University of Economics in Prague

Faculty of Finance and Accounting

Department of Banking and Insurance

MASTER THESIS

Application of time series models and machine learning methods for stock returns prediction

Author: Oleksandr Vodolazskyi

Supervisor: Ing. Milan Fičura

Academic Year: 2018

Declaration of Authorship

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

Prague, _____ _____

Signature

Acknowledgements

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

Abstract

In this diploma thesis we applied time series models and machine learning methods on real stock data to predict future returns movements. The time series models include ARMA and GARCH models. We also used several machine learning algorithms such as logistic regression, gradient boosted trees and Long Short-Term Memory neural networks. The models were used to predict future daily returns movements. Then strategies based on predictions were evaluated by their profitability and riskiness. Also we measured the quality of the models from the point of accuracy and discrimination power. The main goal of the thesis is to compare results of the models application with the Buy & Hold strategy and between each other.

Abstrakt

V této diplomové práci jsme aplikovali modely časových řad a metody strojového učení na reálná akciová data pro predikci pohybů budoucích výnosů. Mezi modely časových řad patří modely ARMA a GARCH. Také jsme požili několik algoritmů strojového učení, jako je logistická regrese, gradient boosted trees a takzvané Long Short-Term Memory neuronové sítě. Tyto modely byly použité pro predikci pohybů budoucích výnosů akcí. Strategie založené na předpovědích byly vyhodnoceny podle jejich ziskovosti a rizikovosti. Také jsme změřili kvalitu modelů z hlediska přesnosti a diskriminační schopnosti. Hlavním cílem práce je porovnání výsledků aplikace zvolených modelů se strategií Buy & Hold a mezi sebou.

Contents

Introduction

The ability to predict the movement of the stock market is what every investor or trader would like to possess. Unfortunately, this is one of the most difficult tasks in financial world. There are mane approaches how to make investment and trading decisions. Among them we can distinguish time series models which try to find the stochastic process describing the behaviour of stock prices or returns. Time series models have been used for many years for modeling stock returns.

In recent years machine learning methods are gaining the popularity as an effective tool for making predictions in different areas. One of these areas is the stock market where machine learning algorithms can learn to predict its future behaviour.

The main goal of the thesis is to apply time series models and machine learning methods for prediction futures stock returns and to examine how effective they are in this task. We want to measure the performance of the strategies based on different models and compare them between each other and with the simple Buy & Hold strategy. Our aim is to verify whether the time series models and machine learning methods are able to beat the market systematically. Obtained results of our research can help to understand the potential of the models and to identify directions of further exploration.

The thesis is divided into four chapters. In the first chapter we describe different approaches for calculation stock returns and explain why logarithmic returns is the most common type. In the second chapter is devoted to theoretical basis of time series models such as ARMA and GARCH. In the third chapter we introduce basic principles of machine learning and tasks which it is able to solve. Also we describe theoretical background of several machine learning methods such as logistic regression, gradient boosted trees and Long Short-Term Memory neural networks. The last chapter is devoted to the results of application of time series models and machine learning methods for stock returns prediction.

Chapter 1

Stock Returns

In Chapter 1 we present the basic principles of stock returns calculation. The Chapter is divided into three sections where we describe gross returns, net returns and log returns.

1.1 Gross returns

The goal of trading on stock markets is to make a profit. The amount of a profit or a loss from a trading strategy depends on changes in prices of stocks and the amount of stocks being traded. Hence traders are rather interested in a relative measure of their profits in order to estimate how well a certain strategy performs. Returns are able to provide this measure because they are expressed as a relation of changes in price to the initial price.

Campbell, Lo, and MacKinlay (1997) give two main reasons for using returns. First, for average investors, return of an asset is a complete and scale-free summary of the investment opportunity. Second, return series are easier to handle than price series because the former have more attractive statistical properties, such as stationarity and ergodicity.

Let P_t be the price of a stock at time index *t*. For the sake of simplicity, we assume that a stock pays no dividends.

For one period from date *t - 1* to date *t* the *simple gross return* can be calculated as:

$$\frac{P_t}{P_{t-1}} = 1 + R_t \tag{2.1}$$

Returns are scale-free, meaning that they do not depend on units (dollars, cents, etc.).

The gross return over the *k* periods is the product of the *k* single-period gross returns (from time *t - k* to time *t*):

$$\frac{P_t}{P_{t-k}} = \frac{P_t}{P_{t-1}} \times \frac{P_{t-1}}{P_{t-2}} \times \ldots \times \frac{P_{t-k+1}}{P_{t-k}} = 1 + R_t(k) = (1 + R_t)(1 + R_{t-1}) \ldots (1 + R_{t-k}) \tag{2.2}$$

Thus, the *k*-period simple gross return is just the product of the *k* one-period simple gross returns. This is called a compound or cumulative return.

1.2 Net returns

Net return over the period from time *t - 1* to time *t* is:

$$R_t = \frac{P_t}{P_{t-1}} - 1 = \frac{P_t - P_{t-1}}{P_{t-1}} \tag{2.3}$$

P_t - P_{t-1} in the numerator is the revenue over the period from time *t - 1* to time *t* and P_{t-1} is the initial price of a stock. Therefore, the net return can be considered as the relative profit.

1.3 Log returns

Log returns, also called *continuously compounded returns*, are denoted by r_t can be obtained by taking the natural logarithm of the simple gross return of a stock:

$$r_t = ln(1 + R_t) = ln\left(\frac{P_t}{P_{t-1}}\right) = p_t - p_{t-1} \tag{2.4}$$

where $p_t = \ln(P_t)$ is called the *log price.*

There several advantages of using log returns.

First, if we assume that prices are distributed log normally, then $ln(1 + r_t)$ is normally distributed.

Second, small values of log return can be approximately interpreted as the simple net return:

$$r_t \approx R_t, \qquad R_t \ll 1$$

The difference between the functions of net and log returns is showed on the following figure.

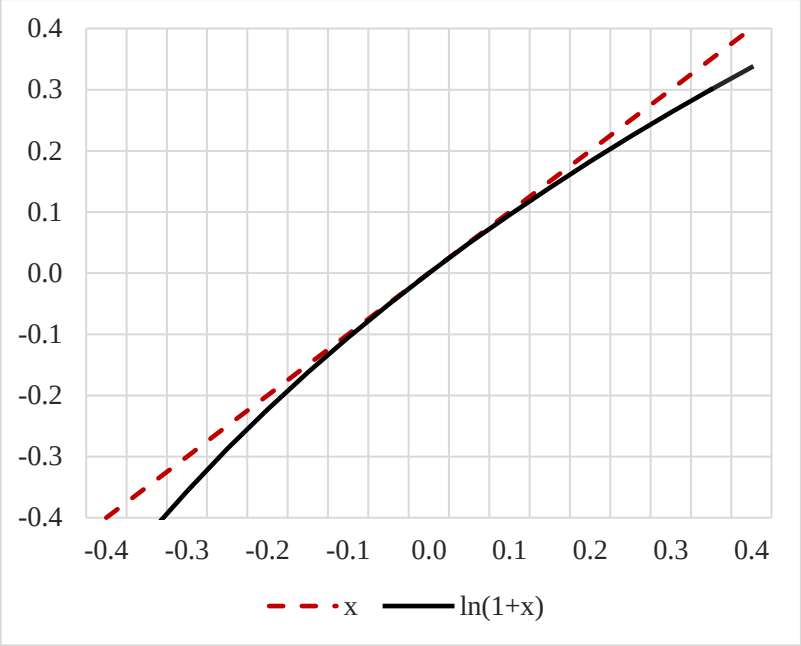

Figure 1.1: Net and log returns (Author's own work)

As can be seen from the graph, for large values the log function gives us smaller returns than simple net. If simple net return can be interpreted as the percentage change of the price, then log returns can also be approximately interpreted as the percentage change of the price.

Third advantage of using log returns is time-additivity. A *k*-period log return is simply the sum of the single-period log returns:

$$\begin{aligned} r_t(k) &= \ln\big(1 + R_t(k)\big) = \ln\big((1 + R_t)(1 + R_{t-1}) \dots (1 + R_{t-k+1})\big) \\ &= \ln(1 + R_t) + \ln(1 + R_{t-1}) + \cdots + \ln(1 + R_{t-k+1}) \\ &= r_t + r_{t-1} + \cdots + r_{t-k+1} \end{aligned}$$

Fourth advantage is numerical stability: addition of small numbers is numerically safer than multiplying small numbers. For many interesting problems, this is a serious potential problem. To solve this, either the algorithm must be modified to be numerically robust or it can be transformed into a numerically safe summation via logs.

Chapter 2

Time series models

Chapter 2 is devoted to the basic theory of time series models. Specifically, we describe main definitions used in time series analysis, present ARMA models for time series forecasting and GARCH models for volatility forecasting.

2.1 Stationarity and autocorrelation function

A *time series* is a chronologically ordered sequence of values of a variable at equally spaced time intervals. Considering log returns r_t of a stock as an ordered collection of random variables over time, we have a time series $\{r_t\}$.

One of the most important concepts in time series analysis is stationarity.

Definition 1. A time series $\{r_t\}$ is said to be *strictly stationary* if for each k, t, and n, the joint distribution of $(r_t, \ldots, r_{t+k})$ is identical to that of $(r_{t+n}, \ldots, r_{t+k+n})$.

Strict stationarity is a very strong assumption that is hard to verify empirically, because it requires that all aspects of behavior of a time series process are unchanged in time. Rather than strict stationarity the weaker form of it is usually assumed.

Definition 2. A time series $\{r_t\}$ is *weakly stationary* if the mean of r_t $\mathrm{E}(r_t)$ and the covariance between r_t and r_{t+k} $\mathrm{Cov}(r_t, r_{t+k})$ are constant and do not depend on t.

Henceforth, we will use the term stationary to mean weakly stationary. Usually stock prices are not stationary, but it is common to assume that stock returns are stationary.

The stationary means that statistical structure of the series is independent of time. It allows preserving model stability, i.e. the model which parameters and structure are stable in time. Stationarity matters because it provides a framework in which averaging (used in AR and MA processes that we will be described further) can be properly used to describe the time series behaviour.

Examples of non-stationary and stationary time series are showed on the following graphs.

Figure 2.1: Development of Google stock price, non-stationary process (Author's own work)

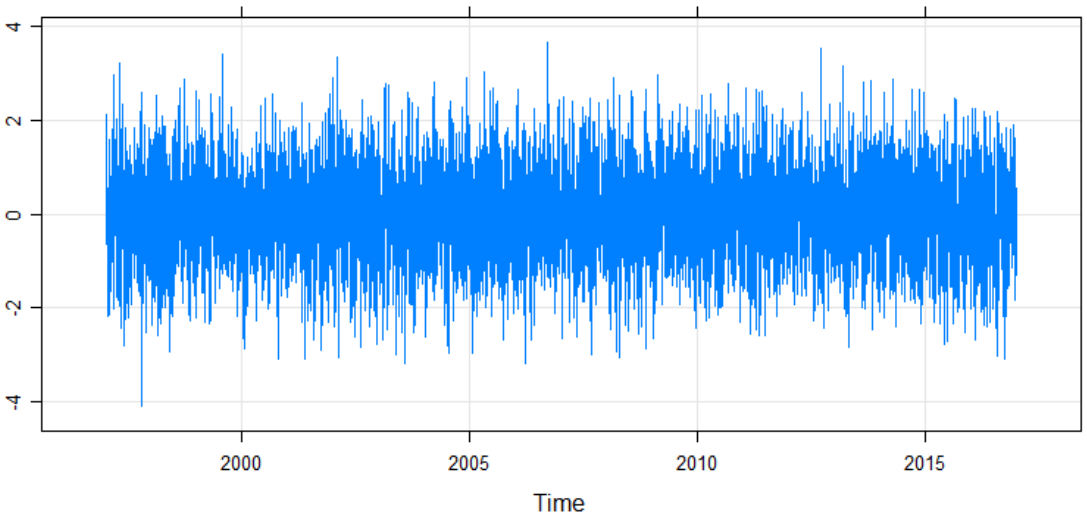

Figure 2.2: Randomly generated time series from a normal distribution, stationary process (Author's own work)

The covariance Cov(r_t, r_{t-k}) is called the lag-k *autocovariance* of r_t, which is in fact a covariance between realizations of the one variable in different points in time.

The autocovariance can be written as:

$$\gamma_k = Cov(r_t, r_{t-k}) = E[(r_t - \mu)(r_{t-k} - \mu)] \tag{3.1}$$

Autocovariance is closely related to autocorrelation of the time series process. The correlation coefficient between r_t and r_{t-k} is called the lag-k autocorrelation of r_t. and can be defined as

$$\rho_k = \frac{Cov(r_t, r_{t-k})}{\sqrt{Var(r_t)Var(r_{t-k})}} = \frac{Cov(r_t, r_{t-k})}{Var(r_t)} = \frac{\gamma_k}{\gamma_0} \tag{3.2}$$

The autocorrelation function (ACF) shows linear dependency between r_t and its past values. For any given stationary time series $\{r_t\}$ we can estimate sample autocovariance and autocorrelation functions. First, let's define the sample mean of r_t as $\bar{r} = \sum_{t=1}^{T} r_t/T$. Then the sample autocovariance function can be estimated as follows:

$$\hat{\gamma}_k = T^{-1} \sum_{i=1}^{T-k} (r_t - \bar{r})(r_{t-k} - \bar{r}) \tag{3.3}$$

Using the formula stated above we can estimate the sample autocorrelation function:

$$\hat{\rho}_k = \frac{\hat{\gamma}_k}{\hat{\gamma}_0} \tag{3.4}$$

Plotting the ACF can help to understand an autocorrelation structure of a time series. Usually the sample ACF is plotted with test bounds which are used for testing the null hypothesis that an autocorrelation coefficient is equal to 0.

For the purposes of demonstrations how ACF plot can be constructed we randomly generated 100 numbers from a normal distribution and plot ACF functions with test bounds with 5% significance level.

We can see in figure 2.3 that the autocorrelation coefficients for 20 lags lie in the test boundaries meaning that they are equal to 0 on the 5% significance level. We know this is true in that case because the data for the plot were generated from a normal distribution.

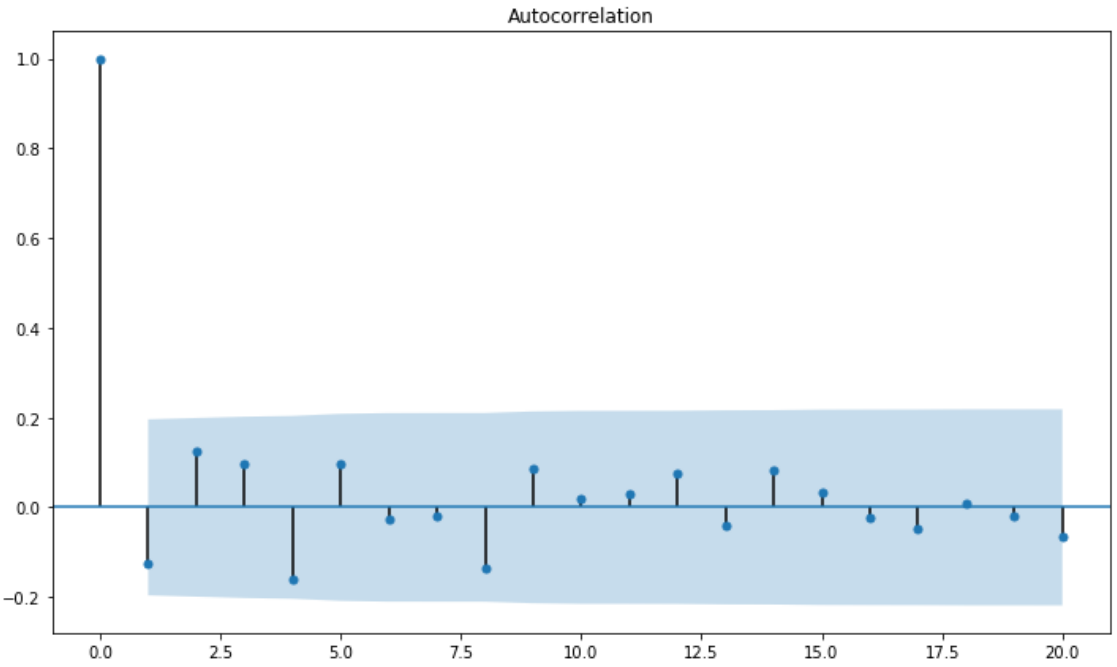

Figure 2.3: The sample ACF plot of a randomly generated sequence from a normal distribution (Author's own work)

An alternative way to test the autocorrelation coefficients is to use the Ljung-Box test. This test is used for testing whether all the autocorrelation coefficients are equal to 0 simultaneously. The null hypothesis states that the data are independently distributed (i.e. the autocorrelations are 0). The test is provided using the following statistic:

$$Q(m) = T(T+2)\sum_{k=1}^{m}\frac{\hat{\rho}_k{}^2}{T-k}$$

The null hypothesis H_0 is rejected if $Q(m) > \chi_\alpha^2$, where χ_α^2 denotes the $100(1-\alpha)$th percentile of a chi-squared distribution with m degrees of freedom.

2.2 Autoregressive models

Autoregressive (AR) models are based on the idea that the current value of the time series, r_t, can expressed as a function of k past values. A simple autoregressive model can be written in the following way:

$$r_t = \phi_0 + \phi_1 r_{t-1} + \varepsilon_t \tag{3.5}$$

where $\{\varepsilon_t\}$ is assumed to be a white noise series with mean zero and variance $\sigma_\alpha{}^2$. In the time series literature, model (3.5) is denoted as an autoregressive model of order 1 or an AR(1) model. The term *autoregression* refers to the regression of the series on its own past values (Tsay 2010).

In a similar way we can define a generalization of the AR(1) model which is the AR(p) model:

$$r_t = \phi_0 + \phi_1 r_{t-1} + \cdots + \phi_p r_{t-p} + \varepsilon_t = \phi_0 + \sum_{i=1}^{p} \phi_i r_{t-i} + \varepsilon_t \qquad (3.6)$$

The AR(p) model can be considered as a multiple linear regression model with lagged values serving as the explanatory variables. This understanding will help us in following chapters.

The autoregressive processes have, in general, infinite non-zero autocorrelation coefficients that decay with the lag. The AR processes have a relatively "long" memory, since the current value of a series is correlated with all previous ones, although with decreasing coefficients.

AR models have several properties that should be described. Consider AR(1) model and let's assume that the series is weakly stationary. Under this assumption we have $E(r_t) = \mu$, $Var(r_t) = \gamma_0$ and $Cov(r_t, r_{t-k}) = \gamma_k$, where μ and γ_0 are constant and γ_k is a function of k. Takin the expectation of equation 3.5 we obtain

$$E(r_t) = \phi_0 + \phi_1 E(r_{t-1})$$

Under the stationary condition of a constant mean we have

$$\mu = \phi_0 + \phi_1 \mu$$

$$\mu = \frac{\phi_0}{1 - \phi_1}$$

The result gives us two conclusions. First, the mean of r_t exists if $\phi_1 \neq 1$. Second, $E(r_t) = 0$ if $\phi_0 = 0$. Also, we can express $\phi_0 = (1\text{-}\phi_1)\mu$and using it we can rewrite our AR(1) model in the following way:

$$r_t - \mu = \phi_1(r_{t-1} - \mu) + \varepsilon_t \qquad (3.7)$$

By repeating the substitution we can express the original equation of AR(1) model as follows:

$$r_t - \mu = \sum_{i=0}^{\infty} \phi_1^i \varepsilon_{t-i} \tag{3.8}$$

Acquired equation represents r_t - μ as a linear function of ε_{t-i}. Having applied this property and the independence of the series $\{\varepsilon_t\}$ we can obtain $E[(r_t - \mu)\ \varepsilon_{t+1}] = 0$. By the stationarity assumption $Cov(r_{t-1}, \varepsilon_t) = E[(r_{t-1} - \mu)\ \varepsilon_t] = 0$. If we the square and then the expectation of equation (3.7), we obtain

$$Var(r_t) = \phi_1^2 Var(r_{t-1}) + \sigma_\varepsilon^2 \tag{3.9}$$

where σ_ε^2 is the variance of ε_t. Under the stationarity assumption that variance is constant we can obtain the following expression of variance:

$$Var(r_t) = \frac{\sigma_\varepsilon^2}{1 - \phi_1^2} \tag{3.10}$$

Consequently, the weak stationarity of an AR(1) model implies that $-1 < \phi_1 < 1$, that is, $|\phi_1| < 1$. This condition is necessary and sufficient for an AR(1) model to be weakly stationary. In order to have better understanding how ϕ_1 affects an AR(1) model we can plot it for different values of the coefficient. On the first graph with $\phi_1 = 0$ we can see white noise because only random part ε_t left in the model. As far as ϕ_1 is increasing the data are becoming less stationary. And finally, when ϕ_1 takes a value 1, the data are not stationary anymore. An AR(1) model with $\phi_0 = 0$ and $\phi_1 = 1$ is called *random walk*.

The ACF of r_t satisfies the following equation:

$$\rho_l = \phi_1 \rho_{l-1}, \quad for\ l > 0.$$

This result says that the ACF of a weakly stationary AR(1) series decays exponentially with rate φ_1 and starting value $\rho_0 = 1$.

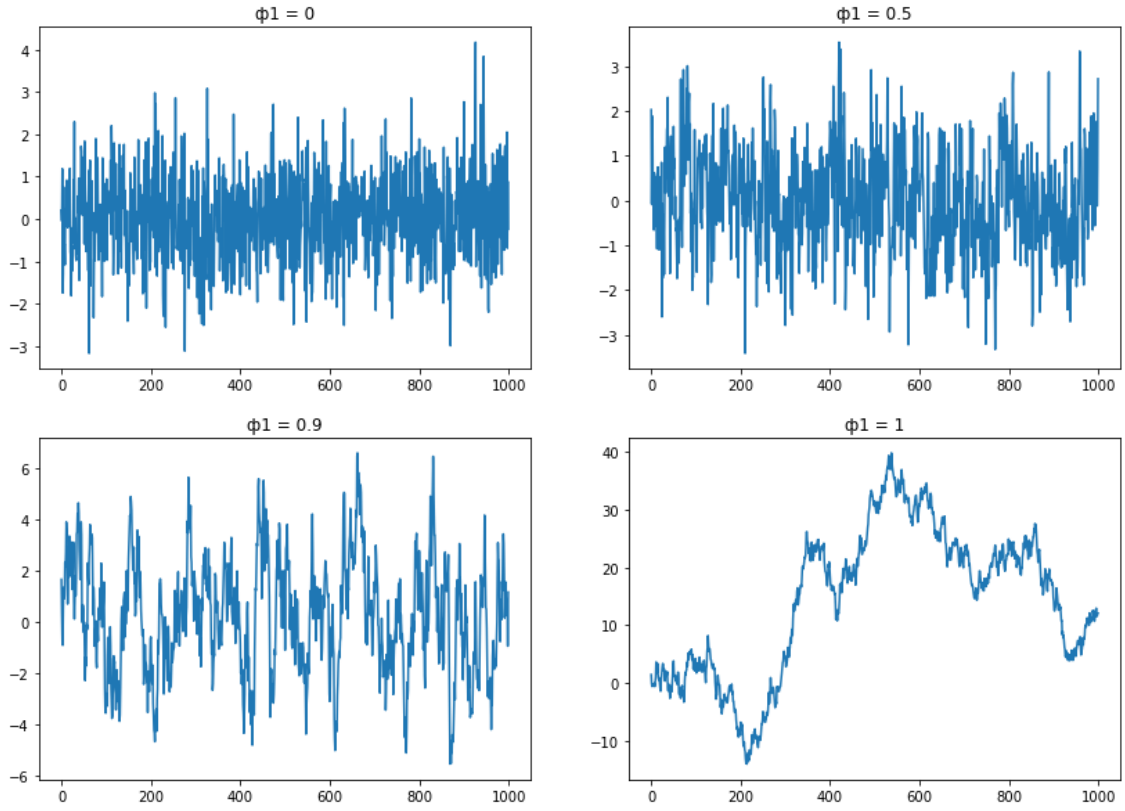

Figure 2.4: Randomly generated data by AR(1) model with different values of ϕ_1 (Author's own work)

2.3 Moving-average models

The idea behind AR processes is to feed past data back into the current value of the time series. Rather than use past values of the time series in a regression, a moving average (MA) model uses past values of the white noise process ε_t. MA(1) model is defined to be

$$r_t = c_0 + \varepsilon_t - \theta_1 \varepsilon_{t-1} \tag{3.11}$$

where c_0 is a constant and $\{\varepsilon_t\}$ is a white noise series. And similarly, we can construct a generalized form of MA process – MA(q) model:

$$r_t = c_0 + \varepsilon_t - \theta_1 \varepsilon_{t-1} - \ldots - \theta_q \varepsilon_{t-q} \tag{3.12}$$

Moving-average models are always weakly stationary because they are finite linear combinations of a white noise sequence for which the first two moments are time invariant. For example, taking expectation of MA(1) model, we have:

$$E(r_t) = c_0,$$

which is constant and does not depend on time. In a similar way, taking variance of MA(1) model, we have:

$$Var(r_t) = (1 + \theta_1^2)\sigma_\varepsilon^2,$$

which is also invariant with respect to time. These two properties can be applied to the general MA(q) model.

Autocorrelation structure of the general MA(q) model is described by the following rules:

$$\rho_k \begin{cases} \neq 0, k \leq q \\ = 0, k > q \end{cases}$$

This means that MA(q) series is linearly dependent on first *q*-lagged values. That is why it is called "finite-memory" model (Tsay 2010).

The other property of MA models is *invertibility*. We can rewrite the MA(1) model with zero mean as $\varepsilon_t = r_t + \theta_1 r_{t-1}$. If we continue the substitution we will obtain:

$$\varepsilon_t = r_t + \theta_1 r_{t-1} + \theta_1^2 r_{t-2} + \theta_1^3 r_{t-3} + \ ...$$

This equation expresses the shock at time *t* as a linear combination of the present and past returns. If $|\theta_1| < 1$ then we say that MA(1) model is *invertible*.

2.4 ARMA models

An ARMA model combines the ideas of AR and MA models into a compact form so that the number of parameters used is kept small, achieving parsimony in parameterization. A time series r_t follows an ARMA(1,1) model if it satisfies:

$$r_t - \phi_1 r_{t-1} = \phi_0 + \varepsilon_t - \theta_1 \varepsilon_{t-1} \tag{3.13}$$

The left-hand side of the (3.13) is the AR component of the model and the right-hand side gives the MA component. The constant term is φ_0. For this model to be meaningful, we need $\varphi_1 \neq \theta_1$; otherwise, there is a cancellation in the equation and the process reduces to a white noise series.

A general ARMA(p, q) model can expressed in the following form:

$$r_t = \phi_0 + \sum_{i=1}^{p} \phi_i r_{t-i} \ + \ \varepsilon_t - \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \tag{3.14}$$

where $\{\varepsilon_t\}$ is a white noise series and *p* and *q* are nonnegative integers.

ARMA(1,1) models also has several properties. Under condition of a weakly stationarity the mean of r_t is defined as follows:

$$E(r_t) = \mu = \frac{\phi_0}{1 - \phi_1}$$

Assuming a weakly stationarity of the time series r_t we can also express the variance:

$$Var(r_t) = \frac{(1 - 2\phi_1\theta_1 + \theta_1^2)\sigma_\varepsilon^2}{1 - \phi_1^2}$$

Because the variance is positive, we need $|\varphi_1| < 1$.

The ACF of ARMA(1,1) time series is ruled by the following equation:

$$\rho_l = \phi_1\rho_{l-1}, \quad for\ l > 1.$$

In such a way we can conclude the the ACF of an ARMA(1,1) model behaves almost like that of an AR(1) model except the fact that the exponential decay starts with lag 2.

There are two methods for identifying the order of AR or MA part in ARMA models:

1. ACF and PACF
2. Information criteria

The ACF and PACF are not informative in determining the order of ARMA models, therefore we should focus on information criteria. There are two well-known information criteria: Akaike information criterion (AIC) and Bayesian information criterion (BIC).

The Akaike information criterion attempts to balance the complexity of the model, which in this case means the number of parameters, with how well it fits the data. If we take the likelihood function for a statistical model, which has *k* parameters, and *L* maximises the likelihood, then the Akaike Information Criterion is given by:

$$AIC = -2\,ln(L) + 2k \tag{3.15}$$

The preferred model, from a selection of models, has the minimum AIC of the group. One can see that the AIC grows as the number of parameters, *k*, increases, but is reduced if the negative log-likelihood increases. Essentially it penalises models that are tending to overfit.

The Bayesian information criterion has similar behaviour to the AIC in that it penalises models for having too many parameters. The difference between the BIC and AIC is that

the BIC is stricter with penalisation of additional parameters. The Bayesian information criterion is defined as:

$$BIC = -2\,ln(L) + k\,ln(n) \tag{3.16}$$

where *n* is the number of data points in the time series.

Once an ARMA(*p*,*q*) model is specified, its parameters can be estimated by either the conditional or exact-likelihood method. In addition, the Ljung–Box statistics of the residuals can be used to check the adequacy of a fitted model.

2.5 GARCH models

ARMA models are used to model the conditional expectation of a process given the past, but in an ARMA model the conditional variance given the past is constant. In financial data we usually see that a volatility is not constant in time. Therefore, GARCH models are widely used for volatility modeling.

GARCH stands for Generalized Autoregressive Conditional Heteroscedasticity. A collection of random variables is heteroskedastic if there are subsets of variables within the larger set that have a different variance from the remaining variables. For example, if a non-stationary time series possesses seasonal or trend effects, then the variance of the series increases with the seasonality or the trend. This type of variability is known as heteroscedasticity. Conditional heteroscedasticity means that variance at one time has a positive relationship with variance at one or more previous time steps. This leads to the fact that periods of high volatility tend to follow periods of high volatility and vice versa.

We should start explanation with a model called ARCH or Autoregressive Conditional Heteroscedasticity. The basic idea of ARCH models is that the shock ε_t of an asset return is serially uncorrelated, but dependent, and the dependence of ε_t can be described by a simple quadratic function of its lagged values (Tsay 2010).

An ARCH(m) model can be written as follows:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{m} \alpha_i \varepsilon_{t-i}^2, \qquad \varepsilon_t = \sigma_t \epsilon_t \tag{3.17}$$

where $\{\epsilon_t\}$ is a sequence of independent and identically distributed (i.i.d.) random variables with mean zero and variance 1, $\alpha_0 > 0$, and $\alpha_i \geq 0$ for $i > 0$.

Although the ARCH model is relatively simple, it often requires many parameters for proper description of an asset return volatility. The ARCH model is appropriate when the error variance in an asset returns series follows an AR model. If we apply ARMA for the error variance, the result will be a GARCH model. The shock ε_t of an asset return follows a GARCH(m, s) model if

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{m} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{s} \beta_j \sigma_{t-j}^2, \qquad \varepsilon_t = \sigma_t \epsilon_t \tag{3.18}$$

where $\{\epsilon_t\}$ is a sequence of independent and identically distributed (i.i.d.) random variables with mean zero and variance 1, $\alpha_0 > 0$, $\alpha_i \geq 0$, $\beta_j \geq 0$ and $\sum_{i=1}^{max(m,s)} (\alpha_i + \beta_i) < 1$.

To see the strengths and weaknesses of GARCH models we should look at the GARCH(1,1) model:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2, \qquad 0 \leq \alpha_1, \beta_1 \leq 1, (\alpha_1 + \beta_1) < 1$$

It is obvious that large ε_{t-1}^2 and σ_{t-1}^2 values lead to large values of σ_t^2. It means that large ε_t follows by large ε_{t-1}. This behavior is known in financial time series as volatility clustering.

Further in the thesis ARMA and GARCH models described above will be used for modeling different stock returns and making prediction of their movements in the future.

Chapter 3

Machine learning methods

In Chapter 3 we introduce the basics of machine learning tasks and describe several machine learning methods. In the first part of chapter we distinguish different problems where machine learning can be applied. In the second part we focus on several machine learning methods for classification that will be used in Chapter 4.

3.1 Machine learning tasks

Machine learning is a very popular topic nowadays and there are several reasons for that. The most exciting thing is that machine learning provides the ability to automatically obtain deep insights, recognize unknown and invisible patterns in data, and create high performing predictive models from data without being explicitly programmed. Machine learning opens new doors for research in finance, especially in exploration of stock prices behaviour.

The formal definition of machine learning was stated by Tom M. Mitchell (1997).

Definition 3. A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P* if its performance at tasks in *T*, as measured by *P*, improves with experience *E*.

The goal of machine learning algorithms is to learn from data, in order to build generalizable models that give accurate predictions, or to find patterns, particularly with new and unseen similar data.

Machine learning tasks are usually classified into two broad categories: *supervised* and *unsupervised* learning.

- **Supervised machine learning** can apply what has been learned in the past to new data using labeled examples to predict future events. For example, when learning to classify handwritten digits, a supervised learning algorithm takes thousands of pictures of handwritten digits along with labels containing the correct number each image represents. The algorithm will then learn the relationship between the images and their associated numbers and apply that learned relationship to classify completely new images (without labels) that the machine hasn’t seen before.

    - **Semi-supervised learning** - the learning algorithm is provided with a mixture of labeled and unlabeled data. The systems that use this method are able to considerably improve learning accuracy.
    - **Active learning** is similar to semi-supervised learning, but the algorithm can "ask" for extra labeled data based on what it needs to improve on.
    - **Reinforcement learning** is a method that interacts with its environment by producing actions and discovers penalties or rewards. The goal is maximizing lifetime/long-term reward (or minimizing lifetime/long-term penalty).
- **Unsupervised learning** - the learning algorithm is provided with unlabeled examples. Generally, unsupervised learning is used to uncover some structure or pattern in the data. This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use.

There are two tasks of supervised learning: *regression* and *classification*. In regression tasks the goal is to predict a continuous target variable. Examples include a person's age, height, or income, the value of a house, and the price of a stock. In contrast, in classification tasks the goal is to predict discrete target variable. For example, a person's gender (male or female), the brand of product purchased (brand A, B, or C), whether a person defaults on a debt (yes or no), or a cancer diagnosis (James, Witten, Hastie and Tibshirani 2013).

Regression involves fitting a model to data. The goal is to understand the relationship between one set of variables - the dependent or target variables (y) - and another set - the independent or explanatory variables ($\boldsymbol{X}$ or x). In cases of just one dependent and one explanatory variable, we have simple regression. In scenarios with more than one explanatory variable, we have multiple regression. In scenarios with more than one dependent variable, we have multivariate regression.

Classification problems are where the target variables are discrete, and they represent some categories or classes. For binary classification, there are only two classes ($y \in \{0,1\}$). Otherwise, the classification problem is called a multiclass classification problem - there are more than two classes.

The most widely used algorithms for supervised learning are:

- Support Vector Machines
- Linear regression

- Logistic regression
- Naive Bayes
- Linear discriminant analysis (LDA)
- Decision trees
- k-nearest neighbor algorithm
- Neural networks

The two main unsupervised learning tasks are *clustering* the data into groups by similarity and *reducing dimensionality* to compress the data while maintaining its structure and usefulness. In contrast to supervised learning, there is no precise metric for how well an unsupervised learning algorithm is doing. Performance is usually subjective and domain-specific.

The most known algorithms for unsupervised learning are:

- K-means
- Hierarchical clustering
- Mixture models
- Manifold learning algorithms
- Principal component analysis (PCA)
- Singular value decomposition
- Neural networks (Autoencoders, Deep Belief Nets, etc.)

We can conclude that the scope of machine learning algorithms is wide and not all of them are of our interest for the aim of the thesis. As the main goal is to predict stock returns we can reduce this task to a binary classification problem: whether the price will go up or down in the future, whether the return will be positive or negative in other words. That is why further we will focus on regression and classification machine learning algorithms.

3.2 Supervised machine learning algorithms

There are many methods in machine learning for classification problems. Some of them are relatively simple, some are more advanced. The goal of this thesis is to predict stock returns or the direction of the price movement. This is a typical classification problem. In this part of the chapter we will consider several algorithms that can solve this type of

problems: logistic regression as the basic algorithm for classification, decision trees and their ensembles and artificial neural networks.

3.2.1 Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification or the log-linear classifier. Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables, which are usually (but not necessarily) continuous, by estimating probabilities.

An explanation of logistic regression begins with an explanation of the logistic function. The logistic function is useful because it can take an input with any value from negative to positive infinity, whereas the output always takes values between zero and one and hence is interpretable as a probability. The logistic function *σ(z)*, also known as a *sigmoid* function, is defined as follows:

$$\sigma(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}} \tag{4.1}$$

A graph of the logistic function is shown in figure 3.1.

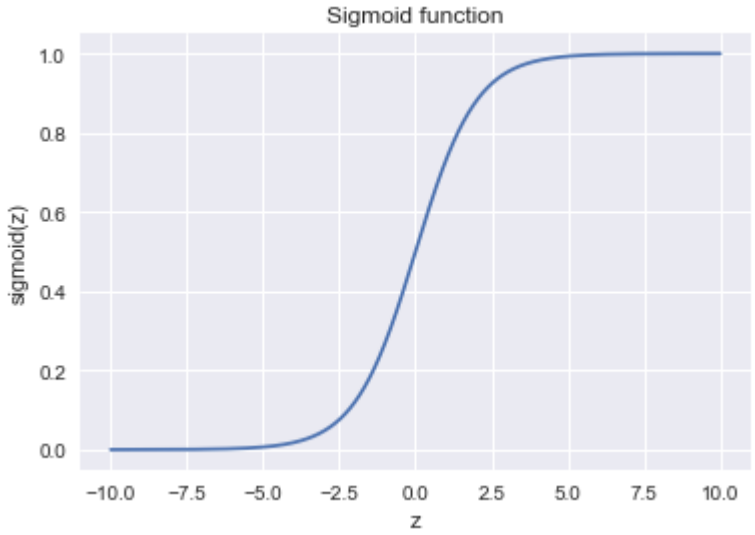

Figure 3.1: Logistic (sigmoid) function (Author's own work)

The input parameter *z* of the logistic function is a linear function that is expressed in the following way:

$$z = w_0 + w_1 x$$

This equation can be written in the matrix representation:

$$z = \boldsymbol{w}^T \boldsymbol{X}, \tag{4.2}$$

where $\boldsymbol{w^T}$ is a vector of weights and $\boldsymbol{X}$ is a matrix of explanatory variables.

Now the logistic function can be expressed as:

$$\sigma(\boldsymbol{w}^T \boldsymbol{X}) = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{X}}} \tag{4.3}$$

The sigmoid function can be interpreted as a probability that the target variable is equal to 1: $\sigma(\boldsymbol{w}^T \boldsymbol{X}) = P(y = 1|\boldsymbol{w}^T; \boldsymbol{X})$. Since we make a binary classification, we want to output a label, not a continuous value. Then we might say that y = 1 if σ(z) ≥ 0.5 and y = 0 if σ(z) < 0.5. The line that forms this divide is an example of a decision boundary.

The regression coefficients are usually estimated using maximum likelihood estimation. Unlike linear regression with normally distributed residuals, it is not possible to find a closed-form expression for the coefficient values that maximize the likelihood function, so that an iterative process must be used instead.

The negative log-likelihood for logistic regression is given by:

$$NLL(\boldsymbol{w}^T \boldsymbol{X}) = -\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \left[y_i ln\left(\sigma_i(\boldsymbol{w}^T \boldsymbol{x_i})\right) + (1 - y_i) ln\left(1 - \sigma_i(\boldsymbol{w}^T \boldsymbol{x_i})\right) \right] \tag{4.4}$$

This is also called the cross-entropy error function. Another way of writing this equation is following. Suppose yi ∈ {−1, +1} instead of yi ∈ {0, 1}. Then we have:

$p(y = 1) = \frac{1}{1+e^{-\boldsymbol{w}^T X}}$ and $(y = -1) = \frac{1}{1+e^{\boldsymbol{w}^T X}}$. Hence:

$$NLL(\boldsymbol{w}^T \boldsymbol{X}) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} ln\left(1 + exp(-\boldsymbol{y_i} \boldsymbol{w}^T \boldsymbol{x_i})\right) \tag{4.5}$$

Minimizing the negative log likelihood is equivalent to maximizing log likelihood. Hence, our goal is to minimize the function in the equation 4.5. We can achieve that using optimization algorithm known as *gradient descent*. Gradient descent perhaps is the most common optimization algorithm in machine learning.

The function we want to minimize or maximize is called the objective function. When we are minimizing it, we may also call it the cost function, loss function, or error function.

Suppose we have a function $y = f(x)$, where both x and y are real numbers. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. We can thus reduce $f(x)$ by moving x in small steps with opposite sign of the derivative. This technique is called gradient descent.

For functions with multiple inputs, we must make use partial derivatives. The gradient generalizes the notion of derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all of the partial derivatives, denoted $\nabla_x f(x)$. The gradient points directly uphill, and the negative gradient points directly downhill. We can decrease f by moving in the direction of the negative gradient. This is known as the method of steepest descent or gradient descent. Gradient descent proposes a new point:

$$x' = x - \alpha \nabla_x f(x) \tag{4.6}$$

where α is the learning rate, a positive scalar determining the size of the step. If α is too small, the minimization algorithm takes long time to converge, if α is too big, the algorithm can diverge. Gradient descent converges when every element of the gradient is zero or, in practice, very close to zero (Goodfellow, Bengio and Courville 2016). An illustration of how gradient descent algorithm works is shown in figure 3.2.

For logistic regression the gradient descent algorithm takes the following form:

$$w_j^{new} = w_j^{old} - \alpha \cdot \sum_{i=0}^{N} \left[\sigma\left(\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}^{(i)}\right) - y^{(i)}\right] x_j^{(i)} \tag{4.7}$$

where $\sum_{i=0}^{N} \left[y^{(i)} - \sigma\left(\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}^{(i)}\right)\right] x_j^{(i)}$ is a gradient of the negative log likelihood.

One problem with ML estimation is that it can result in *overfitting*. The reason that the MLE can overfit is that it is picking the parameter values that are the best for modeling the training data; but if the data is noisy, such parameters often result in complex functions. In order to prevent overfitting, we should use *regularization*.

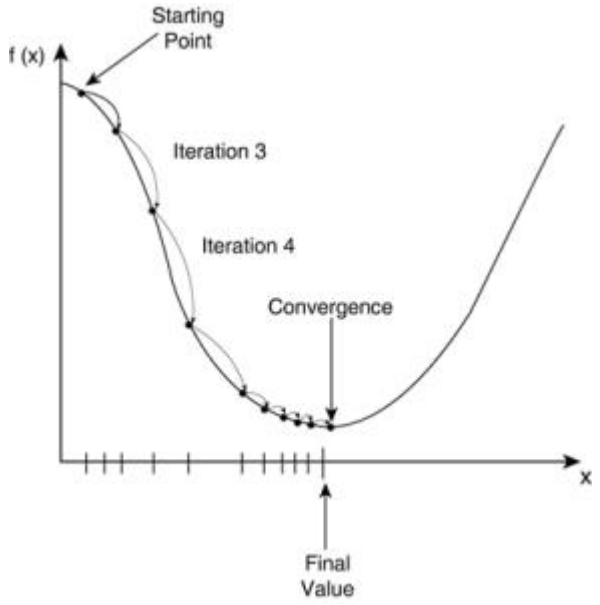

Figure 3.2: Gradient descent method (Credit towardsdatascience.com webpage)

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. In intuitive terms, we can think of regularization as a penalty against complexity. Increasing the regularization strength penalizes large weight coefficients. Our goal in an unregularized model is to minimize the cost function, i.e., we want to find the weights that correspond to the global cost minimum. Now, if we regularize the cost function, we add an additional to our cost function that increases as the value of your parameter weights $\boldsymbol{w}$ increase:

$$J(\boldsymbol{w}) = NLL(\boldsymbol{w}) + \lambda \boldsymbol{w}^T \boldsymbol{w} \qquad (\mathbf{4.8})$$

where λ controls the strength of the regularization. This type of regularization is known as L_2 regularization.

3.2.2 Decision trees

Decision trees are defined by recursively partitioning the input space and defining a local model in each resulting region of input space. This can be represented by a tree, with one leaf per region. Decision trees can be applied to both regression and classification problems, but we will focus on classification decision trees only.

A general schema that represents a decision tree can look as a diagram in figure 3.3.

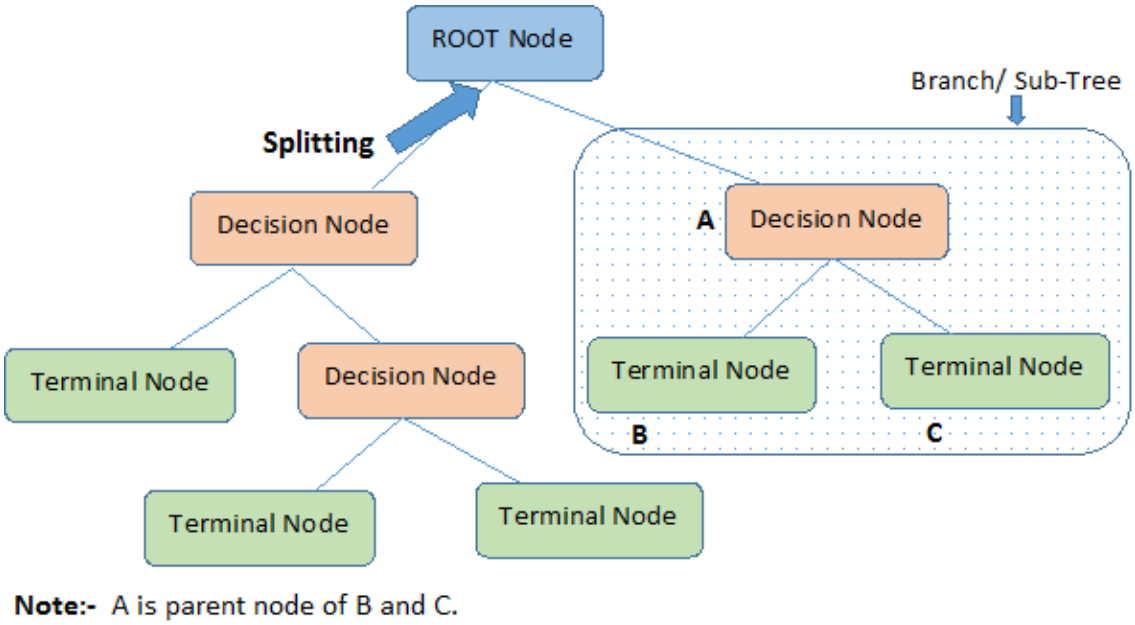

Figure 3.3: Schema of a decision tree (Credit www.analyticsvidhya.com webpage)

Root node represents entire population or sample and it is divided then into two or more homogeneous sets. Decision node is a sub-node that splits further into other sub-nodes. Leaf or terminal node is a node that has no sub-nodes.

Basic algorithm for building a decision tree looks as follows:

1. Start with all the data in the root node.
2. Find some criteria, which splits outcomes the best way
3. Divide the data into two groups
4. Repeat until the groups are too small or sufficiently homogeneous
5. Prediction of a terminal node is the most common outcome (in case of classification task) or the mean of outcomes (in case of regression task)

There are several criteria that measure the quality of the split for classification decision trees.

Misclassification error is the fraction of the training observations in the node that do not belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk}) \tag{4.9}$$

where $\hat{p}_{mk}$ represents the proportion of training observations in the m_{th} node that are from the k_{th} class. However, misclassification error is not sufficiently sensitive for tree-growing, and in practice two other measures are used.

The **Gini index** is defined by:

$$G = \sum_{k=1}^{K} \hat{p}_{mk} (1 - \hat{p}_{mk}) \tag{4.10}$$

The Gini index is a measure of total variance across the *K* classes. It takes a small value if all of the $\hat{p}_{mk}$ are close to zero or one. For this reason the Gini index is referred to as a measure of node purity - a small value indicates that a node contains generally observations from a single class.

An alternative to the Gini index is **cross-entropy**, given by:

$$H = -\sum_{k=1}^{K} \hat{p}_{mk} \ln \hat{p}_{mk} \tag{4.11}$$

One can show that the cross-entropy will take on a value near zero if the $\hat{p}_{mk}$ are all near zero or near one. Therefore, like the Gini index, the cross-entropy takes a small value if the m_{th} node is pure.

For example, if we have two classes and $\hat{p}_{m1} = p$, then:

- Misclassification error: $\min(p, 1 - p)$
- Gini index: $2p \cdot (1 - p)$
- Cross-entropy: $-p \cdot \ln p - (1 - p) \cdot \ln(1 - p)$

This situation is shown in figure 3.4.

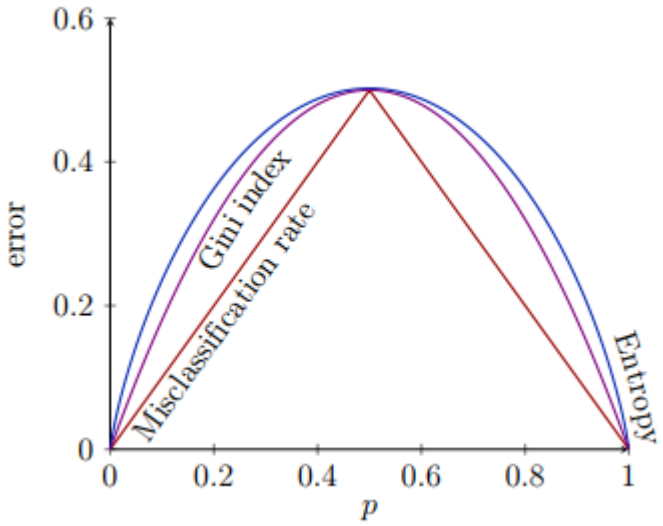

Figure 3.4: Node impurity measures for binary classification (Credit github.com/diefimov/MTH594_MachineLearning webpage)

To prevent overfitting, we can stop growing the tree if the decrease in the error is not sufficient to justify the extra complexity of adding an extra subtree. The standard approach, however, is to grow a full tree, and then to perform pruning. To determine how far to prune back, we can evaluate the cross-validated error on each subtree, and then pick the tree whose CV error is within 1 standard error of the minimum (Murphy 2012).

Decision tree models are popular for several reasons: they are easy to interpret, they can easily handle mixed discrete and continuous inputs, they perform automatic variable selection, they are relatively robust to outliers, they scale well to large data sets, and they can be modified to handle missing inputs.

On the other hand, decision trees have some disadvantages. First, they do not have the same level of predictive accuracy as other kinds of model. Another problem is that trees can be unstable. In other words, a slight change in the input data can lead to a large change in the final tree. However, by aggregating many decision trees, using methods like bagging and boosting, the predictive performance of trees can be substantially improved.

3.2.3 Bagging and boosting

Usually, with increasing complexity of the model prediction error reduces due to lower bias in the model. But at some point, the variance starts to increase and at the end the model is overfitted. It is imported to maintain a balance between these two types of errors. This is known as the trade-off management of bias-variance errors, which is shown in figure 3.5.

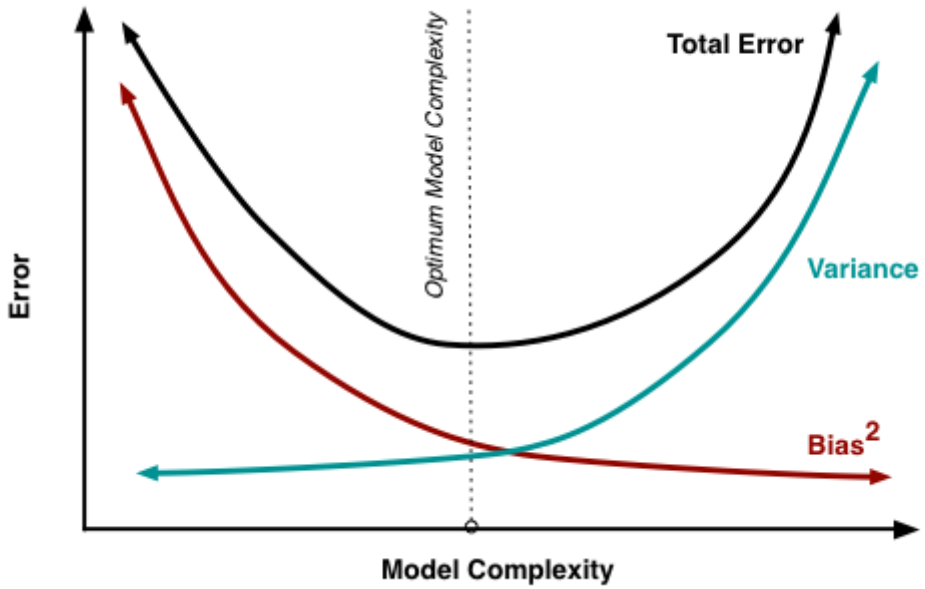

Figure 3.5: Optimal model complexity (Credit www.analyticsvidhya.com webpage)

One way to reduce the variance of an estimate is to average together many estimates. For example, we can train *M* different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble:

$$f(x) = \sum_{m=1}^{M} \frac{1}{M} f_m(x) \qquad (4.12)$$

where f_m is the m_{th} tree. This technique is called *bootstrap aggregation* or *bagging*.

Unfortunately, repeating run of the same learning algorithm on different subsets of the data can lead to highly correlated predictors, which limits the amount of variance reduction that is possible. *Random forests* provide an improvement over bagged trees by decorrelating the trees. As in bagging, we build a number forest of decision trees on

bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of *m* predictors is chosen as split candidates from the full set of *p* predictors. The split is used only for those *m* predictors. In practice usually $m = \sqrt{p}$ is used (James, Witten, Hastie and Tibshirani 2013).

Another approach for improving the predictions from a decision tree is *boosting*, which is a generally can be applied to many statistical learning methods. Boosting works in a similar way to bagging, except that the trees are grown sequentially: each tree is grown using information from previously grown trees. Boosting does not involve bootstrap sampling. Instead each tree is fit on a modified version of the original data set.

One of the realization of the boosting is *gradient boosting* algorithm. The basic principles of gradient boosting are as follows: given a loss function and a weak learner (e.g., regression trees), the algorithm seeks to find an additive model that minimizes the loss function. The algorithm is typically initialized with the best guess of the response. The gradient is calculated, and a model is then fit to the residuals to minimize the loss function. The current model is added to the previous model, and the procedure continues for a specified number of iterations.

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. That is why an important part of gradient boosting are regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting. There are several regularization methods for gradient boosting such as tree constraints, shrinkage, random sampling and penalized learning.

There are a number of ways that the trees can be constrained:

- **Number of trees**. The general rule is to keep adding trees until no further improvement is observed.
- **Tree depth.** In practice, usually, shorter trees are preferred.
- **Number of nodes** can constrain the size of the tree.
- **Number of observations** per split sets a minimum constraint on the amount of training data at a training node before a split can be considered.
- **Minimum improvement to loss** is a constraint on the improvement of any split added to a tree.

Another constrain on a learning process is *shrinkage*. Instead of adding the predicted value for a sample to previous iteration's predicted value, only a fraction of the current predicted value is added to the previous iteration's predicted value. This fraction is commonly referred to as the learning rate and is parameterized by the symbol, λ. This parameter can take values between 0 and 1 and becomes another tuning parameter for the model.

There is a modification of the gradient boosting algorithm called *stochastic gradient boosting*. It involves a random sampling scheme: at each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.

Another useful regularization techniques for gradient boosted trees is to *penalize* tree complexity using L_1 or L_2 regularization of the leaf weight values of the trees.

3.2.4 Artificial neural networks

Artificial neural networks (ANNs) are computing systems inspired by the biological neural networks that constitute animal brains. An ANN is based on a collection of connected units or nodes called artificial neurons (a simplified version of biological neurons in an animal brain). Each connection (a simplified version of a synapse) between artificial neurons can pass a signal from one to another.

The most common type of neural networks is feedforward neural network or multilayer perceptron (MLP). The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input x to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{w})$ and learns the value of the parameters $\boldsymbol{w}$ that result in the best function approximation.

An example of a multilayer perceptron's structure is shown in figure 3.6.

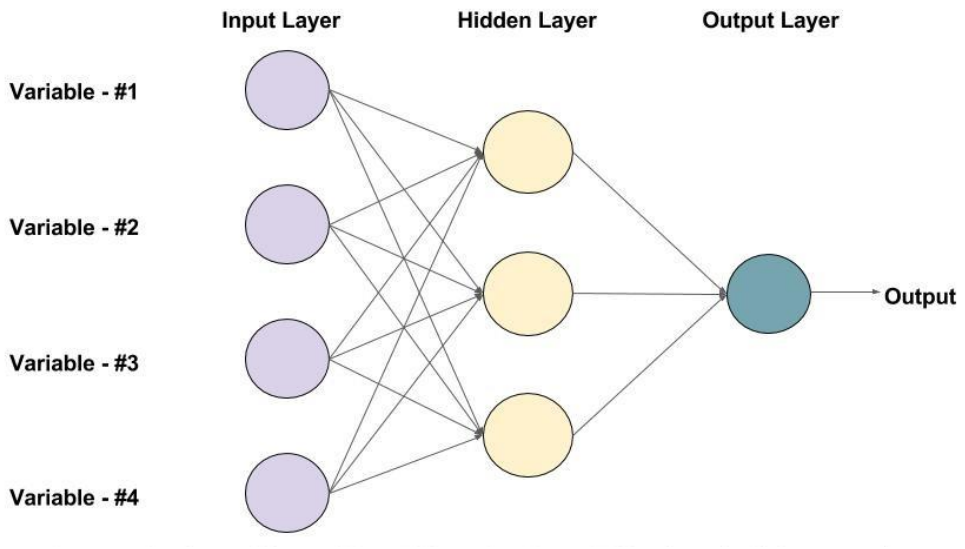

Figure 3.6: An example of a feedforward network with 1 hidden layer (Credit www.learnopencv.com webpage)

Given neural network has an input layer, an output layer, and a hidden layer. In general, there can be multiple hidden layers. Each node, called neuron, in the layer can be thought as the basic processing unit. A schematic diagram of the process unit is given in figure 3.7.

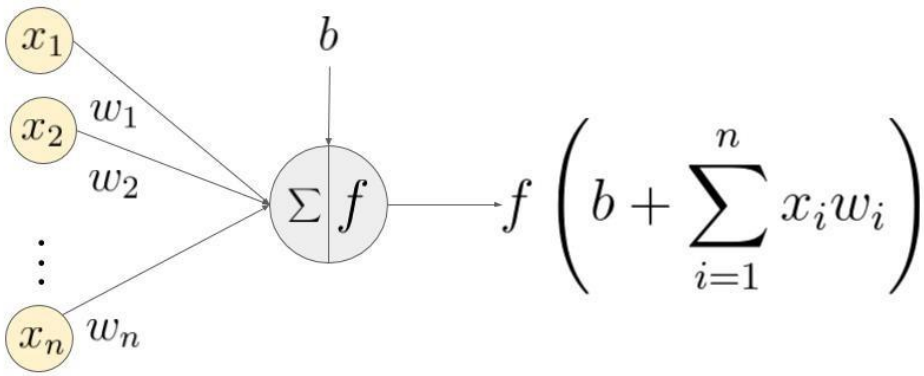

Figure 3.7: An example of a neuron showing the input ($x_1 \ldots x_n$), their corresponding weights ($w_1 \ldots w_n$), a bias (b) and an activation function applied to the weighted sum of the inputs (Credit www.learnopencv.com webpage)

As seen above, a neuron calculates the weighted sum of its inputs and then applies an activation function that can be linear or nonlinear. There are three most used activation functions:

1. Already known *sigmoid* function: $\sigma(\boldsymbol{w}^T\boldsymbol{X}) = \frac{1}{1+e^{-w^T X}}$
2. *Hyperbolic tangent* function: $tanh(\boldsymbol{w}^T\boldsymbol{X}) = 2\sigma(2\boldsymbol{w}^T\boldsymbol{X}) - 1$
3. *Rectified Linear Unit* (ReLU): $f(x) = max(0, x)$

The training of a feedforward network is carried out using the *backpropagation* algorithm. The training samples are passed through the network and the output obtained from the network is compared with the actual output to compute the value of some predefined error function. The error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. To adjust weights properly a gradient descent algorithm is used. The derivative of the error function with respect to the network weights should be calculated, and the weights are changed such that the error decreases.

The other type of neural networks that is of interest for the thesis are *recurrent neural networks*. RNNs are a family of neural networks for processing sequential data.

A typical RNN is shown in figure 3.8.

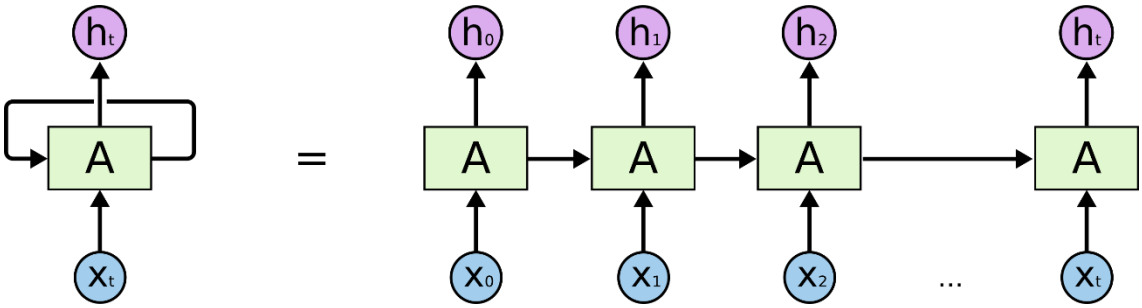

Figure 3.8: A general structure of a recurrent neural network (Credit colah.github.io webpage)

As can be seen from the graph above the output of the hidden layer in a recurrent neural network is fed back into itself. For learning RNNs an extension of a backpropagation algorithm is used. It is called *backpropagation through time* or BPTT. BPTT works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the network, and one output. Errors are then calculated and accumulated for each timestep. The network is rolled back up and the weights are updated.

For recurrent neural networks, we would want to have long memories, so the network can connect data relationships at significant distances in time. However, the more time steps we have, the more chance we have that backpropagation gradients will either accumulate and explode or vanish down to nothing. This issue can be resolved by applying modified form of RNNs – the *Long Short-Term Memory Networks* or LSTM.

LSTM networks are a special kind of RNN that is capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and then were popularized by many people in following work. LSTMs work well on a large variety of problems and are now widely used.

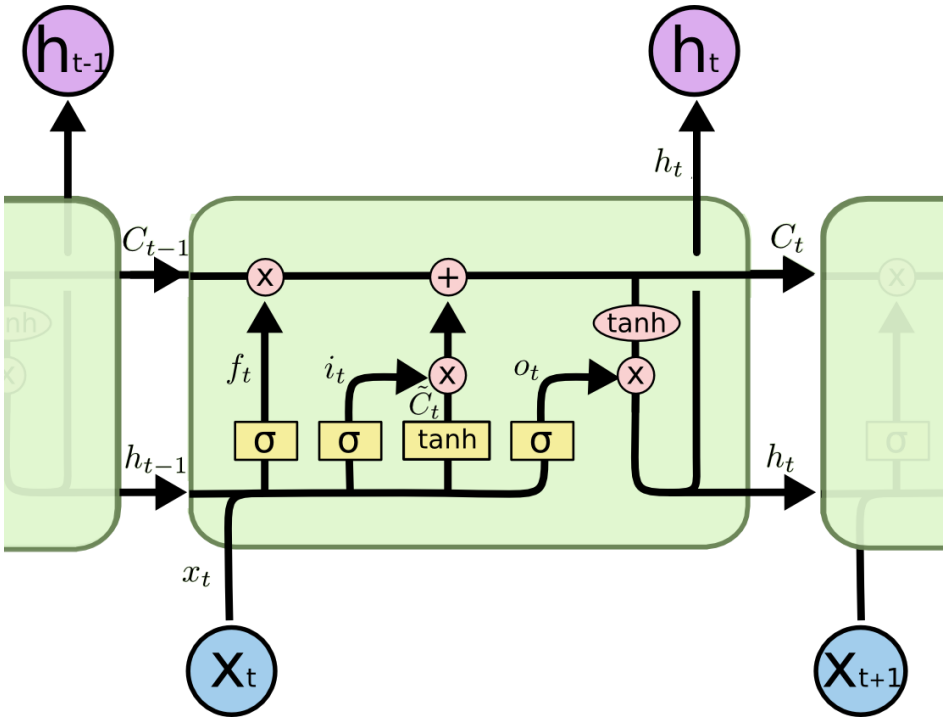

Figure 3.9: The repeating module of LSTM that has four interacting layers (Credit colah.github.io webpage)

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, they have four. The structure of LSTMs is shown in figure 3.9.

A common architecture is composed of a **memory** cell, an **input gate**, an **output gate** and a **forget gate**.

A forget gate is responsible for removing information from the cell state. It is a sigmoid function that takes an output vector from the previous cell $\boldsymbol{h_{t-1}}$ and the input vector $\boldsymbol{x_t}$:

$$\boldsymbol{f_t} = \boldsymbol{\sigma}\left(\boldsymbol{W_f x_t} + \boldsymbol{U_f h_{t-1}} + \boldsymbol{b_f}\right) \tag{4.13}$$

where $\boldsymbol{W_f}$ and $\boldsymbol{U_f}$ are weights matrices and $\boldsymbol{b_f}$ is a bias vector.

A memory cell or a cell state consists of two parts. First, a sigmoid layer called the input gate controls which values will be updated:

$$\boldsymbol{i_t} = \boldsymbol{\sigma}(\boldsymbol{W_i x_t} + \boldsymbol{U_i h_{t-1}} + \boldsymbol{b_i}) \tag{4.14}$$

Next, a tanh layer creates a vector of new candidate values, $\widetilde{\boldsymbol{C}}_{\boldsymbol{t}}$, that could be added to the state:

$$\widetilde{\boldsymbol{C}}_{\boldsymbol{t}} = \tanh(\boldsymbol{W_C x_t} + \boldsymbol{U_C h_{t-1}} + \boldsymbol{b_C}) \tag{4.15}$$

Then a new cell state is created in the following way:

$$\boldsymbol{C_t} = \boldsymbol{f_t} * \boldsymbol{C_{t-1}} + \boldsymbol{i_t} * \widetilde{\boldsymbol{C}}_{\boldsymbol{t}} \tag{4.16}$$

where operator $*$ denotes Hadamard product.

The output vector is based on another sigmoid layer called output gate and the cell state filtered by tanh activation function:

$$\boldsymbol{o_t} = \boldsymbol{\sigma}(\boldsymbol{W_o x_t} + \boldsymbol{U_o h_{t-1}} + \boldsymbol{b_o}) \tag{4.17}$$

$$\boldsymbol{h_t} = \boldsymbol{o_t} * \tanh(\boldsymbol{C_t}) \tag{4.18}$$

Such a complicated structure allows the error to remain in the LSTM unit's memory when error values are back-propagated from the output. This transition continuously feeds error back to each of the gates until they learn to cut off the value. Thus, backpropagation through time is effective at training an LSTM unit to remember values for long durations.

Chapter 4

Stock returns prediction

In Chapter 4 we apply models described in previous chapters to make predictions of stock returns. In the first section we describe methodology of how experiments are conducted and methods for obtaining and preprocessing data. In the second section we present the results of ARMA and GARCH models application. The third section is devoted to the outcomes of machine learning methods application.

4.1 Methodology

In order to achieve the goals of the thesis we have to automate the process of modeling and forecasting. Therefore we used R and Python programming languages.

R was chosen for ARMA and GARCH modeling because it has comprehensive libraries such as *tseries* and *rugarch*. These libraries allow to easily build and fit ARMA and GARCH models with optimal parameters and to make forecasts conveniently. *PerformanceAnalytics* library has a wide range of indicators for measuring performance of strategies based on our predictions.

On the other hand, Python offers rich and efficient libraries for applying machine learning methods. We used Python version 3.6 with *pandas* and *numpy* libraries for data manipulation. *Scikit-learn* library contains a variety of machine learning methods and it was used for logistic regression. *LightGBM* library was developed by Microsoft for building gradient boosted trees. The library uses computationally efficient algorithms therefore the time needed for training a model is lower than in other libraries. *Keras* and *Tensorflow* libraries were used for constructing LSTM neural networks. *Keras* is an API for *Tensorflow* therefore it makes easier to build ANNs.

Making precise point predictions of stock returns is a very complicated task. Therefore, much more useful information for us is the sign of that prediction, whether return is positive or not. The direction of a future stock movement gives us an information to make a trading decision whether to buy or to sell a stock. That is why we consider stock returns prediction task as a classification problem.

The process for making time series modeling consists of the following steps:

1. Download the stock data containing daily information about Open, High, Low, Close prices and Volume.
2. Compute daily log returns from adjusted Close prices. This is the only column used for modeling.
3. Split the data into train and test datasets.
4. Fit ARMA and GARCH models on train + rolling window dataset. As we make 1-day ahead prediction we need to extend our train dataset by 1 day from test set on each iteration.
5. Make a prediction for the following day. If the prediction is positive then we expect upward direction of the stock movement and go long, otherwise we go short.
6. Calculate returns of the strategy based on the predicted direction and realized return on the respective day.
7. Calculate accuracy and Gini coefficient for measuring the quality of classification.
8. Calculate performance metrics for the strategy based on our predictions and for the benchmark Buy & Hold strategy: total return; return per annum; return per trade (per day); Sharpe ratio (annualized); Maximum Drawdown; p-value for t-test whether the return per trade is greater than 0.
9. Save the results to a csv file.

The general process for machine learning methods is quite similar:

1. Download the stock data containing daily information about Open, High, Low, Close prices and Volume.
2. Compute daily log returns from adjusted Close prices and generate several extra predictors from the original data:1-day lagged Open and Close prices; difference between Open and 1-day lagged Close prices; number of month, day and day of week. These features along with Open, High, Low, Close prices and Volume can possibly help algorithms to learn better.
3. Split the data into train and test datasets.
4. Transform the data using Standard Scaler from the *Scikit-learn* library.
5. Train models on the train dataset using 10-folds cross-validation for finding optimal input parameters. It means that we train our models on 9 folds and then validate the result on 10th. This operation repeats 10 times in order to validate

on all the folds. Then we choose parameters for the model which demonstrated the best average score on the validation data and the model on the entire train dataset. For splitting the train data into 10 folds we use *TimeSeriesSplit* function from the *Scikit-learn* library. This function keeps the time structure of the data and prevents from training on the future data and validate on the past one.

6. Make predictions of stock movement's direction for the test data. If the prediction is 1 then we go long. If the prediction is −1 we go short.
7. Calculate returns of the strategy based on the predicted direction and realized return on the respective day.
8. Calculate accuracy and Gini coefficient for measuring the quality of classification.
9. Calculate performance metrics for the strategy based on our predictions.
10. Save the results to a csv file.

At the end all the results are merged into one table to compare the models with each other and the benchmark.

4.2 Data

We conducted experiments on 20 stocks from a list of the biggest companies by market capitalization that are traded on New York Stock Exchange and NASDAQ Stock Exchange. All data were downloaded from Quandl. It is a service that provides financial data including historical stock prices. Quandl has R and Python APIs that allows to download data from their servers in convenient format.

Companies and their tickers used in the thesis are listed in the table 4.1.

Table 4.1: List of stocks used in experiments

Company	**Ticker**
Apple Inc.	AAPL
Amazon.com, Inc.	AMZN
Alphabet Inc.	GOOGL
Microsoft Corporation	MSFT
JPMorgan Chase & Co.	JPM
Johnson & Johnson	JNJ
Exxon Mobil Corporation	XON
Walmart Inc.	WMT

Intel Corporation	INTC
Chevron Corporation	CVX
International Business Machines Corporation	IBM
The Procter & Gamble Company	PG
The Boeing Company	BA
The Coca-Cola Company	KO
PepsiCo, Inc.	PEP
NVIDIA Corporation	NVDA
McDonald's Corporation	MCD
Amgen, Inc.	AMGN
General Electric Company	GE
Honeywell International Inc.	HON

Our datasets cover the period from 03/01/2006 to 29/12/2017. We split the data into train and test sets. The train data covers the period from 03/01/2006 to 30/12/2014 (2264 observations). The test data covers the period from 31/12/2014 to 29/12/2017 (754 observations), which is approx. 20% of all observations.

In the process description for machine learning methods we mentioned that the data should be transformed using Standard Scaler. It standardizes features by removing the mean and scaling to unit variance.

$$x' = \frac{x - \bar{x}}{\sigma} \tag{4.1}$$

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature do not look like standard normally distributed data.

4.3 Quality and performance metrics

The results of our experiments should be quantified in order to understand how good our prediction models are. To do that we use two groups of metrics. The first one defines the quality of classification. Metrics in this group are accuracy and Gini coefficient.

Accuracy or Fraction Correct (FC) is the simplest statistic for measuring the quality of binary classification. Accuracy should not be confused with Accuracy Ratio (AR), which is also called Gini coefficient. Accuracy it is the ratio of the number of correct classifications to the total number of correct or incorrect classifications:

$$\frac{True\ positive + True\ negative}{Total\ population} \tag{4.2}$$

Gini coefficient is a quantitative measure of the discriminatory power in classification models. Gini coefficient can be calculated using cumulative accuracy profile (CAP). The CAP of a model represents the cumulative number of positive outcomes along the y-axis versus the corresponding cumulative number of a classifying parameter along the x-axis. An example of CAP curve is shown in figure 4.1.

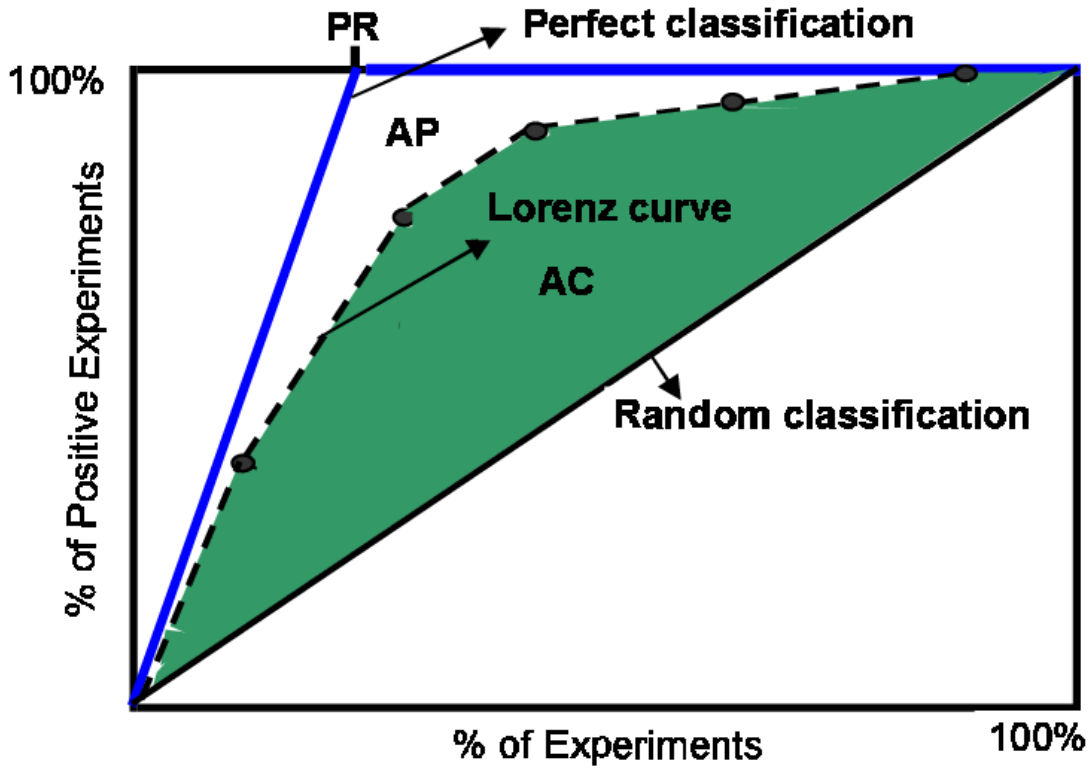

Figure 4.1: CAP curve (Credit www.researchgate.net webpage)

Gini coefficient is defined as the ratio of the area *AC* between the CAP of the validating model and the CAP of the random model, and the area *AP* between the CAP of the perfect

model and the CAP of the random model. However, for the purposes of the thesis we will use another formula for computation of Gini coefficient:

$$Gini = 2AUC - 1 \tag{4.3}$$

The reason for using this formula is that the *Scikit-learn* library allows to compute ROC AUC score only.

The second group of metrics are strategy performance metrics. Since our goal is to predict returns in order to make a trading decision we want to measure how good the strategy based on our predictions is.

Total return is return gained for all period when strategy operated. In our case this is a return gained for entire test period. Total return can be obtained as a sum of daily log returns. Return per annum is an annual return. It can be calculated as an average yearly log returns. Return per trade is an average daily log return.

The Sharpe ratio is the average return earned in excess of the risk-free rate per unit of volatility.

$$S = \frac{\bar{r}_t - r_f}{\sigma_r} \tag{4.4}$$

Sharpe ratio represents risk-adjusted return and allows to compare strategies with almost the same rate of return. Generally, the greater the value of the Sharpe ratio, the more attractive the risk-adjusted return.

Maximum Drawdown (MDD) is another measure of the risk of the strategy. MDD is the maximum loss from a peak to a trough of a portfolio, before a new peak is reached. MDD is calculated as a percentage:

$$MDD = \frac{Trough\ value\ - Peak\ value}{Peak\ value} \tag{4.5}$$

MDD is used to compare the relative riskiness of one strategy with another. Usually, the more maximum drawdown is the more risky the strategy is.

The last metric we use in the thesis is a p-value of the t-test. T-test is a statistical test. One sample t-test can be used for testing whether the mean of a population has a value specified in the null hypothesis. This is two-sided test therefore we can conduct a one-sided test to define whether the mean return is greater than 0 or not. The outcome of the t-test is *p*-value. P-value determines a probability that the null hypothesis is true with

some significance level α. If p-value is greater than α then the null hypothesis is not rejected. If the p-value is less than α, then the null hypothesis is rejected in favor of the alternative hypothesis. We set the significance level to 5% in our thesis. If the p-value from t-test conducted for our strategies is less than 5% then we can say that the average return per trade is greater than 0.

4.4 Results of time series models

As we already mentioned we fit ARMA and GARCH models on the train and window rolling data. It means that for each day of prediction we should add one day to the train data in order to extend the set on which we fit the models. But first we should detect the orders of ARMA and GARCH models. The common approach for GARCH models on stock markets is to use (1,1) orders.

In case of ARMA we can apply *auto.arima* function from the *forecast* package. This function selects the best model from the specified range of *p* and *q* orders by minimum AIC. By default the maximum value for *p* and *q* is 5. Empirically we identified that higher orders usually do not demonstrate lower AIC values so we keep default maximum ARMA orders. Then selected ARMA orders enter the specification of GARCH model which is available in *rugarch* package. This library allows to construct GARCH models and include ARMA as a mean model in its specification.

Once the specification is constructed we can carry out the fitting of the model using the *ugarchfit* function, which takes the specification object and numerical optimization solver. We have chosen hybrid regime, which tries different solvers in order to increase the likelihood of convergence. If the ARMA-GARCH model does not converge then we produce a positive prediction, which is obviously a guess, but in most cases the model converges.

The process of fitting ARMA-GARCH models is highly time consuming. It took approx. 15-20 minutes to make all predictions for the test data of a one ticker. When all predictions are computed we calculate all performance metrics and merge them into one table. We conducted the same procedure of calculating performance metrics for Buy&Hold as well in order to have the results as a benchmark. The results of the benchmark and ARMA-GARCH models ordered by accuracy are presented in the following tables.

Table 4.2: The results of Buy & Hold strategy

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
MCD	Buy & Hold	0.5610	-0.0703	0.6917	0.0009	0.2312	1.4475	-0.1152	0.0062
NVDA	Buy & Hold	0.5597	-0.0109	2.2960	0.0030	0.7674	1.9337	-0.1895	0.0004
BA	Buy & Hold	0.5570	-0.1099	0.9053	0.0012	0.3026	1.4177	-0.2910	0.0072
HON	Buy & Hold	0.5451	-0.0627	0.4861	0.0006	0.1625	0.9850	-0.1419	0.0444
INTC	Buy & Hold	0.5432	-0.0772	0.3365	0.0004	0.1126	0.5205	-0.3272	0.1843
AMZN	Buy & Hold	0.5418	-0.0346	1.3266	0.0018	0.4440	1.5773	-0.2019	0.0033
GOOGL	Buy & Hold	0.5345	-0.0179	0.6857	0.0009	0.2292	1.0490	-0.1092	0.0350
MSFT	Buy & Hold	0.5345	-0.0783	0.6865	0.0009	0.2294	1.0200	-0.1705	0.0390
WMT	Buy & Hold	0.5332	-0.1032	0.2158	0.0003	0.0721	0.3712	-0.4308	0.2605
PEP	Buy & Hold	0.5332	-0.0807	0.3160	0.0004	0.1056	0.8077	-0.1010	0.0814
AMGN	Buy & Hold	0.5279	0.0093	0.1546	0.0002	0.0517	0.2162	-0.2556	0.3542
KO	Buy & Hold	0.5265	-0.0799	0.1728	0.0002	0.0577	0.4518	-0.1212	0.2174
AAPL	Buy & Hold	0.5219	-0.0423	0.4786	0.0006	0.1602	0.6989	-0.3048	0.1137
JPM	Buy & Hold	0.5186	-0.0073	0.6141	0.0008	0.2053	0.9631	-0.2330	0.0481
IBM	Buy & Hold	0.5172	-0.0133	0.0628	0.0001	0.0210	0.1102	-0.3304	0.4244
JNJ	Buy & Hold	0.5106	0.0038	0.3724	0.0005	0.1245	0.8973	-0.1348	0.0605
XOM	Buy & Hold	0.5080	-0.0734	0.0061	0.0000	0.0020	0.0112	-0.2840	0.4923
CVX	Buy & Hold	0.5053	-0.1215	0.2269	0.0003	0.0758	0.3384	-0.4473	0.2792
PG	Buy & Hold	0.5027	-0.0525	0.1043	0.0001	0.0348	0.2511	-0.2774	0.3321
GE	Buy & Hold	0.4987	0.1206	-0.2795	-0.0004	-0.0934	-0.4688	-0.4557	0.7911

Table 4.3: The results of time series models

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
MCD	ARMA-GARCH	0.5623	-0.0397	0.6694	0.0009	0.2237	1.4006	-0.1267	0.0078
NVDA	ARMA-GARCH	0.5597	-0.0109	2.2960	0.0030	0.7674	1.9337	-0.1895	0.0004
HON	ARMA-GARCH	0.5517	-0.0395	0.5408	0.0007	0.1807	1.0964	-0.1436	0.0291
AMZN	ARMA-GARCH	0.5511	0.0209	1.1753	0.0016	0.3933	1.3960	-0.1834	0.0080
BA	ARMA-GARCH	0.5411	-0.1237	0.7148	0.0009	0.2389	1.1176	-0.3871	0.0268
WMT	ARMA-GARCH	0.5371	-0.0221	0.4787	0.0006	0.1600	0.8243	-0.1894	0.0772
GOOGL	ARMA-GARCH	0.5345	-0.0179	0.6857	0.0009	0.2292	1.0490	-0.1092	0.0350
PEP	ARMA-GARCH	0.5345	-0.0298	0.3903	0.0005	0.1305	0.9985	-0.1321	0.0423
MSFT	ARMA-GARCH	0.5252	-0.0369	0.5222	0.0007	0.1745	0.7752	-0.2362	0.0902
PG	ARMA-GARCH	0.5252	0.0061	0.3075	0.0004	0.1028	0.7413	-0.2528	0.1001
INTC	ARMA-GARCH	0.5232	-0.0946	0.2393	0.0003	0.0801	0.3701	-0.2697	0.2613
XOM	ARMA-GARCH	0.5186	-0.0245	0.0109	0.0000	0.0036	0.0200	-0.3009	0.4862
KO	ARMA-GARCH	0.5186	-0.0829	0.1116	0.0001	0.0373	0.2918	-0.1282	0.3070
AAPL	ARMA-GARCH	0.5166	-0.0458	0.5625	0.0007	0.1882	0.8217	-0.2478	0.0780
AMGN	ARMA-GARCH	0.5106	0.0067	0.3420	0.0005	0.1143	0.4784	-0.2265	0.2041
IBM	ARMA-GARCH	0.5066	-0.0288	0.1004	0.0001	0.0336	0.1762	-0.3446	0.3803
JPM	ARMA-GARCH	0.5053	-0.0070	0.3035	0.0004	0.1015	0.4754	-0.1642	0.2056
JNJ	ARMA-GARCH	0.5053	0.0039	0.2912	0.0004	0.0973	0.7010	-0.2266	0.1128
GE	ARMA-GARCH	0.5053	0.0740	0.0972	0.0001	0.0325	0.1630	-0.1860	0.3890
CVX	ARMA-GARCH	0.4960	-0.1288	-0.0644	-0.0001	-0.0215	-0.0960	-0.6901	0.5659

The largest accuracy was demonstrated by MCD ticker (McDonald's Corporation): 56.23%. The lowest accuracy has CVX (Chevron Corporation): 49.60%, meaning that model correctly predicted direction in less than 50% cases. Also we can see that the tickers from the top of the table mostly have p-values for t-test lower than 5%, meaning we can accept alternative hypothesis that average return per trade is greater than zero.

In order to investigate the dependencies between accuracy and strategy performance metrics better we can build a correlation matrix.

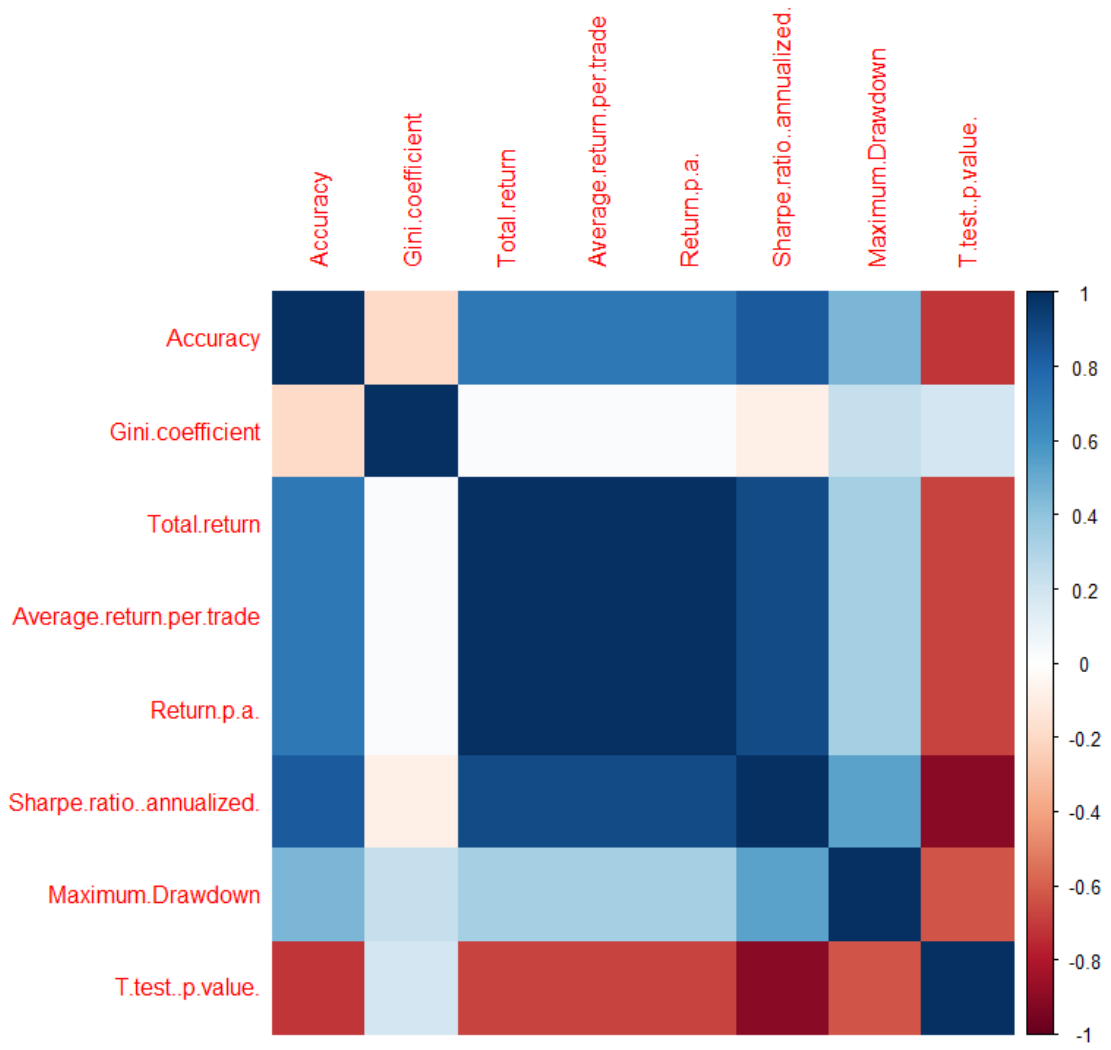

Figure 4.6: Correlation plot for performance metrics

As we can see from the plot there is a strong positive correlation between accuracy and returns gained from the strategy.

In order to visualize the results we can also plot cumulative returns of ARMA-GARCH model in comparison to Buy & Hold strategy over entire test period for several best and worst performed tickers.

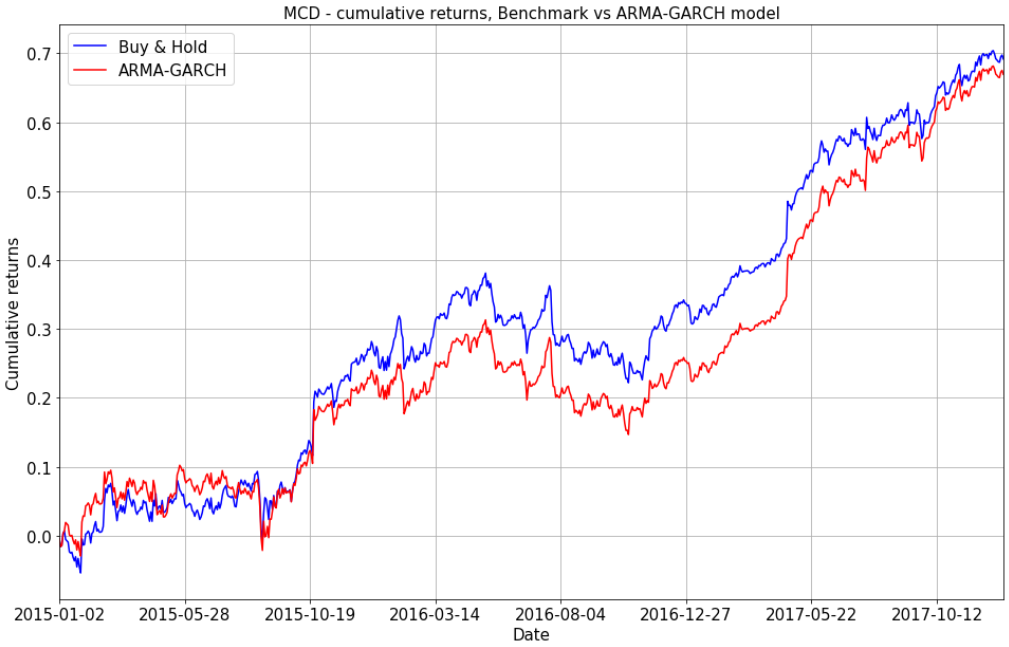

Figure 4.2: Cumulative returns of the stock with the largest accuracy - MCD

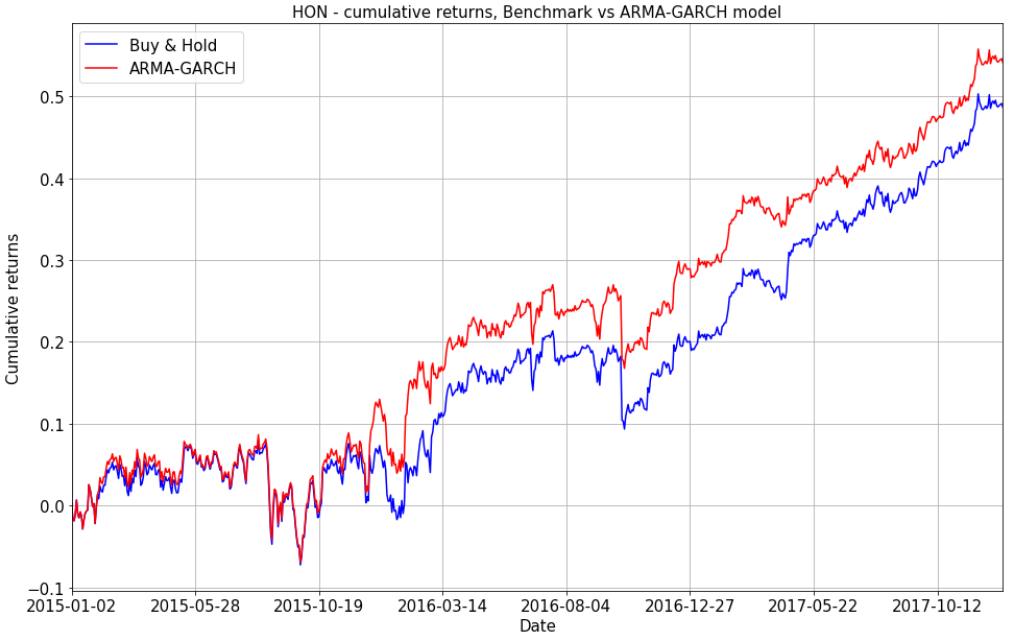

Figure 4.3: Cumulative returns of the HON stock

As we can from the plots above stocks with the most accurate models almost replicated the curve for Buy & Hold strategy.

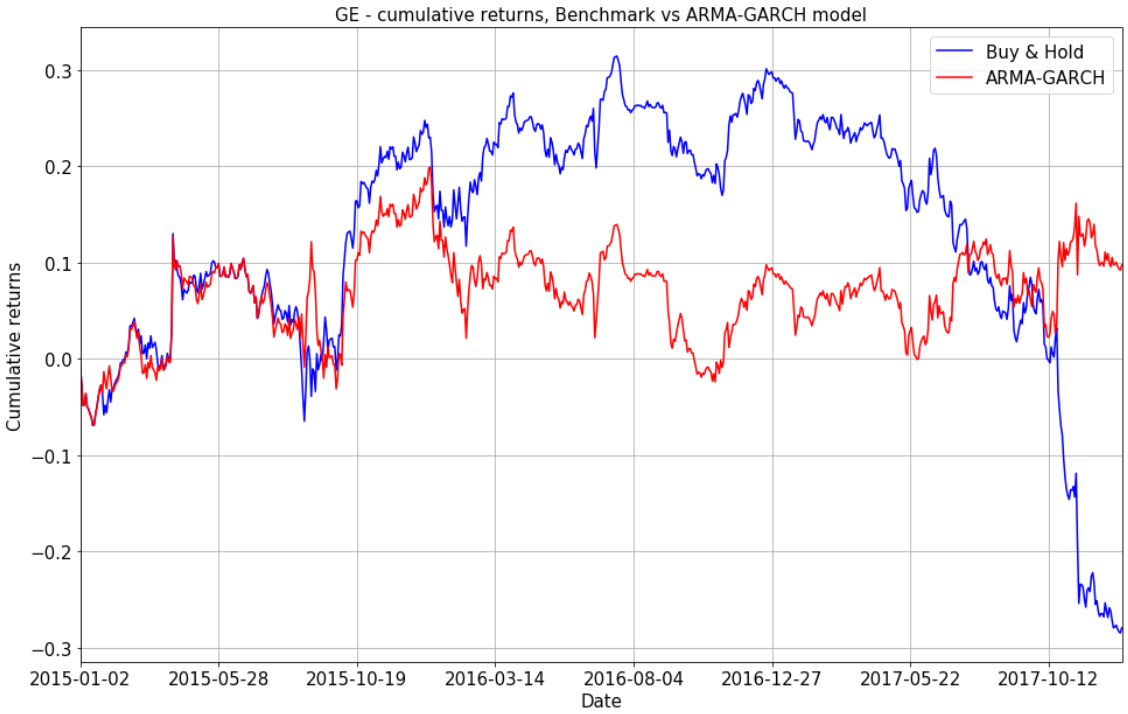

Figure 4.4: Cumulative returns of the GE stock

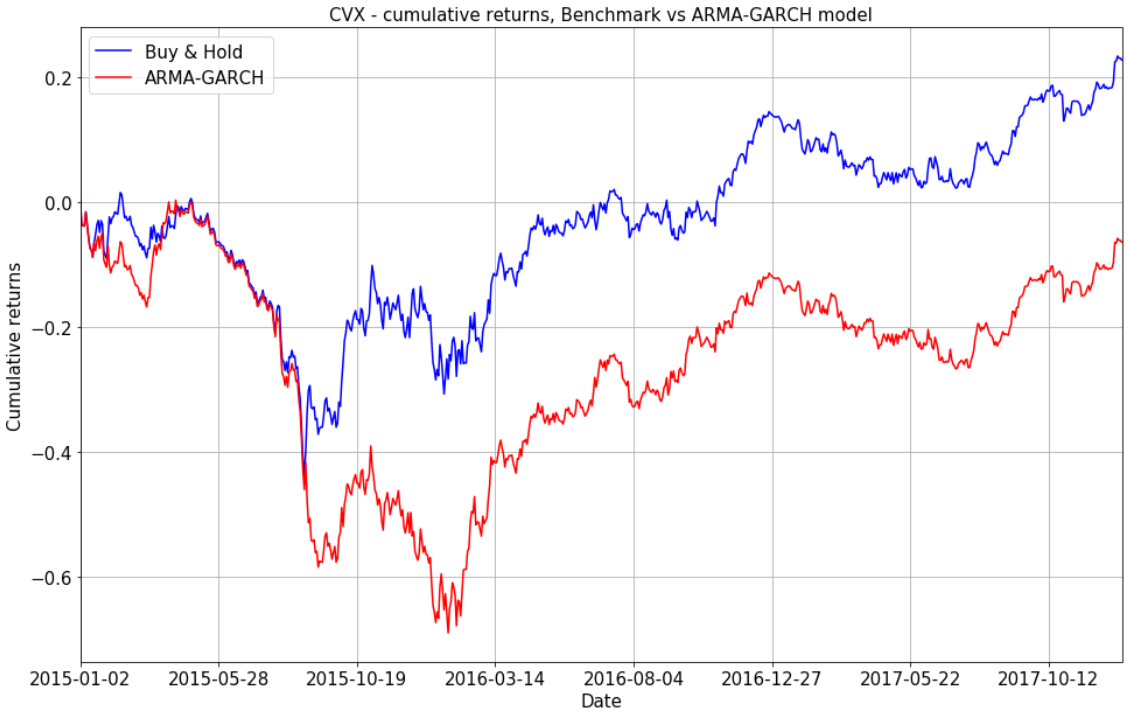

Figure 4.5: Cumulative returns of the CVX stock

ARMA-GARCH strategy for the CVX stock could not demonstrate higher total return than Buy & Hold strategy, but in case of GE ticker, the model was able to reach better performance than the benchmark despite low value of accuracy.

We can also calculate performance metrics for a portfolio of all stocks in order to see how good our models are in terms of portfolio trading. Prices of our stocks do not lay on the same scale and much different from each other. Therefore we construct a portfolio in such a way that initial investment to each stock is approx. 1000 USD. Then we apply trading rules according to our models separately for each stock and recalculate the value if the portfolio for each day in the testing period. It allows us to derive returns for the whole portfolio and to compute performance metrics. Metrics for measuring the quality of classification are not applicable in this case.

Performance metrics of the portfolio for Buy & Hold and ARMA-GARCH strategies are shown in the table 4.4.

Table 4.4: Performance metrics of the portfolio for ARMA-GARCH and benchmark strategies

Model	**Total return**	**Average return per trade**	**Return p.a.**
Benchmark (Buy&Hold)	0.6927	0.0009	0.2315
ARMA-GARCH	0.6744	0.0009	0.2254
Model	**Sharpe ratio (annualized)**	**Maximum Drawdown**	**T-test (p-value)**
Benchmark (Buy&Hold)	1.6256	-0.1196	0.0025
ARMA-GARCH	1.7747	-0.1088	0.0011

We can see that Buy & Hold strategy seems to be more profitable in a portfolio view. On the other hand, maximum drawdown is lower in ARMA-GARCH models and Sharpe ratio is a bit higher. This tells us that the strategy based on time series models' predictions could be considered as less risky strategy. However, the results of both strategies are very close to each other.

4.5 Results of machine learning methods

First, we build a logistic regression model. Using *Scikit-learn* package the process of building is straightforward. In order to find best parameters of the model we use 10-fold cross-validation. Then we select parameters that demonstrated the best mean accuracy score. The only parameter we optimize is L2-regularization term called C. Lower values of C means stronger regularization. Therefore we specify a range of the regularizer as a linear space of 20 values between 0.001 and 1. Results of the parameter selection and respective validation and test accuracies for all stocks are shown in the table 4.4.

Table 4.5: The best regularization parameters for all stocks

Ticker	**C**	**Validation accuracy**	**Test accuracy**
AAPL	0.8423	0.5161	0.5265
AMZN	0.0010	0.5132	0.5424
GOOGL	0.7371	0.5249	0.4861
MSFT	0.6845	0.5098	0.4662
JPM	0.0010	0.5078	0.5285
JNJ	0.7371	0.5180	0.5099
XOM	0.0536	0.5141	0.5563
WMT	0.2113	0.5132	0.5073
INTC	1.0000	0.5137	0.5504
CVX	0.0536	0.5273	0.5139
IBM	0.4742	0.5083	0.5232
PG	1.0000	0.5161	0.4967
BA	0.3165	0.5102	0.5536
KO	0.1062	0.5249	0.5086
PEP	0.2639	0.5220	0.4914
NVDA	0.9474	0.5244	0.4808
MCD	0.0010	0.5356	0.5616
AMGN	0.2639	0.5049	0.5430
GE	1.0000	0.5224	0.4993
HON	0.1062	0.5068	0.5391

Selected parameters are then used to train the models on the entire train dataset and to make predictions for the test data. Performance metrics of the logistic regression models for all stocks ordered by accuracy are shown in the table 4.6.

The best accuracy was gained by MCD ticker as in the time series models. But among top of the list we can also see XOM and INTC tickers, which improved their accuracy against time series models from 51.86% to 55.57% and from 52.32% to 54.98% respectively. On the other hand, NVDA and MSFT tickers dropped to the down demonstrating accuracy lower than 50%.

In order to see an impact of changed classification quality on profitability we can plot cumulative returns of the strategy in comparison to the benchmark for several stocks.

We can see in figures 4.6 and 4.7 that the logistic regression model for XOM and INTC stocks managed to significantly outperform Buy & Hold strategy.

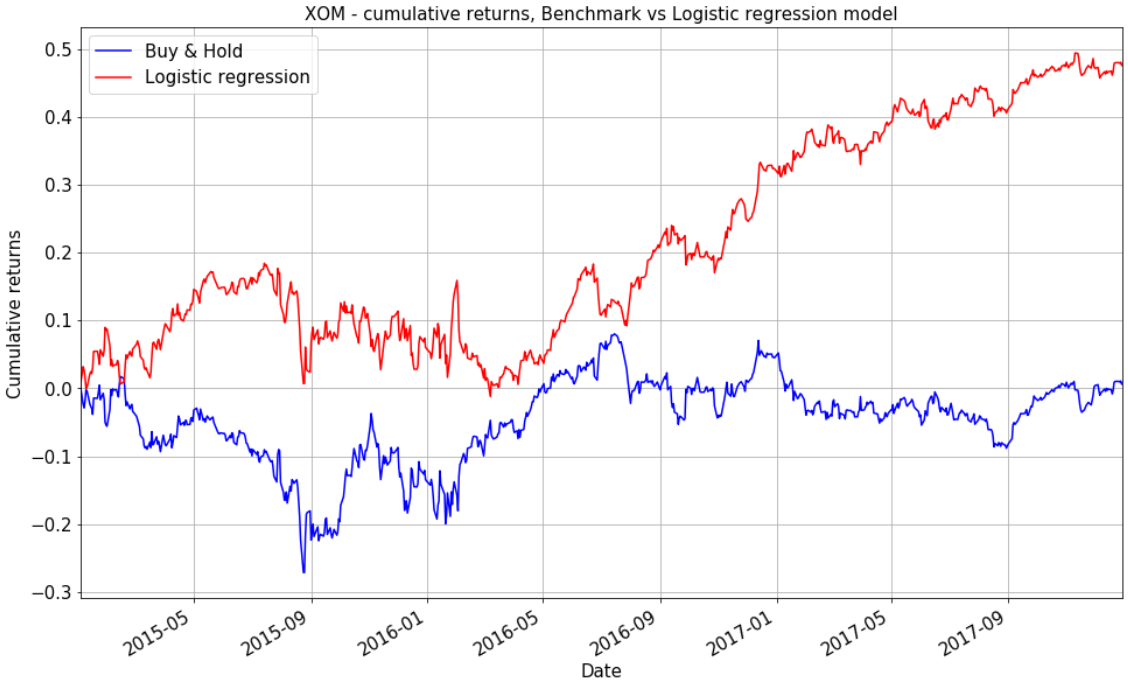

Figure 4.6: Cumulative returns of the XOM stock

Table 4.6: The results of logistic regression models

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
MCD	Logistic regression	0.5610	0.0033	0.6308	0.0008	0.2108	1.3193	-0.1216	0.0114
XOM	Logistic regression	0.5557	0.1083	0.4757	0.0006	0.1590	0.8735	-0.1657	0.0656
BA	Logistic regression	0.5531	0.0039	0.9158	0.0012	0.3061	1.4342	-0.2456	0.0067
INTC	Logistic regression	0.5498	0.0464	0.7714	0.0010	0.2582	1.1962	-0.1659	0.0195
AMGN	Logistic regression	0.5424	0.0564	0.5018	0.0007	0.1677	0.7023	-0.2580	0.1124
AMZN	Logistic regression	0.5418	0.0085	0.8033	0.0011	0.2688	0.9522	-0.2888	0.0501
HON	Logistic regression	0.5385	0.0028	0.3253	0.0004	0.1087	0.6585	-0.2596	0.1275
JPM	Logistic regression	0.5279	0.0218	0.7576	0.0010	0.2532	1.1893	-0.2552	0.0200
AAPL	Logistic regression	0.5272	0.0298	0.7556	0.0010	0.2529	1.1049	-0.1398	0.0283
IBM	Logistic regression	0.5225	0.0311	0.4030	0.0005	0.1347	0.7076	-0.3208	0.1107
CVX	Logistic regression	0.5133	0.0197	0.4633	0.0006	0.1548	0.6916	-0.5376	0.1160
JNJ	Logistic regression	0.5093	-0.0024	0.3747	0.0005	0.1252	0.9027	-0.1124	0.0594
KO	Logistic regression	0.5080	-0.0260	0.1526	0.0002	0.0510	0.3990	-0.1264	0.2452
WMT	Logistic regression	0.5066	-0.0275	-0.3170	-0.0004	-0.1059	-0.5455	-0.8446	0.8271
GE	Logistic regression	0.5000	-0.0010	-0.3110	-0.0004	-0.1039	-0.5216	-0.3928	0.8164
PG	Logistic regression	0.4960	-0.0095	-0.0795	-0.0001	-0.0266	-0.1914	-0.1775	0.6297
PEP	Logistic regression	0.4920	-0.0340	0.0309	0.0000	0.0103	0.0789	-0.1778	0.4457
GOOGL	Logistic regression	0.4867	-0.0162	0.3333	0.0004	0.1114	0.5090	-0.3086	0.1894
NVDA	Logistic regression	0.4814	0.0195	-1.1944	-0.0016	-0.3992	-1.0005	-1.2405	0.9580
MSFT	Logistic regression	0.4668	0.0003	-0.5218	-0.0007	-0.1744	-0.7747	-0.6179	0.9097

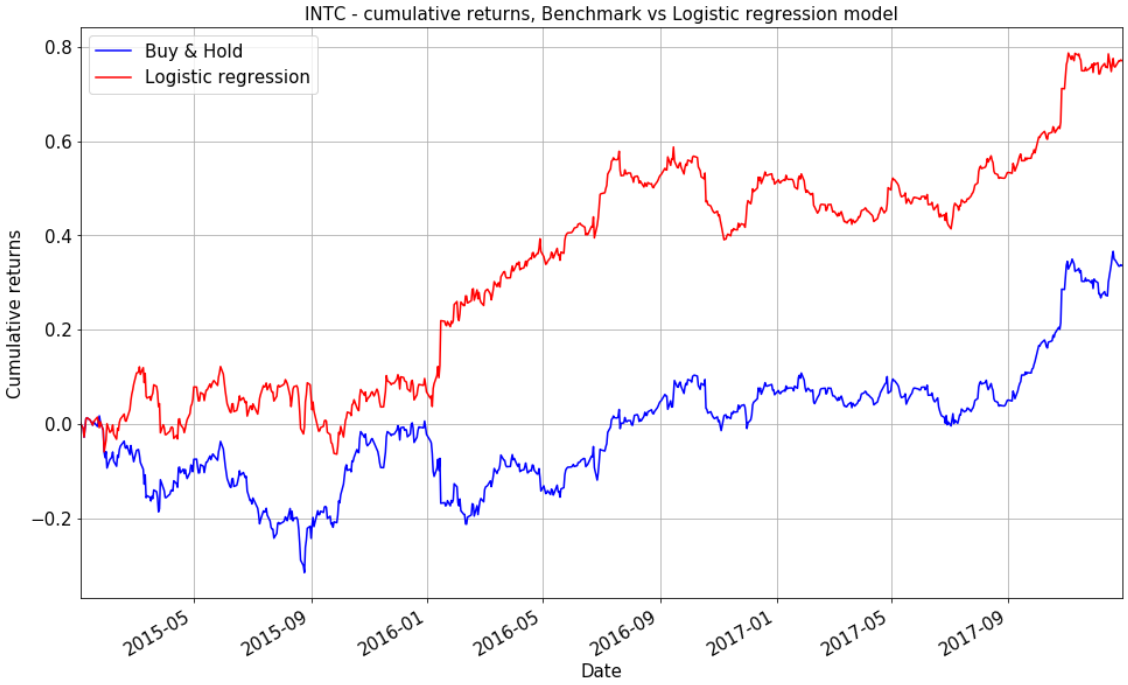

Figure 4.7: Cumulative returns of the INTC stock

However, NVDA and MSFT stocks demonstrated very poor performance.

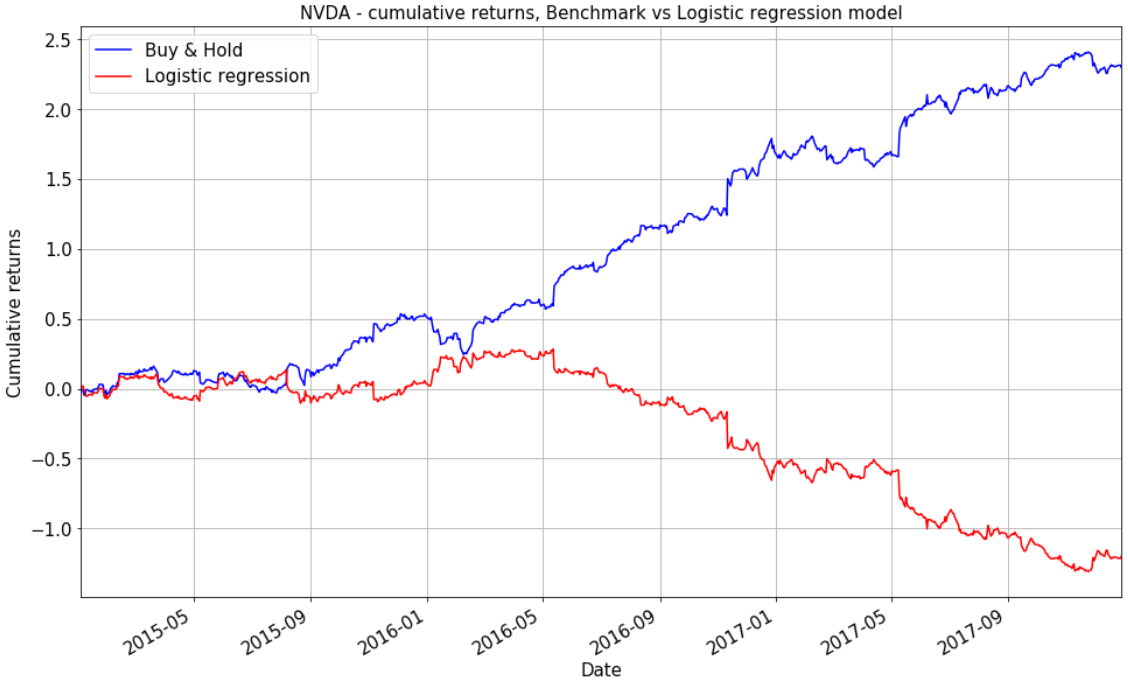

Figure 4.8: Cumulative returns of the NVDA stock

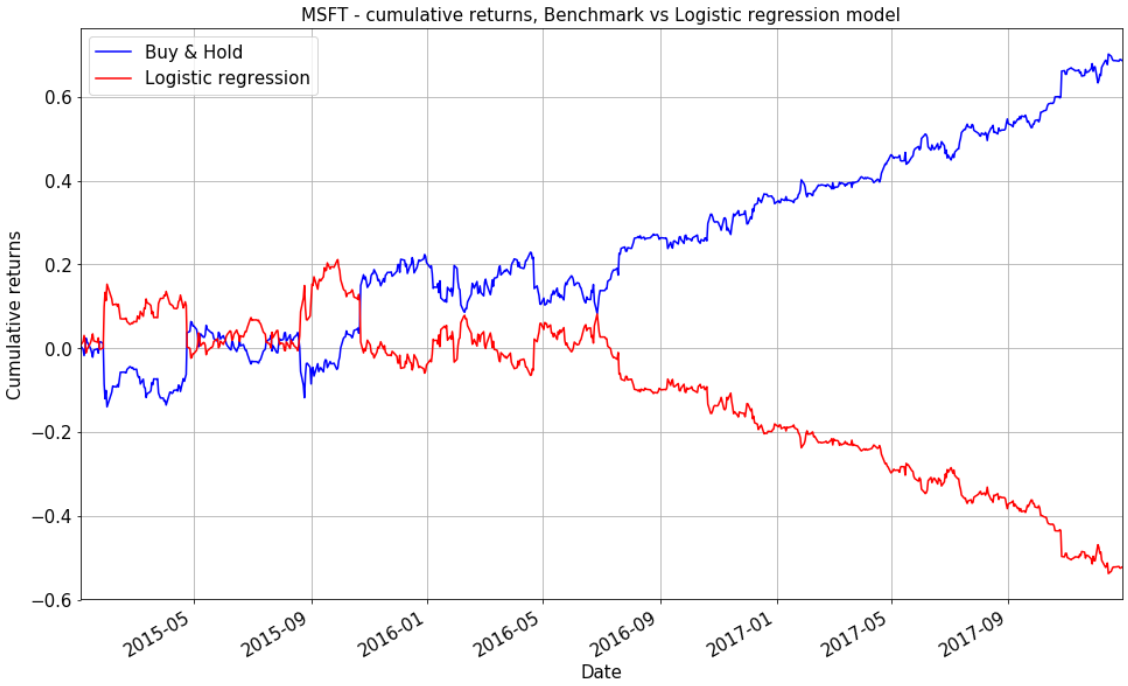

Figure 4.9: Cumulative returns of the MSFT stock

Cumulative returns for them are almost mirrored with respect to the benchmark. The reason of such poor performance may be the fact that returns in the train period were mostly negative and the model is not able to predict positive directions.

Portfolio performance of the logistic regression models in comparison with the benchmark are shown in table 4.7.

Table 4.7: Performance metrics of the portfolio for logistic regression and benchmark strategies

Model	**Total return**	**Average return per trade**	**Return p.a.**
Benchmark (Buy&Hold)	0.6927	0.0009	0.2315
Logistic regression	0.3723	0.0005	0.1244
Model	**Sharpe ratio (annualized)**	**Maximum Drawdown**	**T-test (p-value)**
Benchmark (Buy&Hold)	1.6256	-0.1196	0.0025
Logistic regression	1.7331	-0.0620	0.0014

In the table above we observe that logistic regression models demonstrated poorer performance. However, the maximum drawdown is almost 2 times lower. This results in higher Sharpe ratio compared to the benchmark. Therefore the strategy based on logistic regression can be considered as less risky providing higher risk-adjusted returns.

The next model is gradient boosted trees (GBT). We use *LightGBM* package to build the trees. This package is built on very effective algorithms that makes the process of training models faster than in other packages. GBT have more input parameters and process of finding optimal values for them can take a huge amount of time. Therefore we kept some parameters constant and tried to find optimal values only for two of them. Our goal during parameters selection is prevent overfitting so we set the maximum depth of the tree to 3, number of leaves to 5 and number of trees to 1000. Higher number of trees usually increase the generalization ability of the model. Also we set subsample ratio to 0.8, which is the fraction of observation to be selected for each tree. The two parameters we are selecting are learning rate for boosting algorithm and *colsample_bytree* – subsample ratio of columns for constructing each tree. Results of GBT's predictions are shown in the table 4.8.

At first glance, obviously there is a poorer overall performance. Almost half of the stocks achieved accuracy lower than 50% and only three stocks have an average return per trade greater than 0 according to the results of t-test. MCD stock, which had the best accuracy in time series and logistic regression models now shows the worst. Similarly, AMZN and BA tickers dropped from the top to the down. However, AAPL stock significantly improved accuracy and demonstrated the highest total return among all the stocks.

Table 4.8: The results of gradient boosted trees

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
INTC	LightGBM	0.5511	0.0549	0.7999	0.0011	0.2677	1.2406	-0.2290	0.0162
AMGN	LightGBM	0.5504	0.0857	0.3868	0.0005	0.1293	0.5412	-0.2896	0.1747
HON	LightGBM	0.5451	0.0314	0.5974	0.0008	0.1997	1.2118	-0.1212	0.0182
AAPL	LightGBM	0.5445	0.1025	1.0314	0.0014	0.3452	1.5114	-0.1465	0.0046
IBM	LightGBM	0.5212	0.0220	0.1615	0.0002	0.0540	0.2834	-0.3305	0.3121
XOM	LightGBM	0.5199	0.0394	0.0074	0.0000	0.0025	0.0136	-0.2127	0.4906
WMT	LightGBM	0.5106	0.0174	0.0030	0.0000	0.0010	0.0052	-0.2645	0.4964
JNJ	LightGBM	0.5093	-0.0021	0.3262	0.0004	0.1090	0.7857	-0.1547	0.0873
KO	LightGBM	0.5027	-0.0321	0.1716	0.0002	0.0574	0.4488	-0.1338	0.2189
PG	LightGBM	0.5000	0.0000	0.1712	0.0002	0.0572	0.4125	-0.1488	0.2379
PEP	LightGBM	0.5000	0.0254	0.0389	0.0001	0.0130	0.0993	-0.2570	0.4318
NVDA	LightGBM	0.4920	0.0115	-1.1626	-0.0015	-0.3886	-0.9738	-1.1718	0.9537
GE	LightGBM	0.4854	-0.0295	-0.1715	-0.0002	-0.0573	-0.2875	-0.4463	0.6904
MSFT	LightGBM	0.4841	0.0171	0.0779	0.0001	0.0260	0.1155	-0.3894	0.4209
JPM	LightGBM	0.4814	-0.0341	-0.4120	-0.0005	-0.1377	-0.6455	-0.5363	0.8677
GOOGL	LightGBM	0.4801	0.0041	-0.4735	-0.0006	-0.1582	-0.7235	-0.5511	0.8944
CVX	LightGBM	0.4801	-0.0422	-0.2488	-0.0003	-0.0831	-0.3711	-0.7129	0.7394
BA	LightGBM	0.4775	0.0441	-0.2803	-0.0004	-0.0937	-0.4373	-0.5565	0.7752
AMZN	LightGBM	0.4622	-0.0038	-1.0278	-0.0014	-0.3440	-1.2196	-1.0452	0.9823
MCD	LightGBM	0.4536	0.0142	-0.3443	-0.0005	-0.1151	-0.7184	-0.5107	0.8928

Plots of cumulative returns of several stock are shown on the following figures.

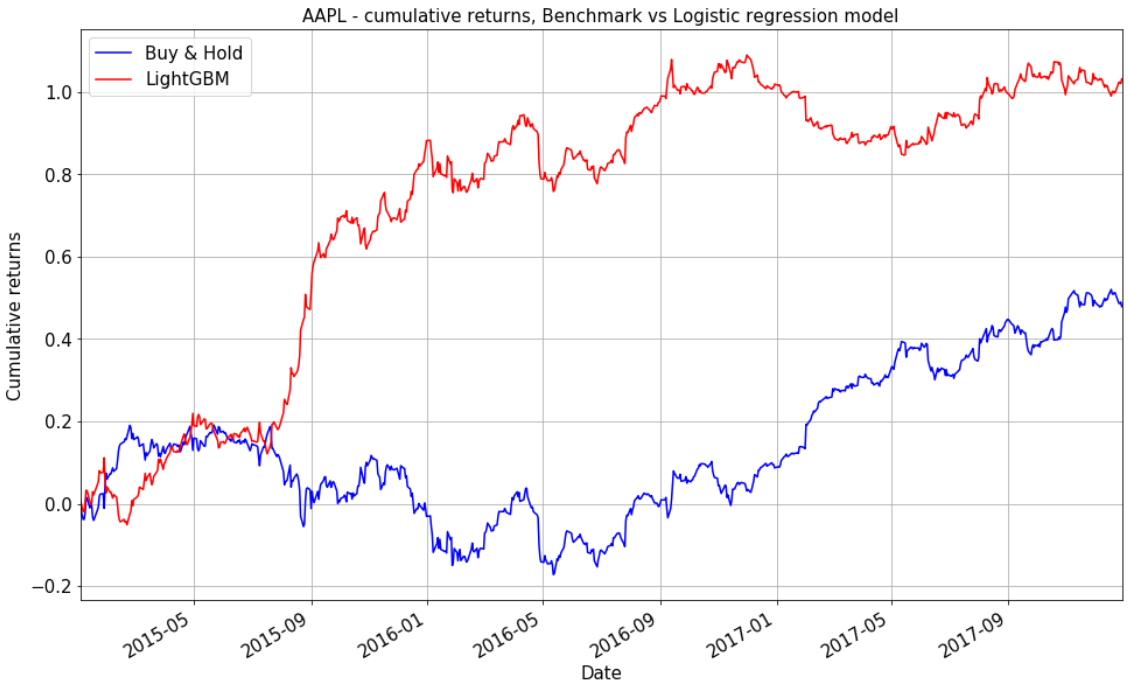

Figure 4.10: Cumulative returns of the AAPL stock

We can notice from the plot of the AAPL cumulated returns that the strategy based on GBT gained the major return during downward trend of the actual returns. In the same time there is a negative performance during the upward trend at the beginning of 2017. The reason of that the model learnt to predict downward directions because the training data can possess higher fraction of negative returns than positive.

In figure 4.11 we see that at the beginning of the test period the LightGBM model demonstrated higher returns against benchmark but then started to fail when cumulative returns of the benchmark went up.

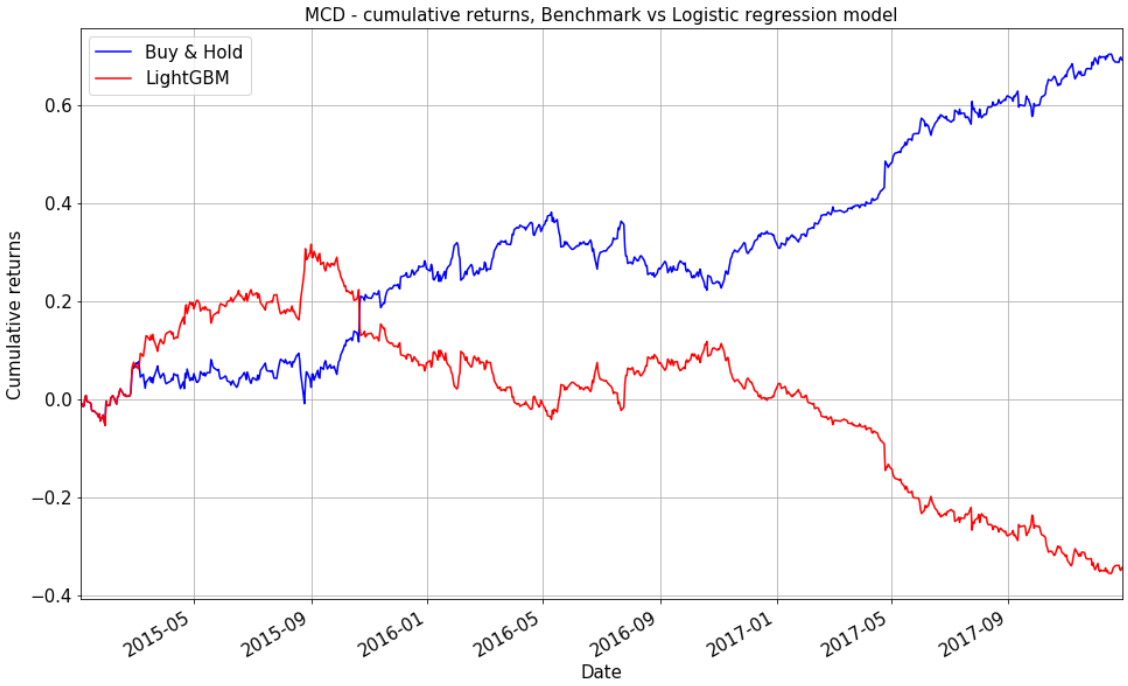

Figure 4.11: Cumulative returns of the MCD stock

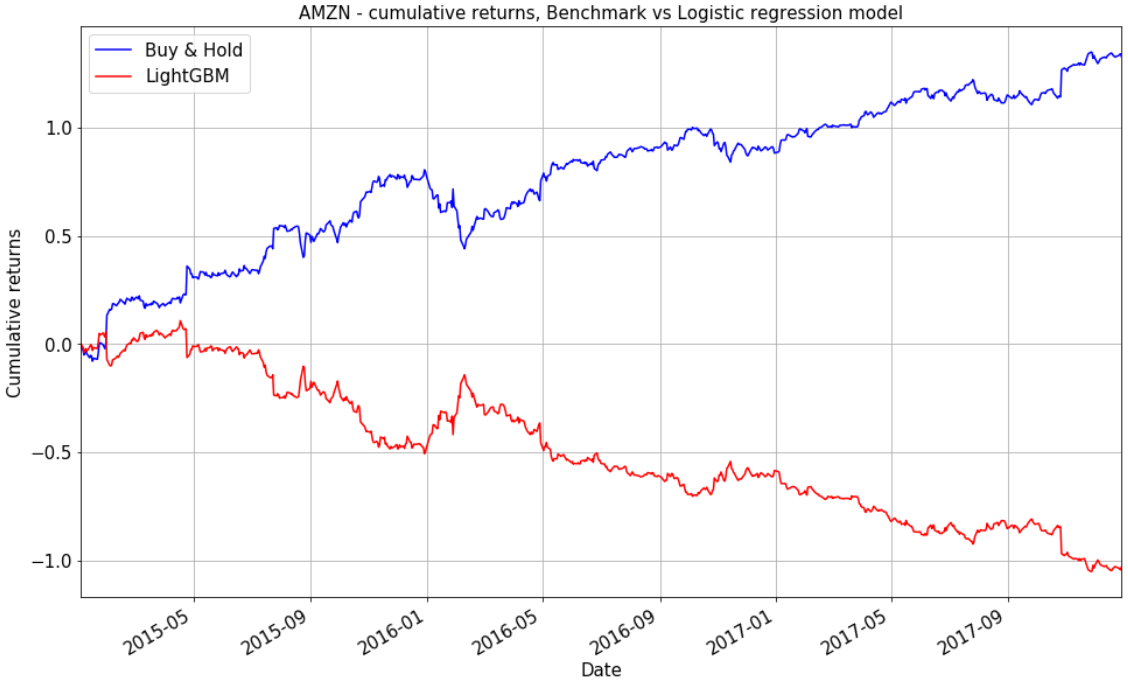

Figure 4.12: Cumulative returns of the AMZN stock

We see the similar picture with the AMZN stock. Again, the reason of such behavior of the model may lie in the different structure of returns in the train and test data.

We can compare performance metrics of the portfolio for LightGBM and benchmark strategies in the table 4.9.

Table 4.9: Performance metrics of the portfolio for LightGBM and benchmark strategies

Model	**Total return**	**Average return per trade**	**Return p.a.**
Benchmark (Buy&Hold)	0.6927	0.0009	0.2315
LightGBM	0.1189	0.0002	0.0398
Model	**Sharpe ratio (annualized)**	**Maximum Drawdown**	**T-test (p-value)**
Benchmark (Buy&Hold)	1.6256	-0.1196	0.0025
LightGBM	0.8006	-0.0740	0.0833

Total return of the strategy based on LightGBM's prediction is much lower that the benchmark. According to T-test the average daily return of LightGBM model is not higher than 0 on the 95% confidence level.

The last machine learning method that we used is Long Short-Term Memory neural network. We used *Keras* API with *Tensorflow* library as a backend. Our neural network consists of one LSTM layer and one sigmoid layer, which takes an output of the LSTM layer and makes a binary prediction of the next day's direction. Our input training data should be a three dimensional tensor in the following format [samples, timesteps, features]. Samples is a number of our observations, timesteps is a size of a look-back window. After several experiments we set timesteps parameter to 10 (this represent two business weeks) as a relatively stable window for different time structures of the stocks. We also fixed a size of batch and number of epochs in order to have training time on an acceptable level. Two parameters that we wanted to optimize are dropout and number of units in the LSTM layer. The dropout represents a fraction of the units to drop for the linear transformation of the inputs and of the recurrent state. This parameter is needed to prevent early overfitting. We used for dropout values of 0.2, 0.4, 0.6 and number of units were 5, 10, 20. The time needed to find optimal parameters and make predictions for all stocks was almost 4.5 hours. Results of LSTM neural networks are presented in the table 4.10.

Table 4.10: The results of LSTM neural networks

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
MCD	LSTM	0.5477	0.0473	0.5078	0.0007	0.1697	1.0606	-0.0927	0.0335
HON	LSTM	0.5477	0.0087	0.5598	0.0007	0.1871	1.1351	-0.1713	0.0250
BA	LSTM	0.5451	-0.0079	0.7519	0.0010	0.2513	1.1760	-0.3235	0.0211
INTC	LSTM	0.5445	0.0038	0.3113	0.0004	0.1042	0.4815	-0.3514	0.2027
PEP	LSTM	0.5332	0.0004	0.3422	0.0005	0.1144	0.8750	-0.0784	0.0653
AMGN	LSTM	0.5318	0.0120	0.1137	0.0002	0.0380	0.1590	-0.3450	0.3917
GOOGL	LSTM	0.5305	-0.0008	0.6329	0.0008	0.2115	0.9680	-0.1346	0.0472
IBM	LSTM	0.5239	0.0458	0.2079	0.0003	0.0695	0.3647	-0.2330	0.2642
WMT	LSTM	0.5212	-0.0075	-0.1429	-0.0002	-0.0477	-0.2457	-0.4119	0.6645
JPM	LSTM	0.5199	0.0028	0.6255	0.0008	0.2090	0.9810	-0.2230	0.0451
JNJ	LSTM	0.5199	0.0199	0.4010	0.0005	0.1340	0.9663	-0.2517	0.0475
AAPL	LSTM	0.5193	-0.0037	0.3035	0.0004	0.1016	0.4429	-0.4406	0.2221
KO	LSTM	0.5186	-0.0083	0.2387	0.0003	0.0798	0.6246	-0.0847	0.1402
AMZN	LSTM	0.5139	-0.0260	0.0495	0.0001	0.0166	0.0586	-0.3696	0.4597
GE	LSTM	0.5119	0.0227	0.3001	0.0004	0.1003	0.5034	-0.1903	0.1921
XOM	LSTM	0.5053	0.0068	0.3214	0.0004	0.1074	0.5896	-0.4323	0.1541
PG	LSTM	0.4775	-0.0478	-0.3020	-0.0004	-0.1009	-0.7282	-0.4158	0.8959
MSFT	LSTM	0.4708	-0.0191	-0.0428	-0.0001	-0.0143	-0.0635	-0.3609	0.5437
CVX	LSTM	0.4682	-0.0683	-0.1427	-0.0002	-0.0477	-0.2128	-0.4363	0.6436
NVDA	LSTM	0.4576	-0.0289	-0.9308	-0.0012	-0.3111	-0.7791	-1.0329	0.9109

The results in table clearly show that LSTM networks could not reach better accuracy and performance than other models. The highest accuracy score 54.77% has MCD ticker but it is less than in logistic regression and ARMA-GARCH models. At the down of the list we again can see MSFT and NVDA stocks. Highly likely that these results that demonstrated each machine learning method can be explained by different patterns in train and test data for these tickers.

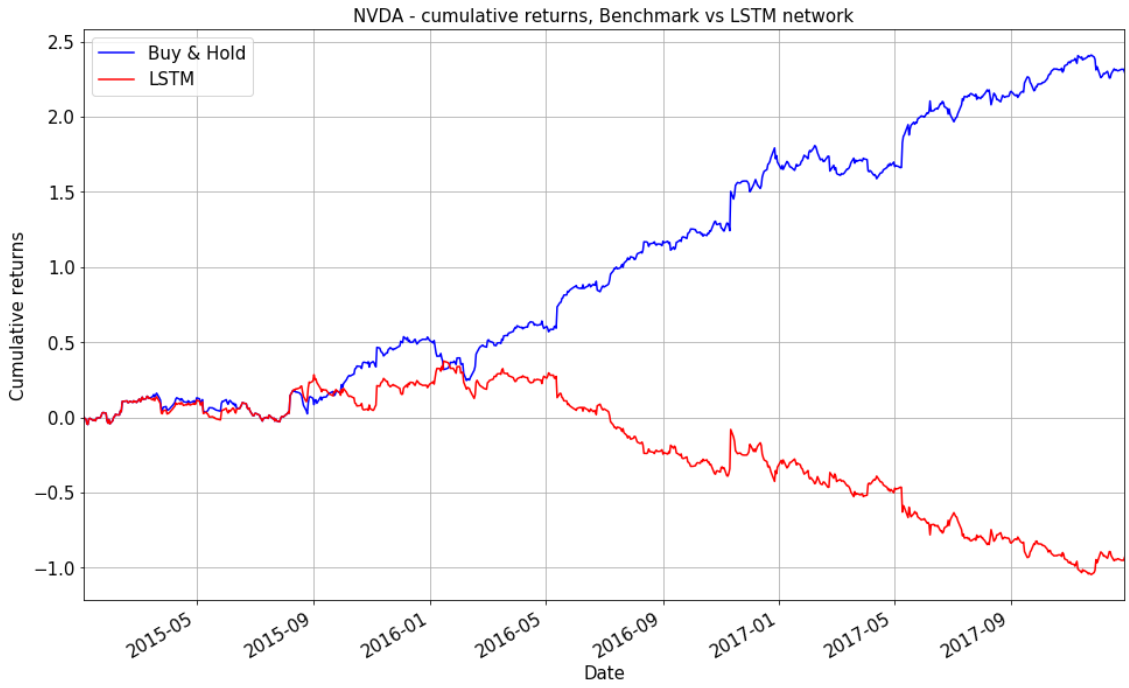

Figure 4.13: Cumulative returns of the NVDA stock

We see on the plot above that behavior of cumulative returns of NVDA stock for the LSTM network is very similar to other machine learning models.

When we take a look at cumulative returns of stocks with highest accuracy we can observe that LSTM networks make their predictions very close to Buy & Hold strategy. This case is shown in figure 4.14.

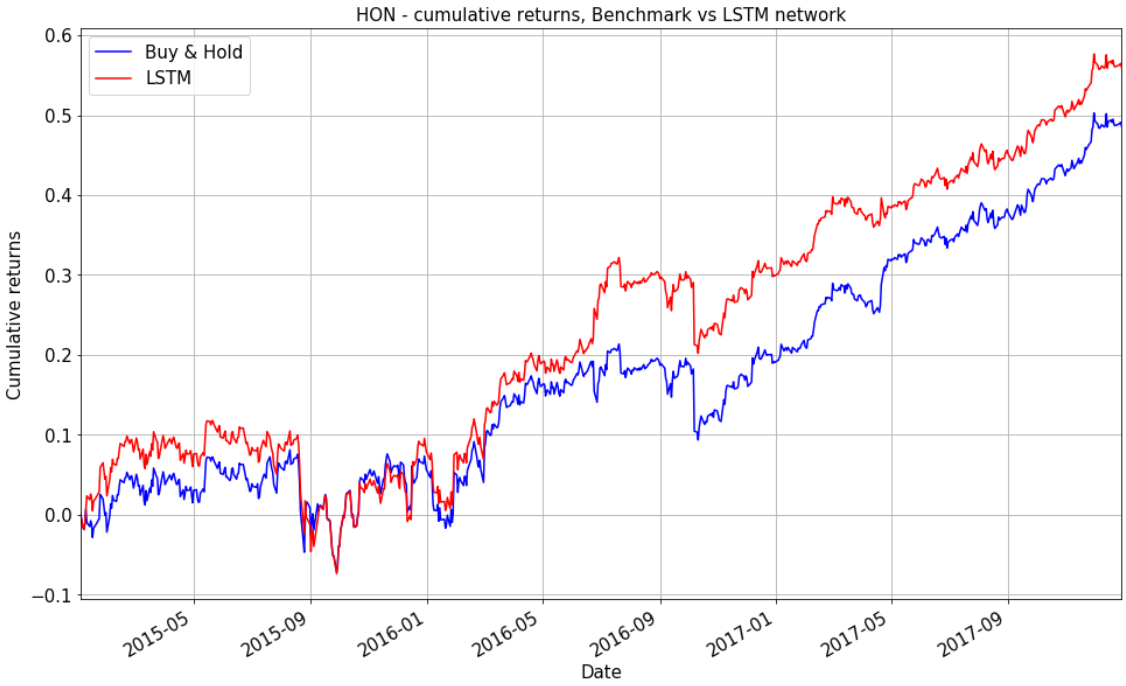

Figure 4.14. Cumulative returns of the HON stock

Portfolio results for LSTM networks in comparison to the benchmark are shown in the table 4.11.

Table 4.11: Performance metrics of the portfolio for LSTM and benchmark strategies

Model	**Total return**	**Average return per trade**	**Return p.a.**
Benchmark (Buy&Hold)	0.6927	0.0009	0.2315
LSTM	0.3234	0.0004	0.1081
Model	**Sharpe ratio (annualized)**	**Maximum Drawdown**	**T-test (p-value)**
Benchmark (Buy&Hold)	1.6256	-0.1196	0.0025
LSTM	1.3919	-0.1372	0.0081

Profitability of LSTM networks is not as good as the benchmark. In the same time this is the only one model that demonstrated higher maximum drawdown compared to Buy & Hold.

In order to compare all models we created merged table with all results in the following pages.

Table 4.12: The results of all models and the benchmark strategy

Ticker	Model	Accuracy	Gini coefficient	Total return	Average return per trade	Return p.a.	Sharpe ratio (annualized)	Maximum Drawdown	T-test (p-value)
AAPL	Buy & Hold	0.5219	-0.0423	0.4786	0.0006	0.1602	0.6989	-0.3048	0.1137
	ARMA-GARCH	0.5166	-0.0458	0.5625	0.0007	0.1882	0.8217	-0.2478	0.0780
	Logistic regression	0.5299	0.0346	0.8076	0.0011	0.2703	1.1814	-0.1298	0.0207
	LightGBM	0.5445	0.1025	1.0314	0.0014	0.3452	1.5114	-0.1465	0.0046
	LSTM	0.5193	-0.0037	0.3035	0.0004	0.1016	0.4429	-0.4406	0.2221
AMZN	Buy & Hold	0.5418	-0.0346	1.3266	0.0018	0.4440	1.5773	-0.2019	0.0033
	ARMA-GARCH	0.5511	0.0209	1.1753	0.0016	0.3933	1.3960	-0.1834	0.0080
	Logistic regression	0.5418	0.0085	0.8033	0.0011	0.2688	0.9522	-0.2888	0.0501
	LightGBM	0.4622	-0.0038	-1.0278	-0.0014	-0.3440	-1.2196	-1.0452	0.9823
	LSTM	0.5139	-0.0260	0.0495	0.0001	0.0166	0.0586	-0.3696	0.4597
GOOGL	Buy & Hold	0.5345	-0.0179	0.6857	0.0009	0.2292	1.0490	-0.1092	0.0350
	ARMA-GARCH	0.5345	-0.0179	0.6857	0.0009	0.2292	1.0490	-0.1092	0.0350
	Logistic regression	0.5053	0.0171	0.5386	0.0007	0.1800	0.8233	-0.1979	0.0774
	LightGBM	0.4801	0.0041	-0.4735	-0.0006	-0.1582	-0.7235	-0.5511	0.8944
	LSTM	0.5305	-0.0008	0.6329	0.0008	0.2115	0.9680	-0.1346	0.0472
MSFT	Buy & Hold	0.5345	-0.0783	0.6865	0.0009	0.2294	1.0200	-0.1705	0.0390
	ARMA-GARCH	0.5358	-0.0237	0.2823	0.0004	0.0943	0.4187	-0.2271	0.2346
	Logistic regression	0.4602	-0.0184	-0.8752	-0.0012	-0.2925	-1.3021	-0.9024	0.9877
	LightGBM	0.4841	0.0171	0.0779	0.0001	0.0260	0.1155	-0.3894	0.4209
	LSTM	0.4708	-0.0191	-0.0428	-0.0001	-0.0143	-0.0635	-0.3609	0.5437

JPM	Buy & Hold	0.5186	-0.0073	0.6141	0.0008	0.2053	0.9631	-0.2330	0.0481
	ARMA-GARCH	0.5053	-0.0070	0.3035	0.0004	0.1015	0.4754	-0.1642	0.2056
	Logistic regression	0.5385	0.0607	0.4782	0.0006	0.1598	0.7494	-0.2081	0.0976
	LightGBM	0.4814	-0.0341	-0.4120	-0.0005	-0.1377	-0.6455	-0.5363	0.8677
	LSTM	0.5199	0.0028	0.6255	0.0008	0.2090	0.9810	-0.2230	0.0451
JNJ	Buy & Hold	0.5106	0.0038	0.3724	0.0005	0.1245	0.8973	-0.1348	0.0605
	ARMA-GARCH	0.5053	0.0039	0.2912	0.0004	0.0973	0.7010	-0.2266	0.1128
	Logistic regression	0.5093	-0.0003	0.2995	0.0004	0.1001	0.7212	-0.1747	0.1063
	LightGBM	0.5093	-0.0021	0.3262	0.0004	0.1090	0.7857	-0.1547	0.0873
	LSTM	0.5199	0.0199	0.4010	0.0005	0.1340	0.9663	-0.2517	0.0475
XOM	Buy & Hold	0.5080	-0.0734	0.0061	0.0000	0.0020	0.0112	-0.2840	0.4923
	ARMA-GARCH	0.5080	-0.0533	-0.1316	-0.0002	-0.0440	-0.2413	-0.3445	0.6617
	Logistic regression	0.5305	0.0665	0.3520	0.0005	0.1177	0.6460	-0.1734	0.1321
	LightGBM	0.5199	0.0394	0.0074	0.0000	0.0025	0.0136	-0.2127	0.4906
	LSTM	0.5053	0.0068	0.3214	0.0004	0.1074	0.5896	-0.4323	0.1541
WMT	Buy & Hold	0.5332	-0.1032	0.2158	0.0003	0.0721	0.3712	-0.4308	0.2605
	ARMA-GARCH	0.5199	-0.0535	0.1113	0.0001	0.0372	0.1915	-0.2834	0.3703
	Logistic regression	0.5119	-0.0140	-0.3570	-0.0005	-0.1193	-0.6143	-0.8392	0.8559
	LightGBM	0.5106	0.0174	0.0030	0.0000	0.0010	0.0052	-0.2645	0.4964
	LSTM	0.5212	-0.0075	-0.1429	-0.0002	-0.0477	-0.2457	-0.4119	0.6645

INTC	Buy & Hold	0.5432	-0.0772	0.3365	0.0004	0.1126	0.5205	-0.3272	0.1843
	ARMA-GARCH	0.5246	-0.0935	0.1172	0.0002	0.0392	0.1813	-0.3669	0.3770
	Logistic regression	0.5153	-0.0070	0.1178	0.0002	0.0394	0.1822	-0.4087	0.3764
	LightGBM	0.5511	0.0549	0.7999	0.0011	0.2677	1.2406	-0.2290	0.0162
	LSTM	0.5445	0.0038	0.3113	0.0004	0.1042	0.4815	-0.3514	0.2027
CVX	Buy & Hold	0.5053	-0.1215	0.2269	0.0003	0.0758	0.3384	-0.4473	0.2792
	ARMA-GARCH	0.5027	-0.0995	-0.3628	-0.0005	-0.1213	-0.5414	-0.6997	0.8253
	Logistic regression	0.5133	0.0195	0.3094	0.0004	0.1034	0.4616	-0.5478	0.2124
	LightGBM	0.4801	-0.0422	-0.2488	-0.0003	-0.0831	-0.3711	-0.7129	0.7394
	LSTM	0.4682	-0.0683	-0.1427	-0.0002	-0.0477	-0.2128	-0.4363	0.6436
IBM	Buy & Hold	0.5172	-0.0133	0.0628	0.0001	0.0210	0.1102	-0.3304	0.4244
	ARMA-GARCH	0.5212	0.0081	0.2506	0.0003	0.0838	0.4398	-0.2249	0.2235
	Logistic regression	0.5000	-0.0123	0.1495	0.0002	0.0500	0.2623	-0.3710	0.3251
	LightGBM	0.5212	0.0220	0.1615	0.0002	0.0540	0.2834	-0.3305	0.3121
	LSTM	0.5239	0.0458	0.2079	0.0003	0.0695	0.3647	-0.2330	0.2642
PG	Buy & Hold	0.5027	-0.0525	0.1043	0.0001	0.0348	0.2511	-0.2774	0.3321
	ARMA-GARCH	0.5040	-0.0411	0.0996	0.0001	0.0333	0.2400	-0.2327	0.3391
	Logistic regression	0.5040	0.0067	0.1597	0.0002	0.0534	0.3848	-0.1384	0.2529
	LightGBM	0.5000	0.0000	0.1712	0.0002	0.0572	0.4125	-0.1488	0.2379
	LSTM	0.4775	-0.0478	-0.3020	-0.0004	-0.1009	-0.7282	-0.4158	0.8959

BA	Buy & Hold	0.5570	-0.1099	0.9053	0.0012	0.3026	1.4177	-0.2910	0.0072
	ARMA-GARCH	0.5570	-0.1099	0.9053	0.0012	0.3026	1.4177	-0.2910	0.0072
	Logistic regression	0.5557	0.0289	1.0037	0.0013	0.3355	1.5732	-0.1967	0.0033
	LightGBM	0.4775	0.0441	-0.2803	-0.0004	-0.0937	-0.4373	-0.5565	0.7752
	LSTM	0.5451	-0.0079	0.7519	0.0010	0.2513	1.1760	-0.3235	0.0211
KO	Buy & Hold	0.5265	-0.0799	0.1728	0.0002	0.0577	0.4518	-0.1212	0.2174
	ARMA-GARCH	0.5252	-0.0687	0.1713	0.0002	0.0572	0.4479	-0.1212	0.2194
	Logistic regression	0.5066	-0.0282	0.2083	0.0003	0.0696	0.5447	-0.1117	0.1732
	LightGBM	0.5027	-0.0321	0.1716	0.0002	0.0574	0.4488	-0.1338	0.2189
	LSTM	0.5186	-0.0083	0.2387	0.0003	0.0798	0.6246	-0.0847	0.1402
PEP	Buy & Hold	0.5332	-0.0807	0.3160	0.0004	0.1056	0.8077	-0.1010	0.0814
	ARMA-GARCH	0.5424	-0.0164	0.2269	0.0003	0.0758	0.5796	-0.1229	0.1582
	Logistic regression	0.4814	-0.0514	0.0734	0.0001	0.0245	0.1874	-0.1718	0.3730
	LightGBM	0.5000	0.0254	0.0389	0.0001	0.0130	0.0993	-0.2570	0.4318
	LSTM	0.5332	0.0004	0.3422	0.0005	0.1144	0.8750	-0.0784	0.0653
NVDA	Buy & Hold	0.5597	-0.0109	2.2960	0.0030	0.7674	1.9337	-0.1895	0.0004
	ARMA-GARCH	0.5610	-0.0110	2.2782	0.0030	0.7614	1.9185	-0.1895	0.0005
	Logistic regression	0.4814	0.0189	-1.2040	-0.0016	-0.4024	-1.0085	-1.2480	0.9593
	LightGBM	0.4920	0.0115	-1.1626	-0.0015	-0.3886	-0.9738	-1.1718	0.9537
	LSTM	0.4576	-0.0289	-0.9308	-0.0012	-0.3111	-0.7791	-1.0329	0.9109

MCD	Buy & Hold	0.5610	-0.0703	0.6917	0.0009	0.2312	1.4475	-0.1152	0.0062
	ARMA-GARCH	0.5623	-0.0397	0.6694	0.0009	0.2237	1.4006	-0.1267	0.0078
	Logistic regression	0.5610	0.0033	0.6308	0.0008	0.2108	1.3193	-0.1216	0.0114
	LightGBM	0.4536	0.0142	-0.3443	-0.0005	-0.1151	-0.7184	-0.5107	0.8928
	LSTM	0.5477	0.0473	0.5078	0.0007	0.1697	1.0606	-0.0927	0.0335
AMGN	Buy & Hold	0.5279	0.0093	0.1546	0.0002	0.0517	0.2162	-0.2556	0.3542
	ARMA-GARCH	0.5093	-0.0012	0.0754	0.0001	0.0252	0.1055	-0.2879	0.4277
	Logistic regression	0.5424	0.0564	0.4874	0.0006	0.1629	0.6821	-0.2605	0.1192
	LightGBM	0.5504	0.0857	0.3868	0.0005	0.1293	0.5412	-0.2896	0.1747
	LSTM	0.5318	0.0120	0.1137	0.0002	0.0380	0.1590	-0.3450	0.3917
GE	Buy & Hold	0.4987	0.1206	-0.2795	-0.0004	-0.0934	-0.4688	-0.4557	0.7911
	ARMA-GARCH	0.5093	0.0872	-0.0425	-0.0001	-0.0142	-0.0713	-0.2701	0.5490
	Logistic regression	0.5106	0.0205	-0.2512	-0.0003	-0.0839	-0.4212	-0.3173	0.7668
	LightGBM	0.4854	-0.0295	-0.1715	-0.0002	-0.0573	-0.2875	-0.4463	0.6904
	LSTM	0.5119	0.0227	0.3001	0.0004	0.1003	0.5034	-0.1903	0.1921
HON	Buy & Hold	0.5451	-0.0627	0.4861	0.0006	0.1625	0.9850	-0.1419	0.0444
	ARMA-GARCH	0.5398	-0.0667	0.2767	0.0004	0.0925	0.5600	-0.1935	0.1665
	Logistic regression	0.5398	0.0067	0.3638	0.0005	0.1216	0.7367	-0.2823	0.1015
	LightGBM	0.5451	0.0314	0.5974	0.0008	0.1997	1.2118	-0.1212	0.0182
	LSTM	0.5477	0.0087	0.5598	0.0007	0.1871	1.1351	-0.1713	0.0250

In a similar way we can summarize performance metrics of the portfolio for all models in the table 4.13.

Table 4.13: Portfolio results for all models

Model	**Total return**	**Average return per trade**	**Return p.a.**
Benchmark (Buy&Hold)	0.6927	0.0009	0.2315
ARMA-GARCH	0.6744	0.0009	0.2254
Logistic regression	0.3723	0.0005	0.1244
LightGBM	0.1189	0.0002	0.0398
LSTM	0.3234	0.0004	0.1081
Model	**Sharpe ratio (annualized)**	**Maximum Drawdown**	**T-test (p-value)**
Benchmark (Buy&Hold)	1.6256	-0.1196	0.0025
ARMA-GARCH	1.7747	-0.1088	0.0011
Logistic regression	1.7331	-0.0620	0.0014
LightGBM	0.8006	-0.0740	0.0833
LSTM	1.3919	-0.1372	0.0081

Also we can plot cumulative returns of the portfolio of all models.

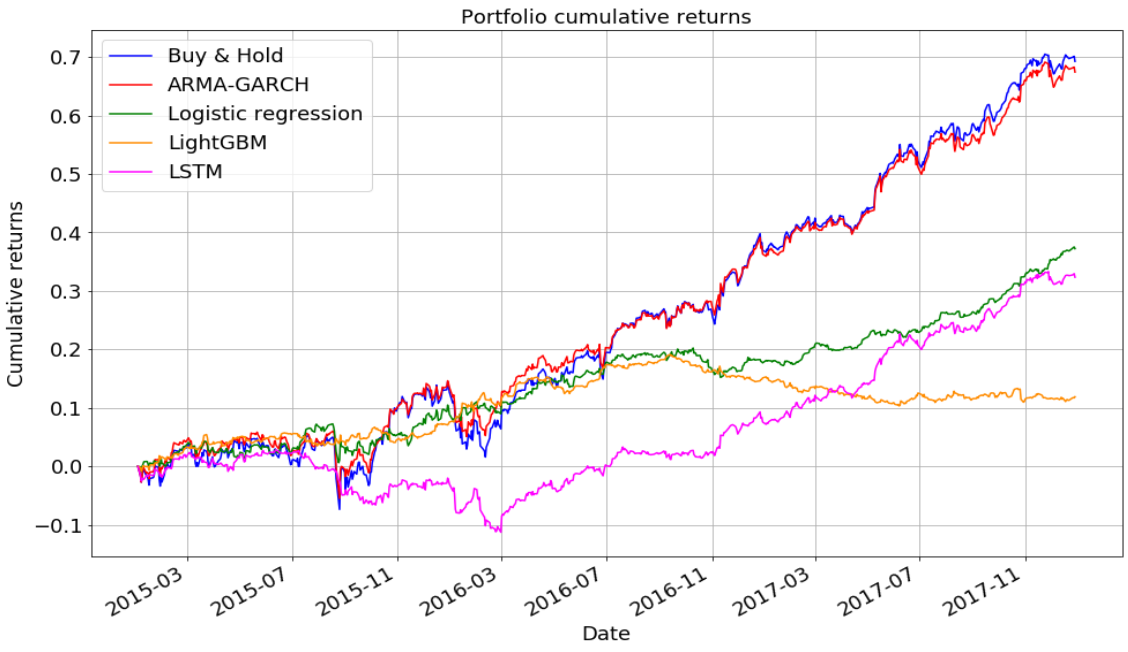

Figure 4.15. Portfolio cumulative returns

According to the table the most profitable strategy is Buy & Hold. However, ARMA-GARCH and logistic regression models demonstrated better risk-adjusted returns according to the Sharpe ratios. We can also see in the figure 4.15 that ARMA-GARCH model almost replicates Buy & Hold curve. This explains very close performance results.

Development of logistic regression curve is smooth and much less volatile than ARMA-GARCH and Buy & Hold curves. That is why logistic regression reached similar Sharpe ratio as ARMA-GARCH while having much lower total return.

LightGBM model has the worst performance among all models. In the first half f the testing period the curve moves similarly to logistic regression curve. But then it starts to go down.

LSTM networks could not provide better results than other machine learning methods in spite of their effectiveness on sequential data. Usually, neural networks demonstrate good results when there are large amounts of the training data. However, the size of historical daily stock data is not enough for LSTM networks to learn complex structures. Supposedly LSTM networks can be successfully applied to lower timeframes such as hours or minutes. However, it is a subject of further research.

Conclusion

The main goal of thesis was to use time series models and several machine learning methods for prediction of stock returns and to test whether applied models can outperform simple Buy & Hold strategy. Also we aimed to compare performance of the models between each other and to explore the potential of machine learning algorithms on financial markets.

In this thesis we described theoretical background of time series models for modeling stock returns such as ARMA and GARCH models for modeling volatility. Also we introduced basic principles of machine learning field and tasks that can be solved using machine learning approaches. We described several methods for classification such as logistic regression and gradient boosted trees. Likewise we investigated theoretical framework of Long Short-Term Memory neural networks that became very popular in recent times by their effectiveness on sequential data.

In the empirical part of the thesis we applied selected models on the real stock data of 20 tickers, which were chosen from the list of top companies by market capitalistion on the New York Stock Exchange and NASDAQ. We used daily stock data over a period from 01.01.2006 to 31.12.2017. The entire period we divide into train (from 01.01.2016 to 31.12.2014) and test (from 01.01.2015 to 31.12.2017) parts. The train data were split into 10 folds for conducting a cross-validation during the training in order to select the best meta parameters of the models. The selected and fitted parameters then were used in models on the test data for making predictions.

In case of time series models only logarithmic returns were used as predictors. To the contrary, machine learning methods allow us to use more predictors so we used all set of stock data such as Adjusted Close, Open, High and Low prices, Volumes. Also we generated some additional features such as log returns, 1-day lagged Open and Close prices, difference between Open and 1-day lagged Close prices, number of month, day and day of week.

To assess the quality of our models we used accuracy and Gini coefficient metrics. For measuring performance of strategies based on predictions produced by the models we used such metrics as total return, return p.a., average return per trade, annualized Sharpe

ratio, maximum drawdown and t-test on whether average return per trade is greater than zero. These metrics allow us to consider our models from different points of view.

The results of ARMA-GARCH models were very close to the results of Buy & Hold strategy. If we look at the cumulative return curve of the portfolio of all stocks we can notice that it almost replicates the development of Buy & Hold strategy. Even if we investigate the results for each stock individually we cannot observe some significant outperform or underperform of Buy & Hold strategy. However, ARMA-GARCH models demonstrated slightly less riskiness in terms of maximum drawdown.

Logistic regression models demonstrated poorer performance than Buy & Hold strategy and ARMA-GARCH. The total return of the portfolio of the stocks was almost 2 times less than the benchmark (Buy & Hold). In the same time individual stocks had very different results. While several stocks significantly outperformed Buy & Hold the other stocks demonstrated almost mirrored development of the cumulative returns. However, from the portfolio point of view the riskiness of logistic regression models was much lower than of Buy & Hold and ARMA-GARCH. That allowed to reach similar risk-adjusted returns as the benchmark.

Gradient boosted trees showed the worst performance and quality of classification among all models. For almost half of the stock the accuracy of predictions was under 50%. From the portfolio point of view the average return per trade according to t-test is not greater than zero on the 95% confidence level.

LSTM networks also did not demonstrate higher performance than Buy & Hold strategy. The reason of that may be the fact that such complicated neural networks usually give good results when there is a big amount of data for training. Unfortunately, daily stock data do not have sufficient number of observations.

We should also note that in the thesis we did not count any commissions or fees in calculation of performance so in real world profitability of the strategies would be even worse.

Generally we can conclude that stock markets seem to be relatively effective and described models are not able to beat the market systematically. In several cases some models demonstrated positive prediction power but the question is whether they can sustain these results in the future.

List of references

ARLT, J., ARLTOVÁ, M., RUBLIKOVÁ, E.: *Analýza ekonomických časových řad s příklady*. Skripta VSE Praha, 148 str., 2002.

BISHOP, Christopher M.: *Pattern Recognition and Machine Learning*. Springer, 2006.

BOLLERSLEV, Tim: *Generalized Autoregressive Conditional Heteroskedasticity*. Journal of Econometrics, 1986

BROCKWELL, P.J., DAVIS, R.A.: *Introduction to Time Series and Forecasting*. Springer, 2002.

BROWNLEE, Jason: *A Gentle Introduction to Backpropagation Through Time*. [online]. Available from: https://machinelearningmastery.com/gentle-introduction-backpropagation-time/

BROWNLEE, Jason: *A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning*. [online]. Available from: https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

BROWNLEE, Jason: *Bagging and Random Forest Ensemble Algorithms for Machine Learning*. [online]. Available from: https://machinelearningmastery.com/bagging-and-random-forest-ensemble-algorithms-for-machine-learning/

CAMPBELL, J.Y., LO, A.W., MACKINLAY, A.C.: *The Econometrics of Financial Markets*. Princeton University Press, 1996.

CHEN, Tianqi: *Intoruction to Boosted Trees*. Tutorial [online]. Available from: https://homes.cs.washington.edu/~tqchen/data/pdf/BoostedTree.pdf

CRYER, J.D., CHAN, K.-S.: *Time Series Analysis With Applications in R*. Springer, 2008.

DEEPLEARNING4J. *A Beginner's Guide to Recurrent Networks and LSTMs*. [online]. Available from: https://deeplearning4j.org/lstm

FLACH, P.: *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

GOODFELLOW, I., BENGIO, Y., COURVILLE A.: *Deep Learning* [online]. MIT Press, 2016. Available from: http://www.deeplearningbook.org/

HALLS-MOORE, Michael: *ARIMA+GARCH Trading Strategy on the S&P500 Stock Market Index Using R* [online]. Available from: https://www.quantstart.com/articles/ARIMA-GARCH-Trading-Strategy-on-the-SP500-Stock-Market-Index-Using-R

HAMILTON, J.D.: *Time Series Analysis*. Princeton Unoversity Press, 1994.

HASTIE, T., TIBSHIRANI, R., FRIEDMAN, J.: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

HOCHREITER, Sepp & SCHMIDHUBER, Jürgen: *Long Short-term Memory*. Neural computation 9(8):1735-80, December 1997.

HYNDMAN, Rob J., ATHANASOPOULOS, G.: *Forecasting: principles and practice* [online]. Available from: https://www.otexts.org/fpp

JAMES, G., WITTEN, D., HASTIE, T., TIBSHIRANI, R.: *An Introduction to Statistical Learning with Applications in R*. Springer, 2013.

KARPATHY, Adnrej: *The Unreasonable Effectiveness of Recurrent Neural Networks*. [online]. Available from: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

KUHN, M., JOHNSON, K.: *Applied Predictive Modeling*. Springer, 2013.

MIKUSHEVA, Anna: *Time Series Analysis*. Lecture notes [online]. Available from: https://ocw.mit.edu/courses/economics/14-384-time-series-analysis-fall-2013/lecture-notes/

MURPHY, Kevin P.: *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

NG, Andrew, BONEH, Dan: *Machine learning course*. Lecture notes [online]. Available from: http://cs229.stanford.edu/syllabus.html

OLAH, Christopher: *Understanding LSTM Networks*. [online]. Available from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

RASCHKA, S.: *Python Machine Learning*. Packt Publishing, 2015.

RICHERT, W., COELHO, Luis Pedro*: Building Machine Learning Systems with Python*. Packt Publishing, 2013.

RUPPERT, D.: *Statistics and Data Analysis for Financial Engineering*. Springer, 2011

SCHAPIRE, Robert E.: *The Strength of Weak Learnability*. Machine Learning. Boston, MA: Kluwer Academic Publishers. 5 (2): 197–227, 1990.

SHUMWAY, R.H., STOFFER, D.S.: *Time Series Analysis and Its Applications with R Examples*. Springer, 2011.

SRIVASTAVA, Pranjal: *Essentials of Deep Learning : Introduction to Long Short Term Memory*. [online]. Available from: https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/

TSAY, Ruey S.: *Analysis of Financial Time Series*, 3rd Edition. John Wiley & Sons, August 2010

WERBOS, Paul J.: *Backpropagation through time: what it does and how to do it*. Article [online]. Available from: http://axon.cs.byu.edu/~martinez/classes/678/Papers/Werbos_BPTT.pdf

Appendix

R code:

```
library(Quandl)
library(tseries)
library(forecast)
library(rugarch)
library(ggplot2)
library(MLmetrics)
library(PerformanceAnalytics)

stocks =
c('AAPL','AMZN','GOOGL','MSFT','JPM','JNJ','XOM','WMT','INTC','CVX','IBM'
,'PG','BA','KO','PEP','NVDA','MCD','AMGN','GE','HON')

path = 'University/Data/'
Quandl.api_key('-ssfpXQYbXiJG1FnjoUE')

# Downloading stock prices
for (stock in stocks){
  assign(paste(stock), Quandl(paste('WIKI/', stock, sep=''), start_date =
  "2006-01-01", end_date = "2017-12-31", type = "xts"))
}

# Fitting ARMA+GARCH models and forecasting returns for each stock
for (stock in stocks){

  # Calculate log returns and add additional columns
  data = get(stock)
  data$LogReturns = Return.calculate(data$`Adj. Close`, method = 'log')
  data$LogReturns[1] = 0
  data$TrueDirection = ifelse(data$LogReturns < 0, -1, 1)
  data$PredictedReturns = 0
  data$PredictedDirection = 0
  test_data = data['2015-01-01/']
  predictionsLength = length(data$LogReturns) -
  length(data$LogReturns['/2015-01-01'])

  # Find optimal arma model
  arimaModel = auto.arima(data$LogReturns['/2015-01-01'], ic = 'aic',
  stepwise = F)

  garch = ugarchspec(variance.model = list(garchOrder = c(1,1)),
  mean.model = list(armaOrder =
  arimaorder(arimaModel)[c(1,3)], include.mean = TRUE),
  distribution.model = 'std')
```

```
  # Make predictions
  for (i in 0:(predictionsLength-1)){
    window = length(data$LogReturns['/2015-01-01'])+i
    trainReturns = data$LogReturns[1:window]
    garchFit = tryCatch(ugarchfit(
      garch, trainReturns, solver = 'hybrid'
      ), error=function(e) e, warning=function(w) w)

    if(is(garchFit, "warning")) {
    test_data$PredictedDirection[index(data$LogReturns[(window+1)])]=1
    print(paste(stock,index(trainReturns[window]),1,"warning",sep=","))
    } else {
    garchForecast = ugarchforecast(garchFit, n.ahead=1)
    prediction = garchForecast@forecast$seriesFor
    test_data$PredictedDirection[index(data$LogReturns[(window+1)])] =
    ifelse(prediction[1] < 0, -1, 1)
    print(paste(stock, colnames(prediction), ifelse(prediction[1] < 0,
    -1, 1), sep=","))
    }
  }

  # Calculate cumulative returns and save data to csv files
  test_data$PredictedReturns =
  test_data$LogReturns*test_data$PredictedDirection
  test_data$CumulativeReturns = cumsum(test_data$LogReturns)
  test_data$CumulativePredictedReturns =
  cumsum(test_data$PredictedReturns)
  assign(paste(stock, '_test', sep=''), test_data)
  write.zoo(get(paste(stock, '_test', sep='')),
  paste(path,stock,'_test.csv',sep=''), sep = ',')
}

# Calculating performance metrics for each stock
for(stock in stocks){
  test_data = get(paste(stock, '_test', sep=''))
  results = data.frame()
  results[1:2,'Ticker'] = stock
  results[1, 'Model'] = 'Benchmark (Buy&Hold)'
  results[2, 'Model'] = 'ARMA+GARCH

  results[1, 'Accuracy'] = Accuracy(rep(1, length(test_data$LogReturns)),
  as.numeric(test_data$TrueDirection))
  results[2, 'Accuracy'] =
  Accuracy(as.numeric(test_data$PredictedDirection),
  as.numeric(test_data$TrueDirection))
  results[1, 'Gini coefficient'] = Gini(rep(1,
  length(test_data$LogReturns)), as.numeric(test_data$TrueDirection))
  results[2, 'Gini coefficient'] =
  Gini(as.numeric(test_data$PredictedDirection),
  as.numeric(test_data$TrueDirection))
  results[1, 'Total return'] = Return.cumulative(test_data$LogReturns,
  geometric = F)
  results[2, 'Total return'] =
  Return.cumulative(test_data$LogReturns*test_data$PredictedDirection,
  geometric = F)
  results[1, 'Average return per trade'] = mean(test_data$LogReturns)
```

```
  results[2, 'Average return per trade'] =
  mean(test_data$PredictedReturns)
  results[1, 'Return p.a.'] = Return.annualized(test_data$LogReturns,
  scale = 252, geometric = F)
  results[2, 'Return p.a.'] =
  Return.annualized(test_data$LogReturns*test_data$PredictedDirection,
  scale = 252, geometric = F)
  results[1, 'Sharpe ratio (annualized)'] =
  SharpeRatio.annualized(test_data$LogReturns, scale = 252, geometric=F)
  results[2, 'Sharpe ratio (annualized)'] =
  SharpeRatio.annualized(test_data$PredictedReturns, scale = 252,
geometric = F)
  results[1, 'Maximum Drawdown'] = maxDrawdown(test_data$LogReturns,
  geometric = F, invert = F)
  results[2, 'Maximum Drawdown'] =
  maxDrawdown(test_data$PredictedReturns, geometric = F, invert = F)
  results[1, 'T-test (p-value)'] =
  t.test(as.vector(test_data$LogReturns), alternative =
  'greater')$p.value[1]
  results[2, 'T-test (p-value)'] =
  t.test(as.vector(test_data$PredictedReturns), alternative =
  'greater')$p.value[1]
  assign(paste(stock,'_results',sep=''), results)
  write.table(get(paste(stock, '_results', sep='')),
  paste(path,stock,'_results.csv',sep=''), sep = ',', row.names = F)
}
```

Python code:

```
import pandas as pd
import numpy as np
import quandl
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_auc_score
from scipy.stats import ttest_1samp
import seaborn as sns
import matplotlib.pyplot as plt
import lightgbm as lgb
import warnings
from tensorflow import set_random_seed
from keras.models import Sequential
from keras.layers import LSTM, Dense
from keras.optimizers import RMSprop
```

```
def max_drawdown(cum_returns, invert = True):
    """
    Function to calculate maximum drawdown
    """
    highest = [0]
    ret_idx = cum_returns.index
    drawdown = pd.Series(index = ret_idx)

    for t in range(1, len(ret_idx)):
        cur_highest = max(highest[t-1], cum_returns[t])
        highest.append(cur_highest)
        drawdown[t]= (1 + cum_returns[t]) / (1 + highest[t]) - 1

    if invert:
        return -1 * drawdown.min()
    else:
        return drawdown.min()

def onesided_ttest(returns, mean = 0, alternative = 'greater'):
    """
    Function returns p-value of one-sided t-test
    """
    ttest = ttest_1samp(returns, mean)
    if alternative == 'greater':
        if ttest[0] > 0:
            return ttest[1]/2
        else:
            return 1 - ttest[1]/2

    if alternative == 'less':
        if ttest[0] > 0:
            return 1 - ttest[1]/2
        else:
            return ttest[1]/2

def gini_coef(y_true, y_pred):
    """
    Function to calculate Gini coefficient
    """
    return 2*roc_auc_score(y_true, y_pred)-1

def Sharpe(returns, n=252):
    """
    Function to calculate Sharpe ratio
    """
    sharpe = returns.mean() * np.sqrt(n) / returns.std()
    return sharpe

def train_test_split(df):
    """
    Function to split stock data into train and test data sets
    """
    X_train, X_test = df.drop('Tomorrow Direction', axis=1)[:'2014-12-
    30'], \
    df.drop('Tomorrow Direction', axis=1)['2014-12-31':]
```

```
    y_train, y_test = df['Tomorrow
    Direction'].loc[X_train.first_valid_index():'2014-12-30'], \
    df['Tomorrow Direction']['2014-12-31':]
    return X_train, X_test, y_train, y_test

def lstm_train_test_split(df, window = 10):
    """
    Function to split stock data into train and test data sets for LSTM
    model
    """
    X_train, X_test = df.drop('Tomorrow Direction',
    axis=1)[len(df[:'2006-01-03'])-window:len(df[:'2014-12-30'])], \
    df.drop('Tomorrow Direction', axis=1)[(len(df[:'2014-
    12-31'])-window):len(df[:'2017-12-28'])]
    y_train, y_test = df['Tomorrow Direction'].loc['2006-01-03':'2014-12-
    30'], \
    df['Tomorrow Direction']['2014-12-31':'2017-12-28']
    return X_train, X_test, y_train, y_test

def lstm_preprocess(df):
    """
    Function to preprocess stock data for LSTM model
    """

    # Select only Adjusted columns
    df_copy = df[['Adj. Open','Adj. High','Adj. Low','Adj. Close','Adj.
    Volume']]
    df_copy = df_copy.rename(columns={'Adj. Open':'Open','Adj.
    High':'High','Adj. Low':'Low',
    'Adj. Close':'Close','Adj.
    Volume':'Volume'})

    # Compute log returns
    df_copy['Log Returns'] = np.log(df_copy['Close']) -
    np.log(df_copy['Close'].shift(1))
    df_copy['Log Returns'][0] = 0

    # Add difference between today's Open and yesterday's Close;
    yeasterday's Open and Close
    df_copy['Open_Close'] = df_copy['Open'] - df_copy['Close'].shift(1)
    df_copy['Open_Lag_1'] = df_copy['Open'].shift(1)
    df_copy['Close_Lag_1'] = df_copy['Close'].shift(1)

    # Add month, day and day of week columns
    df_copy['Month'] = df_copy.index.month
    df_copy['Day'] = df_copy.index.day
    df_copy['Day_of_week'] = df_copy.index.dayofweek
    df_copy = df_copy.dropna()

    # Create a target column, which we want to predict
    df_copy['Tomorrow Direction'] = np.where(df_copy['Log
    Returns'].shift(-1) < 0, 0, 1)

    return df_copy
```

```
def preprocess(df):
    """
    Function to preprocess stock data
    """
    # Select only Adjusted columns
    df_copy = df[['Adj. Open','Adj. High','Adj. Low','Adj. Close','Adj.
    Volume']]
    df_copy = df_copy.rename(columns={'Adj. Open':'Open','Adj.
    High':'High','Adj. Low':'Low','Adj. Close':'Close','Adj.
    Volume':'Volume'})

    # Compute log returns
    df_copy['Log Returns'] = np.log(df_copy['Close']) -
    np.log(df_copy['Close'].shift(1))
    df_copy['Log Returns'][0] = 0

    # Add difference between today's Open and yesterday's Close;
    yeasterday's Open and Close
    df_copy['Open_Close'] = df_copy['Open'] - df_copy['Close'].shift(1)
    df_copy['Open_Lag_1'] = df_copy['Open'].shift(1)
    df_copy['Close_Lag_1'] = df_copy['Close'].shift(1)

    # Add month, day and day of week columns
    df_copy['Month'] = df_copy.index.month
    df_copy['Day'] = df_copy.index.day
    df_copy['Day_of_week'] = df_copy.index.dayofweek
    df_copy = df_copy.dropna()

    # Create a target column, which we want to predict
    df_copy['Tomorrow Direction'] = np.where(df_copy['Log
    Returns'].shift(-1) < 0, -1, 1)
    return df_copy

def build_score_model(X_train, y_train, X_test, y_test, units=10,
dropout=0.2):
    """
    Function to compute average validation accuracy for LSTM model
    """

    timesteps = X_train.shape[1]
    features = X_train.shape[2]
    model = Sequential()
    model.add(LSTM(units, dropout=dropout, recurrent_dropout=dropout,
    input_shape=(timesteps, features)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer=RMSprop(lr=0.001),
    metrics=['accuracy'])
    hist = model.fit(X_train, y_train, batch_size=10, epochs=10,
    validation_split=0.1, shuffle=False, verbose=0)

    acc = np.array(hist.history['val_acc']).mean()
    return acc

quandl.ApiConfig.api_key = '-ssfpXQYbXiJG1FnjoUE'
```

```
stocks =
['AAPL','AMZN','GOOGL','MSFT','JPM','JNJ','XOM','WMT','INTC','CVX','IBM',
'PG','BA','KO','PEP','NVDA','MCD','AMGN','GE','HON']

# Download stock data
for stock in stocks:
    vars()[stock] = quandl.get('WIKI/' + stock, start_date='2006-01-01',
    end_date='2017-12-31')

# Fit Logistic regression and predict for all stocks
best_conf = []
for stock in stocks:
    # Preprocess data
    df = preprocess(globals()[stock])
    X_train, X_test, y_train, y_test = train_test_split(df)

    # Find optimal parameters, train model and make predictions
    cv = TimeSeriesSplit(n_splits=10)
    scaler = StandardScaler()
    logreg = LogisticRegression(random_state=42)
    pipeline = Pipeline([
    ('scaler', scaler),
    ('logreg', logreg)
    ])
    param_grid = {
    'logreg__C': np.linspace(0.001,1,20)
    }
    grid = GridSearchCV(pipeline, cv=cv, param_grid=param_grid,
    scoring='accuracy')
    grid.fit(X_train,y_train)
    pred = grid.predict(X_test)

    # Prepare table for test data
    df_test = globals()[stock][['Adj. Open','Adj. High','Adj. Low','Adj.
    Close','Adj. Volume']]
    df_test = df_test.rename(columns={'Adj. Open':'Open','Adj.
    High':'High','Adj. Low':'Low','Adj. Close':'Close','Adj.
    Volume':'Volume'})
    df_test['Log Returns'] = np.log(df_test['Close']) -
    np.log(df_test['Close'].shift(1))
    df_test = df_test['2015-01-01':]
    df_test['True Direction'] = np.where(df_test['Log Returns'] < 0,-1,1)
    df_test['Predicted Direction'] = pred[:-1]
    df_test['Predicted Returns'] = df_test['Predicted Direction'] *
    df_test['Log Returns']
    df_test['Cumulative Returns'] = df_test['Log Returns'].cumsum()
    df_test['Cumulative Predicted Returns'] = df_test['Predicted
    Returns'].cumsum()

    best_conf.append([stock, grid.best_params_['logreg__C'],
    grid.best_score_, grid.score(X_test, y_test)])
    vars()[stock+'_log'] = df_test
    df_test.to_csv('Data/'+stock+'_log.csv')
```

```
# Save best parameters for Logistic regression in a csv file
best_conf = pd.DataFrame(data=best_conf,
columns=['Ticker','C','Validation accuracy','Test accuracy'])
best_conf.to_csv('Data/Best_conf_log.csv')

# Calculate total results for ARMA-GARCH + Logistic regression
total_results = pd.DataFrame(columns=['Ticker','Model','Accuracy','Gini
coefficient','Total return','Average return per trade',
'Return p.a.','Sharpe ratio (annualized)','Maximum Drawdown','T-test
(pvalue)'])
for stock in stocks:
    df_test = globals()[stock+'_log']
    df_results = pd.read_csv('Data/' + stock + '_results_arma.csv')
    df_results.loc[len(df_results), 'Model'] = 'Logistic regression'
    df_results.loc[(len(df_results)-1), 'Ticker'] = stock
    df_results.loc[(len(df_results)-1), 'Accuracy'] =
    accuracy_score(df_test['True Direction'], df_test['Predicted
    Direction'])
    df_results.loc[(len(df_results)-1), 'Gini coefficient'] =
    gini_coef(df_test['True Direction'], df_test['Predicted Direction'])
    df_results.loc[(len(df_results)-1), 'Total return'] =
    df_test['Predicted Returns'].sum()
    df_results.loc[(len(df_results)-1), 'Average return per trade'] =
    df_test['Predicted Returns'].mean()
    df_results.loc[(len(df_results)-1), 'Return p.a.'] =
    df_test['Predicted Returns'].mean() * 252
    df_results.loc[(len(df_results)-1), 'Sharpe ratio (annualized)'] =
    Sharpe(df_test['Predicted Returns'])
    df_results.loc[(len(df_results)-1), 'Maximum Drawdown'] =
    max_drawdown(cum_returns=df_test['Cumulative Predicted Returns'],
    invert=False)
    df_results.loc[(len(df_results)-1), 'T-test (p-value)'] =
    onesided_ttest(returns=df_test['Predicted Returns'])
    df_results.to_csv('Data/' + stock + '_results_log.csv', index=False)
    total_results = total_results.append(df_results)
    vars()[stock+'_results_log'] = df_results

total_results.reset_index(drop=True, inplace=True)
total_results.to_csv('Data/total_results_log.csv', index=False)

# Fit LightGBM and predict for all stocks
best_conf = []
for stock in stocks:
    warnings.filterwarnings(action='ignore', category=DeprecationWarning)

    # Preprocess data
    df = preprocess(globals()[stock])
    X_train, X_test, y_train, y_test = train_test_split(df)

    # Find optimal parameters, train and make predictions
    cv = TimeSeriesSplit(n_splits=10)
    lgbm = lgb.LGBMClassifier(random_state=42, max_depth=3,
    n_estimators=1000, num_leaves=5, subsample=0.8)
    param_grid = {'learning_rate': [0.0001,0.001,0.01,0.1],
    'colsample_bytree': [0.1,0.25,0.5,0.75,1]}
```

```
    grid = GridSearchCV(lgbm, cv=cv, param_grid=param_grid,
    scoring='accuracy')
    grid.fit(X_train,y_train)
    pred = grid.predict(X_test)

    # Prepare table for test data
    df_test = globals()[stock][['Adj. Open','Adj. High','Adj. Low','Adj.
    Close','Adj. Volume']]
    df_test = df_test.rename(columns={'Adj. Open':'Open','Adj.
    High':'High','Adj. Low':'Low','Adj. Close':'Close','Adj.
    Volume':'Volume'})
    df_test['Log Returns'] = np.log(df_test['Close']) -
    np.log(df_test['Close'].shift(1))
    df_test = df_test['2015-01-01':]
    df_test['True Direction'] = np.where(df_test['Log Returns'] < 0,-1,1)
    df_test['Predicted Direction'] = pred[:-1]
    df_test['Predicted Returns'] = df_test['Predicted Direction'] *
    df_test['Log Returns']
    df_test['Cumulative Returns'] = df_test['Log Returns'].cumsum()
    df_test['Cumulative Predicted Returns'] = df_test['Predicted
    Returns'].cumsum()
    best_conf.append([stock, grid.best_params_['learning_rate'],
    grid.best_params_['colsample_bytree'], grid.best_score_,
    grid.score(X_test, y_test)])
    vars()[stock+'_gbm'] = df_test
    df_test.to_csv('Data/'+stock+'_gbm.csv')
    print(stock + ' - Done!')

# Write the best parameters to a csv file
best_conf = pd.DataFrame(data=best_conf, columns=['Ticker','Learning
rate','Subsample ratio of columns',
'Validation accuracy','Test accuracy'])
best_conf.to_csv('Data/Best_conf_gbm.csv')

# Calculate total results for ARMA-GARCH + Logistic regression + LightGBM
total_results = pd.DataFrame(columns=['Ticker','Model','Accuracy','Gini
coefficient','Total return','Average return per trade',
'Return p.a.','Sharpe ratio (annualized)','Maximum Drawdown','T-test
(pvalue)'])
for stock in stocks:
    df_test = globals()[stock+'_gbm']
    df_results = pd.read_csv('Data/' + stock + '_results_log.csv')
    df_results.loc[len(df_results), 'Model'] = 'LightGBM'
    df_results.loc[(len(df_results)-1), 'Ticker'] = stock
    df_results.loc[(len(df_results)-1), 'Accuracy'] =
    accuracy_score(df_test['True Direction'], df_test['Predicted
    Direction'])
    df_results.loc[(len(df_results)-1), 'Gini coefficient'] =
    gini_coef(df_test['True Direction'], df_test['Predicted Direction'])
    df_results.loc[(len(df_results)-1), 'Total return'] =
    df_test['Predicted Returns'].sum()
    df_results.loc[(len(df_results)-1), 'Average return per trade'] =
    df_test['Predicted Returns'].mean()
    df_results.loc[(len(df_results)-1), 'Return p.a.'] =
    df_test['Predicted Returns'].mean() * 252
    df_results.loc[(len(df_results)-1), 'Sharpe ratio (annualized)'] =
```

```
    Sharpe(df_test['Predicted Returns'])
    df_results.loc[(len(df_results)-1), 'Maximum Drawdown'] =
    max_drawdown(cum_returns=df_test['Cumulative Predicted Returns'],
    invert=False)
    df_results.loc[(len(df_results)-1), 'T-test (p-value)'] =
    onesided_ttest(returns=df_test['Predicted Returns'])
    df_results.to_csv('Data/' + stock + '_results_gbm.csv', index=False)
    total_results = total_results.append(df_results)
    vars()[stock+'_results_gbm'] = df_results

total_results.reset_index(drop=True, inplace=True)
total_results.to_csv('Data/total_results_gbm.csv', index=False)

# Train LSTM network and make predictions for all stocks
for stock in stocks:
    # Preprocess data
    window = 10
    df = lstm_preprocess(globals()[stock])
    X_train, X_test, y_train, y_test = lstm_train_test_split(df,
    window=window)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Transform data to tensors
    X_train, y = [], []
    for i in range(window,X_train_scaled.shape[0]+1):
        X_train.append(X_train_scaled[i-window:i])
        y.append(y_train[i-window])
    X_train = np.array(X_train)
    y_train = np.array(y)
    X_test, y_t = [], []
    for i in range(window,X_test_scaled.shape[0]+1):
        X_test.append(X_test_scaled[i-window:i])
        y_t.append(y_test[i-window])
    X_test = np.array(X_test)
    y_test = np.array(y_t)

    # Train LSTM network for different input parameters
    df_params = []
    for unit in [5,10,20]:
        for drop in [0.2,0.4,0.6]:
            score = build_score_model(X_train, y_train, X_test, y_test,
            unit, drop)
            df_params.append([unit,drop,score])

    df_params =
    pd.DataFrame(data=df_params,columns=['Units','Dropout','Accuracy'])

    # Select the best parameters
    best_units = df_params[df_params['Accuracy'] ==
    df_params['Accuracy'].max()]['Units'].values[0]
    best_drop = df_params[df_params['Accuracy'] ==
    df_params['Accuracy'].max()]['Dropout'].values[0]
```

```
    # Train LSTM network on training data with best parameters
    timesteps = X_train.shape[1]
    features = X_train.shape[2]
    batch = 10
    model = Sequential()
    model.add(LSTM(best_units, dropout=best_drop,
    recurrent_dropout=best_drop, input_shape=(timesteps, features)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer=RMSprop(lr=0.001),
    metrics=['accuracy'])
    model.fit(X_train, y_train, batch_size=batch, epochs=10,
    shuffle=False, verbose=1)

    # Make predictions
    pred = np.where(model.predict_classes(X_test, batch_size=batch) == 0,
    -1, 1)

    # Prepare table for test data
    df_test = globals()[stock][['Adj. Open','Adj. High','Adj. Low','Adj.
    Close','Adj. Volume']]
    df_test = df_test.rename(columns={'Adj. Open':'Open','Adj.
    High':'High','Adj. Low':'Low','Adj. Close':'Close','Adj.
    Volume':'Volume'})
    df_test['Log Returns'] = np.log(df_test['Close']) -
    np.log(df_test['Close'].shift(1))
    df_test = df_test['2015-01-01':]
    df_test['True Direction'] = np.where(df_test['Log Returns'] < 0,-1,1)
    df_test['Predicted Direction'] = pred
    df_test['Predicted Returns'] = df_test['Predicted Direction'] *
    df_test['Log Returns']
    df_test['Cumulative Returns'] = df_test['Log Returns'].cumsum()
    df_test['Cumulative Predicted Returns'] = df_test['Predicted
    Returns'].cumsum()

    best_conf.append([stock, best_units, best_drop,
    df_params['Accuracy'].max(),model.evaluate(X_test, y_test,
    batch_size=batch)[1]])

    vars()[stock+'_lstm'] = df_test
    df_test.to_csv('Data/'+stock+'_lstm.csv')
    print(stock + ' - Done!')

    # Write best parameters to a csv file
    best_conf = pd.DataFrame(data=best_conf,
    columns=['Ticker','Units','Dropout','Validation accuracy','Test
    accuracy'])
    best_conf.to_csv('Data/Best_conf_lstm.csv')

# Calculate total results for ARMA-GARCH + Logistic regression + LightGBM
+ LSTM
total_results = pd.DataFrame(columns=['Ticker','Model','Accuracy','Gini
coefficient','Total return','Average return per trade',
'Return p.a.','Sharpe ratio (annualized)','Maximum Drawdown','T-test
(pvalue)'])
for stock in stocks:
```

```
    df_test = globals()[stock+'_lstm']
    df_results = pd.read_csv('Data/' + stock + '_results_gbm.csv')
    df_results.loc[len(df_results), 'Model'] = 'LSTM'
    df_results.loc[(len(df_results)-1), 'Ticker'] = stock
    df_results.loc[(len(df_results)-1), 'Accuracy'] =
    accuracy_score(df_test['True Direction'], df_test['Predicted
    Direction'])
    df_results.loc[(len(df_results)-1), 'Gini coefficient'] =
    gini_coef(df_test['True Direction'], df_test['Predicted Direction'])
    df_results.loc[(len(df_results)-1), 'Total return'] =
    df_test['Predicted Returns'].sum()
    df_results.loc[(len(df_results)-1), 'Average return per trade'] =
    df_test['Predicted Returns'].mean()
    df_results.loc[(len(df_results)-1), 'Return p.a.'] =
    df_test['Predicted Returns'].mean() * 252
    df_results.loc[(len(df_results)-1), 'Sharpe ratio (annualized)'] =
    Sharpe(df_test['Predicted Returns'])
    df_results.loc[(len(df_results)-1), 'Maximum Drawdown'] =
    max_drawdown(cum_returns=df_test['Cumulative Predicted Returns'],
    invert=False)
    df_results.loc[(len(df_results)-1), 'T-test (p-value)'] =
    onesided_ttest(returns=df_test['Predicted Returns'])
    df_results.to_csv('Data/' + stock + '_results_lstm.csv', index=False)
    total_results = total_results.append(df_results)
    vars()[stock+'_results_lstm'] = df_results

total_results.reset_index(drop=True, inplace=True)
total_results.to_csv('Data/total_results_lstm.csv', index=False)

#Calculating portfolio performance metrics
port_res = pd.DataFrame()
temp = []
for stock in stocks:
    df_test = globals()[stock+'_log']
    price = df_test['Close'][0]
    pcs = round(1000/price)
    amounts = [price*pcs,]
    for each in df_test['Log Returns'][1:]:
        amounts.append(amounts[-1]*np.exp(each))
        if (df_test[df_test['Log Returns'] == each].index[0] ==
pd.Timestamp('2017-08-04 00:00:00') \
        or df_test[df_test['Log Returns'] == each].index[0] == '2017-08-
04') and df_test.shape[0] == 753:
            amounts.append(amounts[-1])
    temp.append(amounts)

df_benchmark = pd.DataFrame(temp)
df_benchmark = df_benchmark.transpose()
df_benchmark.columns = stocks

df_benchmark['Portfolio'] = df_benchmark.apply(sum, axis=1)
df_benchmark['Log Returns'] =  np.log(df_benchmark['Portfolio']) -
np.log(df_benchmark['Portfolio'].shift(1))
df_benchmark['Log Returns'][0] = 0
df_benchmark['True Direction'] = np.where(df_benchmark['Log Returns'] <
0, -1, 1)
```

```
df_benchmark['Predicted Direction'] = 1
df_benchmark['Cumulative Log Returns'] = df_benchmark['Log
Returns'].cumsum()
portfolio = df_benchmark[['Portfolio','Log Returns','True
Direction','Predicted Direction','Cumulative Log Returns']]

results = []
results.append('Benchmark')
results.append('Portfolio')
results.append(accuracy_score(portfolio['True Direction'],
portfolio['Predicted Direction']))
results.append(gini_coef(portfolio['True Direction'],
portfolio['Predicted Direction']))
results.append(portfolio['Log Returns'].sum())
results.append(portfolio['Log Returns'].mean())
results.append(portfolio['Log Returns'].mean() * 252)
results.append(Sharpe(portfolio['Log Returns']))
results.append(max_drawdown(cum_returns=portfolio['Cumulative Log
Returns'], invert=False))
results.append(onesided_ttest(returns=portfolio['Log Returns']))
port_res = port_res.append(pd.DataFrame(results).transpose())

for stock in stocks:
    vars()[stock+'_arma'] = pd.read_csv('../Data/' + stock + '_arma.csv',
index_col='Index')
    vars()[stock+'_arma'] = vars()[stock+'_arma'].iloc[:,7:]
    vars()[stock+'_arma'].columns = ['Open', 'High', 'Low', 'Close',
'Volume', 'Log Returns','True Direction','Predicted Returns', 'Predicted
Direction','Cumulative Returns','Cumulative Predicted Returns']

for model in ['arma','log','gbm','lstm']:
    temp = []
    for stock in stocks:
        df_test = globals()[stock+'_'+model]
        price = df_test['Close'][0]
        pcs = round(1000/price)
        amounts = [price*pcs,]
        for each in df_test['Predicted Returns'][1:]:
            amounts.append(amounts[-1]*np.exp(each))
            if (df_test[df_test['Predicted Returns'] == each].index[0] ==
pd.Timestamp('2017-08-04 00:00:00') \
                or df_test[df_test['Predicted Returns'] == each].index[0]
== '2017-08-04') and df_test.shape[0] == 753:
                 amounts.append(amounts[-1])
        temp.append(amounts)

    df_temp = pd.DataFrame(temp)
    df_temp = df_temp.transpose()
    df_temp.columns = stocks

    df_temp['Portfolio'] = df_temp.apply(sum, axis=1)
    df_benchmark['Predicted Returns_'+model] =
np.log(df_temp['Portfolio']) - np.log(df_temp['Portfolio'].shift(1))
    df_benchmark['Predicted Returns_'+model][0] = 0
    df_benchmark['Predicted Direction_'+model] =
np.where(df_benchmark['Predicted Returns_'+model] < 0, -1, 1)
```

```
    df_benchmark['Cumulative Predicted Returns_'+model] =
df_benchmark['Predicted Returns_'+model].cumsum()
    portfolio = df_benchmark[['Portfolio','Log Returns','True
Direction','Predicted Direction_'+model,'Predicted Returns_'+model,
                             'Cumulative Log Returns','Cumulative
Predicted Returns_'+model]]

    results = []
    results.append(model)
    results.append('Portfolio')
    results.append(accuracy_score(portfolio['True Direction'],
portfolio['Predicted Direction_'+model]))
    results.append(gini_coef(portfolio['True Direction'],
portfolio['Predicted Direction_'+model]))
    results.append(portfolio['Predicted Returns_'+model].sum())
    results.append(portfolio['Predicted Returns_'+model].mean())
    results.append(portfolio['Predicted Returns_'+model].mean() * 252)
    results.append(Sharpe(portfolio['Predicted Returns_'+model]))
    results.append(max_drawdown(cum_returns=portfolio['Cumulative
Predicted Returns_'+model], invert=False))
    results.append(onesided_ttest(returns=portfolio['Predicted
Returns_'+model]))
    port_res = port_res.append(pd.DataFrame(results).transpose())

port_res.columns = ['Model','Ticker','Accuracy','Gini','Total
Return','Avg Return','Return p.a.','Sharpe','Max DD','T-test']

port_res.to_excel('../Data/portfolio_results.xlsx', index=False)
```