Comparative Study on different Machine Learning techniques for stock price forecasting

Introduction:

ML has become a very important technology in time series forecasting over various fields where researchers apply it to real world challenges. These fields include Healthcare [1], Energy [2], Agriculture [3], Climate & Weather [4] and financial data. Especially in financial data people have a huge interest in making accurate forecasts about developments of for example companies and stock prices given the huge potential for profit in trading.

In this paper we want to evaluate different Machine Learning models against each other based on their performance of forecasting different stock prices and indices. We therefore observe the current literature in this field:

Literature review:

The paper [5] from Kumbure et al. (2022) showed in a comprehensive review the technological development in the field of stock forecasting. The review includes 138 articles that have been published between 2000 and 2019. It looks on machine learning techniques that have been applied, features that were considered for training and metrices with which results have been measured. Like that they propose a differentiation of features into 4 categories which are “Technical Indicators”, “Macro-Economy”, “Fundamental Indicators” and “others”. They explain the importance and use frequency of these features. In section 5.4.8 that paper [5] illustrates what families of machine learning models have been applied to the task of forecasting stock prices. It shows that at least since the year 2000 Artificial Neural Networks (ANNs) in various forms were experimented with excluding Deep Learning models that are listed separately. That differentiation shows that deep learning models have especially been investigated heavily in the more recent years with the first reviewed paper applying them in 2017.

One representative of Deep Learning is mentioned very regularly in research – Long Short-Term Memory (LSTM). The paper [6] by Matharasi et al. (2025) evaluates two models against each other which are LSTM and XGBoost. Both delivered promising accuracy in forecasting AMD inc. stock data. While the LSTM model delivered slightly better accuracy in forecasts, the XGBoost model could excel in computational efficiency. Both models have been shown to have outstanding predictive performance when forecasting certain stock or index data. Teixeira, D. M., & Barbosa, R. S. (2025) have conducted another experiment [13] that shows their effectiveness in capturing the complexity of stock prices. They evaluate the models LSTM and XGBoost as well as other architectures such as GRU, CNN, RNN, but also different hybrid combinations of these like LSTM + CNN, RNN + GRU, GRU + CNN, LSTM + GRU, and RNN + LSTM. The study trains all of these models on historical stock data from Apple inc. and comes to the result that the standalone models of GRU, LSTM and XGBoost perform the best for the Apple inc. stock data. This is a little counterintuitive since one might think that hybrid models usually outperform their standalone component models since they might merge strengths of both models but as evidenced that is not generally the case. Under section 5 they discuss that the performance depends furthermore on the timeframe of the training data. They state that the gap of the performance between the models is smaller for a shorter time frame of training data. They state that the reason for that is the lower complexity of that time frame with fewer large amplitude oscillations. When choosing a longer time frame for training with a more complex data course, the three earlier mentioned models GRU, LSTM and XGBoost outperform the other models. Another paper [7] by Li, Siyuan. (2024) investigates the performance of a Kalman Filter and different types of LSTM models such as a single layer LSTM, a stacked LSTM, a bidirectional LSTM and a hybrid convolutional neural network (CNN)-LSTM. From those the bidirectional LSTM performed best for low-volatility stocks and the CNN-LSTM model performed best for high volatility stocks. Here the hybrid CNN-LSTM model in fact outperformed the standalone LSTM model. XGBoost models as mentioned in the two papers above seem to have a reasonable performance in forecasting stock prices under certain circumstances. This is evidenced by another experiment [10] by Aiyegbeni Gifty, & Dr. Yang Li. (2024) that was conducted to compare the performance of LSTM, ARIMA and XGBoost models. They also come to the conclusion that the XGBoost model outstands in performance under certain circumstances. They trained and evaluated the models on Google inc. stock data. This again emphasizes the impact that the data has on the performance of each model and it shows as the other studies mention as well, that certain models have advantages in dealing with certain characteristics in data like volatility making the choice of a suitable model very much not trivial. They also emphasize the importance of hyperparameter tuning. Like that they could reduce the Mean Absolute Error (MAE) from 17.63 without hyperparameter tuning down to 15.98 and the Root Mean Squared Error (RMSE) from 30.24 down to 27.34.

While the mentioned approaches of deep learning and ensemble methods have been looked into particularly heavy in the last years as shown in the previously mentioned papers and under section 5.4.8 of paper [5], the new transformer-based architecture has been proven effective in forecasting stock prices. The paper [14] by Yao, Y. (2025) shows its superiority over the previous mentioned models such as XGBoost, LSTM and CNN under certain circumstances.

Conclusion of Literature review:

In general, the advantages and superiorities of different ML algorithms depend strongly on circumstances such as dynamics in the data, feature selection, architecture tuning and hyperparameter tuning. This said, in this study we want to further diversify the results in this area in a way where we can evaluate predictive performance in relation to the characteristics of the data. Out of that we define our research question in the following paragraph.

Research question:

As mentioned in the previous section, to make useful statements about predictive performance of ML algorithms, we need to look on the relation of the performance of the predictions to the data which is used. We observed the dependency of predictive performance on data characteristics during our literature review where we recognized that different ML models can be claimed to be best performing depending on the parameters, input data and setting of the study. For our experiment we want to look on the performance of ML algorithms in forecasting stock prices based on the volatility of the stock data. We want to understand how volatility in the course impacts the forecasting performance of different ML algorithms and we also want to understand which model is most suitable for a certain degree of volatility in the data. To be more precise we are looking for results about the difference in performance of forecasts when using one ML algorithm and stocks of different volatility as input and we want to draw conclusions about the difference in predictive performance when using stock data of a certain volatility as input to train different ML models to see which is best suitable for a certain degree of volatility.

Machine Learning: (This section will talk about the bullet points as general topics but it will focus only on what’s necessary to understand later parts of the thesis, how important the parts are for the different models will be concretized in the experimental setup under methodology)

Artificial Intelligence refers to systems that are designed to simulate aspects of human intelligence such as decision, reasoning, perception and adaptation. The important part here is that these systems are capable of learning based on outcomes. With learning we mean that the system can update itself to make better predictions in response to the difference of the predicted outcome and the actual outcome. They don’t have to rely on silicon-based hardware but often do when referred to AI. Machine Learning is a subset of Artificial Intelligence. As IBM puts it [48], “Machine learning is a branch of [artificial intelligence](https://www.ibm.com/think/topics/artificial-intelligence) focused on enabling computers and machines to imitate the way that humans learn, to perform tasks autonomously, and to improve their performance and accuracy through experience and exposure to more data.“. To understand that further we look at an example that clarifies where the difference between any algorithm and ML algorithms comes from. Let’s say we have two algorithms that are designed to tell you how many years you can expect to live. The first one retrieves the average life expectancy for your country from the internet and then returns the difference between the life expectancy and your age as the number of years you have left to live. This algorithm would adapt in a sense where it updates its input which is the life expectancy but it still would not be considered a machine learning algorithm. In other words, although the input changes, the algorithm itself never updates so it’s considered non-ML. If the algorithm instead would get the number of years the individual actually had left to live once it died and then update its parameters based on how much the prediction was off, then the algorithm would be considered Machine Learning. So, the important difference is that our algorithm directly learns and gets better in its predictions using the value and direction of the error between the predicted and the actual outcome.

Machine Learning is widely accepted to be divided into 3 main subcategories. These subcategories are defined as Supervised Learning, Unsupervised Learning and Reinforcement Learning. There also exist hybrid approaches of these but we don’t need to understand those right now. We just look at these 3 main subcategories. These categories are also referred to as paradigms or learning paradigms.

Supervised Learning:

Supervised Learning is a machine learning technique where labeled data is used for the training. The model makes a prediction for the target variable based on the input and compares the predicted outcome with the actual outcome. This actual outcome is referred to as “label”. The direction and the amount of the error is then processed by the algorithm to learn. We can imagine supervised learning very well as a linear function with multiple input variables and coefficients for every input variable. We are aiming for a certain result value and we want to find the correct coefficients

In that way the algorithm adapts to the actual patterns and relations in the data allowing for better predictions. [48]

Unsupervised Learning:

Unsupervised Learning is a machine learning technique where data is used for the training that is not labeled. So we have no target variable that is to be predicted but instead we just have input data. That means that the algorithm is not designed to predict a certain value based on an input such as in the example with the life expectancy but instead it’s supposed to just work with the input. The algorithm aims for detecting patterns among the data points. We look at an example to get a sense for its usefulness.

Let’s say we have an online shop and we want to give recommendations for items that the customer would be likely to buy so that we sell more products. How can we know which products the customer might buy? Unsupervised learning can be used to improve these recommendations. There are relations or patterns among our items in terms of how they are bought together or separately. Of course, this is based on the taste and preferences of the customers. So, what we are learning is not an objective relation of items but rather how the customers choose them. Some of these patterns might be obvious such as “these chairs match with this table”, but there are many of patterns some of which are very complex and hard to grasp for humans. As we are constantly adding and removing products from our store it would be infeasible to manually explore and then pick these patterns for a recommendation. So, what we do instead is clustering customers into groups. In our example scenario we could say that customers within one cluster share the same taste or preference. The algorithm locates all customers in a multidimensional space where every information a customer can be described with makes up one dimension. the more similar customers are in terms of the information that we use to describe them the closer they are in this space. Areas where many customers are located close together are called clusters. We look at one example for a cluster. Let’s say for whatever reason many customers who buy item a and b also buy item c two weeks later. For humans it can be hard to find patterns as this or much more complex ones. The algorithm would see that there is a higher density of customers in the area of our multidimensional model space for which “bought item a”, “bought item b” and “bought item c” is true. The algorithm looks at how this information fit together and sees that item c is always bought after item a and b was bought and only if both were bought. Like that the simple conclusion is to recommend item c to customers about 2 weeks after they bought item a and b.

The ways unsupervised learning algorithms work differs from architecture to architecture. One common way is to locate the data points in a multidimensional space for which every information about the data is one dimension. Clusters are then defined as areas where the density of data points is higher than elsewhere.

Reinforcement Learning:

Reinforcement learning is a little bit similar to Supervised Learning but the algorithm doesn’t use labeled input data. Instead, the algorithm works with trial and error. Successful results will be rewarded and, in that way, reinforced so the algorithm strengthens that behavior while at the same time bad moves are punished with a penalty and the algorithm learns to avoid that certain behavior.

These 3 Subsets describe *how* we train the system. They specify the data we see, what feedback we get and what objectives to optimize. There is a clear distinction to the ML algorithms. We will discover the 3 algorithms that we are using in this study under methodology. So while the 3 subsets we just mentioned, specify how the algorithm learns in a conceptualized way, the algorithms themselves defines the rules and operations that happen during that learning.

The technique of machine learning that we are using is Supervised Learning. This makes sense because we have an input of stock data and market circumstances and we want the algorithm to learn to predict the next stock price based on that.

The performance of a ML algorithm is measured with by the error. This is usually some way to measure the difference of the predicted value for the target and the actual value of the target. If they differ strongly, we would say that the error is greater and the performance is worse. The same vice versa – if the difference between predicted and actual value is small, we say that we have a small error and a better performance. Concrete ways to measure the performance will be explored under Methodology.

(Methodology):

In our experiment we will evaluate different ML algorithms based on their predictive performance in regards of the data used. We choose an LSTM, an XGBoost and a CNN architecture for this evaluation since they have been shown to be relevant algorithms in the field of stock price forecasting as shown in the literature review. We use non-hybrid, vanilla versions of these models to have more meaningful results. We tune their hyperparameters separately.

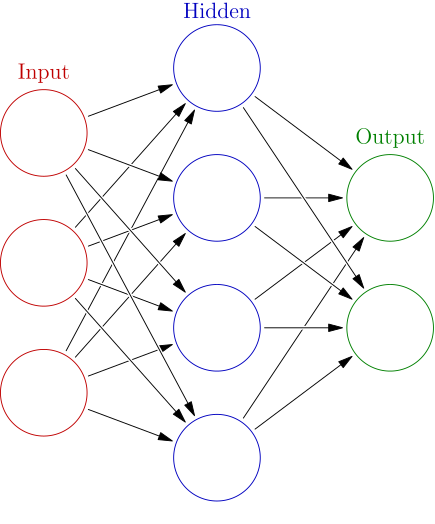
Model selection:

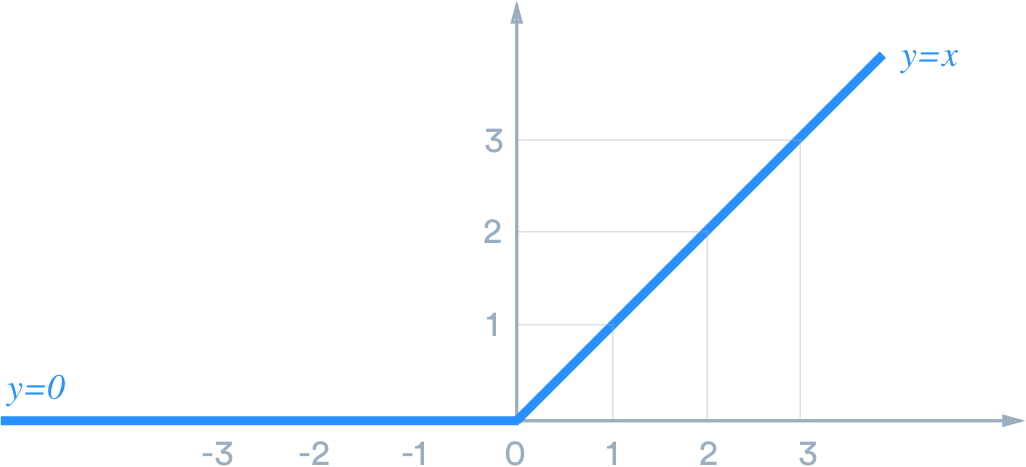
As we understood before Machine Learning can be divided into 3 subsets. Supervised Learning, Unsupervised Learning and Reinforcement Learning. For our task we are looking at Supervised Learning. LSTM, XGBoost and CNN for time series data are some of the best performing Machine Learning algorithms in stock price forecasting as we conclude from our literature review. We will evaluate all three models on our stock data.

We learned earlier that there are 3 main learning paradigms within Machine Learning. For our purposes we are looking at Supervised Learning at the moment. All paradigms can be further divided into different Machine Learning families. These families are groups of ML algorithms. All algorithms within one group or family rely on the same basic idea or architecture of learning. There are many families we could look into but this would be out of the scope of this thesis. The 2 families under Supervised Learning that we are interested in right now are Neural Networks and Ensemble Learning.

Neural Networks:

Neural networks are systems that are inspired by the structure and functionality of the human brain. They consist of neurons which are connected. The artificial neural network adjusts these connections between the neurons when it learns from data. A neural network typically looks like this:

It is important to mention that there are different architectures of neural networks two of which we will explore in more detail under section Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN). They can vary strongly in their functionality, use cases and structure.

The basic version of a neural network is called a feed forward neural network. It always has one input and one output layer. In addition to those the network can consist of an arbitrary number of hidden layers between input and output layer. Each layer including input and output layer consists of at least one neuron. The number of neurons in the input layer needs to match the size of our input. The input necessarily consists of one or more values. So, if our neural network takes images of pixel size (28 x 28) as input then we have 784 values per image and need 784 neurons in the input layer. This applies in our case of a feed forward neural network. We will later see that other architectures like a convolutional neural network for example have a different way of dealing with 2Dimensional input.   
Each neuron is connected with other neurons. These connections are defined by their weight. In our feed forward neural network every neuron has a connection to every neuron of the next layer. Again, this is just the case for our standard feed forward neural network. There exist more specialized structures for which this rule doesn’t apply. The value of a neuron is fed forward to all other neurons it has a connection to. In our basic case the neurons only have connections to neurons of the next layer so the values only move from left to right as the directed arrows between the neurons in the image suggest. When a value is fed forward it is multiplied by the weight of the connection. A neuron receives the weighted values from all neurons that have a connection to it. Within the neurons all these weighted values are added up and the bias of the neuron is added. The value of the bias is learned during training. Before the value is given forward to the next neurons, a function is applied. This happens in each neuron. The function is called activation function. It has this name because the function is what decides whether a neuron is “activated”.  
This is very much inspired by nature. The neurons in our brains receive an input which arrives as electrical signals. Only when the membrane potential exceeds a certain threshold does the neuron “fire” an action potential. Artificial neural networks simulate this behavior with an activation function which also is designed to decide whether and how strong a neuron “fires”. One widely used activation function is called Rectified Linear unit (ReLU). It looks like this:

It “rectifies” or cuts off the negative part and sets every value in it to zero. Every value greater than or equal to zero is retained by a linear function. That’s a very simple understanding of the activation behavior.

We need to introduce the concept of non-linearity here. A linear relation between two variables x and y always has this structure:

Linear functions let us solve linear problems. They always are a straight line when drawn as a graph. For example, the relation of euros to cents is a linear function. You always multiply the amount of euros by 100 to get the number of cents with the same worth.

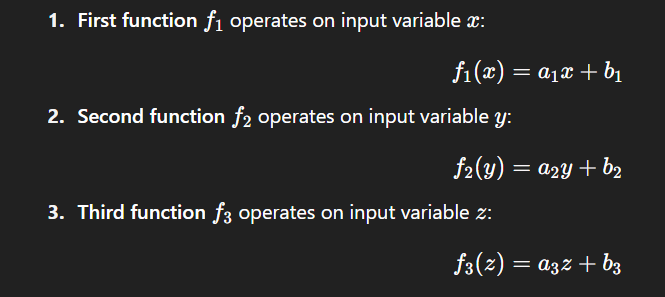
A nonlinear relationship between two variables is everything but a linear relationship. All of those functions are nonlinear.



If we want to model nonlinear relationships, we need a nonlinear function to do that. One example that actually fits nicely to the stock market is compound of interest. If you want to model the course of an investment given a certain interest rate greater than zero, you are facing a nonlinear problem which is only solvable with a non-linear function.

In the fewest cases the problem we want to solve using a neural network is linear. More precisely we usually don’t know the exact relation between the input and output data so we have to assume nonlinearity. Even in the case that the relation is perfectly linear, then we can still use nonlinear functions since they can also model linear relations.

Let’s assume we are having only linear activation functions which is the case that includes having no activation function at all because simply passing on the values would be synonymous to the identity function which is a linear function. Then our whole neural network would only be able to capture linear relationships between input features and the output. This is due to the nature of stacked linear functions. If you stack n linear functions, then no matter how big n is, the resulting function always collapses down to a linear function with n input variables. We look at a simple example that illustrates that.



Stacking:

f1(f2(f3(z))) = f1(f2(a3\*z+b3)) = f1(a2\*(a3\*z+b3) +b2) = a1\*(a2\*(a3\*z+b3) +b2) +b1

= a1\*(a2\*a3\*z+a2\*b3+b2) +b1 = a1\*a2\*a3\*z+a1\*a2\*b3+a1\*b2 +b1

Since a1, a2, a3, b1, b2 and b3 are fixed values, we can compute them together to their effective values.

Weff = a1\*a2\*a3

beff = a1\*a2\*b3+a1\*b2+b1

So, the resulting function is f4(z) = Weff \* z + beff.

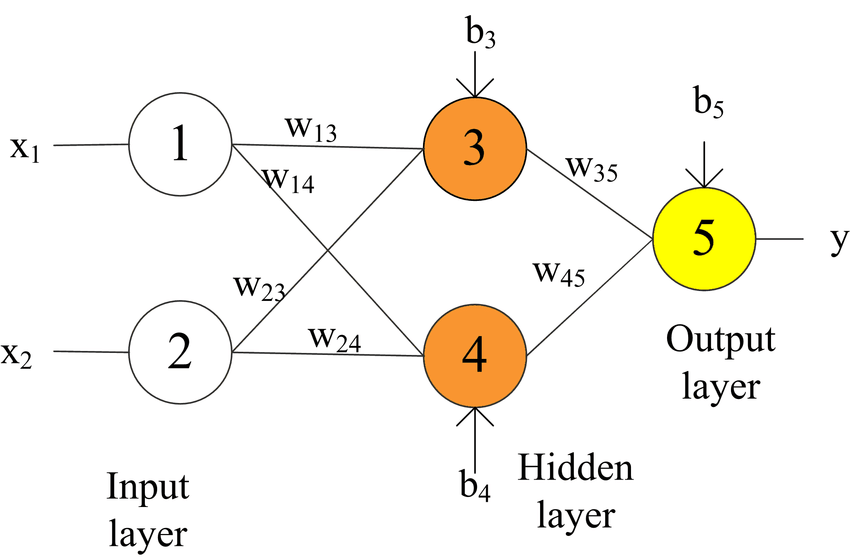
No matter how often we stack linear functions or in the case of neural networks no matter how many layers of neurons with linear activation functions we stack we can never capture non linear relations between input and output.

That’s why we need nonlinear activation functions like ReLU.

We talked about the connections between the neurons and we understood that these connections are defined by their weight and their bias. The weights and biases are learned during training through backpropagation.

To understand the concept of backpropagation we will walk through the process of one prediction using an example network. One prediction or the process of walking the input from the input layer through the network to the output layer is called forward propagation.

The example neural network we are using looks like this:



Where…:

X1 – is the input feature 1

X2 – is the input feature 2

1,2,3,4,5 are the neurons

bn – is the bias in neuron n

wi,j – is the weight of the connection from i to j

y – is the output value

One forward propagation leaves an output. If you have labeled training data and labeled outputs for each input, then you can compare the predicted output with the actual one to learn better weights and biases. We will first only do the process of one forward propagation and then show how the network refines its weights and biases comparing the predicted target value with the actual one.

Example:

X1 = 1.0

X2 = 2.0

Value in neuron 1 = 1.0

Value in neuron 2 = 2.0

W1,3 = 0.5; w2,3 = 0.9; b3 = 0.5

W1,4 = 0.3; w2,4 = -0.4; b4 = 0.6

W3,5 = 0.7; w4,5 = 0.3; b5 = 0.8

The activation function in neuron 3 and 4 is ReLU and the output layer has no activation function.

1. Value x1 and x2 simply arrive in neuron 1 and 2.
2. Neuron 3 receives two values and together with its bias b3 adds them up.

That’s:

X1 \* w1,3 + x2 \* w2,3 + b3

= 1.0 \* 0.5 + 2.0 \* 0.9 + 0.5 = 2.8

Now the activation function is applied on 2.8

Value in neuron 3 = max(0, 2.8) = 2.8

1. Neuron 4 receives two values and together with its bias b4 adds them up.

That’s:

X1 \* w1,4 + x2 \* w2,4 + b4

= 1.0 \* 0.3 + 2.0 \* -0.4 + 0.6 = 0.1

Now the activation function is applied on 0.1

Value in neuron 4 = max(0, 0.1) = 0.1

1. Neuron 5 receives two values and together with its bias b5 adds them up.

That’s:

X3 \* w3,5 + x4 \* w4,5 + b5

= 2.8 \* 0.7 + 0.1 \* 0.3 + 0.8 = 2.79

1. The calculated output y is 2.79

Our calculated output y with 2.79 is what our neural network computes us as result for the input x1 = 1.0 and x2 = 2.0.

Let’s now say that we have an actual output value of 2.1 for this input. Then we can calculate the loss and perform a backpropagation.

The loss E is calculated as:



Where…:

E – is the loss

A5/ y – is the predicted outcome

T – Is the actual outcome

In our case that’s:



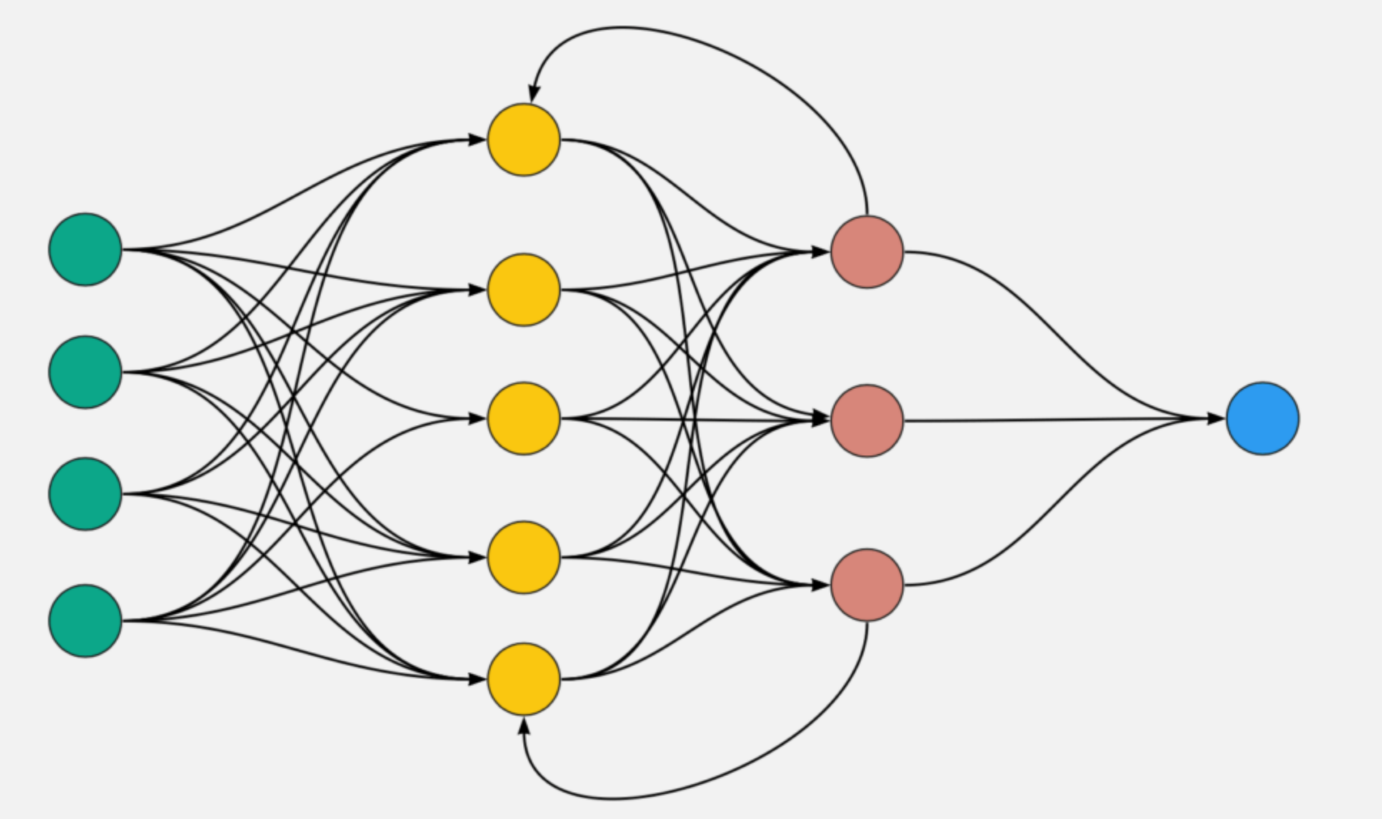
To perform backpropagation we need to calculate the derivates of the

// Further directly from paper into LaTeX //

The structure we just looked at is a very basic version of a neural network. It illustrates all basic mechanisms of modern neural networks. In practice neural networks are much more complex with many hidden layers and many more neurons per layer. Some architectures are even more advanced and introduce different concepts.

One example of these modified neural networks are recurrent neural networks (RNNs). They are designed for and applied to context sensitive problems such as speech, audio or video processing and time series forecasting. RNNs are able to store the context of a sequence. if you for example read a sentence, then in order to make sense of the end of the sentence one needs to remember the beginning of the sentence or at least the key context of the previous words. The same is the case for video data. If you want to make sense of a video clip you cannot just look at the last frame. Instead, you need to look at the whole clip frame after frame while remembering key information from previous frames to understand the content. That’s basically what RNNs are capable of. They can store the context from previous parts of a sequence. That’s why the input of an RNN is not one set of values but instead a sequence of sets of values. We call these sets time steps right now. These time steps are fed into the RNN one after another. After every time step the network stores a hidden state vector. In the base case this hidden state vector captures one value or hidden state for each hidden neuron. Hidden neurons are neurons in hidden layers. These hidden states are stored until the next time step and simply passed as normal values to other neurons. The input for the RNN has the size of the data of one-time step. In the case of a video, it would be the size of one frame. When we feed the first frame into the RNN, a hidden state vector is produced. When we move on to the next frame then the neurons receive this next frame as input and in addition to that the hidden state vector that was produced in the previous time step. Like that the network stores contextual information across time steps and always processes the current input data taking into account the data from previous time steps. In the case of stock price forecasting one time step is one period for which we have input feature values. In our specific context we have daily stock data so one time step is one day.

This is how an RNN can look like:



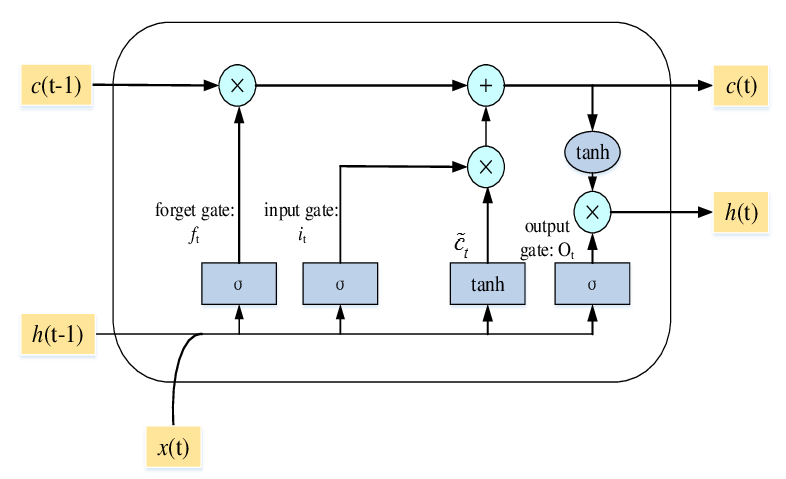
[50]

In that case the green neurons are our input layer, so we know that our input consists of 4 values or in other words each time step carries 4 values as input. We have two hidden layers which are the yellow and the red neurons and we have an output layer which consists of the blue neuron, which means the network computes one output value. All arrows that are directed from left to right symbolize connections used during one forward propagation. The arrows from the first and third red neuron are directed to the left. That means that at time step t we feed the 4 input values which are represented by the green neurons into the network. We perform a forward propagation as we learned before with one exception. the output the first and third red neuron produce at this time step t is stored. When we are done with this forward propagation, we feed the 4 input values from time step t+1 into the network. We do that by assigning the input values to the 4 green input neurons. We again perform a simple forward propagation with the difference that we feed the stored hidden states from the previous time step t into the yellow neurons according to the arrows.

The process of backpropagation hereby works similar to the feed forward neural network that we looked at earlier with the difference that we backpropagate through the time steps. To do so, we would unroll the network along its time steps. Like that we end up with long chains of derivatives for the single weights and biases. The more time steps we have the longer the chains of derivatives are. We proceed without looking at an example here, because in its core functionality it is similar to the back propagation we learned before.

The main thing to keep in mind for RNNs is that they can interpret input data in the context of previous input data. They do that by storing key information from previous time steps.

Long Short-Term Memory (LSTM):

LSTM neural networks are recurrent neural networks. Beyond feeding output values to the next time step, LSTM networks use different gates to facilitate the storing of context specific states. An LSTM network also consists of layers. The greatest difference to other neural networks is that the LSTM cells or units have built in mechanisms to maintain a safe states. Furthermore, LSTM layers always consist of one LSTM cell that acts as multiple units of that cell by storing one value for each unit in the form of state vectors. So, there is one cell per layer but it behaves as many cells since it processes a number of input values from the input vectors parallel with separate weights. There are two state vectors that are maintained by the LSTM network, a cell state and a hidden state. The cell state vector is only passed across time steps and serves as input to the same cell for the next time step. The other state vector is the hidden state which is also passed to the same cell in the next time step but besides that it is passed along layers where it serves as input to the next layer. So, an LSTM cell always receives the cell state vector from the same layer from the previous time step, the hidden state vector from the same layer from the previous time step and an input x which is the raw input for the first LSTM layer or for every following LSTM layer, the hidden state of the previous layer within the same time step.

[57]

Input:

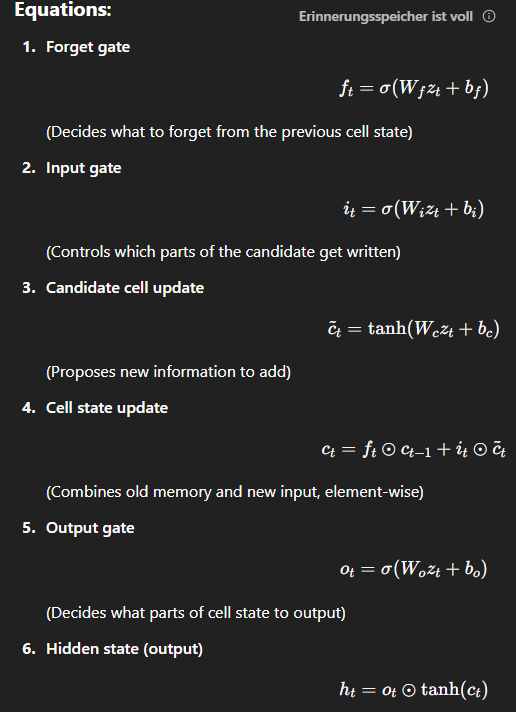
C(t-1) – is the cell state from the same cell from the previous time step

H(t-1) – is the hidden state from the same cell from the previous time step

X(t) – is the input which is the raw input for the first LSTM layer and the hidden state from the previous layer for every following layer.

Output:

C(t) – is the new cell state which will be the input for the same cell in the next time step

H(t) – is the new hidden state which will be the input for the same cell in the next time step and for the next layer.

Z(t) – is the combined vector from concatenating x(t) and h(t-1).

Bi – is the bias vector for the input gate of that cell which stores a bias for every unit of the layer

Bf – is accordingly the bias vector for the forget gate

Sigmoid (…) – is the activation function that is applied at the gates and introduces nonlinearity

Tanh (…) – is another activation function

In the image above we can see how the LSTM cell processes the input. The equations for the gates, the outputs and the inputs are described above to understand how the LSTM cell computes the values. Z(t) is defined as a combined vector that simply concatenates the input x(t) and the previous hidden state h(t-1). The LSTM network stores weight and bias matrices. These weight matrices hold values for each gate for each LSTM unit for each input element. And the bias matrices hold biases for each gate for each LSTM unit but they are the same for all input elements, so the same bias is added to every element of the combined input vector z(t). The sigmoid and tanh functions are applied as part of the gates to introduce nonlinearity to the model. We understood why this is an important concept for Machine Learning. In relation to the basic neural network, we looked at in the beginning, the LSTM has only one cell per layer but still processes vectors, so a multivariate input. It also differs by maintaining certain safe states to capture context or past events. That make it very suitable and strong for time series problems and it is the reason it was observed to perform good on stock price forecasting.

In practice the LSTM layers are typically followed by fully connected layers that make predictions out of the learned patterns. That is not technically necessary, the final hidden state vector can also serve as output but in case the output needs to be further classified or put into a different shape fully connected layers help with that.

Convolutional Neural Networks:

Another type of neural networks is a Convolutional Neural Network (CNN). These are explicitly performant with spatial data, for example images. That is true because they detect local patterns and maintain spatial relation of the input data. To understand what that means we look at how CNNs function. In our explanation we will explain their functionality using an image as input and after that we discover how they can be applied to time series data.

CNNs mainly consist of three different types of layers:

* Convolutional Layer
* Pooling Layer
* Fully-Connected Layer

A series of convolutional and pooling layers usually is supposed to extract important features from the input while fully-connected layers in the end make sense of the extracted features and classify the input.

The first layer of the network typically is a convolutional layer. It receives a 3D input tensor. In the case of an image that is (height(pixels) \* width(pixels) \*channels (color channels)). The layer is mainly defined by its parameters for which…

Number of filters – is the number of filters that are applied on the input of that layer

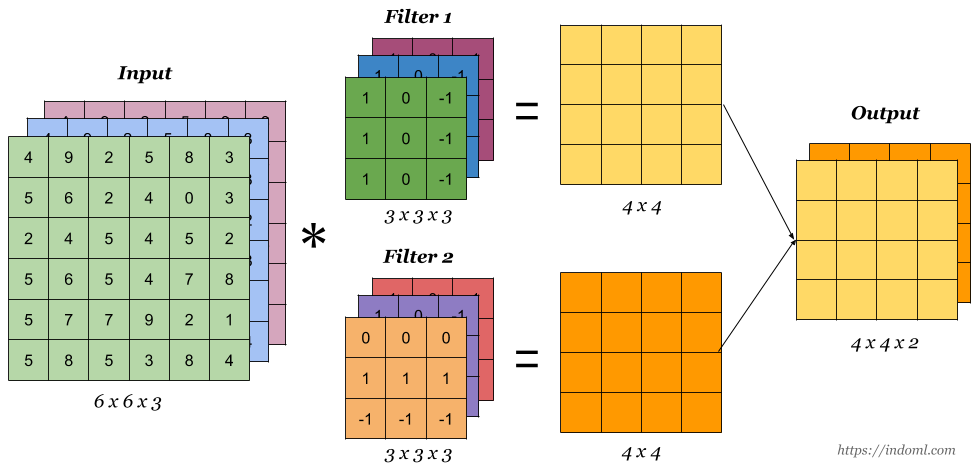
Kernel size – is the size of the filters

Stride – is the step size with which the filters slide over the input

There are more parameters with which you can for example better define the nature of filters. We explain them in more detail in the section hyperparameter tuning but for now we only need to understand the functionality taking into account these 3 parameters.

The convolutional layer applies filters on the input. These filters are supposed to extract information about the occurrence of certain patterns. 3x3xchannels or 5x5xchannels is the typical shape of these filters but while they usually have the number of channels as one dimension, they can have any size for the other two dimensions. The filters are tensors themselves that carry one value in each cell. When a filter is applied to input it slides over the input with a step size which is defined as the stride parameter. A stride of one means the filter moves one cell per step. We said that the filter slides over the input. That means that the filter overlaps with certain cells of the input. In an overlap, the values of the two cells one from the input and one from the filter are multiplied together and the products of all overlaps at one filter position is summed up. Applying one filter results in a feature map for that filter. The summed-up values of one position of a filter are part of the value for the cell of the feature map at that position. In addition to that we add a bias value. Every cell in a feature map is added with the same bias so there is one bias value per filter. In that way we apply all filters of the convolutional layer and end up with one feature map for each filter. The idea behind the feature maps is that the filters have values in their cells in an order which, when multiplied with the input result in a large total sum. A large sum in that case means that a strong occurrence of the pattern encoded in the filter is detected at that position. One feature map now tells us for each position how strong the feature that it is designed to detect occurs at that position. After applying all filters, we end up with one feature map per filter.

Let’s look at one example. In this example we see our input which seems to be an RGB image and we see two filters that have coefficients for every color channel. Unfortunately, we can’t see the other channels right now so we only apply the upper most channels of the filter to the upper most channel of the input. Actually, every channel of the filter would be applied to the according channel of the input.[51]



Like that the filter is applied. Here we see, that the green filter would result in the highest value if the cells on the left side are as big as possible while the cells on the right side of the filter are as close to zero as possible. This sounds like the filter could be detecting a vertical line.

The next step is to add the bias for each cell in the feature map.

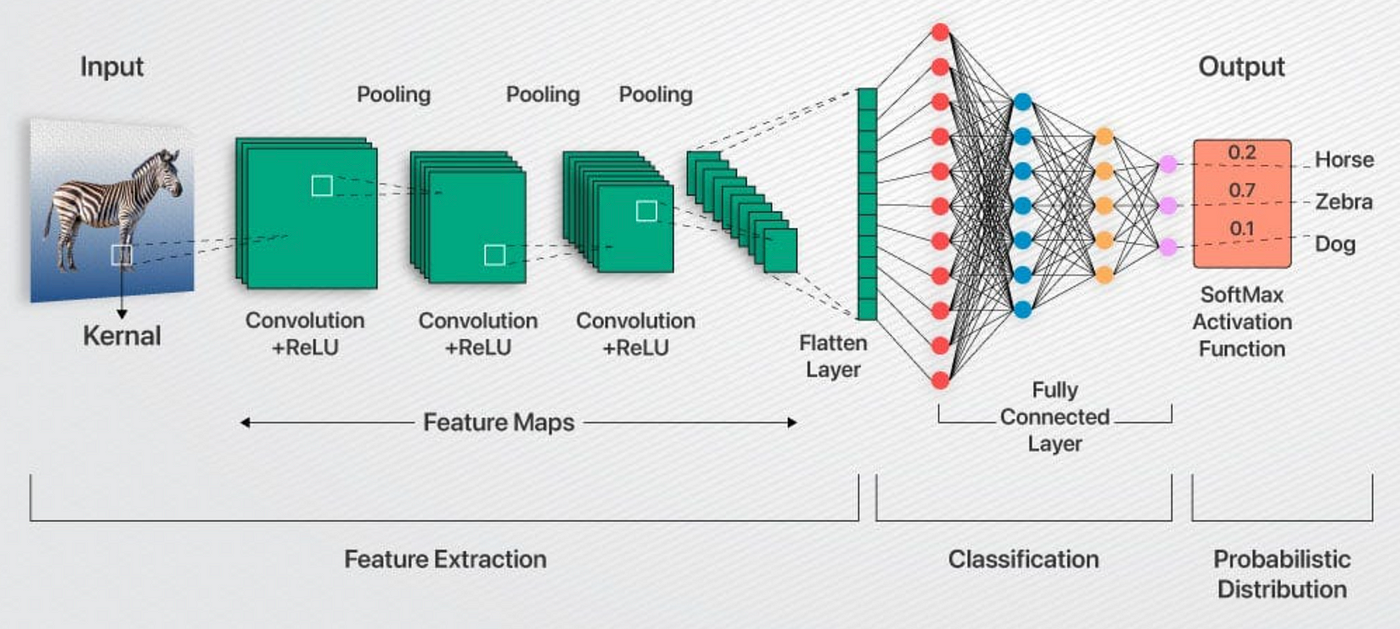
Right after every convolutional layer we apply an activation function to introduce nonlinearity to the model. We looked at this concept earlier in this section. The activation function (e.g., ReLU) is simply applied to every cell of every feature map.

Like that we get the feature maps that will make up the channels for the next layer.

A convolutional layer is typically succeeded by a pooling layer. The pooling layer is designed to reduce the dimensions of the feature maps. This is an advantage for computational efficiency but it especially makes the model more robust. In the sense of an image, we can imagine it as reducing its resolution. We not just reduce the resolution randomly but instead when we merge pixels, we choose to keep the most important one to retain the most important information. The reason it makes the model more robust is that it filters out unnecessary information about the image and like that makes itself less susceptible to overfitting. So, pooling It is the process of down sampling the feature maps. One common approach for this is Max-Pooling. When Max-Pooling is performed we choose a frame size which is often 2 x 2. We move this frame over the original feature map and create a new feature map with reduced dimensions. For each position of the frame, we choose out of the 4 cells the highest value and take it as new value that represents this square at which the frame was. Commonly a stride of 2 is used. That means we move the frame 2 cells to the right so that it captures the 2x2 square of cells next to our first one.

Like that we reduce the dimensions of the feature maps.

This series of convolutional layer followed by a pooling layer can repeat a certain number of times. Each time more complex patterns can be detected. We imagine it like that. In the first convolutional layer we can only detect simple features such as edges, curves and maybe colors or other simple patterns. In the next convolutional layer, we apply again a certain number of filters but now the channels have changed. They changed from being the three-color channels of an image to being maps that talk about occurrences of other features in that image. Where we before, assuming an RGB image, we had 3 channels, we now have the number of feature maps as channel size. So, if the first layer applied 32 filters, then we got 32 feature maps which means in the second layer the filters have a size like (filter\_height \* filter\_width \* 32). That enables the network to capture relations across different features. It can detect more complex patterns like objects by combining simple patterns like colors or edges.

This series of convolutional and pooling layers are followed by fully-connected layers which use the extracted features. They apply a softmax function that takes the extracted features and creates probabilities of class membership that sum up to one. The whole process is illustrated below.

The network learns the filter weights and the biases for each filter. They are refined through backpropagation by calculating the loss. Filters that detect useful features will get larger weights and activate even more after that. The other filters that don’t contribute to a better classification or send wrong signals will be weakened so they fire less.

In this study we are dealing with time series data so how can we use this architecture for time series data? The answer is quite simple; we reduce the dimensionality of the input by one. We can it imagine as having multiple features per period instead of multiple channels and the height\*width dimension per channel simply gets reduced to a sequence length which is equivalent to a table with just one column. The filters now instead of being 3D tensors as for images are just 2D Arrays that slide along the timesteps.

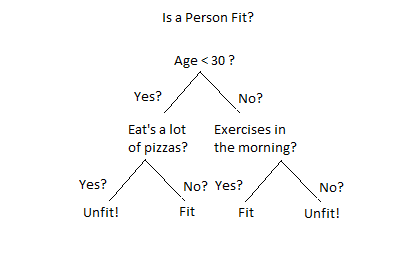
XGBoost:

XGBoost models work differently, they are no neural networks. In order to understand how they function we first need to understand two smaller concepts one of which is decision trees and the other one is random forests.

Decision Trees [54]:

Decision Trees are visual represented as an upside-down treelike structure. Each internal node can be called decision node. It contains a simple yes or no question. Each of these decision nodes split the branch into two sub branches for which one represents the path you take when you answer yes to the question in the decision node and the other one represents the path that you take when the answer is no. At the end of the branches there are other decision nodes that further split the tree and make it deeper and wider until all path end in a leaf node that makes a prediction for the data that belongs to that path.

[53]



That is a simple example for a decision tree. It is designed to tell whether a person is fit based on its age, its pizza eating behavior and its exercise routine. The root node splits the data into two subsets. One subset contains all persons who are under 30 years of age and the other one contains every person who is at least 30 years of age. The subset with the people who are younger than 30 years is split again into subsets of people who are younger than 30 and eat a lot of pizza and into a subset with people who are younger than 30 and don’t eat a lot of pizza. Every person in the subset in which people are younger than 30 and eat a lot of pizza is considered unfit according to the decision tree and every person who is younger than 30 but doesn’t eat a lot of pizza is considered fit. The same practice is applied to the people that are at least 30 years of age and either exercise in the morning and don’t exercise in the morning.

That is how a decision can look like.

But how are they constructed based on the data? In general, for Boolean features in the data simple yes or no question if sufficient. If we have continuous values for a feature, we simply split the data using a threshold in the scale of that feature. We ask “is the value smaller than our threshold or is it at least our threshold?”. Beyond that, splits are chosen according to another rule which can be different in trees. This rule is supposed to measure the quality of a split and one common rule that is used is the entropy-based information gain. When we use the information gain to choose a split then the splits are not performed randomly but instead chosen in a way so that the entropy of the data within the two created subsets is as low as possible. We can understand the entropy in a subset to be low when the class concentration in this subset is high. So, when 80% of the data belongs to one class and the other 20% belong to another class than the entropy is much higher as if we have 5 parts with 20% of the data that each belong to separate classes. Like that the data is split most efficiently starting at the root node. In practice we define a list of parameters that limit the tree in certain dimensions or their splitting behavior and in that way prevent overfitting. We can specify the maximum depth of a tree, the minimum number of samples per leaf and the minimum impurity decrease required to split. In the section hyperparameter tuning we will discover what concrete parameters we tune for our tree-based XGBoost model.

So, we want to emphasize the behavior of a decision tree. It “learns” from the data in the moment it is created by recursively splitting the data into subsets that are increasingly pure in terms of their classification.

Random Forests [55]:

Random forests come from a concept called “ensemble learning”. It is a concept that is based on the fact that more models sometimes can make better predictions than fewer models when aggregated together. We learned before, that decision trees perform classification or regression by recursively asking yes or no questions to the data that split the data into subsets of highest possible purity.  
When we apply ensemble learning to decision trees it means that we train multiple trees – hence the name forest – and then combine their predictions by using them as votes for the final prediction. In order to make sense out of that we need to create different trees in the process and we need to make sure that we do not end up with the same tree multiple times since just having the same tree multiple times doesn’t improve our predictions. We can ensure that the trees are all slightly different with altering different parameters for each tree. The technical term for that is uncorrelatedness among the trees. One possibility to ensure uncorrelatedness is called bootstrapping. This is creating smaller data sets out of our data set. With normal decision trees we feed the whole data set to the tree. However, with bootstrapping we create many slightly different subsets of the data and feed these different subsets to the trees. Being fed different data generally means that the trees roll out differently. Combining the predictions of many trees that are different because bootstrapping was used is called bootstrap aggregating or for short “Bagging”.

Another method we can use to introduce variation to our trees is by shuffling which features a certain tree sees. This concept is called “feature randomness”. While with bootstrapping we divided the data into many different subgroups of smaller size but with the same number of features, feature randomness relies on creating subspaces of features to make the trees behave differently. So, if we have features A, B and C in our data then one tree might only see features A and B while another tree only sees the features B and C.

Every tree makes a contribution to the final prediction in form of a vote. In the case of regression, the overall prediction is the average of the predictions of the single trees while in classification the class that was predicted by most trees is the overall prediction. The idea behind it is straight forward. When we use multiple slightly different trees to average their outcome then we can benefit from statistically better and smoother performance. This is due to the fact that the ensemble prediction of those trees generally outperforms every prediction of a single tree that it consists of.

XGBoost [56]:

Ok, after getting to know decision trees and random forests we now look at how exactly an XGBoost model works. Its name is short for extreme gradient boosting.

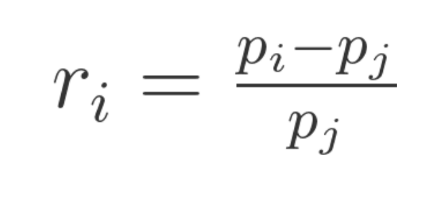
XGBoost models use a principle called boosting. It works by combining weak learners sequentially to improve the predictive accuracy with each learner. The model starts with one prediction for each data point. For regression tasks that prediction typically is the mean of all target values. With the first prediction the model computes the residuals or errors from every data point to the prediction. After that the first tree is learned. It takes the computed residuals as well as the first and second derivative of the loss function and learns splits that group similar residuals together. Once the tree is constructed it carries leaf nodes which data points belong to that share a similar residual as this was the feature after which the tree split. Now an adjustment for the initial prediction is computed for each leaf node by taking into consideration the first and second derivative of the loss function. The first derivative tells us simply how strong and in which direction we reduce our error but it doesn’t tell us how far we can go until we again would worsen the prediction. Because of that we use the second derivative that tells us how strong the course of the loss function is curved. To be more precise it tells us how strong the gradient of the function changes. The extreme in extreme gradient boosting stands for a couple of mechanisms the model uses to make it fast, efficient and accurate. One of these mechanisms is to use the second derivative to compute the adjustment value. Another mechanism which is used is called L2- regularization. It regularizes the weights of leaf nodes by giving penalties for too large values at the leaves preventing the model from overfitting. Other regularizations are used to generalize the model and prevent overfitting for example a maximum tree depth value. After learned the adjustment values are multiplied with a learning rate that is fix for the whole model and then added to the prediction. This process can repeat a lot of times. Every iteration creates a new tree that again adjusts the predictions of all data points towards the right direction. If the learning rate is lower, one tree adjusts the prediction only slightly so more trees are needed but they can have a better accuracy since they correct the predictions with a finer granularity.

Volatility:

Volatility is known as the degree to which data disperses over time or in other words “Volatility is a statistical measure of the dispersion of returns for a given security or market index. It is often measured from either the standard deviation or variance between those returns.”[15]. We understand that the volatility has to be measured to deliver meaningful results. The most common way to measure the volatility of stock courses in finance is by calculating the standard deviation or more precise the sample standard deviation of logarithmic returns [16]. If we do that we get a measure that tells us how strongly the data points differ on average from the mean of the values. That’s quite good to compare the volatility of different stocks or ETFs.

To do so, we first need to calculate the logarithmic returns. We choose to calculate logarithmic returns instead of simple returns. We can understand why we choose the logarithmic return over the simple return by looking at a simple example.

We calculate the simple return with



… where:

Ri = stands for the current data point’s simple return

Pi = stands for the current data point’s value

Pj = stands for the previous data point’s value

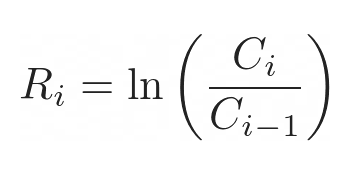
Let’s say we have a stock price development for a company X of the following:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Month 1 | Month 2 | Month 3 |
| Stock price | 100$ | 150$ | 100$ |
| Simple returns | - | +50.00% | -33.33% |

Since the price ends up at 100$ again we know that the total return is 0%. But when looking at the returns we have Month 1-2: +50% and Month 2-3: -33.33%. So, we could assume that the average return is (-33.33%+50%)/2 = 8.335% which is clearly wrong. The mistake we made in our calculation is not considering the compound of interest. We need to understand that the 50$ change from Month 2 to Month 3 is already based on a starting value of 150$ which is different than the starting value for the change in value from Month 1 to Month 2, which was again 50$ but this time based on 100$. To make our numbers make sense, we would need to calculate the total return by considering compound of interest by multiplying the returns instead of adding them up. Then we would get the correct 0% total return. The part where the log returns is a more intuitive way of doing it, is when it comes to looking at returns over single time steps as a human. Let’s extend our course of the stock price by another 4 Months:

Our returns now would be

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Month 1 | Month 2 | Month 3 | Month 4 | Month 5 | Month 6 | Month 7 |
| Price | 100$ | 150$ | 100$ | 150$ | 100$ | 150$ | 100$ |
| Simple Return | - | +50.00% | -33.33% | +50.00% | -33.33% | +50.00% | -33.33% |

If we look at returns like these, we could easily think that we end up with a positive return overall but when looking at the actual stock price we see, that we are just oscillating between two values. When we use the logarithmic return all that gets much more intuitive and easier to use. We calculate the logarithmic return with: [17]

… where:

Ri = stands for the logarithmic return at the current data point

ln = stands for the natural logarithm

Ci = stands for the current data point’s value

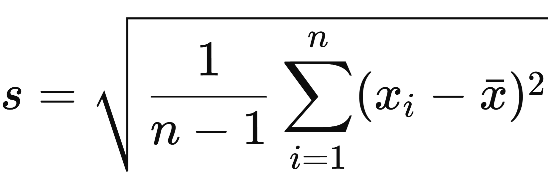
Ci-1 = stands for the previous data point’s value

Let us now also calculate the logarithmic return of our stock course.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Month 1 | Month 2 | Month 3 | Month 4 | Month 5 | Month 6 | Month 7 |
| Price | 100$ | 150$ | 100$ | 150$ | 100$ | 150$ | 100$ |
| Simple Return | - | +50.00% | -33.33% | +50.00% | -33.33% | +50.00% | -33.33% |
| ln Return | - | 0.405 | -0.405 | 0.405 | -0.405 | 0.405 | -0.405 |

If we now add up the logarithmic returns, we are left with 3\* (0.405) – 3\* (0.405) which is 0. So using the logarithmic return instead of actual stock prices itself or simple returns of the stock prices, leaves us with numbers that are detached from the compound of interest effect and give us a much more intuitive understanding of the volatility. We now have our logarithmic returns, which leads us to the next step which is calculating the sample standard deviation based on them.

To calculate the sample standard deviation, we use this formula.



… where:

n = stands for the number of data points that we calculate the sample standard deviation for

xi = stands for the currently observed data point (for us it’s the current logarithmic return)

= stands for the mean of all data points (for us it’s the mean of all logarithmic returns)

In our codebase we write our own pipeline to calculate the volatility in the way we just discussed. For that we use a pre-implemented function for the sample standard deviation which is located in *pandas.DataFrame.std()*.

The standard deviation gives information about how strongly the data points are deviated from the statistical mean. If the standard deviation is lower it means that the data points are on average closer to the mean and if the standard deviation is higher, it means that the data points are spread wider.

We now have defined our metric for volatility and we can start choosing the stock data we want to train the algorithms on.

So far so good. We now have a measure that tells us how volatile the stocks have been in the past. The volatility score is averaged over our time frame which is 5 years. We could go ahead and select the stocks based on that score. If we do so, we might see, that stocks with the same volatility score seem to have much different characteristics in the data.

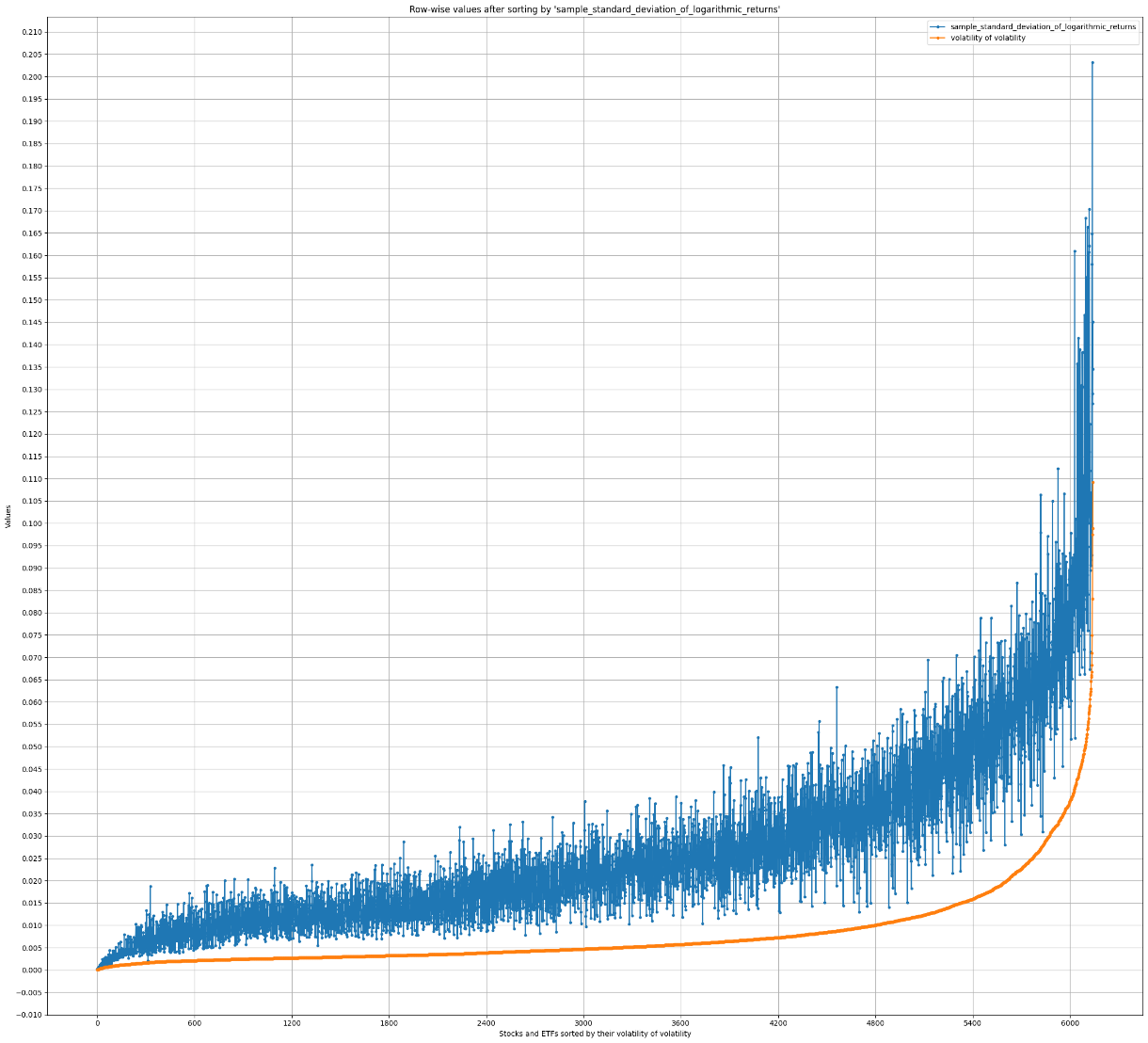
To understand that using an example we take two stocks. One stock is oscillating moderately strong throughout the whole course let’s say its volatility score is 5 at every point in time. The other stock oscillates compared to the first one very little during the first half but a lot during the second half. Let’s say during the first half it has a volatility score of 1 and during the second half it has a volatility score of 9. The average volatility of these two stocks is 5 even though they have very different characteristics and the first stock has a more constant oscillation which doesn’t change as much with time. To account for that problem and to make sure that the stocks we chose are as similar as possible in terms of their volatility we look at another metric called the volatility of the volatility. As one might think this metric simply measures how much the volatility itself changes over time. To measure that we calculate the pure volatility just as described above but not for the whole set of data but instead multiple times for rolling windows. So, we basically compute the volatility for every 75 consecutive data points in the data set. After having the volatility for each of these windows we again calculate, just as explained above, the volatility of these volatility measures. Like that we get the volatility of the volatility for a stock or ETF.

Stock selection:

We have defined how we measure the volatility of a stock course. That enables us now to choose stock data based on which we want to train our models. In the codebase the Jupyter Notebook file ‘Volatility\_Pipeline.ipynb’ completely contains this process. At first, we are using the alpaca-trade-api to retrieve our historical stock data. This API requires you to verify yourself with an API\_KEY and an API\_SECRET. After we created ourself an account and retrieved the credentials, we now create a list which contains common stocks and ETFs listed on AMEX, ARCA, BATS, NYSE, NASDAQ or NYSEARCA [18]. After that we define the time frame for which we want to retrieve the historical data, we choose it to be the past 5 years until 2025-05-07 (yyyy-mm-dd). We then start to make calls to retrieve the data. We batch the call so that we get 200 stock courses with one call. This is due to the rate limit of the alpaca api [19]. Like that we retrieve the historical prices of about 11500 stocks and ETFs.

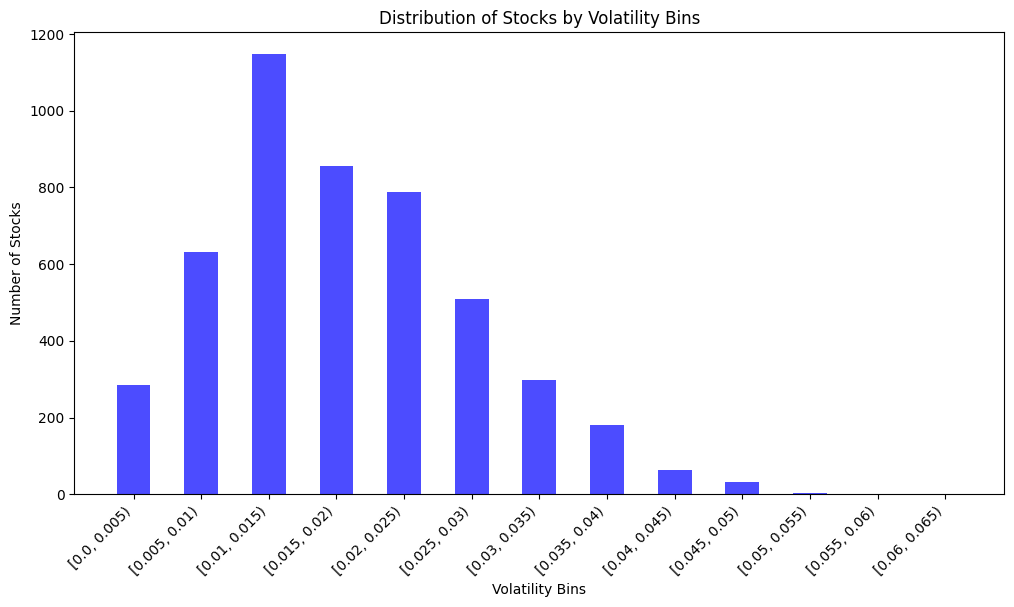
We further process this data to be only left with a table with the stock symbols as columns, the timestamp as index and the close price as a column as well. During preprocessing we would need to account for missing values. It would be much easier to already now just chose stocks and ETFs without missing values. So, we look how many stocks and ETFs have missing values and its about 5000. That means we have about 6500 stocks and ETFs left with no missing values. Since we only need handful of historical courses, we simply decide to drop all that have missing values and just proceed with the ones without any missing value. Besides that, we also drop all historical courses that have the same closing price over at least 30 days. When a stock or ETF has the same closing price over a number of days it can have different reasons but usually means that the asset hasn’t been traded in that time. For whatever reason the price stays the same for at least 30 days, we don’t want such asset courses in our data because it is obviously not natural behavior of the asset price and it gives the algorithm a hard time to learn.

We now want to extract information regarding volatility. To do so we apply the methods of calculating the volatility and calculating the volatility of the volatility for our stocks as described under methodology.Volatility. We apply the functions and store the returned values in a table. As we understood before we the volatility of the volatility to be as low as possible for all stock and ETF courses while at the same time having a wide range of different volatilities of which we use stocks and ETFs to train our model. To filter out stocks and ETFs that have a higher volatility of volatility we need to visualize our data set. For that we draw a graph. The graph shows the measure of volatility (blue) and volatility of volatility (orange) on the y-axis for each stock or ETF and the number of the stock on the x-axis. We sort the stocks by the volatility of their volatility in increasing order.



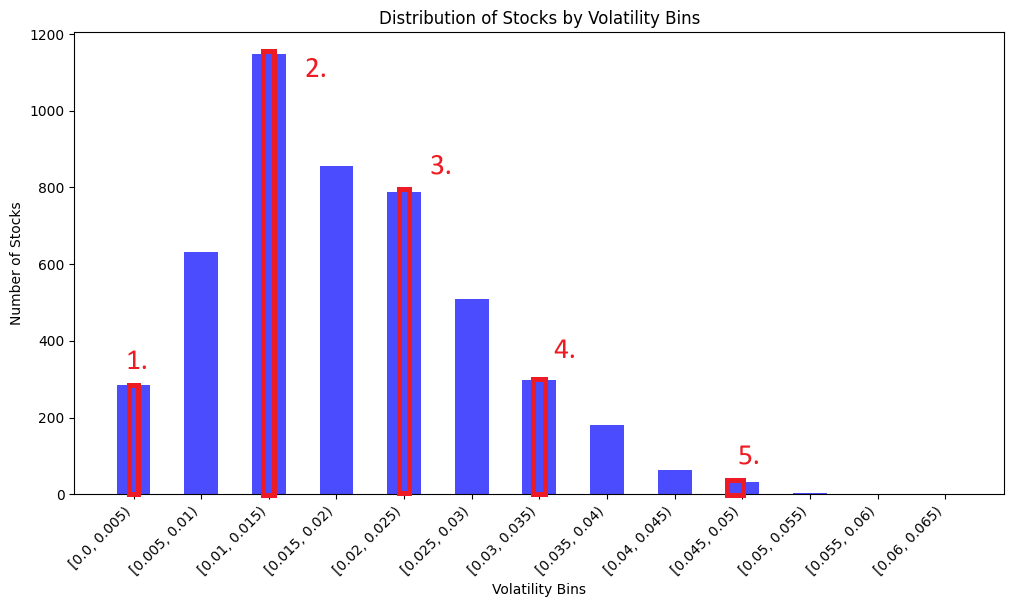
We see that the volatility of the volatility seems to correlate on average with the volatility. The greater the volatility of the volatility the greater the pure volatility. Now, for us two things are most important. The first one is that we want to have stocks from volatilities within a range that is as wide as possible to have a more meaningful result of our experiment. The second important thing is that we want the volatility of the volatility to be as low as possible. We explained before why we want that. Since the volatility seems to correlate with the volatility of the volatility, we need to make a compromise when deciding for a value of the volatility of the volatility which serves as a threshold. We will just drop every stock or ETF that has a volatility of volatility greater than that threshold. We take into consideration that we only need a relatively small number of stocks and ETFs. Besides that we take into consideration that the volatility of volatility seems to increase dramatically on the right side of the graph so we definitely want to cut off at some point before that dramatic increase. After careful consideration we decide to cut off all stocks and ETFs that have a volatility of their volatility of more than 0.01. This is pretty much at stock 4800 in the graph. Like that we ensure to have constant oscillation based on the given volatility while still being able to capture a wide range of stocks and ETFs with different volatilities.

Across the range of volatility, we want to evenly pick 5 values for each of which we again pick 15 stocks or ETFs that are as close as possible to the values in terms of volatility of the historical data. To get a sense for that we first create bins of volatility and sort our stocks and ETFs into these bins. We plot that distribution.



We now want to manually choose 5 bins and 15 stocks or ETFs for each bin for training. To make the results as meaningful as possible we define the ranges of our bins in a way so that we capture the widest possible range of volatility. In other words, we want bin1 and bin5 to be as far apart as possible on the graph and the remaining 3 bins to be in between them while having equal distances to their neighbors. At the same time, we want the stocks and ETFs within one bin to be as similar as possible in terms of their volatility. Based on the distribution which is visible in the plot, we choose our volatility steps to be:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Volatility score interval | 0.00235  – 0.0027 | 0.01243  - 0.01248 | 0.022425  - 0.0225 | 0.032416  - 0.0326 | 0.045  - 0.046 |



This will allow us to train and evaluate the models based on multiple stock data for each value of volatility which we expect to give us a statistically better and closer to the average measurement compared to just using one stock for each step which is susceptible to outliers. We especially expect the models to have a similar predictive accuracy for stocks within the same volatility interval and to show clear jumps in performance when trained based on stocks of different intervals.

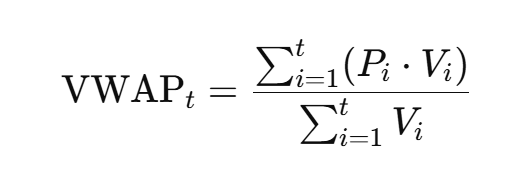
Data Preprocessing:

During the previous section (‘stock selection’) we selected the stocks that can be grouped into 5 intervals or bins of different volatility and which we want our models to be evaluated on. Since we already dropped stocks with missing values, we don’t need to perform any further imputation.

Our historical data so far has the features: open, high, close, low, vwap, volume, trade\_count, symbol – where symbol is the ticker symbol of the stock and will not be used for training.

The other ones are:

* trade\_count, which tells how many transactions have been made in a period.
* Volume, which is the number of shares that have been traded within a period.
* Volume Weighted Average Price (VWAP) is the typical price of a stock or ETF weighted by its volume [36]



Where…

Pi – is the price at which the asset was traded

Vi – is the volume at which the trade executed

t – is the number of trades in the period.

So, if within one period particularly many trades executed at a certain price than this price is weighted more than a price at which few trades executed. That means while the close price only tells you the price at which price the last trade executed, VWAP is averaged over all trades in that day.

The next step is to preprocess the historical data from each asset by adding other features. Some of those are retrieved from APIs and some are calculated using the existing features.

We choose the features based on which the review study by Kumbure et al. [5] has shown to be mainly used in stock price forecasting. We also add some indicators mentioned in the articles “Template:Technical analysis” from Wikipedia [20] and “Technical Indicator: Definition, Analyst Uses, Types and Examples” from Investopedia [21].

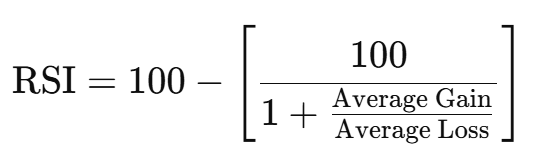
Feature Engineering:

Technical Indicators:

Relative Strength index (RSI):

The RSI is a way to measure speed and directions of price movements. The RSI always lies in the range of zero to 100 and it is commonly considered overbought when above 70 and oversold when below 30. The RSI formula is:

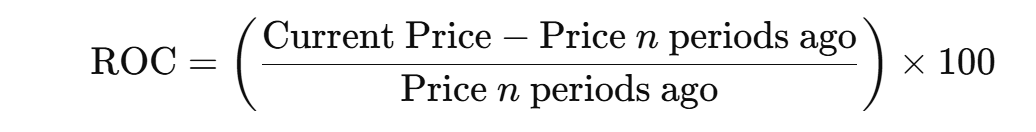
RSI = 100 – [100 / (1 + (Average of Upward Price Change / Average of Downward Price Change))]



Where Average Gain and Average Loss are Exponentially weighted moving averages. [22]

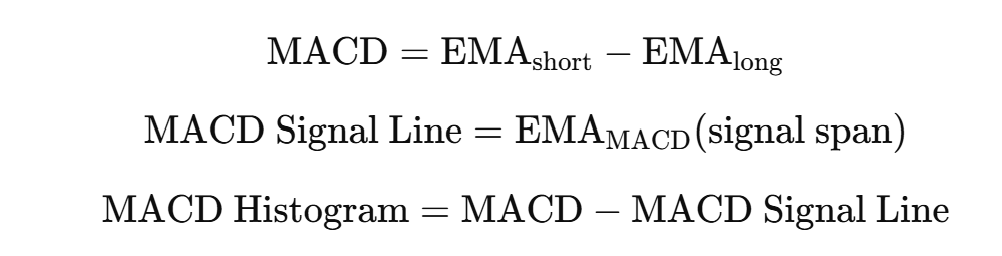
Rate of Change (ROC):

The ROC is an indicator that shows the return in percent not compared to the previous period but to the price n periods ago. A common number for n is 14. [23]



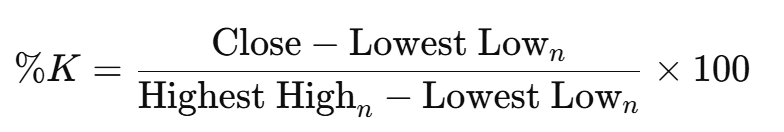
Moving Average Convergence Divergence (MACD):

The MACD is used to measure both the momentum and direction of a price movement. It consists of a the MACD line and the signal line. The MACD line measures the difference between a shorter exponential moving average and a longer exponential moving average. The signal line is again an exponential moving average of the MACD line. Due to the fact that both lines crossing has implications for the price movement and we want the model to better identify that relation, we also add a histogram column that simply is calculated as the difference between MACD line and signal line. [24]



Stochastic Oscillator:

The stochastic Oscillator is another momentum indicator that measures where the current closing price lies in the range of the highest high of a certain past period and the lowest low of that same period in percent. This value is called %K. A second value which is called %D contains the moving average of %K for a short period. Since again the relation of those 2 lines is of importance to us and we need to make the machine learning model consider this relation even though it cannot see the the lines on a chart we add a stochastic difference value that is calculated as the difference of these 2 values. [25]

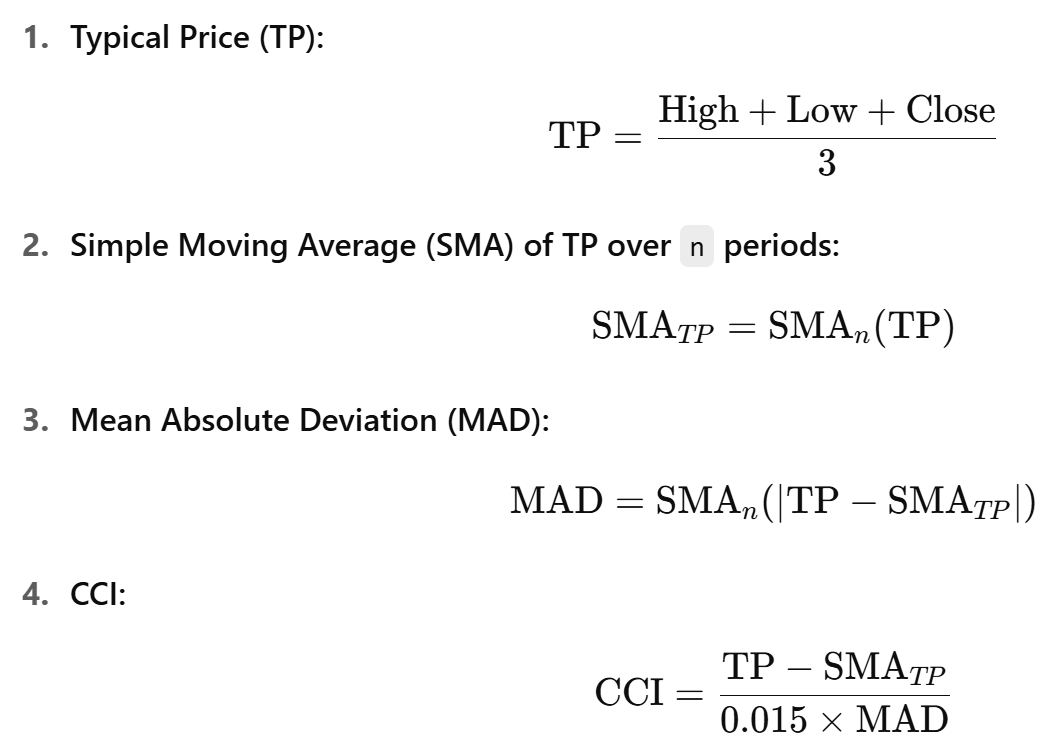






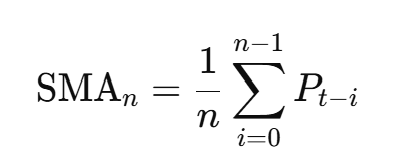
Commodity Channel Index (CCI):

The CCI measures how far a price is from its statistical average indicating overbought or oversold market conditions. [26]



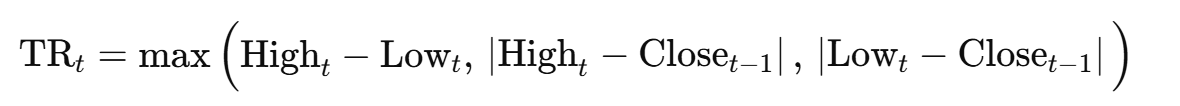
Moving Averages:

We add multiple simple moving averages with different window sizes. Moving averages smooth out price data and indicate an upwards trend when increasing and a downwards trend when decreasing. Besides that they can indicate overbought or oversold conditions. [27]



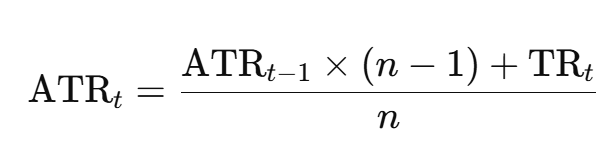
Average True Range (ATR):

The ATR is a volatility indicator that tells how much the price moves on average per period. We therefore first look at the True Range (TR) for every period. This is the maximum price difference within that period. We add that difference to our feature set.



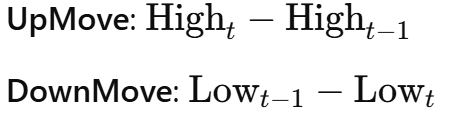
We use the previous day’s close price because the current day’s open price can differ from the previous day’s close price due to overnight news that led to changes in the sentiment. [30]

After that we use wilder’s smoothing to smooth out the True Range over a certain windows size, usually 14, so that we understand how volatile certain periods are. [31]

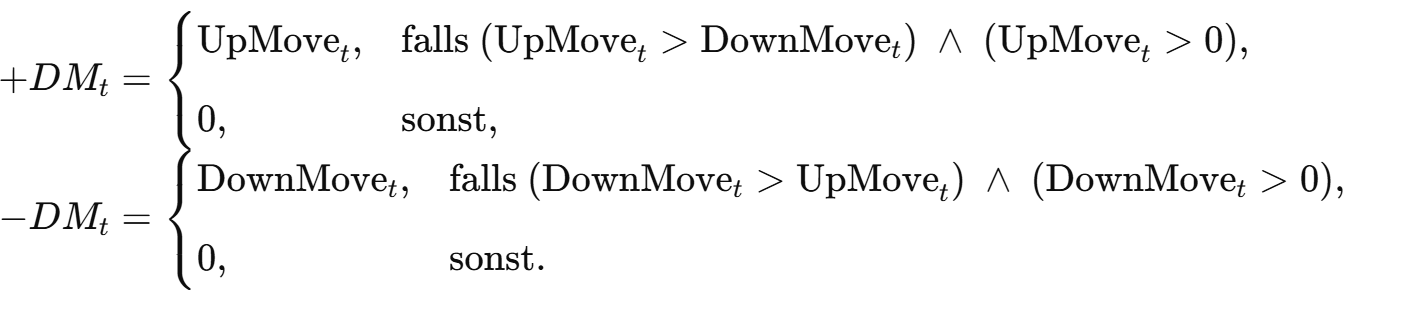


Average Directional Index (ADI):

The ADI uses the ATR as a component. The ADI and its components measure how much of the Average movement within a certain window which we derive from the ATR is upwards movement and how much is downwards movement. For that we calculate our up and downwards movement like this. [32]



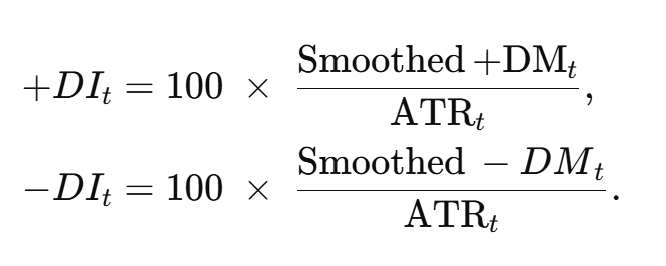
UpMove contains the difference of the high values so that increases have a positive value. DownMove contains the difference of low values in a way that downward movements have positive values and upwards movements have negative values.



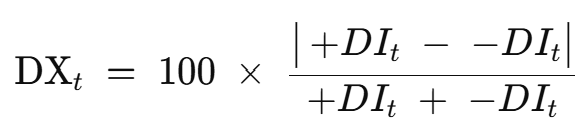
+DM now stores all absolute increases in the high value for periods in which the low value decreased less than the high value increased.

-DM works vice versa.

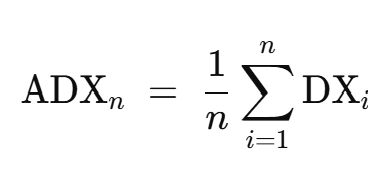
We now average these 2 values over a window that must have the same size as the window size n from our ATR. Like that we get the average up and down movement in absolute values for the given window. We calculate the percent to which our over all movement which we derive from the ATR is upwards movement or downwards movement.



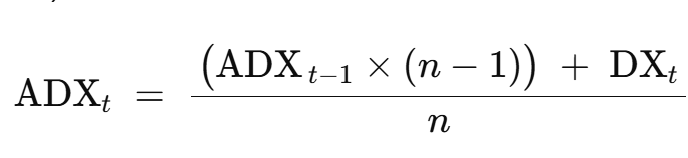
The next step lets us calculate the Directional Movement Index (DX). DX tells us in percent how one-sided a movement is. If up and down movement have the same portion of the overall movement then DX is 0%.



The last step is to calculate the actual ADX. We again use wilder’s smoothing. That is a recursive function so we need to calculate the first ADX value manually by simply averaging the previous n DX values.



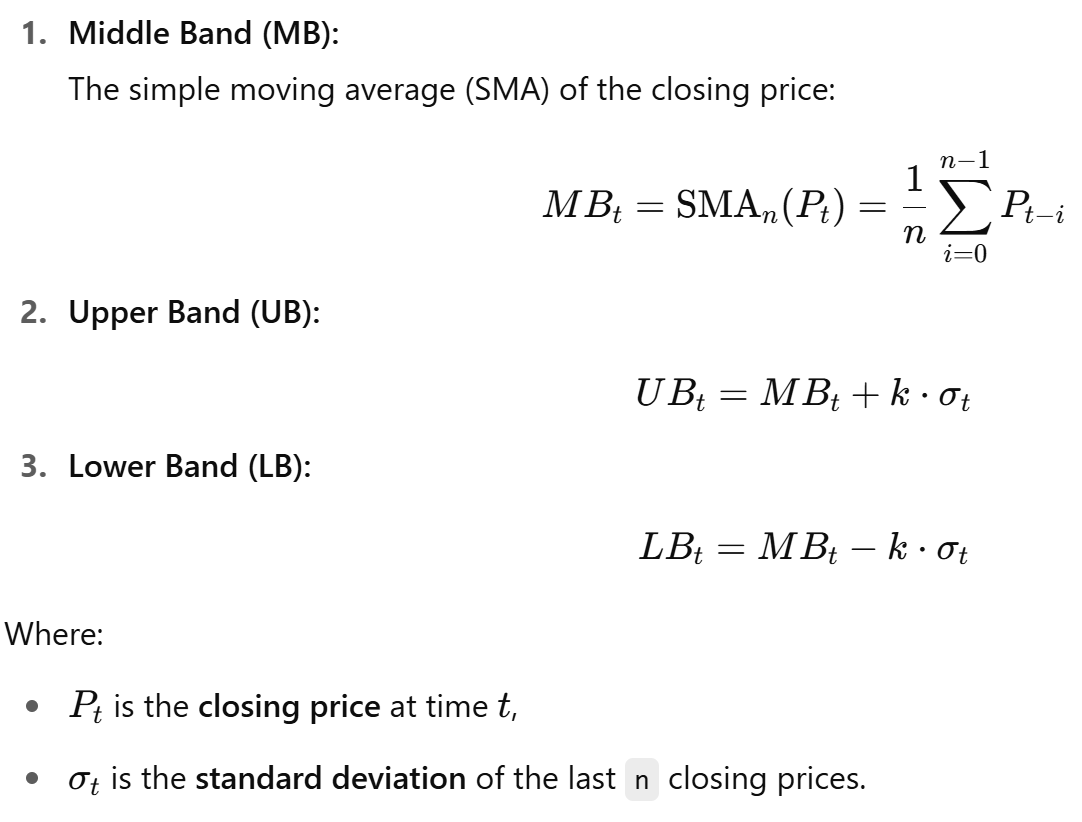
Any further ADX value is calculated recursively.



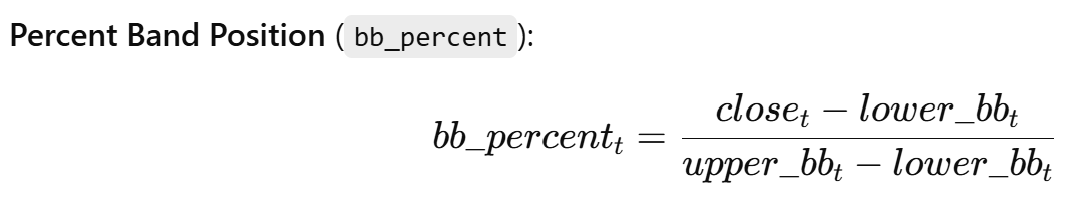
Bollinger Bands:

The Bollinger Bands are indicators to gauge the volatility of a stock to understand whether they are over or undervalued. [33]

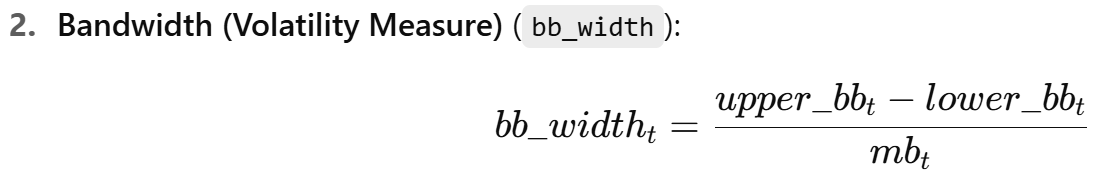
They consist of 3 lines:



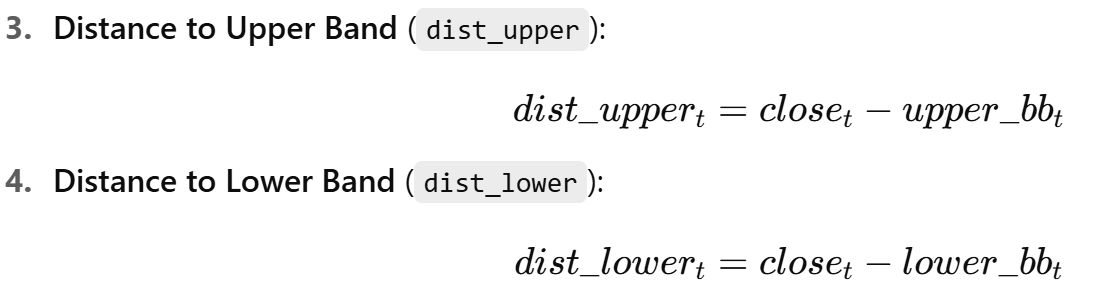
Since we use this indicator for machine learning purposes, we will add the following features to emphasize relations between the bands or our close price and the bands.



* Measures from 0 to 1 where the close price is within lower and upper band

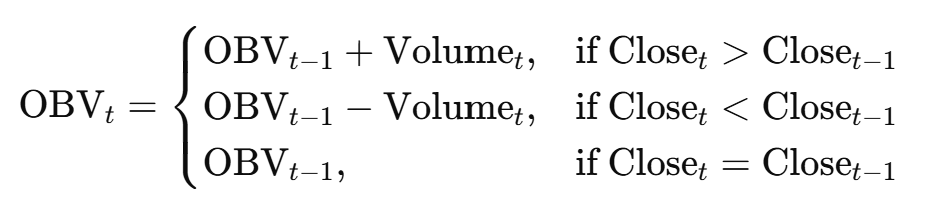


* The band width tells how volatile the course is.

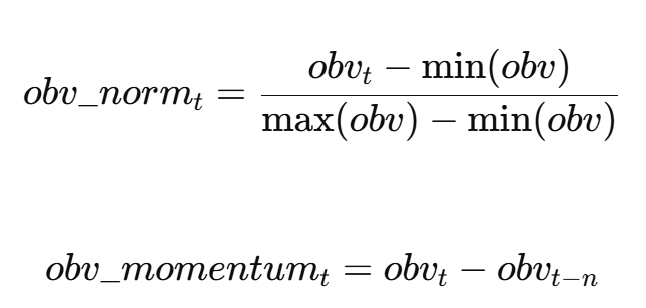


On-Balance Volume (OBV):

The On-Balance Volume Indicator is a momentum indicator that uses volume flow to predict price movement. It assumes that volume precedes the price – so a rising OBV signals buying pressure and a falling OBV signals selling pressure. [34]

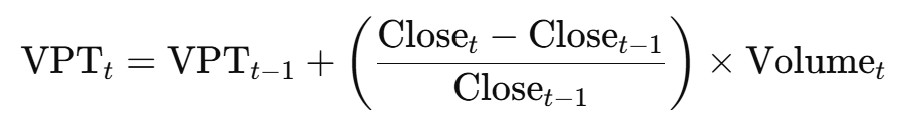


Besides the absolute OBV we add a normalized OBV for better machine learning suitability and an OBV momentum that compares the OBV to the OBV n periods before.

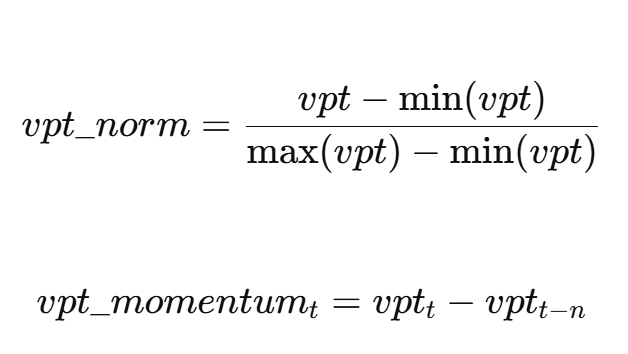


Volume Price Trend (VPT):

The VPT is a momentum indicator that combines the price change with volume. Its purpose is to track how strongly volume aligns with or contradicts price changes. [35]



For Machine Learning purposes we also add the VPT in a normalized way and the VPT momentum.



Macro-economic features:

We add Macro-economic features to capture global or US wide circumstances and market context [38]. We use the “FRED API” version 0.5.2 to retrieve them [37]. Some of the indicators are not published daily. For example, the GDP is published quarterly. We therefore use the interpolate function that comes with the pandas library. It looks for brackets of NaN values and draws a line from the valid value before and the valid value after the bracket to fill the rows in between. So, if we have a list such as [10, NaN, NaN, NaN, 18, NaN, 26] then the interpolation of this list would result in this list: [10, 12, 14, 16, 18, 22, 26]. This allows us to simulate a realistic and smooth course of the features.

S&P 500:

One important macro-economic indicator is the S&P500 index. Since it includes the biggest 500 US companies it is an important measure of the health of the overall us economy.

GDP(USA):

The Gross domestic Product is the value of all good and services produced by a country in a given period. It is a great indicator since it reflects the economic performance of a country [38].

Unemployment rate:

The unemployment rate can capture strengths and weaknesses in the economy. It is a great indicator for economic activity [38].

Consumer price index (CPI):

The consumer price index measures the monthly change in prices paid by urban US customers for different basic products and services [39].

Personal consumption expenditures (PCE):

The personal consumption expenditures is an indicator that measures how much US households spend on goods and services [41].

Industrial production index (IPI):

The industrial production index measures the output in manufacturing, gas, oil, mining and electric industries [42].

federal funds rate:

The federal fund rate is the target interest rate range at which commercial banks are supposed to borrow money [43].

10Y treasury rate:

The 10-year treasury is the interest rate which the US government pays to borrow money over a decade. It is an important benchmark for other interest rates and it tells about expectations of the US government about inflation and economic growth [44].

Retail sales:

Retail sales is a main indicator for economic health because it provides information about the buying power of consumers and bigger retail sales usually lead to more profit for companies which has a positive effect on the stock price [40].

Housing starts:

Housing starts is the number of newly started construction projects in a given period that provides information about the housing market and general economic activity [45].

Exchange rates:

Exchange rates are a good way to measure the economic health in comparison to economies from other countries [46] [47]. We include the exchange rates USD-EUR, USD-GBP and USD-JPY.

Calendar Features:

We also add the following calendar features to capture patterns for example the Monday effect [28] and the January effect [29].

* Day of week (1-7)
* Day of year (1-365)
* Month (1-12)
* Year (yyyy)
* Quarter (1-4)

In total we now have included 60 features in our data set that are products or byproducts of our calculations.

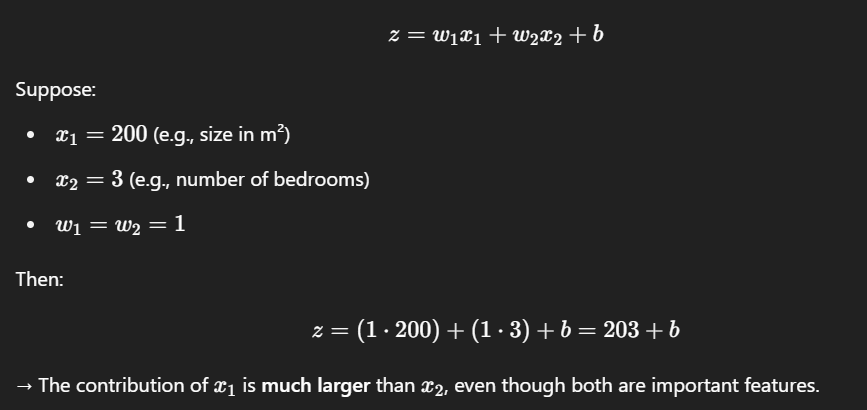
We also already add a ‘target’ column with the next days close price for training.

We continue with performing further preprocessing steps, but we have to perform them in separate pipelines since the LSTM and the CNN model need distinct preprocessing steps compared to our XGBoost model.

Further Pipeline for LSTM and CNN:

One important step we need to perform on data for neural networks is normalization. This is due to the nature of neural networks – they are sensitive to the scale of input features. Without normalization features with large values would dominate features with small values which would lead to uneven learning. We look at an example to understand that dynamic. Neural networks as we learned before work by taking vectorized inputs. That means they take a list of numerical values as input. Each of these values is multiplied by the according weight and then typically added together.

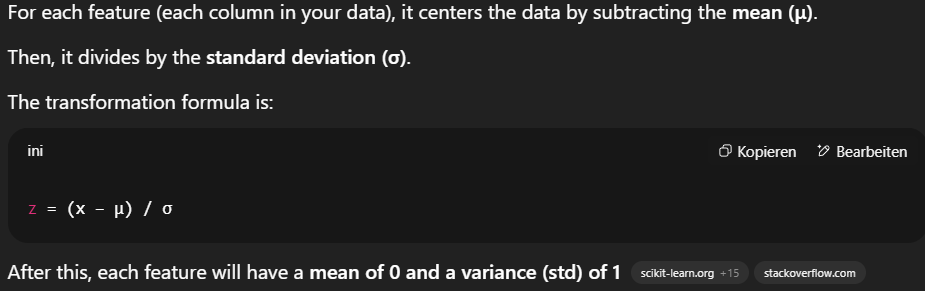
Let’s say we have the following equation for one neuron and we assume that features x1 and x2 are equally important to the result:



As we can see. The contribution of x1 to the result is much larger even though we know that both features are equally important, it’s just that x1 has a larger scale. One might argue that the network could compensate this larger scale by using a much smaller weight for input feature x1, but then we are interfering with the nature of the neural network which really is to weigh particular features heavier because they have been observed to have a greater impact on the result. This is not the case for us because we assume that our features both have an equal impact on the result. If we already have different scales for our inputs at the beginning of the network, it would be as if we predefine an importance for each feature. In case we know that particular features are in fact more important than others and we know the relation of this importance among the features, than the practice to weigh the more impactful ones heavier would make sense, but usually the network is unbeatable in figuring out the individual importance of each feature or combination of features. Now it is totally true that you could simply have much smaller weights for those much larger inputs. But the problem with that lies in the nature of how the network learns its weights. They are computed based on a fixed learning rate which is an absolute value for the whole network and for the full training process. The learning rate tells us how much a weight is updated. If the learning rate is very small it means that we can converge our weights to very precise values with a fine granularity but it also means that we need more iterations to get to this correct value because we are doing much smaller steps. So, if the weights are forced to lie within different ranges because our input features have different scales, then the learning process within the whole network would be asymmetrical and some weights would learn very slow and barely reach a good value while others constantly overshoot and never converge to an accurate enough number.

This is why normalization is needed.

A scaler scales every input feature so that all have the same scale. We use the StandardScaler() from the sklearn.preprocessing library in python. [49]



Like that the mean is 0 and 68% of the values lie within -1 and 1.

To apply the scaler on our features we first have to fit it or in other words make it learn the scale by which it has to recalculate our values. If we would fit it on the whole data set then information about the scale of our validation and test data sets would leak through into the scaler. So, to maintain a clean pipeline we need to first split our data into training, validation and test sets, then fit the scaler only on the training set and after that apply it on all data sets. The target variable hereby is the only feature that doesn’t need to be scaled. In fact, it’s much easier to not scale it because we would otherwise need to revert the scaling for our predictions.

The normalization or scaling is actually the only preprocessing step we are undertaking here. Another important step for neural networks for time series data is windowing. Windowing allows us to have a series of past x periods as input. We want to optimize the window size as a hyperparameter so we decide to implement the windowing under section hyperparameter tuning.

Further pipeline for XGBoost:

The XGBoost model requires a different preprocessing pipeline. As we learned earlier the input for a tree-based model looks different to the input of our LSTM or CNN model. For our XGBoost model the input data is not a collection of windows that have the shape (n\_periods x n\_features) but instead it’s a collection of single row features or 1D vectors that contain all features. That means we can’t simply use past periods as input. The way to work around that and still use XGBoost models for time series forecasting is to add lag values. Lag values always belong to a specific feature and they invoke the value of a past period of that feature. So, a lag value of 3 for the feature “close” would mean that we add 3 columns as new features where one is the “close” price 3 days ago, one is the “close” price 2 days ago and the third one is the “close” price from the previous day. Like that we can add different lag sizes for each feature and in that way add the necessary temporal context for our current period. Our concrete approach is documented in the codebase.

As we learned before XGBoost is a tree-based model and tree-based models only care for the proportions of the data matter but not for their absolute value. When we normalize, the proportions of the values within one feature column don’t change but only their absolute value gets resized. As we learned

### Functionality of Tree based models and why normalization is not needed.

Since we have no windowing to perform on the data for the XGBoost model, the last preprocessing step is to split the data into train, validation and test set.

Hyperparameter tuning:

When we looked into the different Machine Learning models that we are using, we understood what adjustable hyperparameters there are. We will tune these hyperparameters for each combination of model and data set separately so that we conduct our experiment based on optimal performance.

Optuna:

To do so we use the optuna library in python, which has a very straight forward workflow. It allows us to define a function that returns a float value and a direction in which we want to optimize that returned value. The directions can be to minimize or to maximize.   
In the function which’s return value is to be optimized ranges for variables can be specified. The function later will be executed a number of times that is predefined and during every iteration the values from the specified ranges for the variables will be selected. The way these values are selected depends on the algorithm that is specified for the hyperparameter suggestion. It can simply suggest random combinations of hyperparameters or it chooses them based on prior performance.  
The function is executed multiple times with different variable values. The outer object which is called study and which executes the function stores the returned value and hyperparameter combination for each iteration and especially for the one iteration that led to the best result. That’s the general workflow of the optuna optimization library.

Our optuna use case:

In our case the function which’s return value we want to optimize is defined to be the training and evaluation step for each model-stock combination. The variables that the study will pick values for are our hyperparameters. The returned value is the forecast error and the optimization direction is to minimize. Like that the study will look for the combination of hyperparameters that lead to the smallest error – exactly what we are looking for.

In our case the study object takes 4 parameters.

* # 5.2. CNN study
* study = optuna.create\_study(
* direction="minimize",
* study\_name="cnn\_regression\_study",
* sampler=optuna.samplers.TPESampler(),
* pruner=optuna.pruners.MedianPruner(n\_startup\_trials=5, n\_warmup\_steps=10)
* )
* Direction … which defines whether the float value of the function should be minimized or maximized.
* Sampler … Which defines the algorithm that suggests new parameters.
* Pruner … which defines the mechanism to abort bad trials

Our sampler tries to pick combinations that have the highest probability to be another “good” guess. A “good” guess is a combination with a performance above a certain threshold and a “bad” guess is a combination with a performance below that threshold.

The pruner aborts a trial instantly when it foresees it to not be a good trial. After n\_startup trials it checks every trial whether after 10 reports of one trial this trial is better or worse than the median trial so far. If worse than the median trial, it prunes the current trial, if better it keeps going. Like that computational efficiency is ensured and the search space is further focused on promising combinations.

Depending on the ML algorithm that we use, we need to tune different hyperparameters.

LSTM:

* N\_units
  + The number of units (neurons) in each LSTM layer which determines the size of the safe state vectors and the model’s capacity to learn patterns over time
* N\_layers
  + The number of stacked LSTM layer. More may facilitate capturing complex patterns while they risk overfitting. Fewer layers can be better in generalizing but may be less accurate
* Dropout\_rate
  + Fraction of total units that is randomly dropped during training to prevent overfitting. The approach is similar to feature randomness in random forests. We want that the model deals with the data when the architecture is slightly altered. Like that we increase the model’s robustness.
* Learning rate
  + Controls the degree to which the model updates its weights after each batch. Smaller values lead to slower more stable learning while large values speed up learning but may overshoot.
* Batch size
  + Number of samples processed before the model updates its weights. Smaller sizes give more updates but noisier gradients while larger values lead to a more stable learning with less volatility in the loss function but require more memory.
* Dense units
  + Number of neurons in the final dense (fully connected) layer. The determine the capacity of the model to combine learned patterns into predictions or classifications.
* Window size
  + The size of the input window. In other words, it’s the number of previous time steps you feed into the network to predict the next value. A windows size of 20 means that you feed the past 20 periods into the network.

CNN:

* N filters
  + The number of convolutional filters in each convolutional layer. They determine the channel dimension of the input for the following layer. More filters can capture more features but increase model complexity and risk overfitting.
* N layers
  + The number of convolutional layers followed by pooling layers in the CNN. More layers enable the model to capture more complex features but increase training time and risk overfitting
* kernel size:
  + The size of the filter that slides over the input. It determines the size of the temporal context that is captured to identify a pattern.
* Pool size
  + The size of the max pooling window that down samples the feature maps to reduce complexity and increase efficiency. A greater pool size reduces the complexity more.
* Dense units
  + The number of neurons in the fully connected layer after convolutional and pooling layers. It determines the model’s capacity to form predictions out of the extracted features.
* Dropout rate
  + The fraction of neurons in the fully connected layers randomly dropped during training to increase the model’s robustness and prevent overfitting.
* Learning rate
  + Controls the degree to which the model updates its weights after each batch. Smaller values lead to slower more stable learning while large values speed up learning but may overshoot.
* Batch size
  + Number of samples processed before the model updates its weights. Smaller sizes give more updates but noisier gradients while larger values lead to a more stable learning with less volatility in the loss function but require more memory.
* Window size
  + The size of the input window. In other words, it’s the number of previous time steps you feed into the network to predict the next value. A windows size of 20 means that you feed the past 20 periods into the network.

XGBoost:

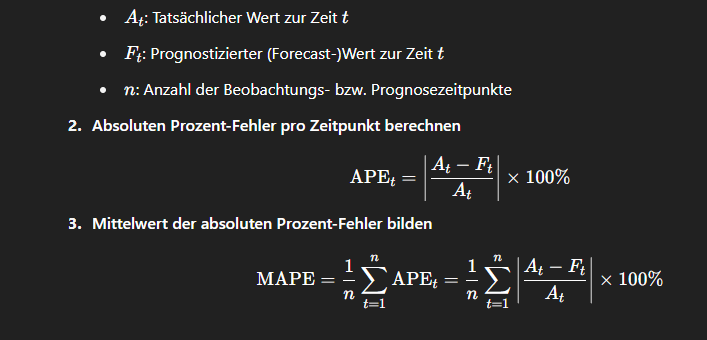
* Max depth
  + The maximum depth of the decision trees. Deeper trees can learn more complex patterns but may overfit.
* Alpha
  + A regularization term on weights that gives a penalty for the absolute value of a weight. It gives an incentive for the trees to make smaller weights. This encourages sparsity which lets the model generalize better
* Lambda
  + A regularization term on weights that gives a penalty for the square of the value. Like that it punishes larger weights more than smaller weights and encourages the tree to have smoother and smaller weights. It can reduce overfitting and prevent single trees to make overly confident predictions.
* Eta
  + The learning rate that determines the degree to which one tree contributes to the final prediction. A lower value means one tree contributes less to the final prediction. More trees would be needed which slows down training but improves generalization.
* Gamma
  + The minimum loss reduction that is required to perform a split. It forces the model to only perform helpful splits and like that reduces the size of the model and helps against overfitting.
* Subsample
  + The fraction of the training data randomly sampled for growing each tree. We earlier read about Bootstrapping to make the trees see different subsets of the data. This value decides how big the fraction from the random subset of the total data set is. It is important for generalization.
* Colsample\_bytree
  + Fraction of features randomly sampled for constructing different trees. This is the “feature randomness” we talked about earlier. Having our trees each considering a slightly different subset of the features helps the model to generalize and prevents overfitting.

Metrices:

When we compare the training and evaluation process of our three models, we use different metrices. The two main things we want to look at in this section is metrices to measure the predictive performance of the models and the computational efficiency of the training process. We start with the performance metric.

One of the most used metrices to assess a prediction for stock prices is the Mean Absolute Percentage Error (MAPE). It is well suited for our use case because it measures the error in percentage which makes it very comparable across different stocks because it accounts for the different scales of stock prices. Let’s say we predict a price of 110$ and the actual price is 100$. We have a percentage error of 10% and an absolute error of 10. If we predict 210$ and the actual price is 200$, then we have a percentage error of 5% and an absolute error of 10. So, while the absolute error indicates a similar performance does the percentage error tell us that our second prediction is simply more accurate since we are closer to the actual price when we account for the scale.

MAPE: [52]



Besides the accuracy of the predictions, we are interested in the computational efficiency of the training and evaluation process. Therefore, we measured the time the models took to run the full training and evaluation pipeline on our computer. We didn’t perform any other task on the computer while training which allows for a realistic measurement but background processes might still have interfered with the execution times a bit.

Results & Discussion:

Before taking a longer shot at the predictive accuracy and the resulting data we first want to look at the execution times of the three models.

Execution times:

* XGBoost: 59.455 minutes
* LSTM: 1887.202minutes
* CNN: 280.319 minutes

The system we used:

* CPU: 13th Gen Intel(R) Core (TM) i5-13400F 2.50 GHz
* RAM: 16,0 GB
* GPU: no GPU
* Operating System: Windows 11, x86-64bit
* Storage: 1 TB SSD NVMe-M.2-Drives
* Motherboard: Gigabyte B760 Gaming X DDR4 (rev. 1.0) ATX

When we look at that the XGBoost model clearly is the fastest one with only taking about one hour to perform all the training and evaluation steps. Second is the CNN model that took about four hours and 40 minutes. The by far slowest model is the LSTM mode. It took a surprising time of about 31 hours and 27 minutes to perform the training and evaluation. We can clearly see that the XGBoost model runs incredibly faster with about an astonishing 31.5x speed-up over the LSTM model and a 4.7x speed-up over the CNN. That is definitely a relevant difference.

Future Research:

* Add other variables as hyperparameters and tune them as well, like lag sizes for xgboost

1. Ghazaleh Babanejaddehaki, Aijun An, and Manos Papagelis. 2025. Disease Outbreak Detection and Forecasting: A Review of Methods and Data Sources. ACM Trans. Comput. Healthcare 6, 2, Article 13 (April 2025), 40 pages. https://doi.org/10.1145/3708549
2. Qureshi, M., Arbab, M.A. & Rehman, S. Deep learning-based forecasting of electricity consumption. *Sci Rep* **14**, 6489 (2024). https://doi.org/10.1038/s41598-024-56602-4
3. Venkatesan, S., Lim, J., Ko, H., & Cho, Y. (2022). A Machine Learning Based Model for Energy Usage Peak Prediction in Smart Farms. *Electronics*, *11*(2), 218. https://doi.org/10.3390/electronics11020218
4. https://www.ft.com/content/78d1314b-2879-40cc-bb87-ffad72c8a0f4
5. Mahinda Mailagaha Kumbure, Christoph Lohrmann, Pasi Luukka, Jari Porras, Machine learning techniques and data for stock market forecasting: A literature review, Expert Systems with Applications, Volume 197, 2022, 116659, ISSN 0957-4174, <https://doi.org/10.1016/j.eswa.2022.116659>
6. T. Matharasi, S. Lakkaraju and I. Mukherjee, "A Comparative Analysis of Machine Learning Approaches for Stock Price Prediction," *2025 International Conference on Pervasive Computational Technologies (ICPCT)*, Greater Noida, India, 2025, pp. 975-979, doi: 10.1109/ICPCT64145.2025.10941156
7. Li, Siyuan. (2024). Comparison of Machine Learning Models for Stock Prediction. Advances in Economics, Management and Political Sciences. 85. 76-84. 10.54254/2754-1169/85/20240846.
8. Drucker, H., Burges, C. J., Kaufman, L., Smola, A., & Vapnik, V. (1996). Support vector regression machines. *Advances in neural information processing systems*, *9*.
9. Cortes, C., Vapnik, V. Support-vector networks. *Mach Learn* **20**, 273–297 (1995). https://doi.org/10.1007/BF00994018
10. Aiyegbeni Gifty, & Dr. Yang Li. (2024). A Comparative Analysis of LSTM, ARIMA, XGBoost Algorithms in Predicting Stock Price Direction. *Engineering And Technology Journal*, *9*(8), 4978–4986. https://doi.org/10.47191/etj/v9i08.50
11. Gifty, A., & Li, Y. (2024). A Comparative Analysis of LSTM, ARIMA, XGBoost Algorithms in Predicting Stock Price Direction. *Engineering and Technology Journal*, *9*(8), 4978-4986
12. Gururaj, V., Shriya, V. R., & Ashwini, K. (2019). Stock market prediction using linear regression and support vector machines. *Int J Appl Eng Res*, *14*(8), 1931-1934.
13. Teixeira, D. M., & Barbosa, R. S. (2025). Stock Price Prediction in the Financial Market Using Machine Learning Models. *Computation*, *13*(1), 3. <https://doi.org/10.3390/computation13010003>
14. Yao, Y. (2025). Stock Price Prediction Using an Improved Transformer Model: Capturing Temporal Dependencies and Multi-Dimensional Features. *Journal of Computer Science and Software Applications*, *5*(2). <https://doi.org/10.5281/zenodo.14832392>
15. <https://www.investopedia.com/terms/v/volatility.asp>
16. <https://www.macroption.com/historical-volatility-calculation/>
17. <https://medium.com/%40manojkotary/simple-returns-vs-log-returns-a-comprehensive-comparative-analysis-for-financial-analysis-702403693bad>
18. <https://alpaca.markets/deprecated/docs/api-documentation/api-v2/assets/?utm_source=chatgpt.com#asset-entity>
19. <https://alpaca.markets/support/usage-limit-api-calls>
20. <https://en.wikipedia.org/wiki/Template%3ATechnical_analysis>
21. <https://www.investopedia.com/terms/t/technicalindicator.asp>
22. <https://www.investopedia.com/terms/r/rsi.asp>
23. <https://www.investopedia.com/terms/p/pricerateofchange.asp>
24. <https://www.investopedia.com/terms/m/macd.asp>
25. <https://www.investopedia.com/terms/s/stochasticoscillator.asp>
26. <https://www.boerse.de/technische-indikatoren/Commodity-Channel-Index-(CCI)-8>
27. <https://www.investopedia.com/terms/m/movingaverage.asp>
28. <https://www.investopedia.com/terms/m/mondayeffect.asp>
29. <https://www.americancentury.com/insights/the-january-effect-and-stock-market-seasonality/>
30. <https://support.zerodha.com/category/trading-and-markets/trading-faqs/general/articles/opening-price-of-a-stock-different-from-its-previous-day-s-closing-price>
31. <https://www.investopedia.com/terms/a/atr.asp>
32. <https://www.investopedia.com/articles/trading/07/adx-trend-indicator.asp>
33. <https://www.investopedia.com/terms/b/bollingerbands.asp>
34. <https://www.investopedia.com/terms/o/onbalancevolume.asp>
35. <https://www.boerse.de/technische-indikatoren/Volume-Price-Trend-(VPT)-56>
36. <https://www.investopedia.com/terms/v/vwap.asp>
37. <https://fred.stlouisfed.org/docs/api/fred/>
38. <https://www.investopedia.com/articles/investing/031413/economic-indicatiors-affect-us-stock-market.asp>
39. <https://www.investopedia.com/terms/c/consumerpriceindex.asp>
40. <https://www.investopedia.com/terms/r/retail-sales.asp>
41. <https://www.investopedia.com/terms/p/pce.asp>
42. <https://www.investopedia.com/terms/i/ipi.asp>
43. <https://www.investopedia.com/terms/f/federalfundsrate.asp>
44. <https://www.investopedia.com/articles/investing/100814/why-10-year-us-treasury-rates-matter.asp>
45. <https://www.investopedia.com/terms/h/housingstarts.asp>
46. <https://www.investopedia.com/articles/economics/08/leading-economic-indicators.asp>
47. <https://www.investopedia.com/terms/e/exchangerate.asp>
48. <https://www.ibm.com/think/topics/machine-learning>
49. <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
50. <https://dataaspirant.com/wp-content/uploads/2020/11/3-Recurrent-Neural-Network-1024x758.png>
51. <https://i.sstatic.net/ondvK.png>
52. <https://en.wikipedia.org/wiki/Mean_absolute_percentage_error>
53. <https://www.xoriant.com/blog/decision-trees-for-classification-a-machine-learning-algorithm>
54. <https://www.ibm.com/de-de/think/topics/decision-trees#:~:text=Ein%20Decision%20Tree%20ist%20ein,als%20auch%20Regressionsaufgaben%20verwendet%20wird>.
55. <https://www.ibm.com/de-de/think/topics/random-forest>
56. <https://www.ibm.com/de-de/think/topics/xgboost>
57. <https://www.researchgate.net/figure/The-structure-of-the-LSTM-unit_fig1_347803411>

Words to explain:

Average  
Exponentially weighted moving averages (EWMA or EMA) with formula  
Simple moving average (SMA) with formula  
period. I would say when you read the whole thing you just write down every word you need to explain.

* A system updates itself (same as third, learning)
* Error (in prediction)
* To learn (when a system updates itself and learns from error)
* Data point
* Period
* Data sets (big tables of periods x features)
* Adding a feature (equals adding a column with that value)
* Classification
* Regression
* generalization