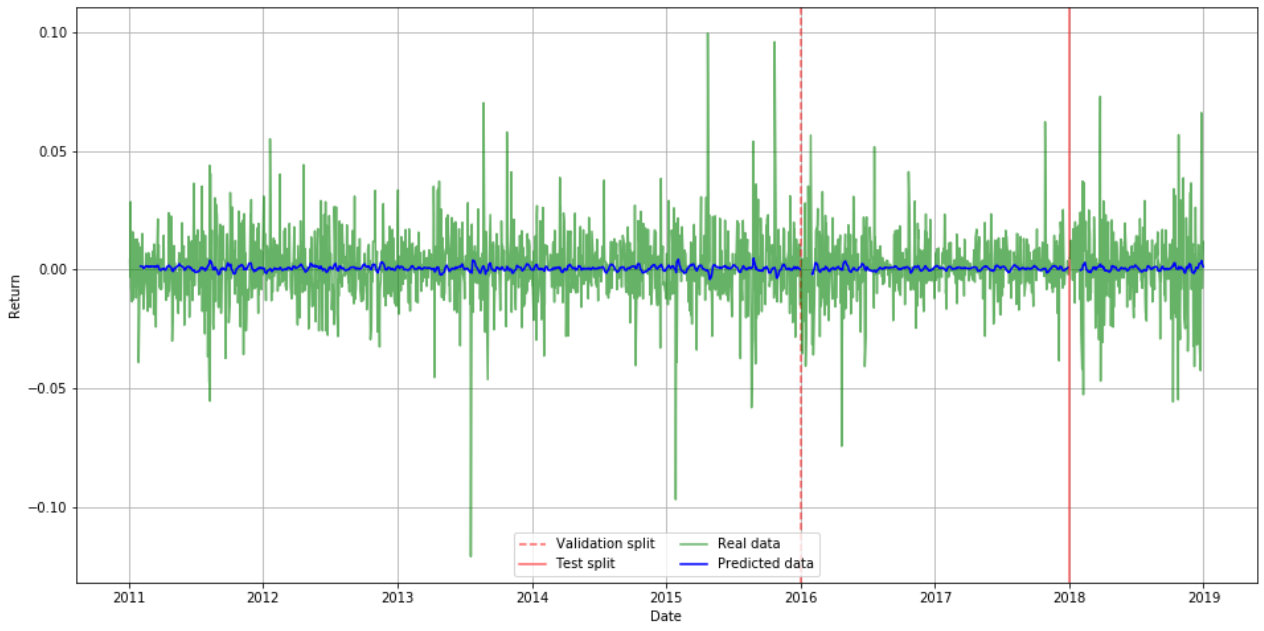

Figure A.6: **MSFT ticker** - Actual and Predicted Log Returns by Univariate LSTM model

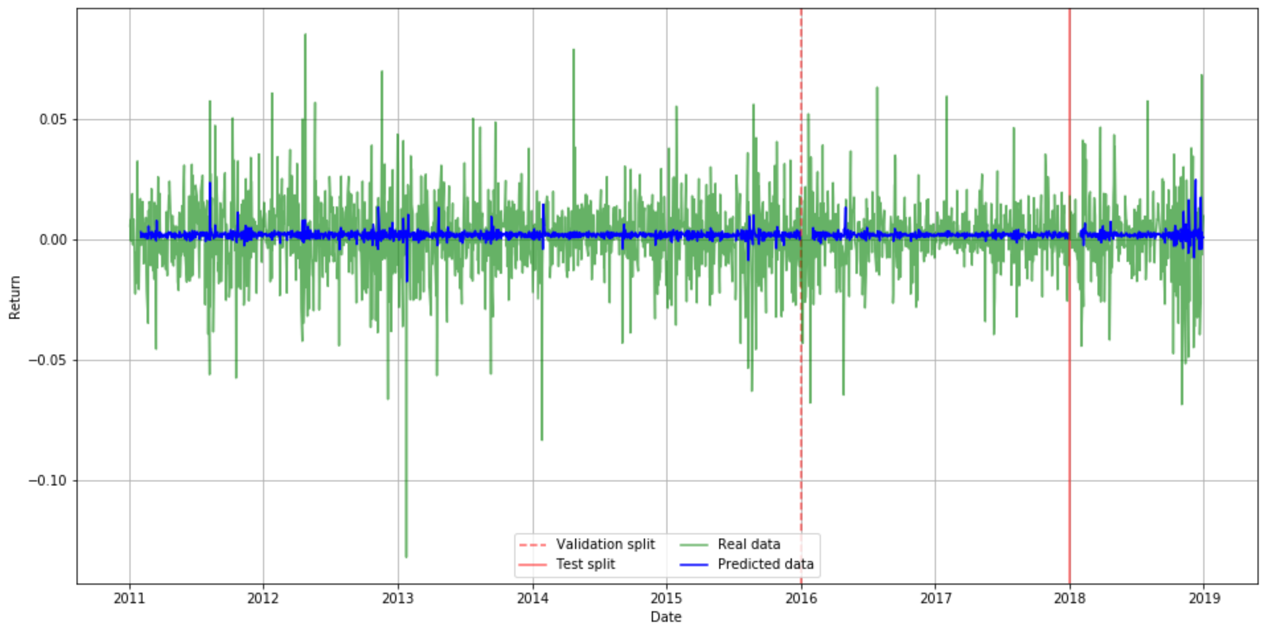

Figure A.7: **AAPL ticker** - Actual and Predicted Log Returns by Univariate LSTM model

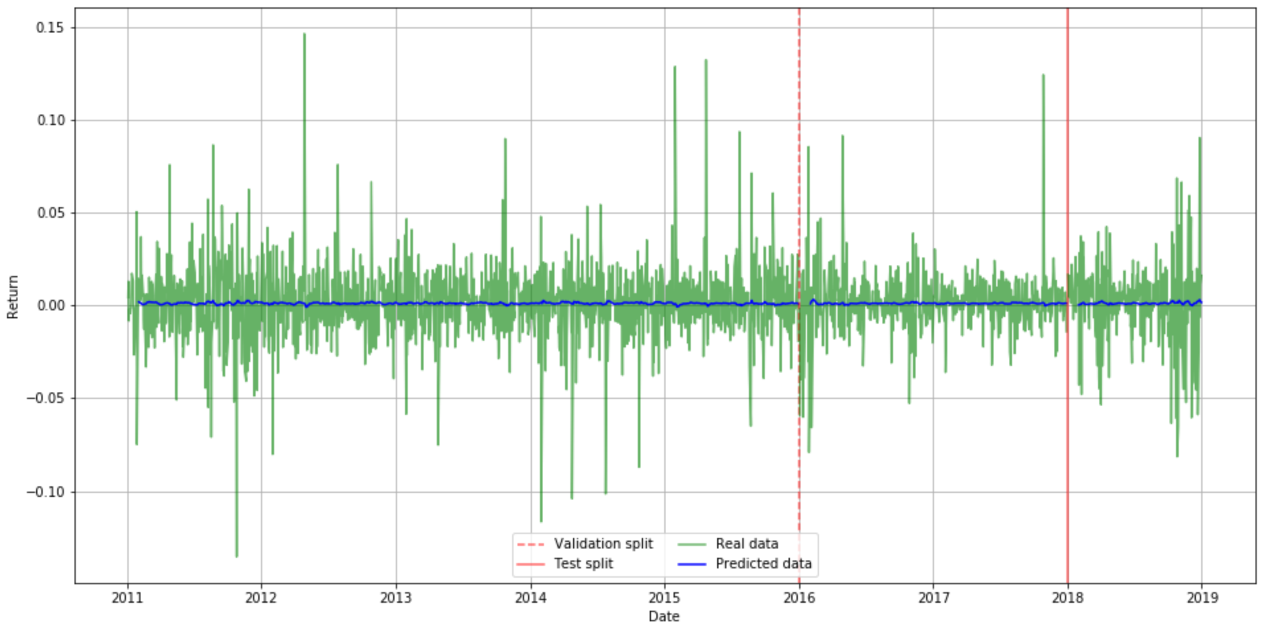

Figure A.8: **AMZN ticker** - Actual and Predicted Log Returns by Univariate LSTM model

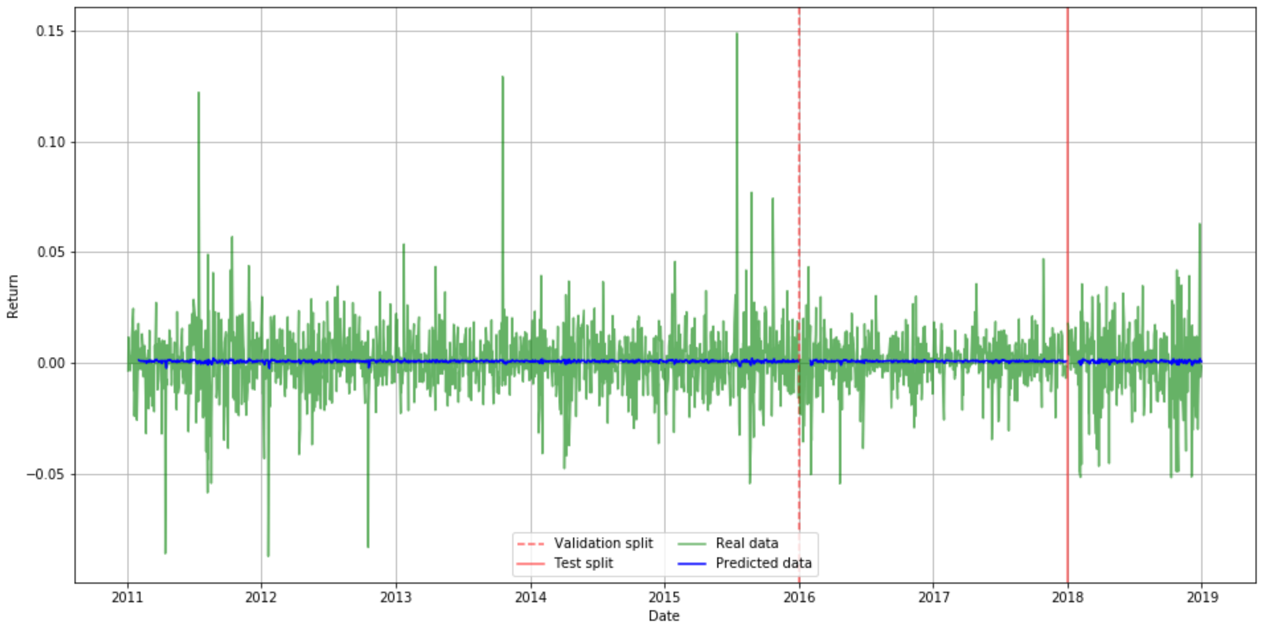

Figure A.9: **GOOG ticker** - Actual and Predicted Log Returns by Univariate LSTM model

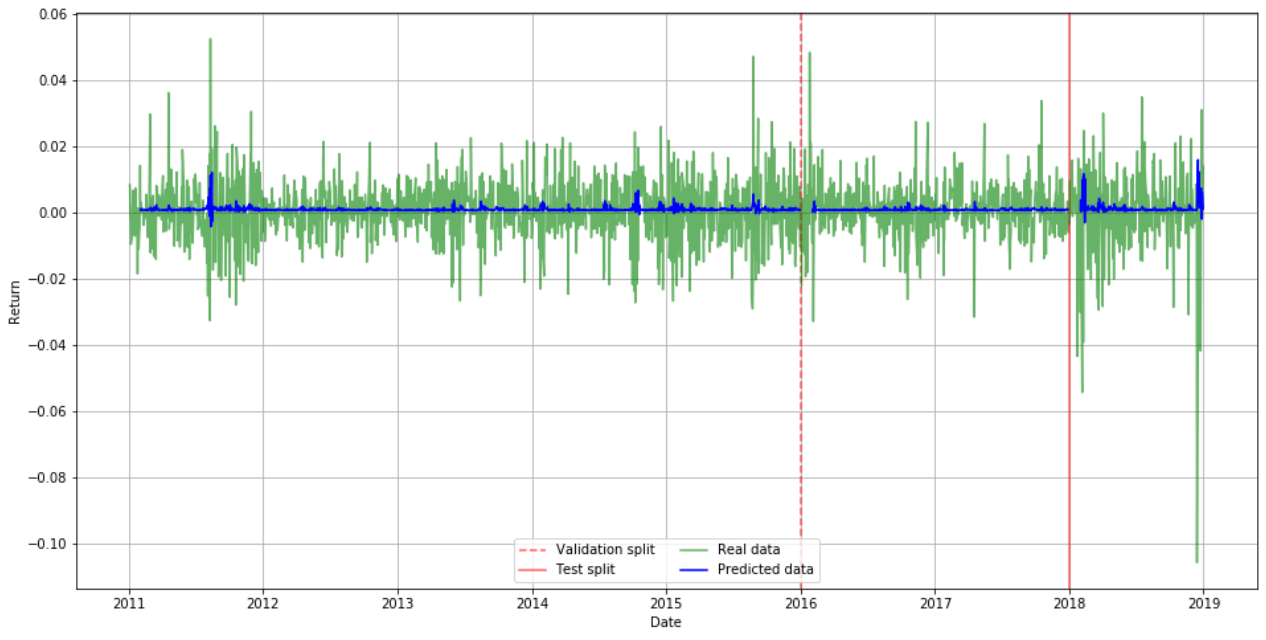

Figure A.10: **JNJ ticker** - Actual and Predicted Log Returns by Univariate LSTM model

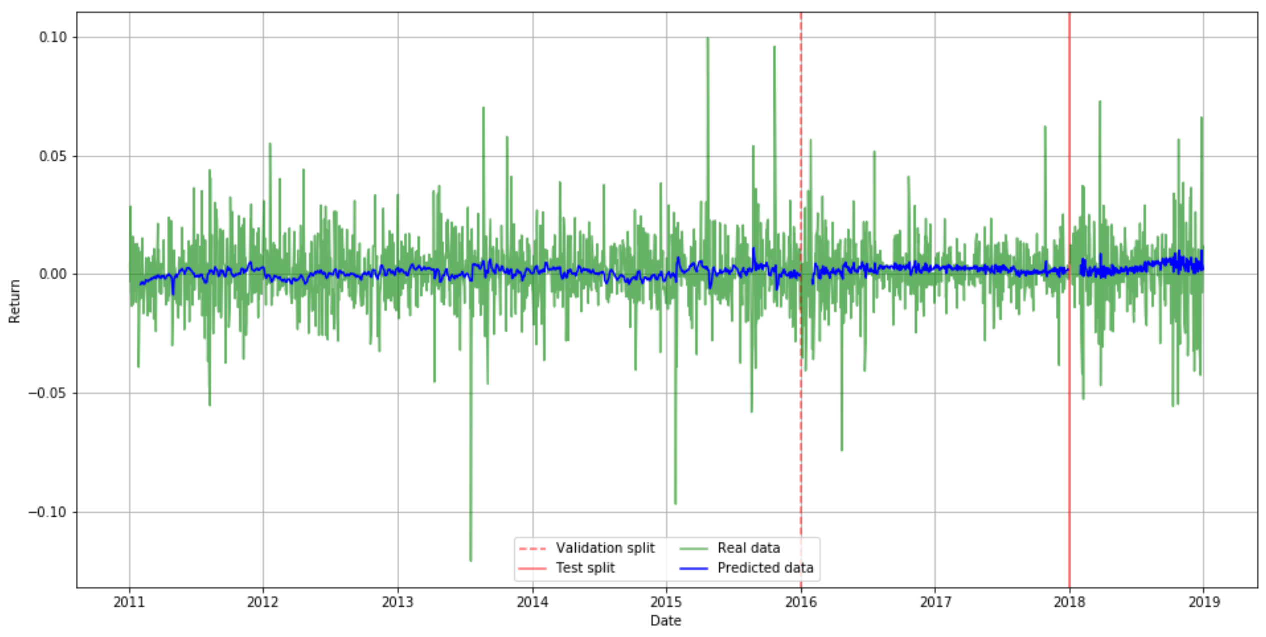

Figure A.11: **MSFT ticker** - Actual and Predicted Log Returns by Multivariate LSTM model

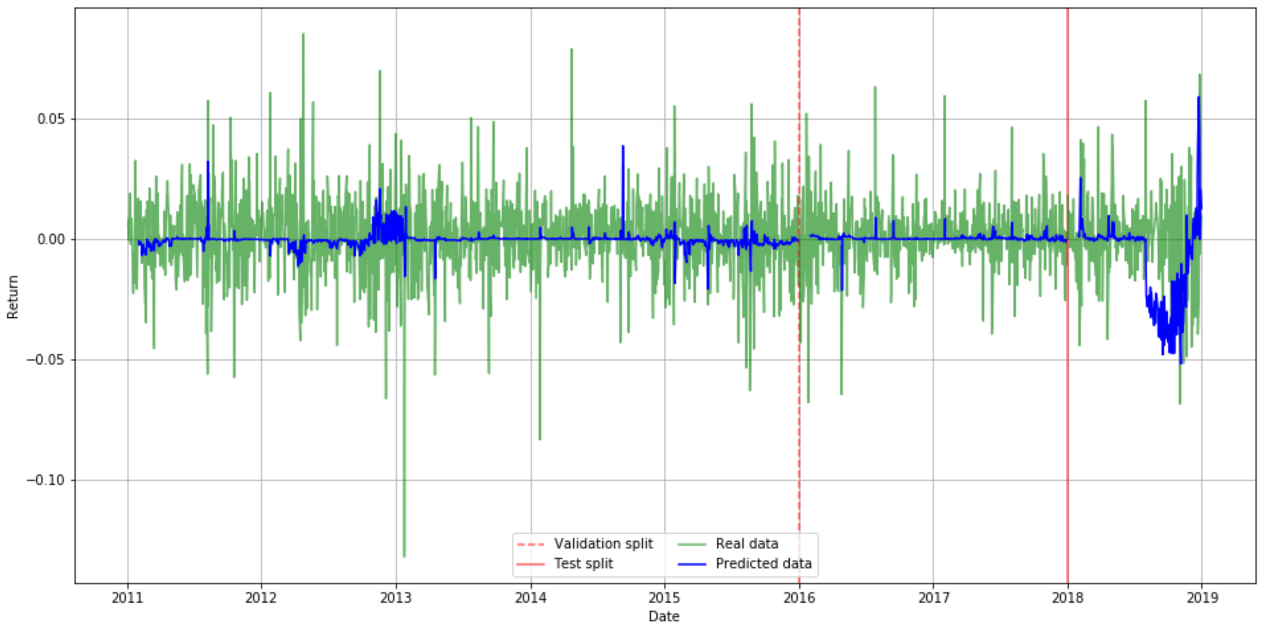

Figure A.12: **AAPL ticker** - Actual and Predicted Log Returns by Multivariate LSTM model

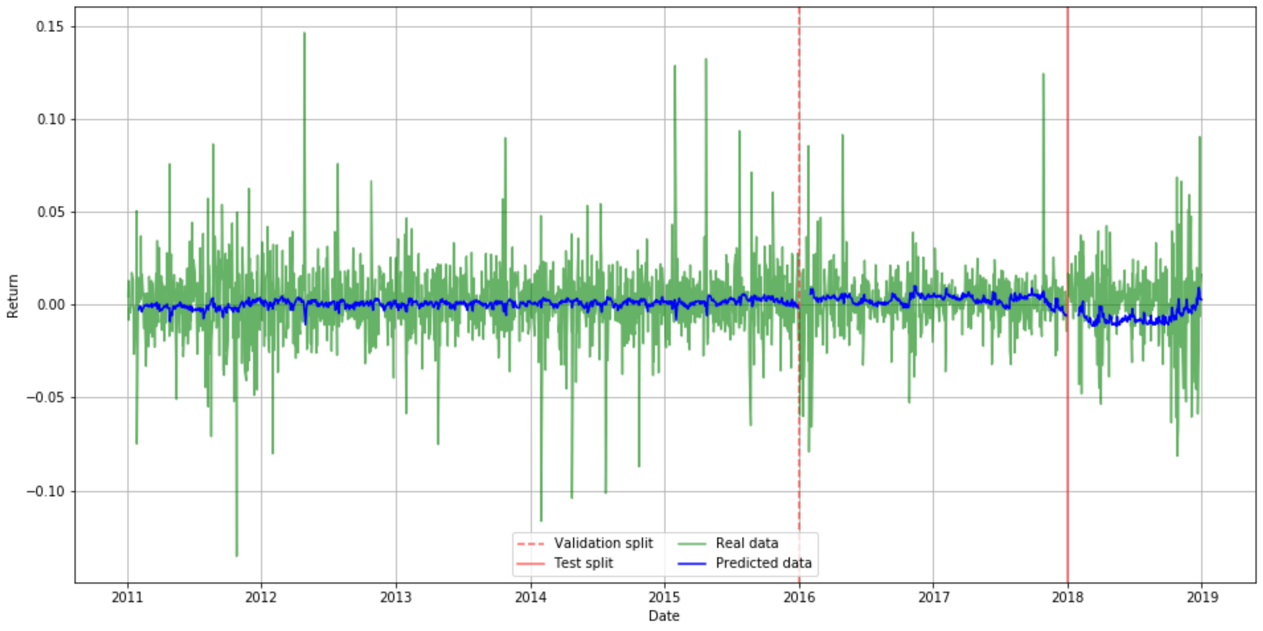

Figure A.13: **AMZN ticker** - Actual and Predicted Log Returns by Multivariate LSTM model

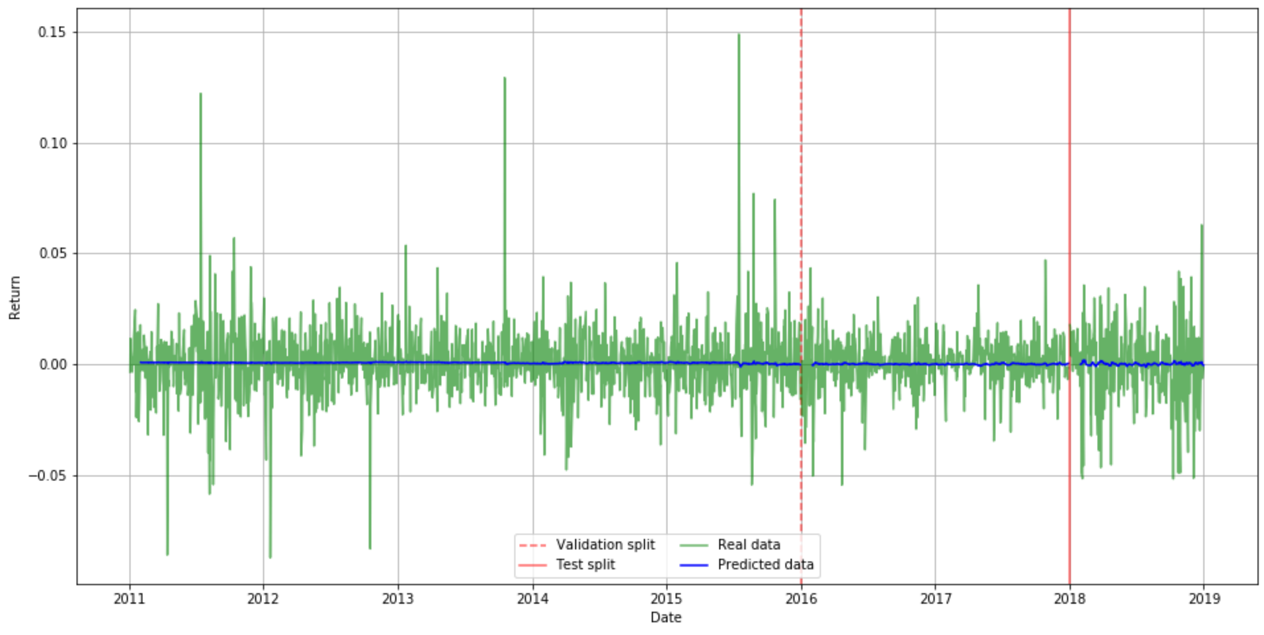

Figure A.14: **GOOG ticker** - Actual and Predicted Log Returns by Multivariate LSTM model

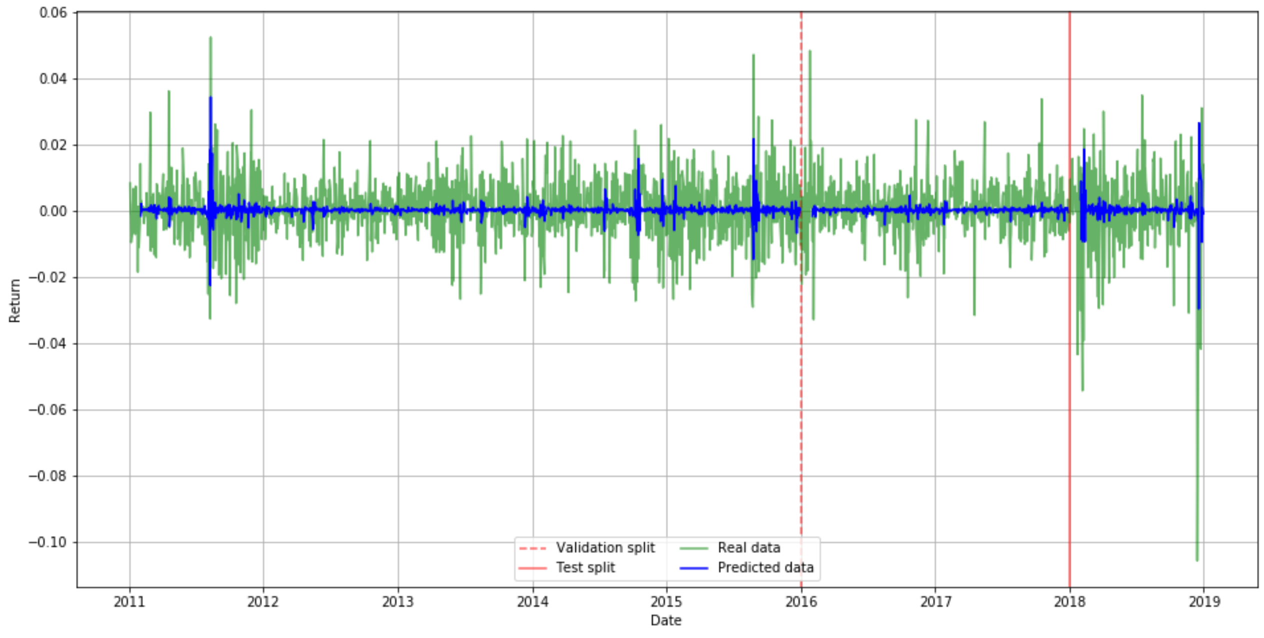

Figure A.15: **JNJ ticker** - Actual and Predicted Log Returns by Multivariate LSTM model

B. Source code - LSTM model

```
# ## Imports and settings
import tensorflow as tf

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import simfin as sf
from simfin.names import *
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import *
from tensorflow import keras
from kerastuner.tuners import RandomSearch
from datetime import date
from sklearn.feature_selection import f_regression,SelectKBest
import pickle
from dm_test import dm_test

sf.set_api_key(api_key="free")
sf.set_data_dir('./simfindata')

def openDF(path):
    with open(path, 'rb') as handle:
        return pickle.load(handle)


# ## Definitions / model settings
# data settings
MARKET = 'us'
TICKERS = ['MSFT', 'AAPL','AMZN', 'GOOG','JNJ']
DATE_FROM = '2011-01-01'
DATE_TO = '2018-12-31'
OFFSET = pd.DateOffset(days=30)
IGNORE_COLUMNS = ['SimFinId','Open','Low','High','Close']
PREDICT_COLUMN = 'Return'
TRAIN_SPLIT = 80
TRAIN_SPLIT_DATE = '2016-01-01'
OUTSAMPLE_SPLIT_DATE = '2018-01-01'
PAST_HISTORY = 20
BATCH_SIZE = 40
BUFFER_SIZE = 10000

#Model fitting settins
TRAIN_EPOCHS=500
STEPS_PER_EPOCH=1 # has effect only when generated data (batch) are inputed

#Tuner settings
MAX_TRIALS = 5
EXECUTIONS_PER_TRIAL = 2

hub = sf.StockHub(market=MARKET, tickers=TICKERS, offset=OFFSET)
tf.random.set_seed(13)

# ## Data work
def get_raw_data(type = 'sinfin'):
    if type == 'sinfin':
        original_df = hub.load_shareprices(variant='daily')
        vol_signals_df = hub.volume_signals(window=20)
        fin_signals_df = hub.fin_signals(variant='daily')
        val_signals_df = hub.val_signals(variant='daily')
        return [original_df, vol_signals_df, fin_signals_df, val_signals_df]

def merge_data(data):
    return pd.concat(data, axis=1)

def filter_by(df, column, val):
    df = pd.DataFrame(df)
    df = df.reset_index()
    df = df.rename(columns={'level_0': 'Ticker', 'level_1': 'Date'})
    df = df[df[column] == val]
    df = df.reset_index(drop=True)
    return df

def create_windows(dataset, target, history_size,
                      target_size, step, single_step=False):
    data = []
    labels = []
    start_index = 0 + history_size
    end_index = len(dataset) - target_size

    for i in range(start_index, end_index):
        indices = range(i-history_size, i, step)
        data.append(dataset[indices])
        if single_step:
            labels.append(target[i+target_size])
        else:
            labels.append(target[i:i+target_size])

    return np.array(data), np.array(labels)

def createReturnColumn(df):
    df['Return'] = np.log(df['Adj.␣Close'])-np.log(df['Adj.␣Close'].shift(1)) #TODO
    # Removes first row, due to NAN
    df = df.iloc[1:]
    return df

raw = get_raw_data()
original = merge_data(raw)
original

# ### Normalize, standardize, split trainset
# Fit scaler from IN df, apply in IN df columns and out df columns
def standardize_columns(tr, val, tst, columns, scaler):
    for col in columns:
        tr[col + '_S'] = scaler.fit_transform(tr[col].values.reshape(-1, 1))
        val[col + '_S'] = scaler.transform(val[col].values.reshape(-1, 1))
        tst[col + '_S'] = scaler.transform(tst[col].values.reshape(-1, 1))
    return [tr,val,tst]

def get_scoped_data(data, columns, predictColumn):
    cols_s = [col + '_S' for col in columns]
    features = data[cols_s]
    values = features.values
    predict_column_index = features.columns.tolist().index(predictColumn)

    return [values, predict_column_index, features]

def feature_selection(df, allColumns, predictedColumn):
     # Feature selection
    cols_without_return = [col for col in allColumns if col != predictedColumn]
    df = df.reset_index()

    df_without = df.loc[:, cols_without_return]
    df_predicted = df.loc[:, predictedColumn]

    fit = SelectKBest(f_regression, k=5).fit(df_without.values, df_predicted.values)
    sc = pd.DataFrame({'col': df_without.columns.tolist(), 'F-scores': fit.scores_, 'p-val': fit.pvalues_})
    sc = sc.sort_values('p-val')
    tmp = sc[sc['p-val'] <= 0.05]
    if tmp.shape[0] == 0:
        tmp = sc[sc['p-val'] <= 0.10]
    if tmp.shape[0] == 0:
        tmp = sc.iloc[0:2, :]
    sc = tmp
    cols = sc['col'].tolist();
    if predictedColumn not in cols:
        cols.append(predictedColumn)
    return cols, sc.sort_values('F-scores', ascending=False)

# ## Model work

def build_model(hp = None):
    if hp == None:
        model = keras.Sequential()
        model.add(tf.keras.layers.LSTM(units=256))
        model.add(tf.keras.layers.Dropout(0.1))
        model.add(tf.keras.layers.Dense(1))
        model.compile(
            optimizer='adam',
            loss='mean_squared_error',
            metrics=['mse']
        )
        return model
    else:
        model = keras.Sequential()
        model.add(tf.keras.layers.LSTM(units=hp.Int('units',
                                                    min_value=16,
                                                    max_value=48,
                                                    step=16)
```

```
                                  ))
        model.add(tf.keras.layers.Dropout(hp.Float('dropout', min_value=0.01, max_value=0.46, step=0.15)))
        model.add(tf.keras.layers.Dense(1))

        optimizer=keras.optimizers.Adam(
                hp.Choice('learning_rate',
                          values=[1e-1, 1e-2, 1e-3, 1e-4]))
        model.compile(optimizer=optimizer, loss='mse', metrics=['mse'])
        return model

def createTuner(build_model, ticker, univariate=False):
    uni = 'uni' if univariate == True else 'multi'
    return RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=MAX_TRIALS,
    executions_per_trial=EXECUTIONS_PER_TRIAL,
    directory='tuner',
    project_name='tune-' + uni + '-' + ticker)

# Attach predicted values of size original-past_history to df
def attachPredictedData(df, attachData, name, past_history):
    outDf = pd.DataFrame(df)
    outDf = outDf.reset_index(drop=True)
    outDf.loc[past_history:len(attachData)+past_history-1, name] = attachData
    return outDf

# ## Processing
def runOneTicker(originalDf, ticker, univariate=True, useTuner=False):
    # Helper instances
    scaler = MinMaxScaler(feature_range=(-1, 1))
    earlyStopping = tf.keras.callbacks.EarlyStopping(
        monitor='val_loss', min_delta=0.0001, patience=50, verbose=0, mode='min',
        baseline=None, restore_best_weights=True
        )

    # Data filtering
    df = filter_by(originalDf, 'Ticker', ticker)
    df = df.sort_values('Date')
    df = df[df.Date >= DATE_FROM]
    df = df[df.Date <= DATE_TO]

    # Create return
    df = createReturnColumn(df)

    # Drop ignore columns and NaN values
    df = df.drop(columns=IGNORE_COLUMNS)
    df = df.dropna(axis=1)

    # Get processing columns
    predictColumnS = PREDICT_COLUMN + '_S'
    columns = None
    if univariate:
        columns = ['Return']
    else:
        columns = [col for col in df.columns.tolist() if col not in ['Date', 'Ticker']]

    # Fallback, dropna for rows
    df = df.dropna(axis='index', subset=columns)

    # Split DF into train, val and outsample data
    insample_data = pd.DataFrame(df[df.Date < OUTSAMPLE_SPLIT_DATE])
    train_data = pd.DataFrame(insample_data[insample_data.Date < TRAIN_SPLIT_DATE])
    val_data = pd.DataFrame(insample_data[insample_data.Date >= TRAIN_SPLIT_DATE])
    outsample_data = pd.DataFrame(df[df.Date >= OUTSAMPLE_SPLIT_DATE])

    # Feature selection
    fscores_df = None
    if univariate:
        selectedColumns = columns
    else:
        selectedColumns, fscores_df = feature_selection(train_data, columns, PREDICT_COLUMN)

    #standardize columns
    standardize_columns(train_data, val_data, outsample_data, selectedColumns, scaler=scaler)

    # create subset of columns/features for model fitting
    train_values, train_column_index, train_df = get_scoped_data(train_data, selectedColumns, predictColumnS)
    val_values, val_column_index, val_df = get_scoped_data(val_data, selectedColumns, predictColumnS)
    outsample_values, outsample_column_index, outsample_df = get_scoped_data(outsample_data, selectedColumns, predictColumnS)

    # configure constants for data splitting and sampling
    past_history = PAST_HISTORY
    # index of next predicting observation
    future_target = 0
    # get every n-th value from window = STEP
    STEP = 1

    #Create input tensors with past time windows for NN
    train_x, train_y = create_windows(train_values, train_values[:, train_column_index],
                                                    past_history,
                                                    future_target, STEP,
                                                    single_step=True)
    val_x, val_y = create_windows(val_values, val_values[:, val_column_index],
                                                past_history,
                                                future_target, STEP,
                                                single_step=True)
    test_x, test_y = create_windows(outsample_values, outsample_values[:, outsample_column_index],
                                                past_history,
                                                future_target, STEP,
                                                single_step=True)

    model=None
    history=None
    hyperparams=None
    # build and fit model
    if useTuner:
        tuner = createTuner(build_model, ticker, univariate)
        tuner.search_space_summary()
        history = tuner.search(x=train_x,
             y=train_y,
             epochs=TRAIN_EPOCHS,
             validation_data=(val_x, val_y),
             callbacks=[earlyStopping]
            )
        tuner.results_summary()
        hyperparams = tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
        model = tuner.get_best_models()[0]
        history = model.fit(x=train_x,y=train_y, epochs=TRAIN_EPOCHS, validation_data=(val_x, val_y), callbacks=[earlyStopping])
    else:
        model = build_model()
        history = model.fit(x=train_x,y=train_y, epochs=TRAIN_EPOCHS, validation_data=(val_x, val_y), callbacks=[earlyStopping])

    # create fitter for inverese scaling for certain column
    fitter = scaler.fit(train_data['Return'].values.reshape(-1, 1))

    # Predict train, val and test data, inverse scale all of them
    # Use when Return was standardized
    predicted_train = model.predict(train_x)
    inv_predicted_train = fitter.inverse_transform(predicted_train)
    inv_actual_train = fitter.inverse_transform(train_y.reshape(-1, 1))

    predicted_val = model.predict(val_x)
    inv_predicted_val = fitter.inverse_transform(predicted_val)
    inv_actual_val = fitter.inverse_transform(val_y.reshape(-1, 1))

    predicted_test = model.predict(test_x)
    inv_predicted_test = fitter.inverse_transform(predicted_test)
    inv_actual_test = fitter.inverse_transform(test_y.reshape(-1, 1))

    outTrain = attachPredictedData(train_data[['Date', 'Return']], inv_predicted_train, 'Predicted', past_history)
    outVal = attachPredictedData(val_data[['Date', 'Return']], inv_predicted_val, 'Predicted', past_history)
    outTest = attachPredictedData(outsample_data[['Date', 'Return']], inv_predicted_test, 'Predicted', past_history)

    return { 'univariate': univariate,
            'useTuner': useTuner,
            'hyperparams': hyperparams,
            'columns': selectedColumns,
            'fscores': fscores_df,
            'rmse_train': np.sqrt(mean_squared_error(inv_actual_train, inv_predicted_train)),
            'rmse_valid': np.sqrt(mean_squared_error(inv_actual_val, inv_predicted_val)),
            'rmse_test': np.sqrt(mean_squared_error(inv_actual_test, inv_predicted_test)),
            'out_test': outTest
           }

#UNI
outHP = pd.DataFrame(columns=['Ticker', 'units', 'dropout', 'learning_rate'])
rmse = pd.DataFrame(columns=['Ticker', 'RMSE␣Train', 'RMSE␣Validation', 'RMSE␣Test'])
outsamples = {}
for ticker in TICKERS:
    tickDict = runOneTicker(original, ticker, univariate=True, useTuner=True)
    hpactual = tickDict['hyperparams']
    outHP = outHP.append({'Ticker': ticker, 'units': hpactual['units'], 'dropout': hpactual['dropout'], 'learning_rate': hpactual['learning_rate']}, ignore_index
          =True)
    rmse = rmse.append({'Ticker': ticker, 'RMSE␣Train': tickDict['rmse_train'], 'RMSE␣Validation': tickDict['rmse_valid'], 'RMSE␣Test': tickDict['rmse_test'
          ]},
                              ignore_index=True)
    outsamples[ticker] = tickDict['out_test']

#MULTI
outHPM = pd.DataFrame(columns=['Ticker', 'units', 'dropout', 'learning_rate'])
outFScoresM = {}
rmseM = pd.DataFrame(columns=['Ticker', 'RMSE␣Train', 'RMSE␣Validation', 'RMSE␣Test'])
outsamplesM = {}
for ticker in TICKERS:
    tickDict = runOneTicker(original, ticker, univariate=False, useTuner=True)

    hpactual = tickDict['hyperparams']
    outHPM = outHPM.append({'Ticker': ticker, 'units': hpactual['units'], 'dropout': hpactual['dropout'], 'learning_rate': hpactual['learning_rate']},
            ignore_index=True)

    outFScoresM[ticker] = tickDict['fscores']
    rmseM = rmseM.append({'Ticker': ticker, 'RMSE␣Train': tickDict['rmse_train'], 'RMSE␣Validation': tickDict['rmse_valid'], 'RMSE␣Test': tickDict['
            rmse_test']},
                              ignore_index=True)
    outsamplesM[ticker] = tickDict['out_test']
```

C. Source code - GARCH model

```
# ## Settings
sf.set_api_key(api_key="free")
sf.set_data_dir('./simfindata')

# data settings
MARKET = 'us'
TICKERS = ['MSFT', 'AAPL','AMZN', 'GOOG','JNJ']
DATE_FROM = '2011−01−01'
DATE_TO = '2018−12−31'
OFFSET = pd.DateOffset(days=30)
OUTSAMPLE_SPLIT_DATE = '2018−01−01'

hub = sf.StockHub(market=MARKET, tickers=TICKERS, offset=OFFSET)

# ## Data work
def get_raw_data(type = 'sinfin'):
    if type == 'sinfin':
        original_df = hub.load_shareprices(variant='daily')
        vol_signals_df = hub.volume_signals(window=20)
        fin_signals_df = hub.fin_signals(variant='daily')
        val_signals_df = hub.val_signals(variant='daily')
        return [original_df, vol_signals_df, fin_signals_df, val_signals_df]

def merge_data(data):
    return pd.concat(data, axis=1)

def filter_by(df, column, val):
    df = pd.DataFrame(df)
    df = df.reset_index()
    df = df.rename(columns={'level_0': 'Ticker', 'level_1': 'Date'})
    df = df[df[column] == val]
    df = df.reset_index(drop=True)
    return df

def createReturnColumn(df):
    df['Return'] = np.log(df['Adj.␣Close'])−np.log(df['Adj.␣Close'].shift(1)) #TODO
    # Removes first row, due to NAN
    df = df.iloc[1:]
    return df

# Load data
raw = get_raw_data()
original = merge_data(raw)


# ## Time series description
def stockDescription(df, ticker):
    out = pd.DataFrame(df['Return']).describe(percentiles=[])
    out = out.rename(columns={'Return': ticker})
    out = out.reset_index()
    out = out.append({'index':
                'kurtosis', ticker: stats.kurtosis(df['Return'])}, ignore_index=True)
    out = out.append({'index':
            'skewness', ticker: stats.skew(df['Return'])}, ignore_index=True)
    return out

def testStationarity(df, ticker):
    trends = ['n', 'c', 'ct']
    labels = {'n': 'No␣trend␣components', 'c': 'Constant', 'ct': 'Constant␣and␣linear␣trend'}
    out = pd.DataFrame(columns=['Ticker','Trend␣component', 'ADF␣Test␣Statistic', 'ADF␣P−Value', 'ADF␣Lags', 'ADF␣Critical␣values␣5%', 'P−P␣Test␣
        Statistic', 'P−P␣P−Value', 'P−P␣Lags', 'P−P␣Critical␣values␣5%'])
    for trend in trends:
        ad = ADF(df['Return'], trend=trend)
        pp = PhillipsPerron(df['Return'], trend=trend)
        out = out.append({'Ticker':ticker,
                    'Trend␣component': labels[trend],
                    'ADF␣Test␣Statistic': ad.stat,
                    'ADF␣P−Value': ad.pvalue,
                    'ADF␣Lags': ad.lags,
                    'ADF␣Critical␣values␣5%': ad.critical_values['5%'],
                    'P−P␣Test␣Statistic': pp.stat,
                    'P−P␣P−Value': pp.pvalue,
                    'P−P␣Lags': pp.lags,
                    'P−P␣Critical␣values␣5%': pp.critical_values['5%'],
                    }, ignore_index=True)
    out = out.round(decimals=2)
    out = out.set_index(['Ticker','Trend␣component'])
    return out

# ## Time series forecasting
def getBestArima(data):
    """ returns best order (p, q) of arima on insample data """
    res = {}
    res_df = pd.DataFrame(columns=['P', 'Q', 'AIC'])
    for p in range(1,4):
        for q in range(0,4):
            model = ARIMA(data, order=(p, 0, q))
            aic = model.fit().aic
            key = 'pq{}{}'.format(p, q)
            res[key] = aic
            res_df = res_df.append({'P': p, 'Q': q, 'AIC': aic}, ignore_index=True)
    res_df = res_df.sort_values(by='AIC').head(1)
    model_best = ARIMA(data, order=(int(res_df['P'].values[0]), 0, int(res_df['Q'].values[0])))
    dw_test = durbin_watson(model_best.fit().resid)
    bp_test = het_breuschpagan(model_best.fit().resid, data)
    return res_df['P'].values[0], res_df['Q'].values[0], dw_test, bp_test[2], bp_test[3]

def fit_arch(data, lags, p, q, val_split_date):
    am = arch_model(data, mean='AR',lags=lags,p=p,q=q, vol='Garch')
    arch_result = am.fit(disp='final', last_obs=pd.to_datetime(val_split_date))
    return arch_result

def getBestArch(data, val_split_date):
    res = {}
    res_df = pd.DataFrame(columns=['lag','P', 'Q', 'AIC'])
    for lag in range(1,4):
        for p in range(1,3):
            for q in range(1,3):
                fitted = fit_arch(data, lag, p, q, val_split_date)
                aic = fitted.aic
                res_df = res_df.append({'lag': lag, 'P': p, 'Q': q, 'AIC': aic}, ignore_index=True)
    res_df = res_df.sort_values(by='AIC').head(1)
    model_best = fit_arch(data, lag, int(res_df['P'].values[0]), int(res_df['Q'].values[0]), val_split_date)
    return model_best, int(res_df['lag'].values[0]), int(res_df['P'].values[0]), int(res_df['Q'].values[0]), int(res_df['AIC'].values[0]),


# ## Plots and outputs
def createOutput(data, forecasts):
    out = pd.DataFrame(data)
    out['Predicted'] = forecasts
    return out

def runOneTicker(originalDf, ticker):
    # Data filtering
    df = filter_by(originalDf, 'Ticker', ticker)
    df = df.sort_values('Date')
    df = df[df.Date >= DATE_FROM]
    df = df[df.Date <= DATE_TO]

    # Create return
    df = createReturnColumn(df)
    df = df[['Date','Return']]
    desc = stockDescription(df, ticker)
    stationarity = testStationarity(df, ticker)

    # Split DF into train, val and outsample data
    insample_data = pd.DataFrame(df[df.Date < OUTSAMPLE_SPLIT_DATE])
    outsample_data = pd.DataFrame(df[df.Date >= OUTSAMPLE_SPLIT_DATE])
    # Set date as index
    insample_data = insample_data.set_index('Date',drop=True)
    stationarity = testStationarity(insample_data, ticker)

    ar_p, ar_q, dw, bp1, bp2 = getBestArima(insample_data)

    outsample_data = outsample_data.set_index('Date',drop=True)
    arch_scaler = 100
    arch_data = pd.DataFrame(df).set_index('Date', drop=True)
    arch_data_scaled = arch_data * arch_scaler
    fitted, garch_lag, garch_p, garch_q, garch_aic = getBestArch(arch_data_scaled, OUTSAMPLE_SPLIT_DATE)

    forecasts = fitted.forecast(horizon=1)
    arch_forecast_scaled = forecasts.mean.dropna()
    arch_forecast = arch_forecast_scaled / arch_scaler

    mse = mean_squared_error(arch_data.iloc[-len(arch_forecast.dropna()):], arch_forecast)
    rmse = np.sqrt(mse)
    garch_output_outsample = createOutput(outsample_data, arch_forecast)

    return {'desc': desc, 'stat': stationarity, 'garch_forecasts': garch_output_outsample, 'arima_stats': [ar_p, ar_q, dw, bp1, bp2], 'garch_stats': [garch_lag,
            garch_p, garch_q, garch_aic, rmse]}
```

```
for ticker in TICKERS:
    tickDict = runOneTicker(original, ticker)
```