June 2021 Cohort

CQF Final Project
Deep Learning for Time Series

Andrea Russo[†]

[†]*Department of Physics and Astronomy, University College London, London WC1E 6BT, UK*

E-mail: andrea.russo.19@ucl.ac.uk, andrearusso.physics@gmail.com

Abstract: Models of price direction forecasting through deep learning algorithms live at the frontier of quantitative finance. In this project we develop an LSTM/GRU based binary classification neural network aimed at 5-days price direction forecasting for Advanced Micro Devices and JP Morgan stocks. We perform feature extraction and selection to address multicollinearity. Subsequently, we implement a deep learning architecture and hypertune its parameter through Bayesian optimisation. We evaluate the model through standard metrics like accuracy, F1-score and ROC curves. We find that the model is subject to slight overfitting but produces satisfying results.

Contents

1 Introduction

1.1 Deep learning in finance

Machine learning is a branch of computer science which focuses on the construction of self-optimizing algorithms. Given a performance metric, these algorithms are able to learn form the input data and tweak their internal parameters to achieve the best possible results.

Regardless of the fact that the first neural network is dated back to 1950 [1], machine learning algorithm have become increasingly popular only in the last decade. The reason behind this surge in popularity is based on the massive increase in the amount of information available. Machine learning algorithms are incredibly data-hungry, therefore their applicability to many fields was limited until recent years, when the amount of data has become sufficient for their implementation. From physics to healthcare, form tech to finance, more and more fields are starting to apply machine learning techniques to optimise and improve performances.

The use of machine learning has become ever so prominent in the world of quantitative finance. Among all the different machine learning algorithms, a lot of attention has been recently drawn to the implementation of deep learning architectures. Differently from most machine learning algorithms, deep learning is a subset of machine learning based on the construction of neural networks (NN). A neural network is a composed by objects called 'neurons' which are organised in connected layers. Each neuron is a mathematical function capable of taking a linear combinations of the inputs and map it to an output. A neuron layer receives its inputs directly from the dataset or from another layer of neurons, process it and passes it on to the next neuron layer. The output of the final layer is then taken as the output of the neural network. This will be further explained in the appropriate section.

1.2 Goals and structure

For this project, we developed and optimized a neural network aimed at assessing price direction forecasting for 2 assets in a 5 days window. This is a binary classification problem in which the deep learning algorithm was required to output a 0 or 1 corresponding to a prediction of a price increase, with 1, or price decrease, with 0. It has been decided that a stationary price was also going to be considered to be a 0 output.

The chosen assets were the two equities corresponding to Advanced Micro Devices stock, with ticker 'AMD', and JP Morgan stock with ticker 'JPM'. These equities were chosen because they belong to two different industry sectors. AMD is a semiconductor/microchip company while JP Morgan operates in the banking and financial sector.

The choice was made in order to avoid sector-wide correlations and have a more unbiased evaluation of the deep learning architecture.

The structure of the report will be the following:

Section 2: we will discuss feature engineering, this comprises everything from data collection to data cleaning, feature extraction and feature selection. The aim of the section is to boil down the feature datasets for each equity to less than a hundred features from the initial number of over two hundred.

Section 3: Once the final features have been selected, we construct the Neural Network. In this section we use Keras and Tensorflow to first generate an appropriate time series form the feature selected, and then to construct a sequential LSTM/GRU neural network for a 5-day price forecasting. The network presents many hyperparameters that we tune using a Bayesian optimiser, aiming for the neural network architecture that gives the best accuracy on the valuation set. We avoid overfitting trough appropriate callback functions.

Section 4: we analyse the results obtained when asking the previously trained networks to perform label prediction on the test dataset. We run a full performance analysis tailored for binary classification algorithms which includes train and test accuracy, precision, recall, F_1-scores confusion matrices, ROC curves plot and probability scatter plots. This allows us to fully evaluate the model. We find that overfitting is avoided and the performances are satisfactory.

Section 5: Lastly, we summarise the main points of the work and highlight any interesting results. We reflect on the model shortcomings and we talk about possible future improvements.

2 Feature engineering

In this section we explain how we collected relevant data and performed feature engineering in order to construct an appropriate feature set for AMD and JPM tickers.

Data collection and feature engineering refer to the process of constructing, scaling and selecting features form a set of collected data such that the features are optimised to be used in a machine learning algorithm. The importance of feature engineering is twofold and the process can be summarised into a sequence of steps namely data collection, data cleaning, feature extraction, feature rescaling and feature selection. Having clean data and features that are rescaled properly improves the accuracy and overall performance of the machine learning algorithm. Moreover, having appropriately rescaled data is fundamental if one wants to compare features with different numerical scales and orders of magnitude. Nevertheless, even if the features are appropriately cleaned and rescaled, their number might be too large. A large number of unfiltered features carries the problem of multicollinearity, meaning that redundant information could be present and slow down the computations. When feature selection is correctly carried out, redundant information in the feature set is minimised. This allows the algorithm to learn the same amount of relevant information from a smaller set of features, improving its speed and performance.

2.1 Data collection

Data collection and cleaning refers to the process of collecting data form reliable sources and checking if the dataset is missing any values or if it is corrupted in any way. For this project, the source of data was chosen to be Yahoo Finance. We downloaded 30 years worth of Open, High, Low, Close, Adjusted Close and Volume data for AMD and JPM. The 30 years span from the 1st of January 1990 to the 5th of January 2022. Additionally, we downloaded the same amount of data for the volatility index (VIX), for the thirty, ten and five years treasury yield (TYX,TNX,FVX) and for the Philadelphia Gold and Silver index (XAU). These extra tickers were downloaded to be used in the construction of features capable of giving insight into the global market conditions, which might be a factor influencing price direction forecasting. The data was checked for missing values but resulted complete. The data was then saved in csv format and loaded into pandas dataframes

2.2 Feature extraction

The aim of feature extraction is to create a list of features from the dataset available. Ideally, all the relevant indicators are included. For this project, we extracted lagged returns with intervals of 5 days up to 55 days, and momentum up to 56 days. Additionally, we made sure that there was no overlap between the momentum and the lagged returns. The lagged returns and the momentum where then transformed into binary data, meaning that a positive values were transformed into ones and a negative or zero values were transformed into zeros. The encoding into binary values helps the machine learning algorithm to achieve better results in binary classification problems.

We also decided to insert features with the explicit purpose of monitoring the volatility in the equities price and in the global market. Periods of high and low volatility should influence the price forecasting of the asset. The role of volatility indicator features was played by the realised volatility computed up to 25 days in the past at intervals of 3 days, and by the Average True Range computed up to 30 days at intervals of 2 days. The value of the VIX index was then inserted as a feature to measure the overall volatility of the market. Much more can be said on the nature of volatility, and some considerations and insights obtained from the 'Advanced Volatility Modeling' are reported in Appendix A

We also included 50-100-150-200 SMA for the XAU gold and silver index. This index was chosen as it was one of the few tickers with enough data retrievable from Yahoo finance to be comparable with the time period of the chosen assets. The role of this ticker is to give an indication about trends of precious metal prices, which are a measure of inflation in the overall market.

Lastly, we included the spread between the 30 years and 5 years US treasury yields and between the 10 years and 5 years US treasury yields. This will also act as an indicator of market trends such as flights to safety and market sentiment.

At this point, we call on the pandas.ta library to create a list of about 200 technical indicators for each asset. The full list of the technical indicators generated can be found in [2]

Having created all the features, we defined the target label. Given that we aimed at a 5-days direction forecast, our target was easily computed by looking at the future returns and taking its sign.

2.3 Feature cleaning and rescaling

Once the features were defined, we moved onto the cleaning and rescaling phase. The first step was to eliminate all the features that had more that 5% of missing data. This was done because backfitting/forwardfitting more than 5% of data implies a real possibility of skewing the results of the feature selection algorithms. After getting rid of those features, the remaining missing values were related to features using a certain lookback period. The missing data were therefore placed at the chronological beginning of the dataset, and we decided to drop them completely since they constituted a negligible amount of data points compared to the full 30 years window.

We then moved to address class imbalances, which can result when the target labels are heavily skewed in one class with respect to another. In the case under consideration, the market tendency of these equities to increase their values led to imbalance in the up moves with respect to the down/stationary moves. This was easily fixed by defining class weights, which forced the machine learning algorithm to spend more training time on the class with less elements, as explained in [3], such that the skewness of the labels was not a concern anymore.

At this point, the datasets were split into train, validation and test dataset with a 60-20-20 split while being careful not to shuffle the data. Moreover, to avoid information leaking, given that we are aiming at a 5 day price direction forecast, the last five data points of the train and validation sets were deleted. This was done to avoid information from the future leaking to past data.

Once all the features had been cleaned, they had to be rescaled. Feature rescaling is a process that presents some subtleties. The choice of rescaling is fundamental in order to have an effective and coherent way of comparing different features. After careful considerations, our choice of scaling method settled for the RobustScaler. The reason behind this choice originated from the fact that none of the features could be reasonably assumed to be Gaussian distributed, especially those put in binary form. This automatically eliminated all the rescaling methods aimed at normalising and centering Gaussian features, like StandardScaler, or aimed at making features Gaussian, like PowerTransformer.

Of the wide variety of rescaling method left, two made sense more than others: MinMaxScaler and RobustScaler. The choice settled for the latter. Given that outliers in returns and price movements often corresponds with market crashes or other relevant events, we felt like it made sense to include them without having to worry about skewing the rescaling of most data points, which is a concrete possibility if MinMaxScaler is used. RobustScaler is a scaler that is particularly resistant to outliers. It works by removing the median and scales the data according to the quantile range specified.

$$x_{scaled} = \frac{x - Q_1(x)}{Q_3(x) - Q_1(x)} \tag{2.1}$$

The centring and scaling statistics of this method are based on percentiles and are therefore not influenced by a few numbers of huge marginal outliers. Outliers themselves are still present in the transformed data. More on feature scaling can be read at [4]

Therefore, the RobustScaler was trained on the training set only, to avoid information leaking, and then applied to the train, validation and test set.

2.4 Feature selection

When constructing any kind of machine learning algorithm, having enough features is fundamental if one's goal is to maximise the predicting power. Nevertheless, an elevated number of features carries a high probability of multicollinearity. Multicollinearity is the presence of correlation between different features and it is detrimental to the speed of the machine learning algorithm. Correlations between features implies that there is a redundancy in the information that a feature adds to the set therefore, when the program processes a set of correlated features, it is wasting time and computational power without gaining much predictive benefits. Moreover, some of the features may not be relevant at all for the needed task.

In order to address these problems, the number of features is reduced through the implementation of feature selecting methods, which are aimed at addressing and removing irrelevant features and multicollinearity. In this project, we opted for a two step feature selection process. First, we run BorutaPy, and then we implement two types of unsupervised learning algorithms on the selected features: K-means clustering and Self Organising Maps. The choice of these two methods of unsuperviesed learning was based on their ability to handle non linear relations between data.

BorutaPy

BorutaPy is a feature selection algorithm based on the RandomForest classifier. Random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees.

BorutaPy takes an 'all relevant' approach, meaning that it works well for getting rid of features which are the least relevant, but has its drawbacks. Specifically, BorutaPy is a linear method and therefore struggles to address non linearity in the feature set. To read more about BorutaPy, check [5]. Therefore, after selecting the most relevant features highlighted by BorutaPy, we packed them into new dataset and we run them through K-means clustering and Self Organising Maps.

K-means clustering

K-means clustering is a method of vector quantisation that aims at partitioning n observations into K clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. Therefore we will then aim at taking only one feature per cluster, which will further reduce the amount of selected features.

Given a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where each observation is a d-dimensional real vector, K-means clustering aims to partition the n observations into $K \leq n$ sets $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} |\mathbf{x} - \boldsymbol{\mu}_i|^2 = \underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} |S_i| \mathrm{Var} S_i \tag{2.2}$$

where $\boldsymbol{\mu}_i$ is the mean of points in S_i. This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \frac{1}{2\,|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2 \tag{2.3}$$

Because the total variance is constant, this is equivalent to maximizing the sum of squared deviations between points in different clusters which follows from the law of total variance.

When running a K-means clustering classification, the number of clusters is an hyper parameter that can be tuned in order to optimize the results. For this project, we resorted to the 'Elbow Method'. The Elbow Method is one of the most popular methods to determine this optimal value of K. It iterates the values of K in a given interval, which was chosen to be $K \in [2, 44]$, and for each iterations measures the value of inertia (the same could have been done measuring distortion). By inertia we mean the sum of squared distances of samples to their closest cluster center. The value of inertia decreases rapidly at first but tends to plateau after a certain number of clusters. To determine the optimal number of clusters, we selected the value of K at the "elbow", which is the point after which the inertia starts decreasing in a linear fashion. In the specific, we chose the value of K corresponding to the 5% relative inertia, which consists of 41 clusters for AMD and 15 for JPM as shown in Figure 1

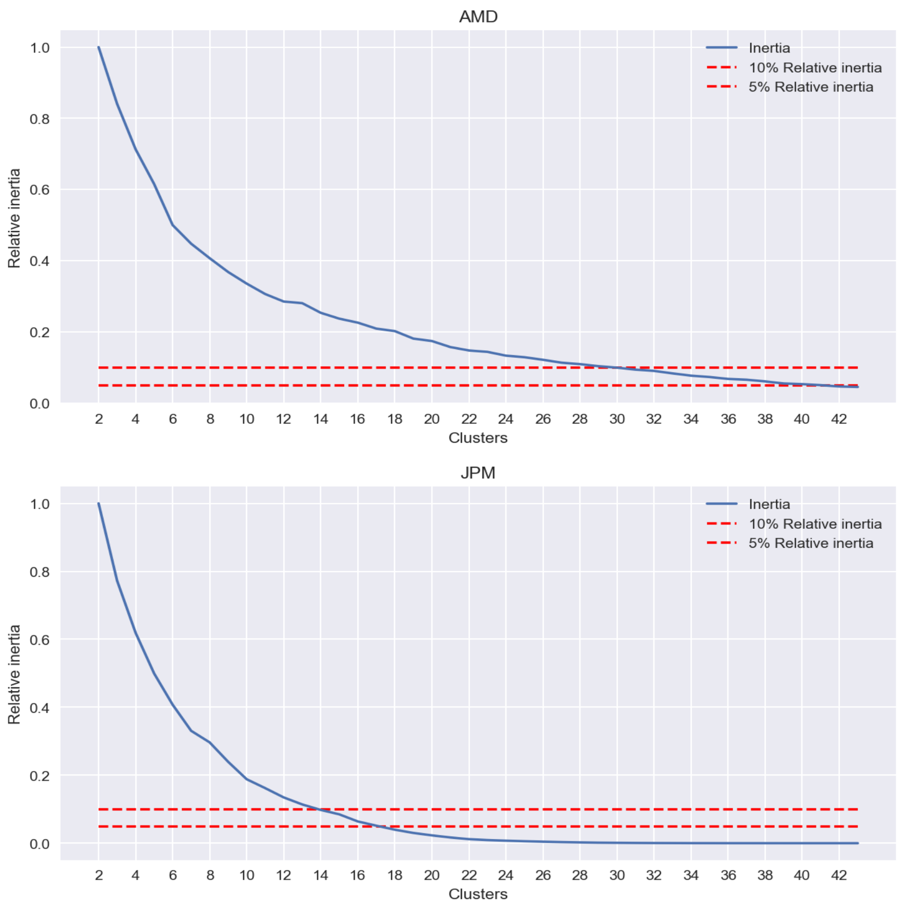


Figure 1: Elbow plots for the cluster inertia of AMD and JPM. The blue line represents the relative inertia while the two red dashed lines show the 5% and 10% inertia thresholds

After having optimised the number of clusters, we run the classifier and arrived at the results showed in the scatter plots of Figures 2 and 3

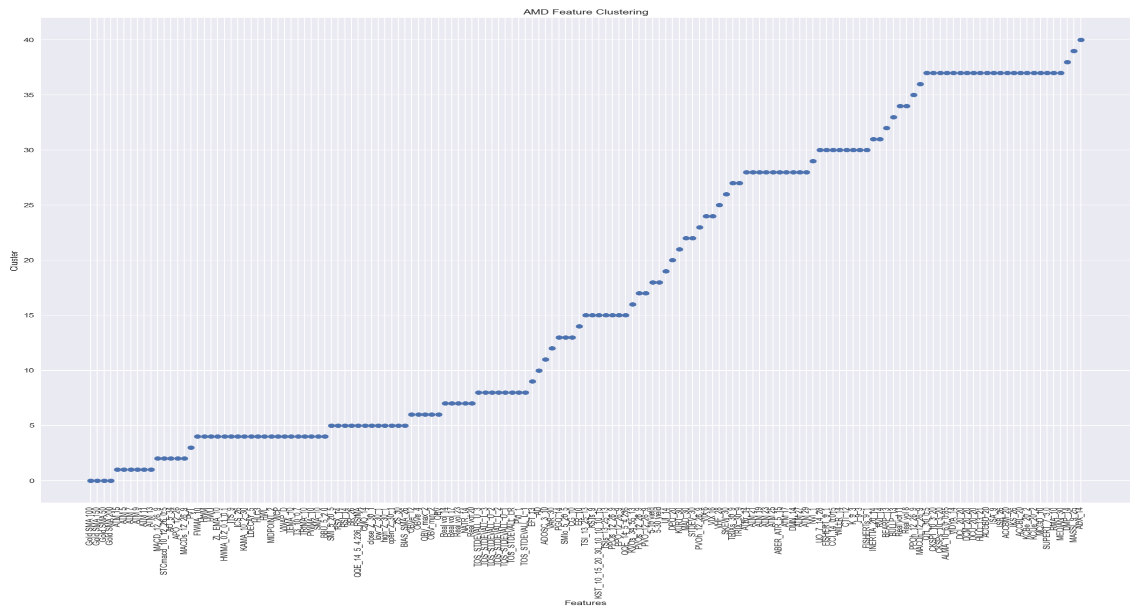


Figure 2: Scatter plot of AMD clusters. On the x axis the features selected by the BorutaPy, on the y axis the cluster number to which they belong.

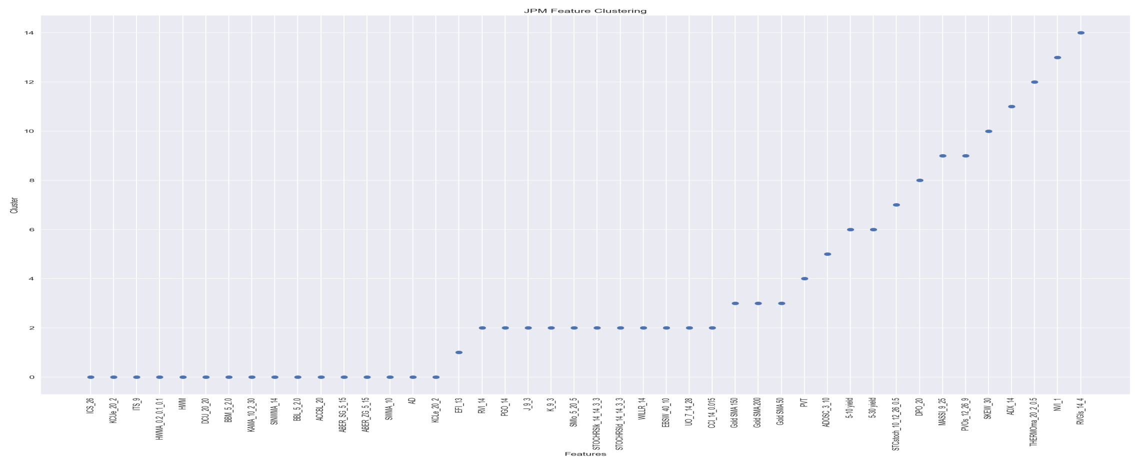


Figure 3: Scatter plot of JPM clusters. On the x axis the features selected by the BorutaPy, on the y axis the cluster number to which they belong.

The features are then subsumed into a dataframe together with their cluster number

Self Organising Maps

SOM is an unsupervised machine learning algorithm used to produce a low-dimensional (typically two-dimensional) representation of a higher dimensional data set while preserving the topological structure of the data. These clusters then could be visualized as a two-dimensional "map" such that observations in proximal clusters have more similar values than observations in distant clusters. Features in the same cluster can be represented by one member of the cluster itself.

Self-organizing maps, like most artificial neural networks, operate in two modes: training and mapping. First, training uses an input data set (the "input space") to generate a lower-dimensional representation of the input data (the "map space"). Second, mapping classifies additional input data using the generated map.

In most cases, the goal of training is to represent an input space with p dimensions as a map space with two dimensions. Specifically, an input space with p variables is said to have p dimensions. A map space consists of components called "nodes" or "neurons," which are arranged as a hexagonal or rectangular grid with two dimensions.[5] The number of nodes and their arrangement are specified beforehand based on the larger goals of the analysis and exploration of the data.

Each node in the map space is associated with a "weight" vector, which is the position of the node in the input space. While nodes in the map space stay fixed, training consists in moving weight vectors toward the input data (reducing a distance metric such as Euclidean distance) without spoiling the topology induced from the map space. After training, the map can be used to classify additional observations for the input space by finding the node with the closest weight vector (smallest distance metric) to the input space vector.

The SOM training utilizes competitive learning. When a training example is fed to the network, its Euclidean distance to all weight vectors is computed. The neuron whose weight vector is most similar to the input is called the best matching unit (BMU). The weights of the BMU and neurons close to it in the SOM grid are adjusted towards the input vector. The magnitude of the change decreases with time and with the grid-distance from the BMU. The update formula for a neuron v with weight vector $\mathbf{W_v}$(s) is

$$W_v(s+1) = W_v(s) + \theta(u, v, s) \cdot \alpha(s) \cdot (D(t) - W_v(s)), \tag{2.4}$$

where s is the step index, t an index into the training sample, u is the index of the BMU for the input vector $\mathbf{D}$(t), $a($ s) is a monotonically decreasing learning coefficient;

$\Theta(u, v, s)$ is the neighborhood function which gives the distance between the neuron u and the neuron v in step s.

The SOM produced for the two assets are showed in the Figure 4. The feature are then subsumed into a dataframe together with their coordinates.

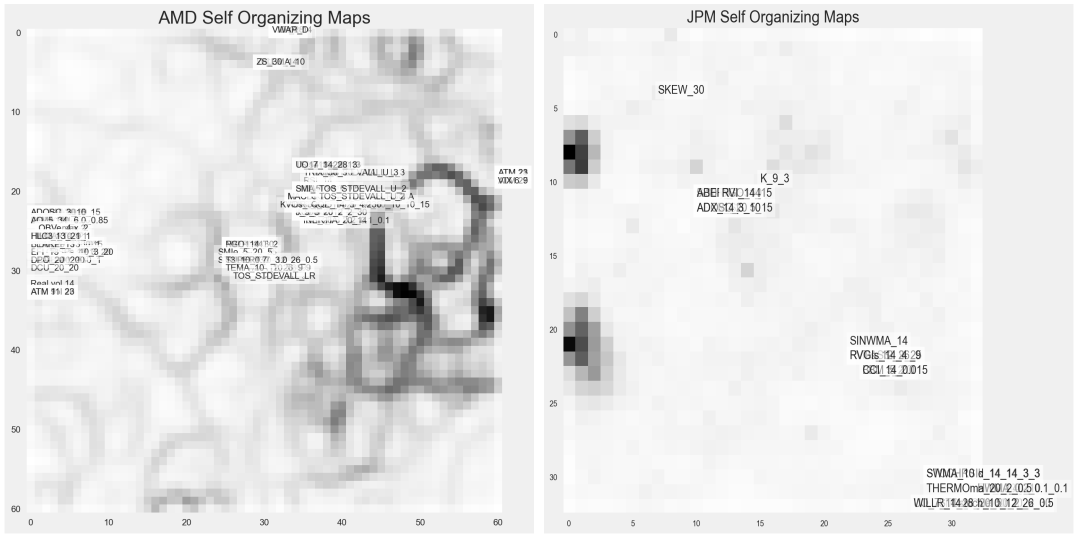


Figure 4: Self Organising maps for AMD and JPM

Selected features

After collecting the dataframes from the K-means and SOM classifier, we had to find a way of selecting which features would have been kept as input for the neural network.

The selection was done according to the following algorithm. We iteratively went through all the clusters outputted by the K-means and randomly selected a feature for each cluster. Given that features in the same cluster are supposed to be equivalent, and some clusters were quite dense, this allowed to already diminish the number of features significantly. Then, for every feature selected in this way, we looked its coordinate values assigned by the SOM algorithm and deleted form the SOM output dataframe all the features with the same coordinate values or at a coordinate value distance of ± 1.

This meant that the features remaining in the SOM output dataframe were considered different enough to those cluster representatives selected from the K-means algorithm. Therefore, we added them to the list composed of the cluster representatives to compose the final feature set.

Lastly, we dropped from the training, validation and test set all the features that did not pass the selection, arriving at the final datasets that were going to be used as inputs to the deep learning algorithm. We created a correlation heath map to display the remaining correlations between the selected features. The correlation heat map for AMD is displayed in Figure 5, while the heat map for JPM is showed in Figure 6

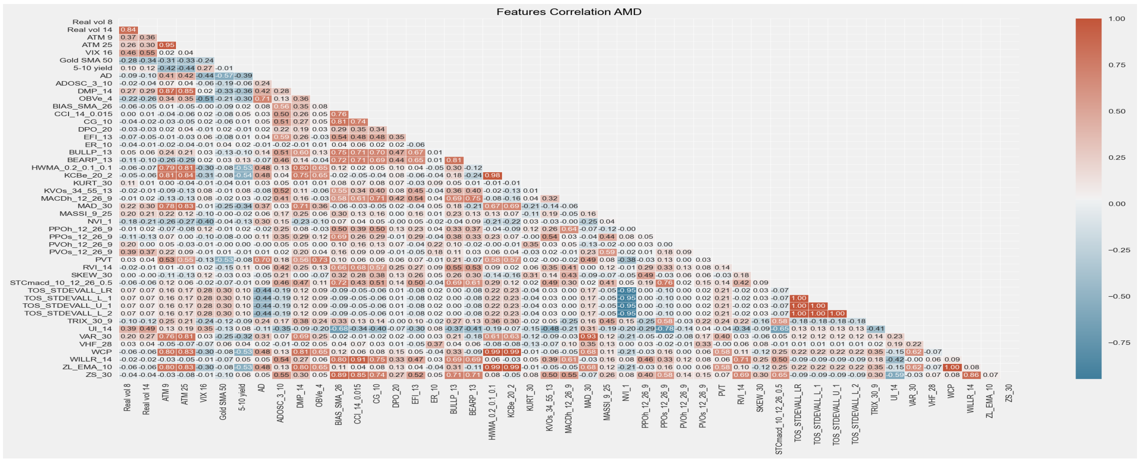


Figure 5: Correlation heath map for AMD. Bright colours indicate stronger correlation coefficients, according to the colour bar. It can be seen that even if there still are some pockets of correlated features, the selected features are not very correlated across the board.

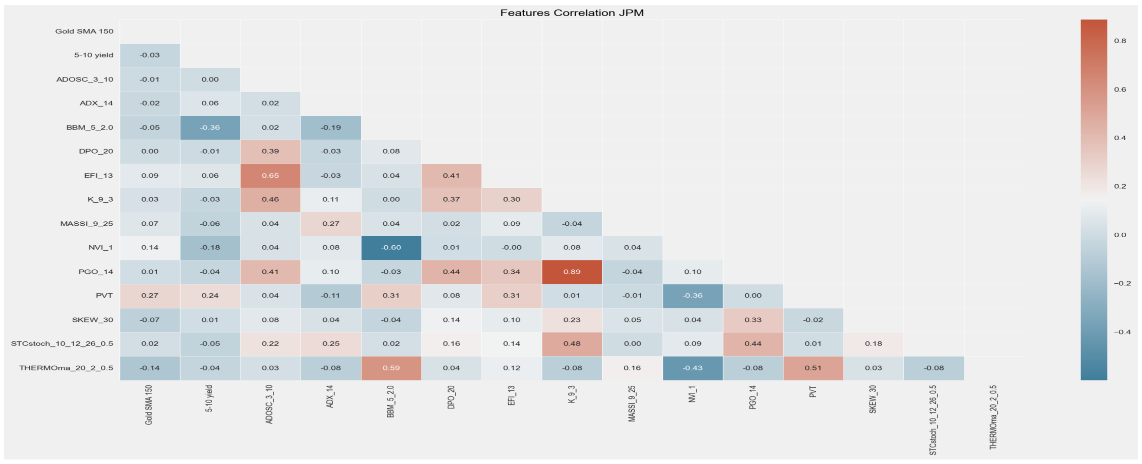


Figure 6: Correlation heath map for JPM. Bright colours indicate stronger correlation coefficients, according to the colour bar. It can be seen that even if there still are some pockets of correlated features, the selected features are not very correlated across the board.

Lastly, before moving to the construction of the neural network architecture, we exploited the Keras TimeseriesGenerator function to transform the time series into data samples with the correct structure for LSTM and GRU layers. This is necessary because the train, validation and test data sets are 2-dimensional, while the input accepted by LSTM and GRU neurons in 3-dimensional. Using TimeseriesGenerator, we can easily reshape the data correctly, and we can define the lookback period. This is the amount of timesteps that will be used by the deep learning network to predict the next label. We set this number to be 32 timesteps, which is a commonly used length and corresponds to slightly more than a month of observations. A good deep down guide on TimeseriesGenerator is [6]

3 Deep Learning architecture

In this section we will explain more in details what is a deep learning neural network and why it suits the goal of price direction forecasting. Moreover, we will describe the architecture chosen for this specific task and its hyperparameter tuning.

3.1 The structure of a Neural Network

The class of machine learning algorithms is quite diverse and can be split up in further subclasses based on the way the targets are defined. Supervised learning algorithms are based on learning functions that map input to output data bases on training samples composed by paired input-output data. This kind of machine learning infers a function from the training data and uses it to map new examples. On the other hand, unsupervised learning aims at the self construction of an internal model for the machine based on the observation and acquisition of patterns as probability densities. The input and the outputs are not labelled in pairs and the user relies of the power of mimicry and self-organisation of the machine. For this project, the choice was made to employ one of the most used form of unsupervised learning, namely neural networks.

Neural networks are an unsupervised learning approach based on the biological webs of neurons present in living creatures. A machine learning neural network is based on a collection of connected units or nodes called artificial neurons. Akin to its biological counterpart, these nodes have the ability to receive a signal from their input channel (or edge), process it with its internal structure, and output another signal through its output channel. The signal is nothing but a real number. After being received, input signals are linearly combined through a series of adjustable weights and passed through a non-linear function which acts as the internal structure of the neuron. The final number is then sent as part of the final output or as input to another neurons. As the neural network analyses the training samples, it learns by adjusting the internal weights of each neuron such that the best output is provided. Neurons are organised in layers which can contain an arbitrary number of them. Each layer may perform a different transformation on their inputs. The simplest networks are composed of just an input and an output layer, but there can be as many 'hidden layers' as one desires nested in between the two. If a neural network has one or more hidden layers it is referred to as a deep neural network and this is exactly the type of structure that has been used for this project.

For this project, we have exploited three kind of neuron layers. Dense layers, Long-Short Term Memory (LSTM) layers and Gated Recurrent Units (GRU) layers.

Of these three, the simplest are the dense layers. They are composed by neurons which take input from all the synapses in the precedent layer, they combine them linearly together in a weighted sum, and act on the result with a nonlinear function like a sigmoid or a hyperbolic tangent. Dense neurons play well the role of output layers due to their simple nature that renders them optimal controlling the way the output is presented, leaving more complex manipulations to the hidden layers.

The drawback of dense layers is the same as their strength. Their simplicity means that they are not well suited in for many different tasks, among which dealing with time series. Therefore, the use of more advanced neurons is required, and our choice led to LSTM and GRU layers.

3.2 LSTM & GRU

Neural networks are supposed to mimic the biological process of learning that takes place in living organism. Due to the extreme variety of problems that a living organism has to deal with, the learning process requires massive amounts of data. However, not all data is presented in the same way and some characteristics of the data might or might not be relevant depending on the objective at hand.

One big distinction can be made between time ordered and not time ordered data. When dealing with price forecasting inputs come from a time series and it is important for neuron layers to be able to handle and take into account the specific time ordering of the data. This differs drastically, for example, from image recognition where the training dataset of images and labels does not carry an intrinsic order. As one can imagine, time ordered data are not dealt optimally by all kinds of neurons. To obviate to this problem, Recurrent Neural Networks (RNN) have been developed.

RNNs are a class of neural networks where nodes are arranged along a temporal sequence according to directed or undirected graphs. Moreover, they posses an internal state that can be used as a memory, allowing them to process finite sequences of inputs. There are different kind of RNN neurons, but not all are well suited for handling long time series. This is the problem of Long-Term dependencies, meaning that basic RNNs struggle when the (tempooral) gap between relevant information and the place at which it is needed is long. In order to deal with this, some advanced forms of RNNs have been developed, among which we can find LSTMs and GRUs.

LSTMs are recurrent neural networks commonly employed when dealing with long time series in the context of deep learning. Their principal characteristic is that they have feedback connections. Therefore, LSTMs can process not only single data points, but also entire sequences of data. The LSTM internal structure is composed by cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. The presence of the forget gate allows the neuron to delete part of the information before outputting it. The drawbacks of LSTM layers is given but their complexity, which impacts negatively on the processing and computing time of the entire network. More can be found on LSTM layers here [7]

GRUs are like a LSTM with a forget gate, but have fewer parameters than LSTM, as they lack an output gate. They are faster at processing data than an LSTM, but they might perform worse on longer datasets. Information about GRU layers can be found here [8, 9]

In this project, we opted for the use of a combination of LSTM and GRu layers. We hoped that combining the strengths of these two types of layers would serve to mitigate their individual weaknesses.

3.3 Neural Network architecture

Given all the previous considerations, it is now time to discuss our choice of architecture for the task of 5 days price direction forecast. The network structure is pretty simple and presents 3 hidden layers only. In the specific, it is composed of four alternating LSTM and GRU layers, organised such that the input layer is an LSTM one. Lastly, a final output dense layer closes the network. The number of neurons on each layer, their dropout rate and their activation function is not specified at this point, except for the final layer, which is composed by a single neuron with a sigmoid activation function. The reason behind this choice is that we are constructing a neural network aimed at binary classification of up/down moves. Hence we expect the output layer to collect all the processed data and output a single real number between zero and one. The choice of the activation function is then given by the properties of the sigmoid function, which acts as a map $f : \mathbb{R} \to [0, 1]$.

As mentioned above, the number of neurons for each layer, their dropout rate and their activation function is left unspecified. The reason behind this choice is that we aimed at tuning these parameters with a Bayesian optimisation algorithm to obtain the best results. More on this in the following subsection.

Lastly, the specification of any machine learning algorithm requires a choice of loss function and optimizer. Given the nature of the task, the chosen loss function was the binary cross entropy. This is optimal for binary classification tasks as it compares each of the predicted probabilities to the actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. The Binary cross-Entropy is defined as

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p(y_i)) + (1 - y_i)\log(1 - p(y_i)) \tag{3.1}$$

where y is the label (1 for up moves and 0 for down/stationary moves) and p(y) is the predicted probability of the move being up for all N steps.

The optimizer chosen was Adam due to its usual good performance. Adam combines the advantages of two other extensions of stochastic gradient descent, namely AdaGrad and RMSprop. The former stands for Adaptive Gradient Algorithm, it maintains a per-parameter learning rate that improves performance on problems with sparse gradients. The latter stands for Root Mean Square Propagation, which also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight, improving the performance on on-stationary problems.

Instead of adapting the parameter learning rates based on the average first moment as in RMSProp, Adam employs the average of the second moments of the gradients. More specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and two parameters named β_1 and β_2 control the decay rates of these moving averages. To read more in-depth about Adam, check [10, 11].

We also decided to take BinaryAccuracy, Precision, Recall and AUC as metrics for the neural network performance.

Moreover, we defined callback functions for the tuning and training processes. The tuning callback was comprised of an EarlyStopping function, while the training callback also included a ModelCheckpoint. The role of the early stopping is to interrupt the training of the network if a metric of choice does not improve over a given amount of epochs. For the tuning process, we set this function to be the valuation loss and the number of epochs associated to 10, meaning that the training would have stopped if the loss function on the valuation set had not improved after 10 epochs. This should prevent overfitting of the model. On the other hand, the training callback was instead defined so that the monitored function was the valuation accuracy, and the epoch number was set to 15. The reasoning behind this choice was based on the idea that, given that the tuning process should have already decided good hyperparameters that keep the valuation loss in check, we wanted the training process to push for a better valuation accuracy. Valuation accuracy was also the monitored metric imposed on the model checkpoint, meaning that the best weights associated with the best performing epoch would be saved.

3.4 Hyperparameter tuning

When constructing any machine learning algorithm, there will always be some parameters that are manually passed as an input from the developer. These are called hyperparameters and they are not optimised by the machine learning algorithm itself. Nevertheless, they can be optimised with other methods. This process is called hyperparameter tuning and it is extremely important for the optimization of the machine learning performance.

For the Neural Network constructed in this project, the hyperparameters that were hypertuned were the number of neurons in the input and hidden layers together with their dropout rates. By dropout rates we refer to the option that each LSTM and GRU layers have of dropping a percentage of the input data in order to optimize the layer learning process by forcing it to work with less information. This helps to prevent overfitting. We also added a dropout layer with tunable dropout rate before the dense output layer, this drops a percentage of the inputted information going to the last layer, and acts as if the dense layer had a 'dropout rate' option within its parameters.

Lastly, we optimise the choice of activation function for the input and hidden layers. We allow the algorithm to choose between 'elu' and 'relu' activation. The difference between the two is that the 'relu' function takes the value of zero for any negative input, while the 'elu' function takes small negative values. The advantage of the latter is that it prevents the problem of zero gradient, which is a problem that can be encountered by the optimizer algorithm if the activation function has points with zero gradient. Since 'elu' activation functions are not necessarily the best choice in every scenario, we allowed it to be a tunable parameter, but we force the same activation on all the layers. This is motivated by the limited capabilities of the computing hardware used to compile the program, which forced us to limit the number of hyperparameters as much as possible in order to be able to compile the network in a finite amount of time. Having more computing power would have allowed us to hypertune more parameters, like having a different activation function for each layer or a tunable number of layers.

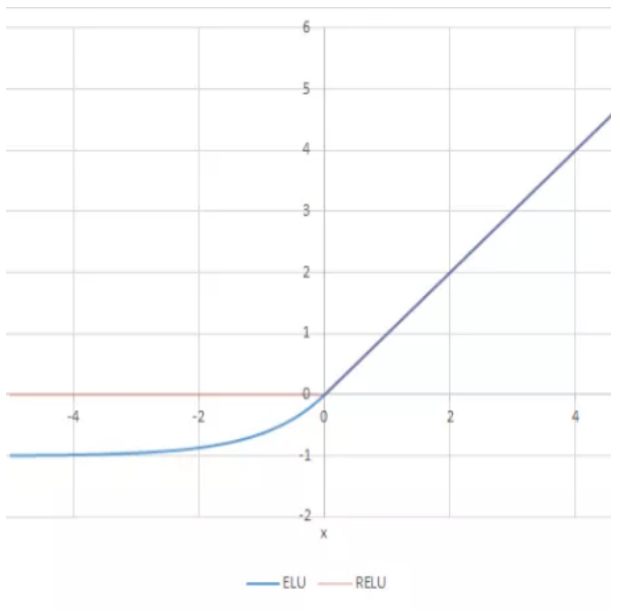


Figure 7: Comparison of elu and relu activation functions. Their behaviour is the same for positive numbers but the relu function is zero on all the negative ones

When performing hyperparameter optimisation, the choice of optimiser is vital. There are many optimisers available with the Keras tuner package, each of them with their strengths and drawbacks. For this project, we decided to employ an algorithm based on Bayesian optimisation. The strength of this approach relies on the fact that Bayesian optimisation is optimal for global optimisation of so called 'black-box' functions and does not assume any functional form. The way in which this is achieved is by treating the unknown objective function as a random function and placing a 'prior' over it. The role of the prior is to capture beliefs about the behaviour of the function and subsequently be updated to form the posterior distribution over the objective function. This latter distribution is in turn used to construct an 'acquisition function' that determines the next query point. The drawback of this optimisation approach is that is computationally intense and hence may be lengthy to compute in lower performing machines.

We present the architecture resulting from one of the runs of the hyperparameter tuning for AMD in Figure 8 and for JPM in Figure 9:

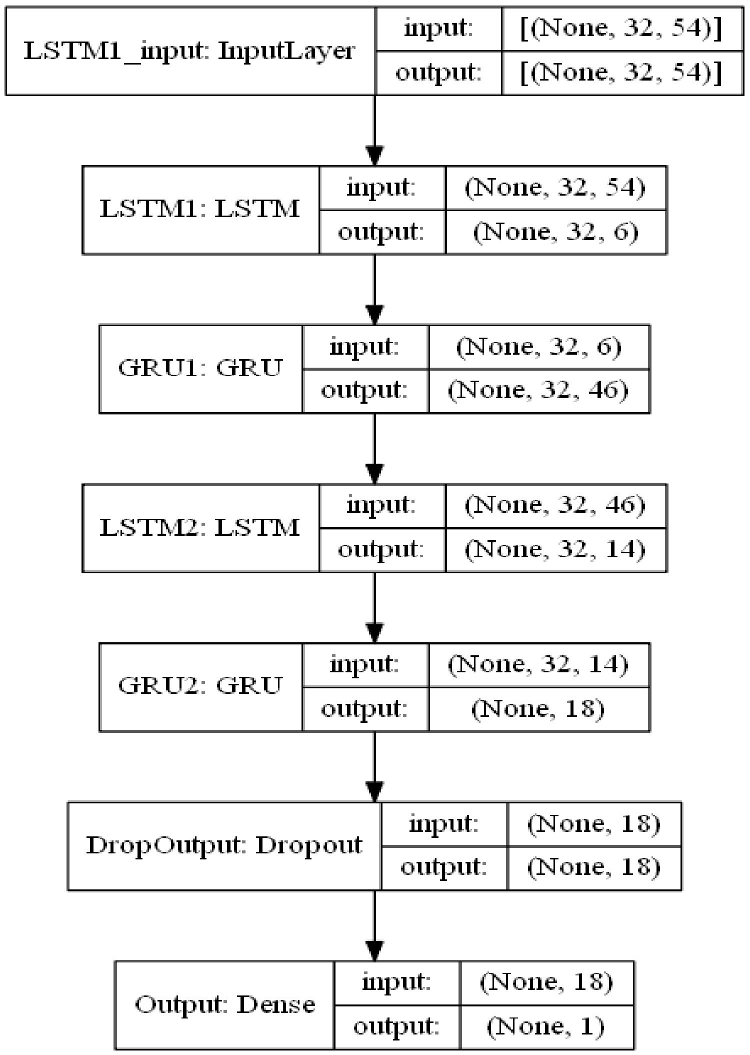


Figure 8: Deep learning architecture for AMD after hyperparameter tuning.

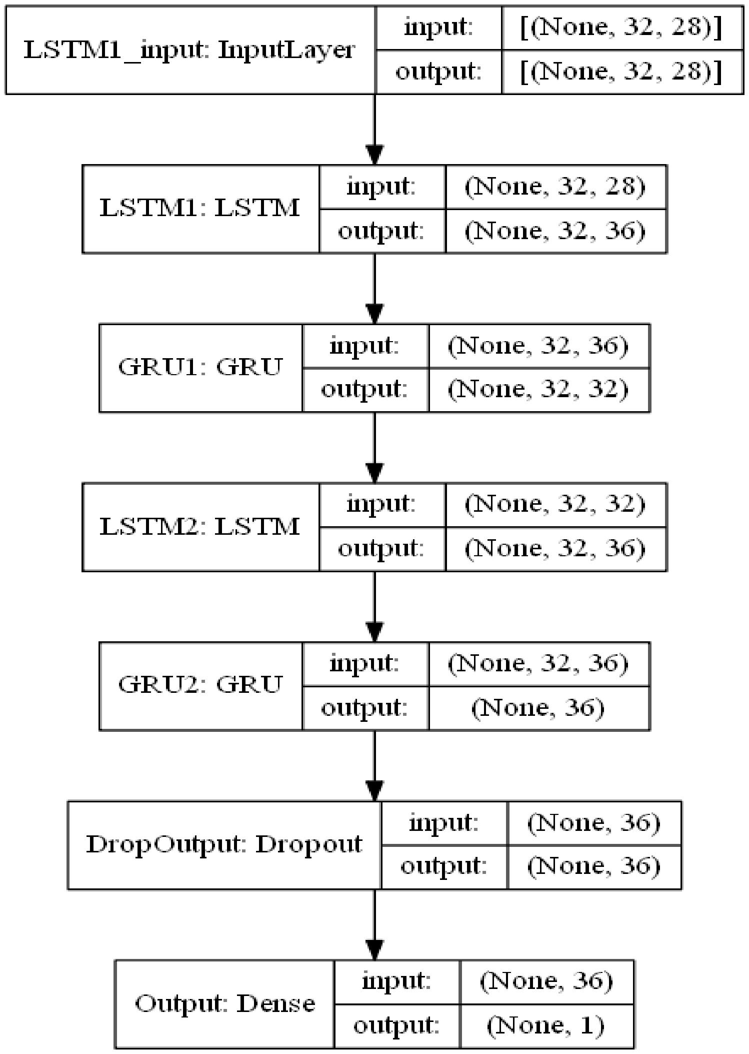


Figure 9: Deep learning architecture for JPM after hyperparameter tuning.

Once the optimal hyperparameters were found, we proceeded by loading them into a Neural Network and training it with the training callback described earlier.

4 Analysis of results

In this section we will describe and analyse the performance of the tuned and trained neural network on the test datasets. We will explain which metrics are used when evaluating a binary classification algorithm and why, and comment on how the network has performed overall and in each evaluation metric.

4.1 Analysing classification algorithms

When evaluating the performance of a binary classification algorithm there is a variety of metrics that can be used. These metrics are fundamental when developing any machine learning model as they allow to infer the performance of the algorithm. In our case, the deep learning network has been allowed to adjust the layers internal weights using train and validation datasets, but we now need to accurately gauge how it will perform on unseen data. Therefore, we will ask the neural network to predict the labels on the test dataset.

4.2 Accuracy and classification reports

The first metrics that we looked at were the train and test accuracy. The accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity's true value, hence evaluating the train and test accuracy gives us a metric for the model overfitting. If the train accuracy is much greater than the test accuracy, it means that the model has become very good at learning the patterns of the train set, without really learning how to make reliable predictions. The Train and test accuracy for this run of the model are reported in Table 1

	Train Accuracy	**Test Accuracy**
AMD	0.7265	0.6891
JPM	0.7423	0.6728

Table 1: Train and test accuracy for AMD and JPM

Immediately after the train and test accuracy, we exploited the sklearn metric function to obtain a classification report of the model results. This function prints the values of the main classification metrics, which are precision, recall, f1-score and support. These four indicators are based on the classification of predicted labels into four categories, namely true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). Precision is calculated as the ratio of true positives with the total number of positive predicted labels

$$P = \frac{TP}{TP + FP}. \tag{4.1}$$

Precision is an indicator of what percentage of the labels that were classified in one way were actually correct. Recall, on the other hand, is defined as the ration between true positives and the sum of true positives and false negatives.

$$R = \frac{TP}{TP + FN}. \tag{4.2}$$

Recall evaluates the true positive rate, which is a measure of the percentage of the labels missed by the classification.

Precision and recall are meaningless by themselves. A classification model could achieve a perfect score in one of them without really being a good model, for example a recall of 1 could be achieved by classifying all the elements as positives, but the model would be useless. It is when they are considered in combination that they become a powerful evaluating tool. F-scores are an harmonic mean of precision and recall. They are a parametrised family of metrics defined as:

$$F_\beta = (1 + \beta)^2 \frac{P \cdot R}{\beta^2 P + R}. \tag{4.3}$$

It is easy to notice how for $\beta > 1$, precision is weighted more than recall, while the opposite happens for $\beta < 1$. Therefore, the F_1 score is a metric that gives the same weight to both precision and recall, and this is the measure that we used.

$$F_1 = 2\frac{P \cdot R}{P + R}. \tag{4.4}$$

The highest value for the F-measures is 1, indicating perfect precision and recall, while the minimum is 0.

The support is simply the number of samples of the true response that lie in that class. The full evaluation for AMD is presented in Table 2, while that for JPM is illustrated in Table 3

AMD	Precision	Recall	F_1-score	Support
0	0.63	0.65	0.64	901
1	0.72	0.70	0.71	1149
Accuracy			0.68	2050
Macro avg	0.67	0.67	0.67	2050
Weighted avg	0.68	0.68	0.68	2050

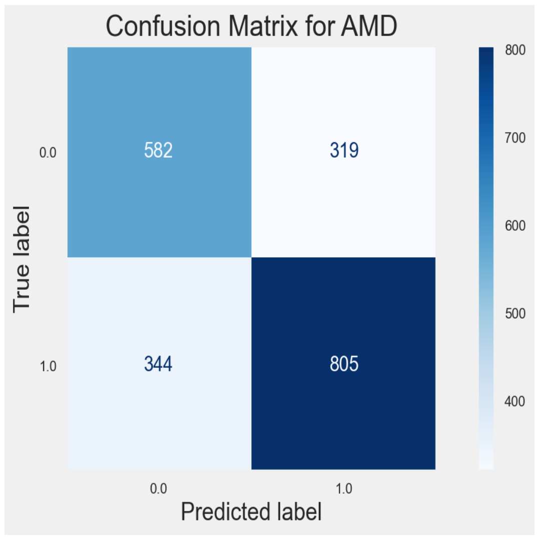
Table 2: Evaluation table for AMD predicted labels

JPM	Precision	Recall	F_1-score	Support
0	0.61	0.71	0.66	898
1	0.74	0.64	0.69	1158
Accuracy			0.67	2056
Macro avg	0.68	0.68	0.67	2056
Weighted avg	0.68	0.67	0.68	2056

Table 3: Evaluation table for JPM predicted labels

4.3 Confusion matrix

The third metric that we used to analyse the model predictions was a confusion matrix, also known as an error matrix. The idea of a confusion matrix is to be a visual representation of the amount of true/false positives and true/false negatives. Even if it is not the most mathematically meaningful evaluation metric, it allows for a quick visual overlook of the outcome of the predicted labels. The confusion matrix obtained for AMD and JPM are represented in Figure 10.



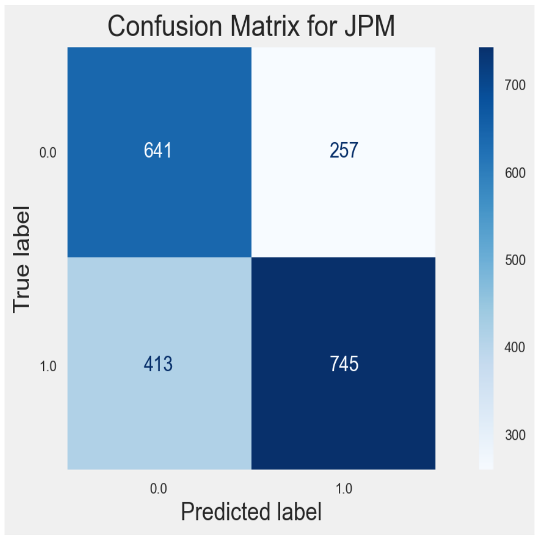


Figure 10: Confusion matrix for the AMD and JPM predicted labels. The four squares correspond to (form the top left going clockwise) true negatives, false positives, true positives and false negatives. The intensity of the blue color is correlated with the number of predicted points belonging to each category, which is written on each square.

4.4 ROC curve

As fourth metric, we evaluated the the Receiving Operating Characteristic (ROC) curve. This curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It is plotted as the true positive rate against the false positive rate. The ROC curve is therefore the sensitivity or recall as a function of fall-out. The graph is split in half by the line representing an equal rate of true positive and false positives, a good model is one whose ROC lies above the equal rates line, while a model with a ROC lower than the equal rates line can be considered worse at binary classification than the flipping of a coin. The amount by which the ROC curve is better than the equal rate curve is given by the area under the curve (AUC). The closest the AUC value is to 1, the better the model is at predicting labels.

Figure 11 shows the AMD ROC curve for this specific run of the model, while the ROC curve for JPM is reported in 12.

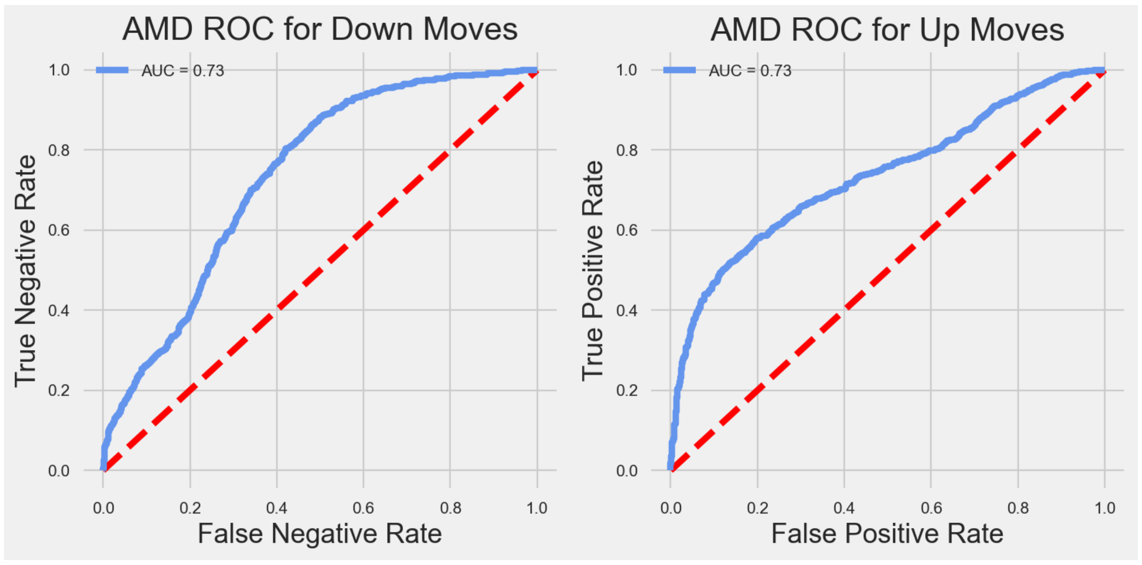


Figure 11: Receiving Operative Characteristic curve for AMD. The red dashed line represents the equal rate line. The blue line is the performance of the model. The area under the curve (AUC) is 0.73, implying that the model is better at classifying up/down moves than the random flip of coin.

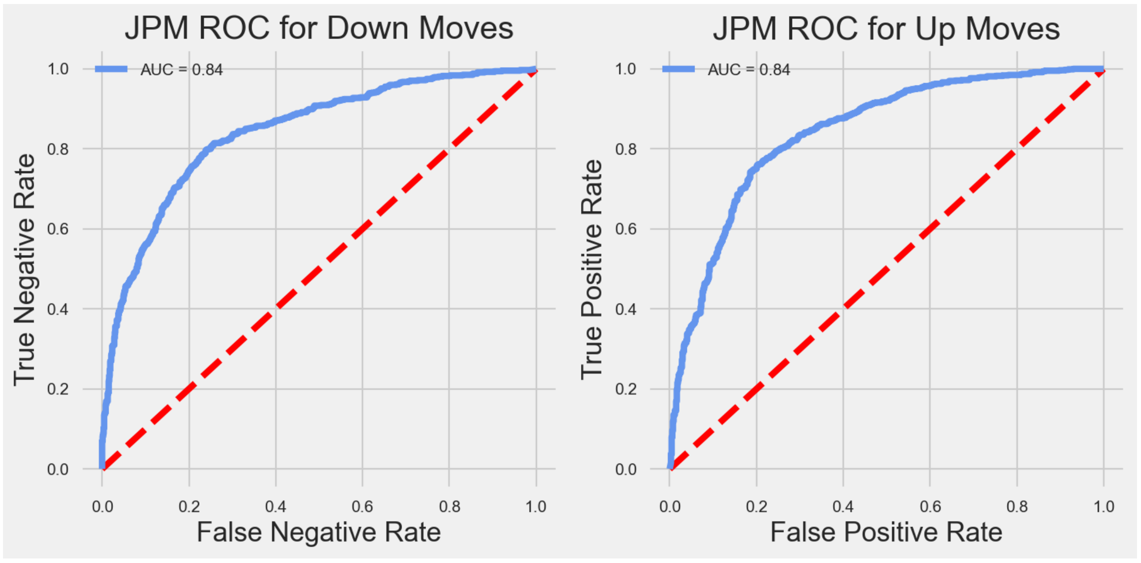


Figure 12: Receiving Operative Characteristic curve for JPM. The red dashed line represents the equal rate line. The blue line is the performance of the model. The area under the curve (AUC) is 0.84, implying that the model is better at classifying up/down moves than the random flip of coin.

4.5 Probabilities scatter plots

The last metric used to gauge the performance of the model are a couple of simple probabilities scatter plot. Plot represents the up/down moves according to their probability, two different colours differentiating the true and false values. Probability scatter plots are useful to infer the certainty with which the classification has been made. If we consider up move as an example, we will have that a scatter plot with true values close to 1 and false values close to 0 belongs to a model with a high confidence in its outputs. This is preferred to a model in which the output probabilities are around 0.5, as they mean that the prediction is not far from a random guess.

The probability scatter plots of this specific run are reported in Figure 13 for AMD and in Figure 14

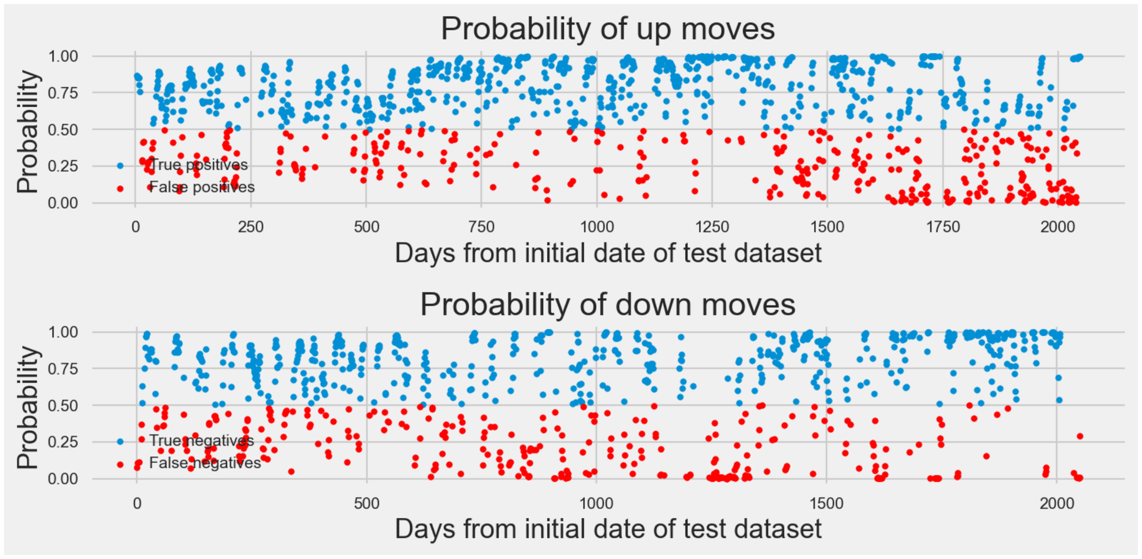


Figure 13: Probability scatter plots for AMD. On the x axis are represented the time steps of the test set index. On the y axis the probability associated with each predicted label. Blue dots indicate true positives and true negatives, while red dots correspond to false positives and false negatives. The plots are quite scattered, indicating that the predictions were not extremely certain, but they were quite better than a random guess.

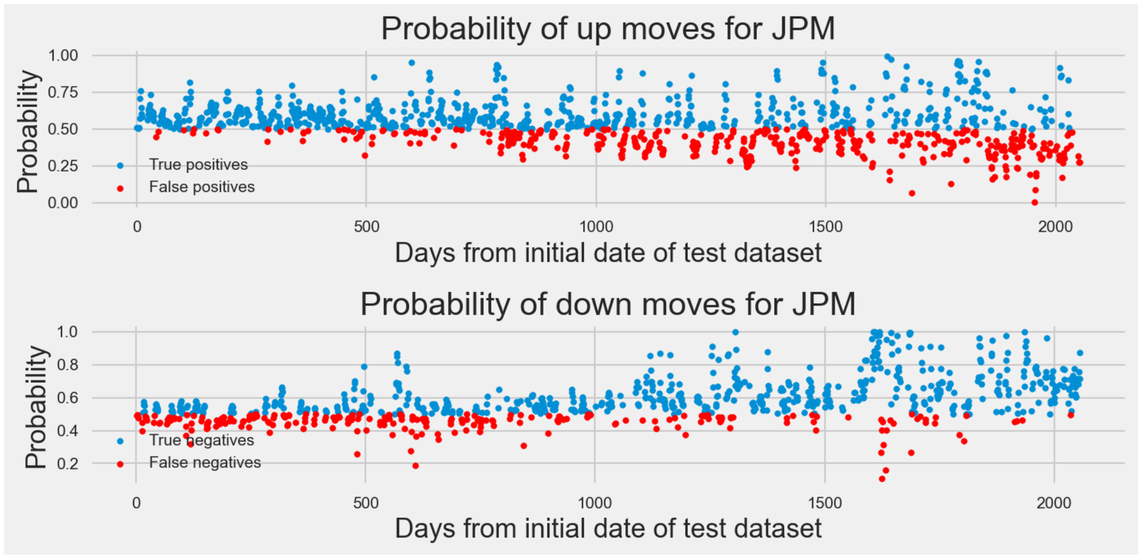


Figure 14: Probability scatter plots for JPM. On the x axis are represented the time steps of the test set index. On the y axis the probability associated with each predicted label. Blue dots indicate true positives and true negatives, while red dots correspond to false positives and false negatives. The plots are quite concentrated around the 0.5 line, meaning that the model was not very certain to which class the predicted labels should have been assigned. Oddly, it seems that the down moves were predicted more accurately toward the end of the time interval.

5 Conclusion

5.1 Summary of the results

According to the evaluation metrics, the model performance was acceptable. Overfitting was avoided as can be inferred from noticing that the train accuracy was better than the test accuracy. Precision and recall were good for both assets, with every value above 0.6, and the F_1 score was around 0.7 on both assets too.

Both confusion matrices showed a greater value of true values with respect to false values, as it can be deduced by the precision and recall scores.

The ROC are satisfactory for both assets, showing a solid separation from the equal rates line.

The probability scatter plots differ substantially between the two assets. AMD scatter plots are more spread with respect to the JPM scatter plots, which are more concentrated along the middle line corresponding to a probability of 0.5. As previously explained, this means that the JPM predicted labels were predicted with less certainty than the AMD ones, even if the result was overall quite good.

5.2 Shortcomings and counterpoints

Price direction forecasting is complicated and riddled with nuances that are hard to capture without spending a long time developing and testing the model. This project was based on a highly simplified model of price prediction, where only up and down moves where concerned. Feature extraction could have been more throughout, for example considering credit spreads and fundamental indicators. The global market indicators were limited in numbers and the gold/silver priced were not represented by ETFs but by an index which is subject to influences outside those related to the price of the precious metals. Unfortunately, time constraints did not allow for the deep analysis required to correctly retrieve, analyse and embed fundamental indicators into the feature set and the XAU ticker used for precious metals had the advantage of having publicly accessible data going further enough to the past to match the other tickers.

The feature selection process was quite effective, reducing the number of features by more than half in both assets and drastically cutting down multicollinearity between the ones selected. Yet, some correlations still remain and further feature selection methods could ave been implemented to get rid of them too.

With regards to the neural network, hyperparameter tuning has proven effective but more hyperparameters, for example a tunable number of layers, could have been defined and implemented. The constraints in this case were given by the limited computing power on which the model was developed and tested, especially given the computing and the time required for Bayesian optimisation. Therefore, we preferred to limit the number of hyperparameters to the point were a good performance was still achieve with at a manageable time cost.

5.3 Future improvements

When considering how the model could be further improved, one could start from implementing the points raised above. Better feature extraction, stricter feature selection and wider hyperparameter tuning could definitely improve the model performance. After having implemented those changes, one could ask if the model would be good enough to be deployed in the market. The answer would still be negative. The reason behind it are many, but one stands out the most: the model is not realistic enough. Therefore, a deeper and more precise formulation of the problem should be thought and implemented. Only then a strategy could be built around the model and eventually deployed.

Nevertheless, the model does what the model has been told to do, and in this simplified formulation of price direction forecasting its performance is satisfactory.

Acknowledgements

We would like to thank the Fitch Learning Institute and the all the teaching staff of the CQF course.

Special thanks to Mr. Kannan Singaravelu for promptly answering any question regarding the project.

Appendix

A Advanced Volatility Modeling

The advanced elective on volatility modelling shed light on the real nature of volatility. Volatility is not a constant nor a smooth time dependent variable as it is assumed in the Blach-Scholes model. Instead, it is more appropriate to treat it as a stochastic variable. There are many different methods of modelling stochastic volatility. The simplest one assume it follows a geometric Brownian motion. The volatility ν_t and its underlying asset S_t would obey the coupled SDEs

$$dS_t = \mu S_t dt + \sqrt{\nu_t} S_t dW_t \tag{A.1}$$

$$d\nu_t = \alpha_{\nu,t} dt + \beta_{\nu,t} dB_t \tag{A.2}$$

where $\alpha_{\nu,t}$ and $\beta_{\nu,t}$, are some functions of ν, and dB_t, is another standard Gaussian that is correlated with dW_t, with constant correlation factor ρ.

These model were quickly replaced by more advanced approaches like the Heston model which describes a mean reverting volatility

$$d\nu_t = \theta(\omega - \nu_t)dt + \xi\sqrt{\nu_t} dB_t \tag{A.3}$$

where ω is the mean long-term variance, θ is the rate at which the variance reverts toward its long-term mean, ξ is the volatility of the variance process.

Unfortunately, there was not much room in this project to apply the variety of numerical techniques taught in the advanced elective. Instead, we decided to use the volatility calculated and retrieved to check if volatility does indeed exhibit mean reverting and clustering behaviours as suggested in [12, 13] model is indeed more sensible than simple geometric Brownian motion.

We present the two days realised volatility of the returns for both AMD and JPM in Figure 15

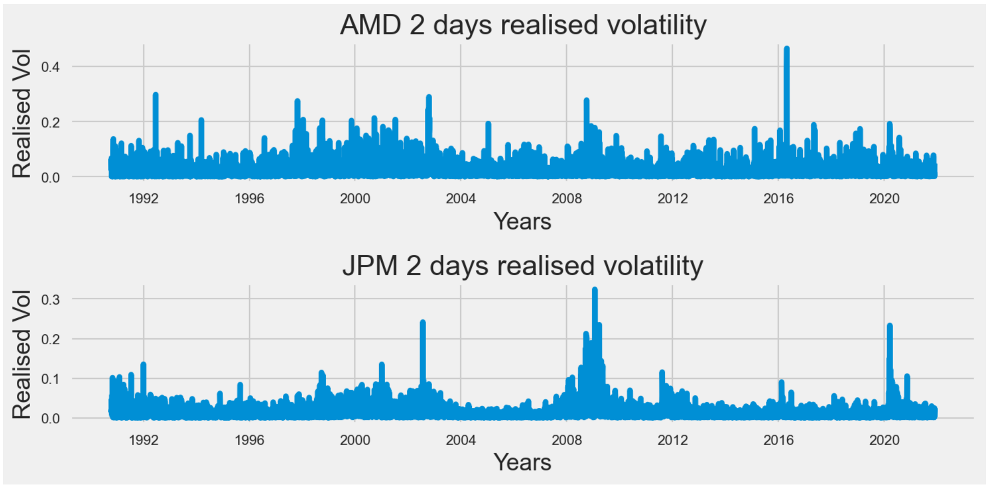


Figure 15: 2 days realised volatility for AMD and JPM. The mean reverting behaviour can be clearly observed, together with the effect of volatility clustering.

We can also explore if this behaviour appears in the global market. Therefore, we plot the VIX index, which is shown in Figure 16

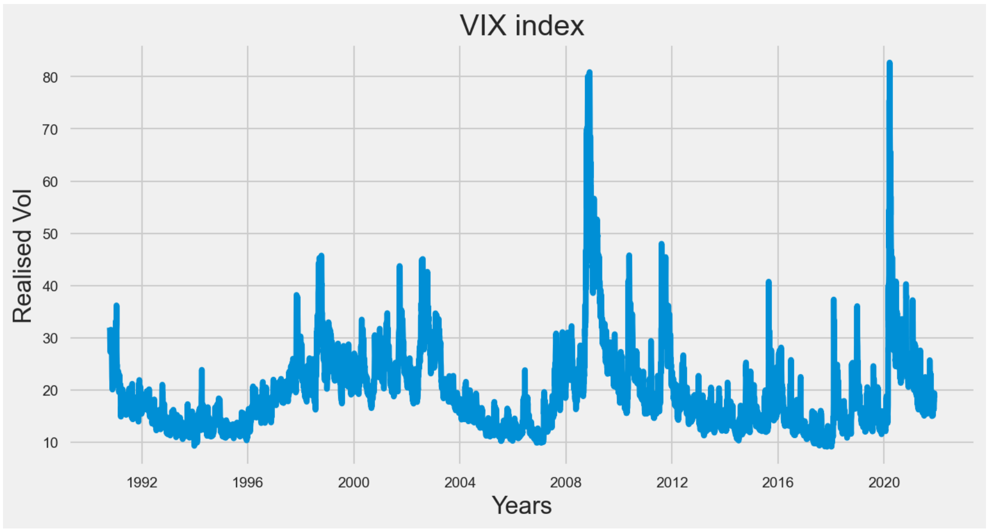


Figure 16: VIX volatility index. The mean reverting and volatility clustering behaviours can be clearly observed here too.

In both figures we can clearly see that volatility does follow a stochastic dynamics, but not the one that would be given by geometric Brownian motion. Volatility spikes around years of financial crisis, and the dot com bubble can be clearly seen together with the 2008 financial crisis and the recent COVID crash.

All these data suggest that volatility clearly cannot be modeled with a constant or smooth function, but complex stochastic models need to be developed.

Given the importance played by volatility is every aspect of the financial world, we are sure that volatility modeling is a field that still has a lot of room to grow.

References

[1] *Timeline of machine learning*, *En.wikipedia.org* .

[2] GitHub, *twopirllc/pandas-ta*, .

[3] *www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights*, .

[4] *towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35*, .

[5] GitHub, *scikit-learn-contrib/boruta-py*, .

[6] *medium.com/swlh/timeseriesgenerator-a-deep-down-with-example-in-python-dfe32dcb2a24*, .

[7] *colah.github.io/posts/2015-08-understanding-lstms*, *Colah.github.io* .

[8] *towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21*, .

[9] $Gated_recurrent_unit$, En.wikipedia.org.

[10] machinelearningmastery.com, *adam-optimization-algorithm-for-deep-learning*, .

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv.org* (2014) .

[12] M. Calabressi, *Alternating direction implicit methods for solutions of the heston stochastic volatility model*, .

[13] D. R. Ahmad, *Stochastic volatility and jump diffusion*, .