Department of Economics And Finance
Master's Degree in Finance
Chair of Asset pricing

Empirical asset pricing via Machine Learning and Fundamental financial metrics: evidence from the European Stock Market

Candidate:
Samuele Troiano

Supervisor:
Prof. Nicola Borri

Co-supervisor:
Prof. Giacomo Morelli

Academic year 2023/2024

Empirical asset pricing via Machine Learning and Fundamental financial metrics: evidence from the European Stock Market

by

Samuele Troiano

Submitted to the Department of Economics and Finance
on September 24, 2024 in partial fulfillment of the requirements for the degree of

MASTER DEGREE IN FINANCE

ABSTRACT

This study examines the predictive power of machine learning algorithms for forecasting and predicting European stock returns. Neural models, tree-based models, and enseble techniques are compared with the goal of determining whether it is possible to obtain economically and statistically significant results using a small number of predictors, primarily firm characteristics and four momentum features, with a relatively small sample of stocks (176). Great results have been gained in studies in the US stock market; the goal of this research thesis is to test whether these results can be carried over to European soil as well, taking into account the major European equity indices. The models exhibit great variability in statistical predictive performance, resulting in significant disparities in economic profitability. The return and risk measures for long-only trading strategies demonstrate that portfolios constructed using machine learning models predictions can outperform the market. It is also shown with long and short strategies that the models are able, with the exception of the Stacking Regressor model, to achieve economically significant results even when downward movements of prices are the object of predictions, again succeeding in outperforming the market portfolio. Overall, neural network models outperform tree-based models and ensemble approaches, in all fields of analysis here treated.

Thesis supervisor: Nicola Borri
Title: Professor of Asset Pricing

Contents

List of Figures

List of Tables

Introduction

In recent times, machine learning has emerged as one of the most powerful tool in financial markets, challenging traditional methodologies for asset pricing and return forecasting. This research examines the potential applications of machine learning algorithms for stock returns prediction using fundamental financial indicators and momentum features within the context of the European stock market. Although numerous studies have demonstrated the efficacy of ML models in the U.S. market, there has been a notable dearth of research in the European context, despite significant differences in market dynamics and regulatory contexts. Given the nonlinear structure of these models, this study aims to test whether the success of machine learning in predicting U.S. stock returns can be replicated in Europe. To achieve this, the study employs neural networks, tree-based models, and ensemble techniques to explore whether a limited set of predictors, a relative small dataset and time span, primarily focusing on firm characteristics, can yield statistically and economically significant stock returns predictions.

The rest of the work is organized as follows: in Chapter 1 the relevant academic literature with respect to stock return predictability and machine learning models is summarized. Special attention has been paid to the Efficient Market Hypothesis and how Machine Learning challenges traditional approaches to stock returns forecasting.
In Chapter 2, i'll describe the most relevant machine learning models used in the current thesis. Afterwards, detailed description of data handling and modeling is given, alongside the discussion on model selection and performance metrics.
Chapter 3 discusses the empirical results from this study. First, i illustrate the predictive accuracy and statistical significance of each machine learning model under analysis. At the end, portfolio performances both for the long-only and long-short strategies are discussed, while considering the economic significance of machine learning models in trading contexts, taking into consideration portfolio turnover and break-even trading costs.

Chapter 1

Literature Review

For years, predictability in stock returns has been one of the debated topics in finance, standing against the Efficient Market Hypothesis. While traditional methods have considered either financial ratios or past prices for return forecasting, recent developments in machine learning techniques offer new paradigms. This section synthesizes academic research on these themes, focusing on the challenges on the EMH and the use of financial ratios in traditional forecasting models and ML techniques in order to predict stock

1.1 Predictability of stock returns and Efficient Market Hypothesis (EMH)

Economists have long been fascinated by the nature and sources of variations in the stock market. By the early 1970s, a consensus had emerged among financial economists suggesting that stock prices were well proxied by a random walk model and that changes in stock returns were basically unpredictable. Fama [1] provides an early, definitive statement of this position. Theoretically, the random walk theory of stock prices is historically preceded by theories relating movements in the financial markets to the business cycle. One notable example concerns the interest of Keynes in the variation in stock returns over the business cycle. The efficient market hypothesis began to take shape in the 1960s from the random walk theory of asset prices advanced by Samuelson [2]. Samuelson showed that, in an informationally efficient market, the changes in prices have to be unforecastable. Indeed, statistical evidence on the random character of the changes in equity prices had already been provided by Kendall [3], Cowles [4], Osborne [5], and many others. However, it was Samuelson's contribution that had given academic respectability for the hypothesis; although the random walk model has been around for many years, having been originally discovered by Louis Bachelier, a French statistician, back in 1900. Whereas a number of studies did find some statistical evidence against the random walk hypothesis, these were dismissed as economically unimportant—trading rules based upon them could not generate profits in the presence of transaction costs—and statistically suspect—they might be due to data mining. For instance, Fama [6] concluded that "there is no evidence of important dependence from either an investment or a statistical point of view." Despite its apparent empirical success, the random walk model remained a statistical statement, rather than a coherent theory of asset prices. For example, it need not hold in markets with risk-averse traders, even under market efficiency. The term 'market efficiency' first received formal attention in the seminal review by Fama [1]; it has since typically been known as the informational efficiency of financial markets and focuses on information as the key to price setting. The efficient markets hypothesis (EMH) more narrowly defines an efficient market as one in which a new piece of information is quickly and accurately reflected in the current security price. In the seminal review, Fama [1] surveys the empirical evidence for the weak-form, semi-strong-form, and strong-form EMH. Being from the first category, he gives relatively wider coverage. Most of the empirical studies before 1970 generally used serial correlation tests and technical trading rules, and their findings strongly indicated that stock markets were weak-form efficient. Exploring the predictability of equity returns then allows an investigation of the degree of efficiency in the equity market. Two decades later, Fama [7] conducts a second review of the market efficiency literature. Instead of anchoring on past returns, he expands the coverage of weak-form EMH to tests of return predictability using other variables such as the dividend–price ratio, earnings–price ratio, book-to-market ratio, and various measures of the interest rates. Event studies and tests for private information are merely the new names for tests of the respective semi-strong-form and strong-form EMH. His review indicates that evidence of return predictability from past returns, dividend yields, and a number of term structure variables is mounting, but he argues this may be spurious and should be treated skeptically.

Fama and French [8] extended work by identifying five common risk factors in the returns on stocks, three of which are stock market factors based on financial information of firms: an overall market factor, and factors related to firm size and book-to-market equity. They use the time-series regression approach of Black, Jensen, and Scholes [9], regressing monthly returns on stocks and bonds on the returns to a market portfolio of stocks and mimicking portfolios for size, book-to-market equity, and term-structure risk factors in returns. The time-series regression slopes are factor loadings that, unlike size or BE/ME, have an unequivocal interpretation as risk-factor sensitivities for bonds as well as for stocks. Their findings indicate that portfolios designed to depict the risk factors of size and book-to-market equity reflect strong comovement in common stock returns, supporting the hypothesis that size and book-to-market equity proxy for sensitivity to common risk factors in stock returns. Their multifactor approach documents that, while the market factor captures a significant portion of variability in stock returns, additional size and book-to-market equity factors provide crucial explanatory power regarding cross-sectional variation in average returns. This evidence challenges the strict interpretation of the EMH, suggesting that stock returns may be somewhat predictable, particularly when considering the multifactor dimensions of risk. In addition to the canonical Fama and French papers, other widely cited studies like Rozeff [10], Campbell and Shiller [11], Cochrane [12], Goetzmann et al. [13], Hodrick [14], and Lewellen [15] report the main predictors of stock returns to be dividend yield, dividend-price ratio, dividend payout, earnings-price ratio, and book-to-market ratio.

Earnings-based measures have also proven useful in predicting stock returns. Basu [16] finds that stocks with low price-to-earnings (P/E) ratios earn higher returns than those with high P/E ratios, suggesting that earnings information is not fully captured by market prices. This

substantiates the idea that financial and accounting information is instrumental in acquiring a better forecast of stock returns, providing investors with valuable insights beyond the scope of market efficiency as traditionally defined. Indeed, it is the integration of financial ratios and accounting information that constitutes the fundamental analysis of a stock, in contrast to trading strategies based on technical indicators like Moving Average or traded volume. The estimate in fundamental analysis is the intrinsic value of a company, which investors try to assess using information provided by the firm through public accounting disclosures and by observing market prices to evaluate whether the market price reflects the firm's financial performance. The outcome of this evaluation is to understand whether a firm is undervalued or overvalued in the market, and then to speculate on the stock's return over the following weeks, months, or years, depending on the variables used for evaluation and the investor's time horizon. Intuitively, if a stock is considered undervalued, investors will expect an increase in the stock price and predict a positive—and sometimes abnormal—excess return. This implies that if investors can identify the crucial variables for assessing firm value, those variables can be input into models used to predict and forecast future stock returns.
Academic literature shows that stock returns can indeed be predicted from historical accounting data, particularly financial ratios [17], [18]; [19]. Fundamental factors such as profitability, solvency, liquidity, and operating efficiency are crucial in developing successful fundamental investment strategies, and there is an optimistic correlation between these cumulative fundamental indicators and high-performance companies [20], [21]. Among these fundamentals, various variables have been identified as influencing stock returns. Khan [22] found that EPS, ROE, cash flow ratio, and D/E had a significant positive impact, while the net profit margin ratio had a significant negative impact on stock returns. Lev and Thiagarajan [23] predicted that earnings-related fundamental signals like sales growth and gross margin are significantly related to stock returns and future earnings predictions. Except for the debt-to-equity ratio (D/E), all other fundamental variables have shown a significant positive relationship with stock returns [24], [25]. Hatta [26] used price-earnings ratio, EPS, D/E ratio, dividend per share, net profit margin, and return on assets as proxies for financial factors in determining stock returns. His findings confirmed that EPS and the price-earnings ratio had a significantly positive relationship, while D/E ratio and net profit margin showed a significantly negative relationship with stock returns. Martani et al. [27] studied the role of cash flow, firm size, market-based ratios, liquidity ratios, profitability ratios, and leverage ratios on stock returns inding negative relationships except for profitability, market-based ratios, and turnover ratios on stock returns.

1.2 Predictability of stock returns and Machine Learning techniques

As pointed out, the ability to predict equity returns has long been a fundamental question in the empirical finance literature. Despite the extensive work on empirical asset pricing, there remains a divide on whether equity returns are predictable. Goyal and Welch [28] favor a negative answer to this question, while others, reviewed later and mentioned above, suggest that variables with predictive power can indeed be found. Our work contributes to the many attempts made over the years to add evidence to the prediction of stock returns within the finance and economics literature.

Recent developments in the machine learning (ML) literature allow for the study of equity return predictability using a vast set of candidate predictors simultaneously, overcoming limitations faced by conventional statistical methods. In some cases, machine learning techniques more than double the performance of leading regression-based strategies from the literature, with predictive gains linked to the non-linear predictor interactions that traditional methods miss [29]. Machine learning techniques are designed to build predictive models even when the number of predictors is extremely large, sometimes exceeding the number of available observations. These algorithms are highly appealing in finance, both for practitioners and researchers, given the abundance of data and the typical challenges associated with predicting stock returns.
Machine learning algorithms provide the ability to study equity return predictability in an agnostic manner. Unlike traditional approaches, machine learning algorithms do not require pre-specifying which predictors to use or the functional form of the predictive model, allowing for both linear and non-linear relationships. This flexibility enables the data to speak for themselves without imposing theoretical constraints on how future returns are determined. However, this comes at the cost of being unable to draw theoretical implications from the model, which is a limitation of this approach. Recently, Gu et al. [30] synthesized empirical asset pricing literature with machine learning methods in a comparative analysis for stock return prediction. They present evidence clearly rejecting the ordinary least squares (OLS) benchmark in favor of machine learning methods in terms of both statistical performance and investor economic performance. The OLS benchmark represents a typical approach in one of two main strands of empirical literature on stock return prediction. Specifically, in cross-sectional stock return predictability research (e.g., [31], [32]), future stock returns are regressed on a few lagged stock characteristics. The second strand, time-series stock return predictability research, focuses on predicting the time-series of returns, often forecasting stock indices using macroeconomic predictors. Due to the underperformance of the linear benchmark in their study, Gu et al. [30] recommend using machine learning techniques to overcome the serious limitations of commonly applied methods.

Gu et al. [30] is one of the first paper to comprehensively use ML methods in cross-sectional predictability research. Many concurrent papers have followed this seminal work, each contributing to the literature by investigating the generality of the conclusions derived by the researchers. This has been done by applying their research design to other stock markets ([33], [32], [34], [35], [36], [37]) for international or regional applications, or to other asset classes ([38] for bond returns). More precisely, I focus on two separate issues: (1) overfitting and irrelevant predictors, and (2) a U.S. bias. The following sections discuss these concerns and how they can be addressed.

The aforementioned literature typically relies on large predictor sets. For example, the predictor set in Gu et al. [30] includes 94 firm characteristics and interactions of each firm characteristic with eight macroeconomic time-series from [28] and 74 industry sector dummy variables. This results in a set of 900+ predictors. Following [30], Leippold et al. [34] even extend this predictor set (i.e., 1160 predictors) while relying on a smaller cross-section (i.e., 3,900 stocks) and time-series (i.e., a study period from 2000 to 2020). Tobek and Hronec [33] and Martens et al. [38] also use more than 100 predictors when investigating stocks internationally and bonds, respectively. Such large predictor sets and the enhanced flexibility of machine learning methods over more traditional prediction techniques such as the OLS benchmark come at the risk of overfitting the data, which may put machine learning methods at a disadvantage. My set of predictors, 28 in total, as they'll be discussed in the section 'Data', are constituted by the main drivers of value firm. The goal of the current thesis is to test whether machine learning approaches work even if the research is conducted without utilizing a large number of predictors from which the model gets knowledge, and the universe of stocks is not sufficiently enough to allow the model to "learn too much".

Another issue is the U.S. bias and lack of external validity in economics [39] and finance [40] research. Karolyi [40] finds that only 16% to 23% of empirical studies published in top finance journals focus on non-U.S. markets, which is disproportionate to their economic significance. This presents two concerns: generalizing conclusions from U.S. data alone can be misleading, as research shows these findings don't always hold internationally (see [41], [42], [43]). Every replication adds value by extending studies out-of-sample (see [44], [45]). Harvey et al. [46] suggest three ways to address multiple testing bias: (1) out-of-sample validation, (2) using statistical frameworks for multiple testing, and (3) looking across multiple asset classes. Like Fieberg et al. [47], I use the first approach, conducting empirical analysis based on European markets. It is interesting to see if a market with a shorter history and less research shows different predictability patterns. Replicating the main findings on stock return predictability outside the U.S. is key to validating these studies' conclusions. Furthermore, in opposed to Fieberg et al. [47], who used 12,000 stocks across multiple markets in the MSCI Developed Europe Index[1] ,so using both developed and emerging markets. I deploy 176 companies from major European indexes (EURO STOXX 50, DAX, CAC40, and AEX). Firms listed in more than one index will only be counted once. The goal of this work is to test whether the results from the discussed accademic literature hold with a different number of predictors, only firm related, and fewer stocks. Other recent studies have examined the forecasting performance of machine learning algorithms on European stock returns, building on GKX's U.S. data analysis. Drobetz and Otto [32] report strong predictive performance using neural networks and support vector machines, employing 22 predictors but focusing on a larger sample of 800 firms. Antonio Marsì [48] conducted a similar analysis, using 45 stock-level characteristics drawn from the EURO STOXX 50 index

The remainder of the current work is organised as follows: Chapter 2 will discuss the machine learning methodology that will be deployed and will illustrate the organisation of the data and predictors used. Chapter 3, on the other hand, will show the results of the empirical research conducted. Specifically, the latter will be divided in classification analysis, statistical analysis, variable importance and portfolios performances based on the machine learning methodologies

[1]That includes: Austria, Belgium, Denmark, Finland, France, Germany, Ireland, Italy, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the United Kingdom.

discussed. The last part, the Conclusions, will draw summaries of the previously conducted analysis.

Chapter 2

Empirical Analysis

2.1 Methodology

In our empirical study, i am dealing with a panel of stocks, where months are indexed as t = 1, ..., T and stocks are indexed as i = 1, ..., N . Accordingly, the future stock return r of asset i at month t + 1 can be defined in general terms as

$$r_{i,t+1} = E_t\left(r_{i,t+1}\right) + \epsilon_{i,t+1}, \tag{2.1}$$

With

$$E_t\left(r_{i,t+1}\right) = g\left(x_{i,t}\right), \tag{2.2}$$

Current expectations about future returns are expressed as a function $\hat{g}(\cdot)$ of a vector of predictor variables $x_{i,t}$ (i.e., firm characteristics). The vectors of predictor variables at time t are defined as a P-dimensional vector $x_{i,t} = (x_{i,t,1}, \ldots, x_{i,t,P})$. Thus, the following methods aim to provide an estimate $\hat{g}(\cdot)$. Despite its flexibility, this framework imposes some important restrictions. The function $\hat{g}(\cdot)$ does not depend on i or t. Since the same functional form is adopted for all time periods and stocks, the model uses all the information in the panel and stabilizes estimates of risk premiums for an individual asset The objective of most Machine Learning models The objective is to find the optimal functional representation for the cross-sectional return y_t based on the information (characteristics) available at previous time point $x_{i,t-1}$ as:

$$\hat{g}(\cdot) = \arg\min_{g \in \mathcal{G}} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(y_{i,t} - g(x_{i,t-1})\right)^2 \tag{2.3}$$

All the models used are designed to predict the true returns by minimizing the out-of-sample mean squared forecast error:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

2.1.1 Regression trees and Random Forest

Regression Trees are a special class of decision trees used for predicting continuous outcomes. As a fundamental tool in statistical modeling and machine learning, they are widely applied due to their intuitive interpretability and ability to capture nonlinear relationships. Regression trees provide a systematic method for partitioning the feature space and modeling complex relationships among predictors. In our application, regression trees group firms by their characteristics, so that observations within each group are similar with respect to their future returns. A Tree algorithm splits the predictor space into K non-overlapping regions $R_1, R_2, R_3 \ldots R_K$. In each region, the fitted value of the target variable is its mean.

$$\hat{y}_k = \frac{\sum_{i=1}^{n} y_i \mathbb{1}(x_i \in R_k)}{\sum_{i=1}^{n} \mathbb{1}(x_i \in R_k)} \tag{2.4}$$

A tree 'grows' by completing a succession of binary splits based on cutoffs for concrete properties at each branching point. As a result, each split adds another layer of depth, allowing for more complicated interaction effects. Beginning with all observations, the tree divides the feature space into two rectangular partitions. The firm characteristic and cutoff value are selected to provide the best fit in terms of predicting inaccuracy.The generated rectangular patches in the predictor space approach the unknown function $\hat{g}(\cdot)$ as the partition's mean. The splitting method continues on one or both sections, resulting in increasingly smaller rectangles until a stopping requirement is satisfied, such as tree depth (J) or no benefit from additional splits. Regions are defined by finding critical values of the predictors such that by fitting y with its mean over the region the MSE is minimized. To sum up the algorithm works following these 3 main steps:

1. Choose a predictor $x_j \in X$ and a value x_j^* such that two regions are defined: $R_1 : x_j \leq x_j^*, R_2 : x_j > x_j^*$. The predictor and its critical value are chosen such that the MSE is minimized:

$$\text{MSE} = \frac{1}{N}\left(\sum_{i \in R_1(s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i \in R_2(s)} (y_i - \hat{y}_{R_2})^2\right) \tag{2.5}$$

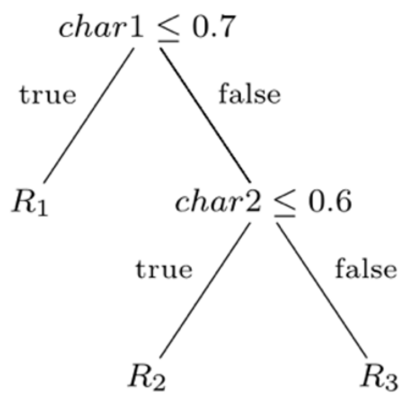
$$(X_j^*, s^*) = \arg\min_{j,s} \text{MSE}(X_j, s) \tag{2.6}$$

   Where $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ are the forecasted mean responses for the regions $R_1(s)$ and $R_2(s)$, respectively, and N is the total number of observations. The critical value x^* is not chosen at random: the model takes each predictor and uses all the values that the predictor j takes in the dataset to identify the optimal one that minimizes the MSE.

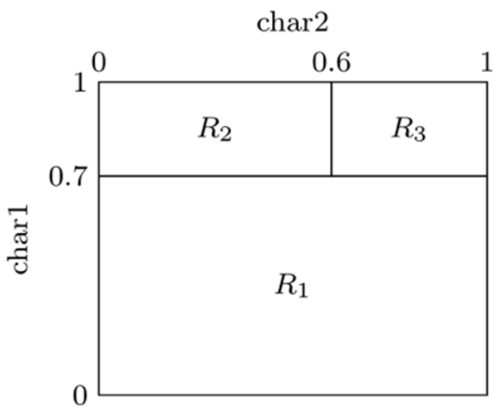
2. Choose a region R_k, a predictor x_j and a value x_j^*, split the region R_k into 2 subregions accordingly and fit the target variable mean to each region. Again the split is made to minimize prediction error.

3. Repeat step 2 recursively until a stopping criterion is met.

Figure 2.1 illustrates a regression tree of depth J = 2 based on exemplary firm characteristics char1 and char2 (2 predictors) and 3 regions formed – the leaves of the three K - These three parameters are critical in tree formation, and their optimal values will be found in the context of Random Forest using grid search, which entails trying every combination of values and selecting the one that results in the lowest RMSE. Figure 2a depicts the tree architecture and its representation in the rectangular feature space. Initially, all data are divided by the firm characteristic char1 using a cutoff value of 0.7. Firm-month observations with values less than that threshold are assigned to the left branch, whilst all other observations are assigned to the right branch.



(a) Regression tree



(b) Partitioned data

Figure 2.1: Regression Trees structure. This figure shows the structure of an example regression tree (a) with two characteristics (char1, char2) and its representation as divided partitions of the predictor space (b)

Formally, the unknown function $\hat{g}(\cdot)$ is approximated by a regression tree as:

$$\hat{g}^{\text{tree}}(x_{i,t}; K) = \sum_{k=1}^{K} c_k \mathbb{1}(x_{i,t} \in R_k), \tag{2.7}$$

Regression trees are commonly used in the ML literature because (as shown in Fig. 2.1), the succession of binary decision rules allows for a very straightforward and intuitive explanation. Furthermore, the underlying structure is perfect for depicting multiway interactions and nonlinearities. However, regression trees are rarely utilized on their own since they are unstable in response to changes in the input data and are prone to significant overfitting difficulties. Unlike OLS, a regression tree can substantially overfit even with few variables, and with enough depth, a perfect fit is possible. In the simplest instance, each leaf node has simply one observation. This is why in machine learning modeling, Random Forests are spread out instead of regression trees.

Random Forests go a step beyond regression trees in using ensemble learning techniques that will enhance predictive accuracy and robustness. They mitigate problems of single trees and aggregate their multiple-tree outputs, each built from a different boot-strapped[1] sample of the data. Random Forests follow the Bagging-Bootstrap Aggregating approach: several

[1]statistical method used to create multiple subsets of the original dataset by sampling with replacement.

regression trees are independently trained on different data subsets, generated via an i.i.d. bootstrap sampling. First, it will create B different bootstrap samples from the original dataset. It builds, for each observation, a regression tree by randomly selecting, at each node, a differ-ent subset $p < P$ of predictors. This ensures diversity among the trees and reduces both the correlation as well as the variance of the estimator (cf. Breiman 2001). It then constructs each tree without pruning[2], allowing maximum complexity to capture diverse patterns. This approach is quite important because the ensemble nature of random forests inherently reduces overfitting due to averaging. The final output of the forest will be the average of each boostrap sample chosen by the model:

$$\hat{g}(x_{i,t}; B, K) = \frac{1}{B} \sum_{h=1}^{B} \hat{g}_b^{\text{tree}}(x_{i,t}; K). \tag{2.8}$$

I will apply grid search alghorithm for choosing the best hyperparameters in the analysis. Grid search approach is used in order to understand how performance of a model measured by RMSE varies in case of change of a certain parameter. Correspondingly, the methodology allows finding the best combination of parameters that minimize the RMSE and provide more accurate predictions. The optimal parameters searched by the `GridSearch` algorithm are the following:

- **n_estimators (B)**: this parameter specifies the number of decision trees that the Random Forest algorithm will create during training. In the formulas it corresponds to B or the so-called bootstrap sample.
- **max_features (K)**: controls the number of features that are randomly chosen to evaluate for each split at a decision node. In the literature, the value chosen is the square root of P (the number of predictors). In this study it's allowed for variation using grid search, setting also the choice between other values.
- **Max_depth (J)**: refers to the maximum number of levels (or depth) from the root node to the deepest leaf node in a decision tree. It essentially limits how many splits can occur along any path from the root to a leaf.

The value of the previously listed parameters used in `GridSearch`, and the one chosen by the optmization procedure are presented in 2.2

2.1.2 Gradient Boosting regression trees

While trees are fit independently in the random forest algorithm, GBRTs are estimated in a manner that is adaptive so as to reduce bias. Thus, B trees are estimated sequentially and combined additively to form an ensemble prediction. The iterative procedure goes as: In step, the gbrt approach computes a first shallow tree to fit the realised returns. This oversimplified

[2]is the removal of superfluous branches or nodes to make the tree less complex, hence less expensive, simpler in its logic, and more general. The ultimate objective of pruning is to improve the generalizing performance of the decision tree by avoiding overfitting to the training data

tree exhibits a high forecast error. Second, it calculates another shallow tree that fits the forecast residuals of the first tree. Its forecasts are summed over to arrive at an ensemble prediction. This forecast component given by this second tree is shrunk-a factor $\nu \in (0{,}1)$ to prevent overfitting of forecast residuals. Every additional shallow tree is fitted to the forecast residual from the previous ensemble prediction; its shrunk forecast component gets added into the ensemble forecast. In other words, on each iteration, b, a new decision tree ĝtree is fitted on the previous residuals added to the ensemble. The contribution of every tree, however, is controlled according to the learning rate shrinkage factor $0 < \nu < 1$, in order for the residuals not to overfit. Finally, given by the GBRT estimation function:

$$\hat{g}(x_{i,t}; B, K, \nu) = \sum_{h=1}^{B} \nu \hat{g}_b^{\text{tree}}(x_{i,t}; K). \tag{2.9}$$

In the context of Gradient Boosted regression trees, the following hyperparameters are set using grid search:

- **n_estimators (B)**: this parameter specifies the number of decision trees that the Gradient Boosted regression trees algorithm will create during training. In the formulas it corresponds to B or the so called boostrap sample
- **Max_depth (J)**: refers to the maximum number of levels (or depth) from the root node to the deepest leaf node in a decision tree. It essentially limits how many splits can occur along any path from the root to a leaf.
- **learning_rate (ν)**: it regolates the learning rate shrinkage factor in order to avoid overfitting.

The value of the previously listed parameters used in `GridSearch`, and the one chosen by the optmization procedure are presented in 2.2

2.1.3 Neural Networks

Neural networks are the most complex method in our empirical analysis. They are highly parameterised - thus suitable to solve very complicated machine learning problems - but they are opaque and hard to interpret. In general, they map inputs - predictors - to outputs - realized returns. Inspired by the functioning of the human brain, a network is composed of many interconnected computational units, called "neurons". But every single neuron on its own contributes very little, whereas a network of neurons works cohesively to improve the performance in prediction.
My analysis focuses on traditional "feed-forward" networks. These consist of an "input layer" of raw predictors, one or more "hidden layers" that interact and nonlinearly transform the predictors, and an "output layer" that aggregates hidden layers into an ultimate outcome prediction. The principle of this approach is to make the predictions through a linear combination of non-linear combinations of predictors, passed through an hidden layer. The structure of a neural network with 1 layer hidden, 4 predictors, 5 nodes is represented in

the figure 2.2 below[3]. The input layer includes predictor variables, and each neuron in this layer represents one of the characteristics of a firm, such as price-to-earnings ratio, market capitalization, and past returns among others. While the output layer contains an estimation of the dependent variable-realised returns. Without any hidden layer, the simplest neural network equals the OLS regression model. Hidden layers added allow us to move from shallow to deep architectures and capture interactions and nonlinear effects. The number of units in the input layer is equal to the dimension of the predictors, for which i chose four in this example and denote $z_1, ., z_4$. The left panel shows the most simple possible network-e. without hidden layers (OLS case). Each of the predictor signals is amplified or attenuated according to a 5-dimensional parameter vector, ω that includes an intercept and one weighing parameter per predictor. The output layer sums up the weighted signals into the forecast:

$$\theta_0 + \sum_{k=1}^{4} z_k \theta_k \tag{2.10}$$

However, like in the simplest OLS instance, the intercept represents the model's bias or unexplained component, and the weights–or omegas–are the coefficients of each predictor, indicating how much influence each has on the prediction outcome. It incorporates more flexible prediction correlations by introducing hidden layers between inputs and outputs. The right panel of Figure 2 shows an example of a hidden layer with five neurons. Each neuron receives information linearly from all of the input units, just like in the simple network on the left. Next, each neuron applies a nonlinear "activation function" $f(\cdot)$ to its aggregated signal before sending it to the next layer. For example, the second neuron in the buried layer converts inputs into outputs as:

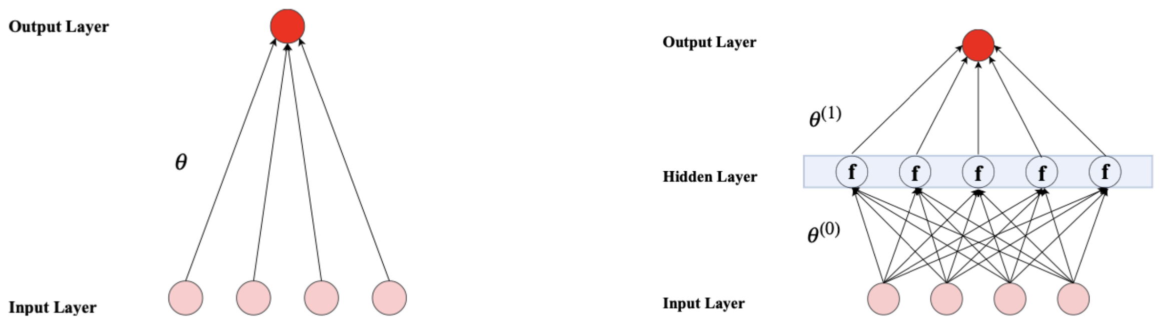
$$x_2^{(1)} = f\left(\theta_{2,0}^{(0)} + \sum_{j=1}^{4} v_j^{(0)} z_j \theta_{j,2}^{(0)}\right) \tag{2.11}$$

The second neuron in the first hidden layer, indicated as $x_2^{(1)}$, analyzes the input data by first performing a linear combination of the input features z_j, the weights $\theta_{j,2}^{(0)}$, and the bias term $\theta_{2,0}^{(0)}$. The linear combination occurs at the input level, whereas the activation function $f(\cdot)$ occurs at the next layer, in this example, the first hidden layer.
In this example, there are a total of $31 = (4 + 1)5 + 6$ parameters (five parameters for each neuron and six weights to aggregate them into a single output). The three most frequent activation functions are the sigmoid (or logistic) function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function. Their expressions are given below:

$$\text{sigmoid :} \quad f(x) = \frac{1}{1 + \exp^{-x}} \tag{2.12}$$

$$\text{tanh :} \quad f(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}} \tag{2.13}$$

[3]S. Gu, B. Kelly, and D. Xiu. "Empirical asset pricing via machine learning". In: The Review of Financial Studies 33.5 (2020), pp. 2223–2273



(a) Neural network with no hidden layer **(b)** Neural network with one hidden layer

Figure 2.2: Neural Network structure. This figure provides diagrams of two simple neural networks with (right) or without (left) a hidden layer. Pink circles denote the input layer, and dark red circles denote the output layer. Each arrow is associated with a weight parameter. In the network with a hidden layer, a nonlinear activation function f transforms the inputs before passing them on to the output

$$\text{ReLU}: \quad f(x) = \max(0, x) \tag{2.14}$$

Since the sigmoid function has a range from 0 to 1 and the tanh between -1 and 1 they usually are adopted in the hidden layer to mimic the behaviour of biological neurons. More precisely, ReLU function is more used in the output layer for classification problems, while linear function, $f(x) = x$, is preferred for regression problems. The choice of the activation functions plays an important role in the architecture of an NN. To see this, notice that if all the activation functions were linear, the NN would boil down to a linear regression model, whereas nonlinear functions such as the sigmoid and the tanh function allow neural networks to discover complex functional relationships that might exist between outputs and inputs.
to implement the model iused Keras,[4].an open-source library that provides a Python interface for constructing ANNs via TensorFlow. The activation function used in the hidden layer is 'relu', while a linear function was used for the output layer. As the optimizer, i choose Adam (Adaptive Moment Estimation), a form of mini-batch gradient descent that adjusts the learning rate for each model parameter at each iteration. It is an optimizer that has been shown to produce better results in terms of accuracy, but it requires more time and computational power.

Neural networks predict the output y by applying learned weights to the input features x. In the simplest linear model, such as ordinary least squares (OLS) regression, the coefficients are analogous to weights, but neural networks extend this by using multiple layers and non-linear transformations. As for the usual the weights in a neural network are optimized by minimizing

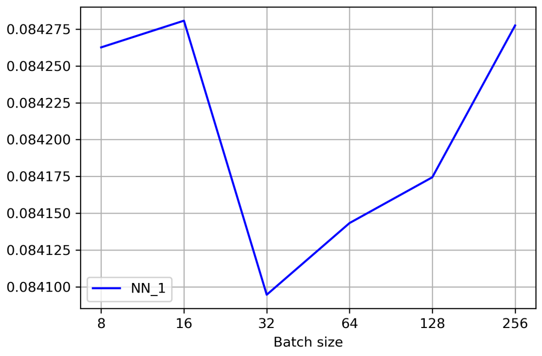
[4]Keras is an open-source library that provides a Python interface for developing ANNs through another library for machine learning called TensorFlow

a loss function. In this case, I used Root Mean Squared Error (RMSE) as the loss function to guide the optimization process In line with [32] and [47], due to the complexity of Neural Networks, I choose to deploy two separate models, reducing computational efforts. The appropriate number of neurons in hidden layers is determined using the geometric pyramid rule [49]:

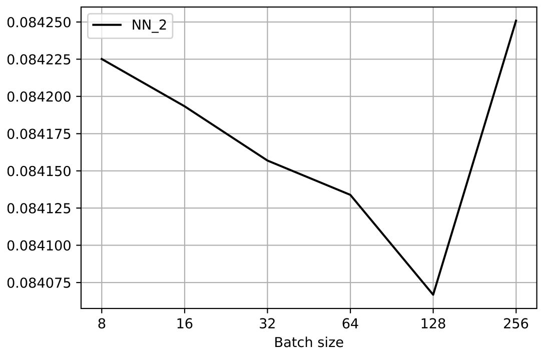
- **NN1**: with 1 hidden layer with 32 neurons.
  $(HL = 1; N = \{32\})$

- **NN2**: with 2 hidden layers with 64 and 32 neurons each.
  $(HL = 2; N = \{64, 32\})$

One of the main parameters set by the search grid is the **batch_size**: it determines how many training examples are used to compute the gradient and update the model's weights in one iteration. In other words, it controls how much data the model "sees" before making an update. Figure 2.3 depicts the minimization procedure for selecting the batch size, with each batch size having a matching RMSE. The grid search algorithm will select the smallest RMSE value that corresponds to a given batch size. For `NN1` the optimal batch size is 32, for `NN2` is 128.



(a) Neural network 1

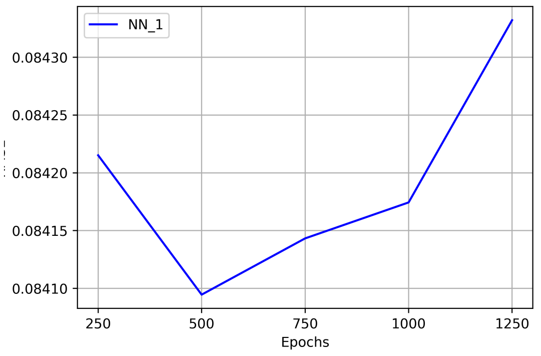


(b) Neural network 2

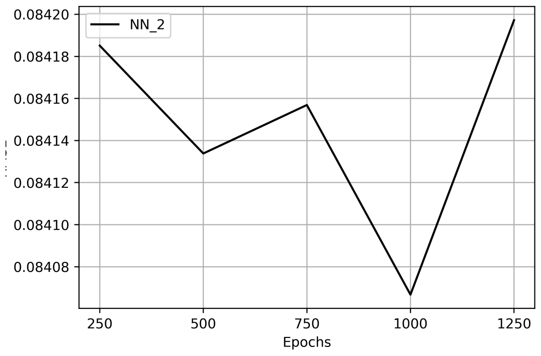
Figure 2.3: RMSE for varied batch sizes. The figure depicts the grid search algorithm's minimization phase. Since cv=3 was chosen, each batch size has three different RMSEs. The graph depicts the minimal values of them

Yet another crucial tuning parameter for the learning algorithm is the **learning rate-lr**learning rate lr, which controls the step size of every parameter update. In the following, the **learning_rate** is set to 'adaptive', which means the learning rate will be changed dynamically across the course of training, based on an initial learning rate, **learning_rate_init**. The search grid also tunes another hyperparameter to avoid overfitting: **alpha**, alias the **L2 regularization parameter**, penalizes large weights by preventing overfitting. L2 regularization adds an additional term to the loss function discouraging the weights from getting too large, which can occur when the model starts memorizing the training data. The smaller the alpha value, the less regularization is applied, and the better the fit of the model.

I also use **early_stopping**, a regularization technique that stops the training process when the model performance on the validation set doesn't show further improvements to prevent overfitting. If set to `True`, it enables early stopping to stop training when the model stops improving performance on the validation set, which saves time and prevents overfitting to the training data. Another important parameter set by the search grid is **max_iter**, or **epochs**, meaning the maximum number of iterations the optimization process can perform. The optmization process is show in figure 2.4 where different values of RMSE are plotted in correspondence of the epochs values. An iteration is one complete pass through the training data. If the training converges early (i.e., the loss function stabilizes), **early_stopping** will stop the process before reaching the iteration limits.



(a) Neural network 1



(b) Neural network 2

Figure 2.4: RMSE for varied epochs. The figure depicts the grid search algorithm's minimization phase. Since cv=3 was chosen, each value of epochs has three different RMSEs. The graph depicts the minimal values of them

2.1.4 Stacking Ensemble Regressor

Stacking is a sophisticated ensemble learning technique designed to enhance predictive performance by strategically combining multiple models. The core idea of stacking is to exploit the benefit of different machine learning models. One method for leveraging the benefits of these distinct models is to train them independently and give their predictions as input to a high-level model, also known as a **meta-model** or **level-1 model**. This meta-model aims to correct any systematic errors that the base models (**level-0 models**) might make, leading to a more accurate and robust final prediction. Mathematically,speaking let $\mathbf{X}$ represent the feature matrix and $\mathbf{y}$ the target variable. Suppose you are working with M base models denoted by $h_1(\mathbf{X}), h_2(\mathbf{X}), \ldots, h_M(\mathbf{X})$. The stacking procedure will deploy through the following steps:

1. **Base Model Training**: Each base model $h_m(\mathbf{X})$ is trained on the dataset $(\mathbf{X}, \mathbf{y})$, resulting in predictions $\hat{y}_m = h_m(\mathbf{X})$ for each model m.

2. **Meta-Model Training**: The predictions from all base models are compiled into a new dataset
$$\mathbf{X}' = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M].$$

The meta-model $h_{\text{meta}}(\mathbf{X}')$ is then trained on this dataset to predict the target variable $\mathbf{y}$.

This essentially encapsulates the original notion of two-layered prediction, with diversity in the base models at the first layer, followed by a meta-model that learns how to optimally combine these base models at the second layer. The meta-model itself might be as relatively simple as a linear regressor, but also might be more complex -if the problem complexity require so- like a neural network, depending on the chain of relationships between the base models outputs.

The main theoretical benefit of stacking is the reduction of generalization error as a result of a balancing effect of individual model biases. Recent literature, such as those by Van der Laan, Polley, and Hubbard [50] suggest that stacking does indeed combine diverse perspectives of a set of competing models to make impressive improvements upon the baseline predictive accuracy. This is because each of the base models is capable of learning to represent different patterns or aspects of the data, thus requiring the meta-model to be quite significant for synthesizing these perspectives into an integrated final forecast.
In the context of analyzing financial markets, stacking has gained more importance because of its power in model combination, which captures varied market phenomena, such as momentum, mean reversion, or volatility clustering. For example, Gu, Kelly, and Xiu [30] show that the accuracy of forecasting stock returns is substantially improved by integrating various signals extracted from financial data by stacking a sequence of machine learning models, including gradient boosting machines and random forests. While the performance of stacking generally develops on the quality of the base models and the complexity of the meta-model. As [51] note, an ensemble stacking can outperform an individual model only when a set of selected base models is adequately diverse and the meta-model parameters are suitably optimized. This is especially true for high-dimensional financial data, where different models would capture different dimensions, and some kind of stacking approach synthesizes them into something more reliable to make a prediction.

2.1.5 Voting Ensemble Regressor

The Voting Ensemble Regressor is one of the simpler yet equally powerful ensemble techniques that, at its core, combines predictions by implemented voting mechanism. In regression tasks, voting generally means averaging the predictions generated by all base models in order to generate a final output. The intuitive idea underlying voting is the reduction of variance in the predictions, using diversity in the base models in order to yield a more robust and stable result. Suppose you have M base models $h_1(\mathbf{X}), h_2(\mathbf{X}), \ldots, h_M(\mathbf{X})$. The prediction of the Voting Regressor, $\hat{y}_{\text{voting}}$, is computed as the mean of the predictions from these models:

$$\hat{y}_{\text{voting}} = \frac{1}{M} \sum_{m=1}^{M} h_m(\mathbf{X})$$

This approach finds its rationale in the law of large numbers, stating that an average of multiple independent estimators should outperform and yield a prediction closer to the true value, under the assumption these estimators are unbiased and their errors are not highly uncorrelated [52]. The performance of a voting ensemble strongly depends on diversity among base models: the less correlated the errors of base models, the larger the potential for improvement by averaging [53]. Voting ensembles are of particular appeal in financial markets, where forecasts generally have a high degree of uncertainty owing to the stochastic nature of asset prices. A voting ensemble can reduce the impacts of outliers and noisy predictions by averaging the outputs of different models, thus making more reliable forecasts. Recent works, [54], show that the performance of voting ensembles is superior to that of single models in the prediction of stock returns by appropriately aggregating several models like support vector machines, decision trees, and neural networks. Another important strength of regressor Voting is simplicity and interpretability, as the final prediction is a mean of the base models' predictions; therefore, it is simple to understand and explain, which can often be a critical choice in financial applications. Another advantage of voting, which has been highlighted by [55], is that such a combination is straightforward, so this technique is especially useful for tasks where model interpretability is valued about as much as predictive accuracy. On the other hand, voting does not explicitly train a meta-model to combine the base models; it relies entirely on the assumption that their combined prediction will be better than the predictions of its individual constituent models. This may turn out to be beneficial in scenarios where the relationships between the models are very rich and could not effectively be modeled by a meta-model. Furthermore, voting ensembles may be less computationally intensive than stacking because they do not require an additional layer of model training.

Thus, both stacking and voting ensemble regressors have their merits regarding the stock return prediction for the European market. While one can combine several models together by a stacked ensemble regressor in a more sophisticated way to potentially capture complex relationships in the data, a voting-based ensemble regressor is far simpler to produce, more interpretable, and yet sometimes quite accurate since a combination of voters reduces the variance in the predictions. Specific dataset characteristics, single model performance, and the trade-off between model complexity, interpretability, and computational load are some of the factors guiding the choice among these methods.

2.2 Data and Predictors

Previous study revealed that there is a U.S. bias in the field of research of economics [39] and finance [40]. In this regard, recent research [46] calls for a higher statistical threshold (t-statistic of 3 rather than 2) in empirical studies on the US stock market, as well as a greater emphasis on other asset classes and equity markets. I follow this demand by investigating the European stock market. More specifically the sample of stocks deployed in the current work includes companies contained in the EURO STOXX 50 index, DAX, CAC40, e AEX . The target variable in my predictive analysis is the monthly return. Monthly price data was sourced from Refinitiv DataStream. As highlighted by [30], using higher-frequency data like monthly returns can significantly improve the predictive power of machine learning models in asset pricing and return forecasting. Summary statistics on returns are provided in Table 2.1. After removing companies that were listed in multiple indices, the sample was narrowed down to 216 firms. The data spans from June 2004 to June 2023, covering a period of 23 years and

providing 229 monthly observations per firm. Firms that lacked data for five or more years were excluded from the sample, leaving a total of 176 companies. The companies included in the sample were also classified according to the NACE code[5] at the first level[6], for managing the sectoral analysis during the model's application phase. The list of NACE codes with respective sectors can be found in APPENDIX A of the Table A.1 The accounting variables used to predict the target variable were also downloaded from Refinitiv DataStream. The starting predictors are 28 and are derived from the extensive empirical asset pricing literature previously discussed, as well as from the CFI website. The complete list and descriptions and summary statistics of these variables, are provided in Table 2.1.
Following Otto [32], I also consider two-way interactions between firm characteristics as predictors, as well as second and third-order polynomials of firm characteristics to capture nonlinear relationship. Due to computational limitations, i only selected some specific predictors for interaction and the integration of higher-order polynomials. The selection is based on the simple correlation with the target variable: the predictors that covary most strongly (positively or negatively) with the target variable were chosen. The list of correlations between predictors and returns is shown in the APPENDIX A in table A.2. Additionally, I constructed momentum indicators as predictors to capture the importance that investors place on past stock movements:

- **mom_1m**: rolling 1-month return - short term reversal
- **mom_6m**: rolling 6-month return
- **mom_9m**: rolling 9-month return
- **mom_36m**: rolling 36-month return - long term reversal

Incorporating rolling window returns into stock return prediction models exploits the momentum effect, where security prices tend to continue moving in the same direction. This enhances the viewpoint and boosts the predictive capability of the models. Various empirical studies by Jegadeesh and Titman [56], Carhart [57], recent works by Han, Lee, and Worah [58], and Nordvig and Firoozye [59] affirm the significance of momentum indicators in machine learning frameworks and validate their utility in financial prediction. Xia, Hu, and Sargent [60] demonstrate that machine learning models incorporating momentum are likely to outperform those that do not.
To ensure accurate computation of firm momentum features without introducing bias, calculations are performed separately within a data subset of each firm. This approach maintains the integrity of predictive features by preventing mixing of calculations across firms. However, NaN values are inevitably generated in the momentum features' calculation, particularly at the initial stages of the time series due to insufficient data points for computing rolling

[5]NACE (Nomenclature of Economic Activities) is a European industry standard classification system used to categorize and systematically organize economic activities. It provides a framework for statistical and economic analysis across various sectors within the European Union

[6]NACE Level 1 consists of 21 broad sectors that classify the main economic activities

returns. It is crucial to address this issue to prevent errors during model training, which necessitates a complete dataset.
In order to handle these NaN values, two imputation methods are deployed: the Spline Interpolation and K-Nearest Neighbors. Spline interpolation is a mathematical process based on fitting a smooth curve through available data points. In particular, cubic splines will be used for interpolating those missing values with the goal of imputed values to keep continuity and, meanwhile, the first and second derivatives smooth. This is a very effective method in financial time series data, where the underlying process is often a smooth trend, therefore making spline interpolation a good choice. Recent studies [61] support this application of the spline interpolation in financial contexts where it has performed well in maintaining the integrity of the time-series data.
Complementary, I used the K-Nearest Neighbors (KNN) Imputation method in addition to spline interpolation. KNN imputation works by finding the k-nearest neighbors, based on Euclidean distance or some other suitable metric, to the observation with the missing value and imputes the missing value with the mean of these nearest neighbors. KNN proves particularly useful when there is, possibly on the local scale, a pattern that might be lost with mere interpolation. For example, if certain stocks tend to exhibit similar momentum, KNN can use that to interpolate among stocks that do not have explicit values available on a specific day or time period. Its robustness concerning the handling of diverse data structures has been tested in many studies, including the recent comprehensive work [62], which underlined its efficiency in financial datasets.
By using Spline Interpolation along with KNN imputation, momentum features were imputed in a manner such that the temporal and cross-sectional characteristics of the data were preserved to retain the predictive integrity of the features.

During the course of the cleaning phase of the dataset, two principal actions were undertaken. firstly, in cases where a predictor is missing for a particular firm i, the median value of the cross-section for the corresponding month t is imputed, following the approach commonly seen in academic literature such as Fieberg et al. [47] and Marsì [48]. Secondly, too correlated predictors were dropped. According to Lewellen [63], the high correlations resulting from the similarity in construction, issuance metrics, and incorporation of comparable firm information, such as profitability measures, do not pose a significant issue in this empirical analysis. The focus of the researcher lies in the collective predictive capability of the machine learning models rather than the individual effects of each predictor. Machine learning models offer a potential solution to address multicollinearity by choosing/reducing variables, thereby enhancing predictive accuracy. Despite this, i construct a correlation matrix to identify the most correlated one, following Otto [32]. The correlation matrix is shown in the APPENDIX A in figure A.1. The aim is to drop variables that exibhit correlation equal or higher of 80%. In my dataset the predictors are **current_ratio** and **eps_basic**, positively correlated respectively with **quick_ratio** and **EPS_diluited**. To enhance and confirm the results of my variable elimination, i performed a variable importance factor (VIF) analysis, that basically measures the correlation of one variable with all the others. Standard practices makes that a variable exhibiting values higher than 5 should be treated accordingly. Fortunate the highest value in my sample is 4.95 as it is shown in table A.3 in the APPENDIX A, meaning that no other variable should be dropped. I refined the initial dataset through two additional steps: firstly, I winsorized all monthly firm characteristics at the 1% and 99% levels to address outliers. Unlike Otto [64], I followed Gu et al. [30] in not winsorizing returns. Secondly, I ranked all firm characteristics on a monthly basis cross-sectionally and normalized the ranks to the $(-1, +1)$ interval, in line with the approach of Kelly et al. [65] and Fieberg et al. [47], instead of a simple linear transformation:

$$\text{normalized\_rank} = \frac{2 \times \text{rank}_i}{n+1} - 1$$

Due to the fact that machine learning methods allow us to overcome the high dimensionality problem, which arises when the number of predictors tends to be very large compared with the number of observations, and at the same time be useful in solving the problem of multicollinearity, they tend to overfit, so i will need to control for the degree of model complexity by tuning respective model hyperparameters. Examples of such hyperparameters include the number and/or depth of trees in boosted regression trees or random forests. In order to avoid overfitting and ensure maximum predictive power out-of-sample, hyperparameters cannot be set a priori, but they have to be adaptively determined from the sample data. When I'm dealing with time series data, the temporal order needs to be preserved. I therefore use a chronological split instead of random splitting. It is important to preserve the temporal order of the data in order to prevent leakage, which can result in overly optimistic evaluations of model performance.
In many circumstances, there is no theoretical guidance on how to tune hyperparameters for new data. To address this, i take a typical technique from current research literature by dynamically picking tuning parameters from a validation sample using `GridSearch`. This method allows us to construct a pseudo out-of-sample scenario by giving the model a variety of options and ranges to choose from and then determining the optimal combinations. By splitting the data into two in-sample sets (testing and validation) and one out-of-sample set (test set), i can successfully optimise hyperparameters by minimizing a loss function, such as RMSE, on a validation sample rather than the training set. This strategy helps to prevent the creation of an overfit model during training, hence enhancing its performance
The values contained in the `GridSearch` and the optimal parameters chosen after the optimization procedure are summarized in Table 2.2.

The The training sample is used to estimate the model for multiple parameter specifications, while the validation sample is used to tune the parameters. Based on the models estimated from the training sample, the RMSE within the validation sample is calculated for each parameter specification. The model with the parameter specification that minimizes the RMSE is used for out-of-sample testing. Note that, because the tuning parameters are chosen from the validation sample, it is not truly out-of-sample. The test sample, however, is used for neither model estimation nor parameter tuning. This is why it is truly out-of-sample and appropriate for evaluating a model's out-of-sample predictive power.

My sample data are then split into three different sets using a fixed window approach:

1. **Test set**: 8.5 years of data, or 100 monthly observations per firm, for a total of 17.600 observations.

2. **Validation set**: 5 years of data, or 60 monthly observations per firm, for a total of 10.560 observations.
3. **Training set**: 5.5 years of data, or 58 monthly observations per firm, for a total of 11.968 observations.

The target objective of the analysis is to predict the magnitude (and the sign) of the next month's return. To do so, the column `monthly_returns` is shifted, and a binary column based on the latter is created for classification purposes. The column `target` will take the value 1 when the return in the next period is > 0, and 0 otherwise.

Model	**Hyperparameter**	**Grid search**	**Optimal value**
RF	`n_estimators (B)`	[300,400,600,800]	800
	`max_features (K)`	[sqrt,log2,0.6]	0.6
	`max_depth (J)`	[25,30,35,45]	25
GBR	`n_estimators (B)`	[350,500,600,750]	750
	`max_depth (J)`	[10,15,20]	10
	`shrinkage_rate` (ν)	[0.001,0.003,0.005]	0.003
NN1	`batch_size`	[8,16,32,64,128,256]	32
	`neurons`	(32,)	(32,)
	`epochs`	[250,500,750,1000,1250]	500
	`learn_rate_init`	[0.001,0.003,0.005]	0.001
	`l2_reg`	[0.001,0.003,0.005]	0.001
NN2	`batch_size`	[8,16,32,64,128,256]	128
	`neurons`	(64,32)	(64,32)
	`epochs`	[250,500,750,1000,1250]	1000
	`learn_rate_init`	[0.001,0.003,0.005]	0.001
	`l2_reg`	[0.001,0.003,0.005]	0.001

Table 2.2: Hyperparameters, search grid values and optimal parameter. This table display the hyperparameters search by grid search for each of the machine learning models deployed and described in the section Methodology. The algorithm searches for the best combination and gives as an output the optimal value. Since vc = 3 is chosen, for each combination, the grid search algorithm will perform the task 3 times and will display the lower RMSE.

Predictor	**Avg**	**Std**	**Description**
p/e	27.0	135.7	Price-to-earnings ratio
p/s	2.3	4.8	Price-to-sales ratio
p/cf	10.5	89.8	Price-to-cash flow ratio
p/b	5.1	70.4	Price-to-book ratio
g_profmargi	0.4	0.2	Gross profit margin
o_profmargi	0.2	0.3	Operating profit margin
n_profmargi	0.1	1.1	Net profit margin
eps	34.7	70.9	Earnings per share
eps_fully	2.1	4.2	Fully diluted EPS
bookval/n	18.8	28.6	Book value per share
tan_bv/n	7.5	23.3	Tangible book value per share
cf/n	4.4	8.2	Cash flow per share
roe	0.2	0.8	Return on equity
roa	0.1	0.1	Return on assets
eps_5y	8.3	19.5	5-year EPS growth
salgro_5y	0.1	0.2	5-year sales growth
quick	1.0	0.8	Quick ratio
d/e	1.2	6.6	Debt to equity ratio
d/v	0.4	1.7	Debt to total capital
w_cap/t_cap	0.1	0.3	Working cap to total cap
inv_turn	0.34	10.8	Inventory turnover
asset_tunr	0.7	0.5	Asset turnover ratio
dy	0.03	0.03	Dividend yield
roa^2	0.4	0.3	Square of ROA
dy^2	0.03	0.0	Square of dividend yield
roe^2	0.7	14.6	Square of ROE
roa*roe	0.1	1.8	ROA multiplied by ROE
mom_1m	0.1	0.1	Momentum over 1 month
mom_6m	0.03	0.2	Momentum over 6 months
mom_9m	0.04	0.3	Momentum over 9 months
mom_36m	0.2	0.5	Momentum over 36 months
monthly_returns	0.0	0.1	Monthly returns

Table 2.1: Summary statistics

Chapter 3

Empirical Results

3.1 Predictive Accuracy

To assess the predictive performance of our models, the analysis of the predictions generated by our model will be based using two different approaches: regression and classification-based. As underlined by of Gu, Kelly, and Xiu [30], the benefit of a dual approach lies in how combining regression and classification metrics can lead to a more holistic evaluation of model performance. While the regression metrics provide a detailed view of the magnitude of the prediction errors and the model's ability to capture variance, the classification metrics provide insight into the practical applicability of the predictions in real-world trading where the focus is often on the direction of returns.
Initially, i analyze the predictions by treating the analysis as a classification problem. It is feasible to do so by setting the target variable and prediction to 0 or 1 depending on their sign, which is analogous to the challenge of forecasting whether a stock's price will rise or fall, with the target being 1 when the return is positive and 0 when it is negative. Thus, i can now use classic classification metrics to evaluate our models, such as accuracy, precision, recall, and F1 score, which are defined as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

This type of classification-based analysis is particularly helpful for practical trading applications, given that most decisions depend on the potential direction of the stock rather than necessarily on the exact size of the movement. Powerful metrics of classification can be if used in a financial context in which often the pre-eminent goal is not exact return values but just to correctly anticipate directional movements. The results regarding these classification metrics are shown in table 3.1 below:

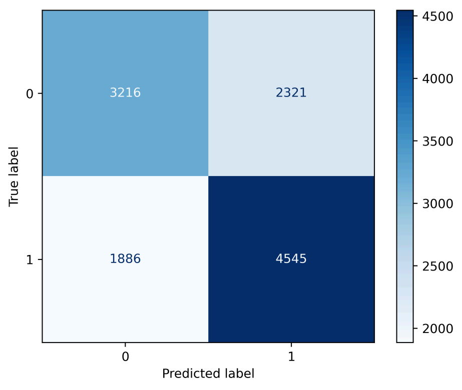
Metric	RF	GBR	NN1	NN2	SR	VR
Accuracy	0.64	0.64	0.69	0.69	0.67	0.69
Precision	0.65	0.65	0.70	0.70	0.70	0.68
Recall	0.70	0.71	0.71	0.73	0.69	0.76
F1 score	0.67	0.68	0.70	0.71	0.69	0.71

Table 3.1: Classification based metrics for each of the machine learning models

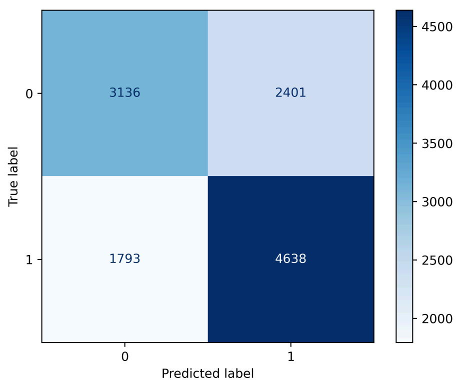
To further enhance the analysis of classification metrics, I constructed confusion matrices for each of the predictive models, showed in figure 3.1. The confusion matrix is a valuable tool descriptor in a classification task as it points to the number of correct and incorrect predictions across classes. It plots true positives, true negatives, false positives, and false negatives as output, giving a clear picture of areas in which the model perform accordingly, and areas in which it needs improvements.
Classification metrics and confusion matrices indicate that neural networks and esemble techniques, in general, tend to slightly outperform tree-based models on most of the relevant performance metrics. In general, given an accuracy rate for all models of between 65% and 70%, it is safe to say that adopting similar models offers visible benefits in terms of correct predictions, compared to a random situation of 50%/50%. In this regard, however, it is also necessary to consider that, on average, stocks tend to increase in value, so taking 50/50 as a benchmark is not entirely correct. However, especially in the short term, a percentage of correct predictions close to 70% is certainly a good starting point. NN2 and Voting Regressor have the highest recall and F1 score, which means these have a better balance between precision and recall. This is especially so for NN2, which seems to have done a pretty good job of reducing the number of false negatives. The Voting Regressor is slightly better in recall, which means an even better ability to identify positive upward movement of a stock, which could be very helpful in the stock return prediction when capturing upward movements is more important. Also the other models, seems to have a good ability of predicting positive returns. This is also confirmed by good percentages of precision. It can also be inferred, from the confusion matrices, that the models are less adept at anticipating downward movements of the stocks under analysis, with respect to the already discussed upward movements.

Secondly i deploy statistical analysis to quantify the models' predictive accuracy, i.e., how well predicted returns reflect realized returns. i will use different metrics, explained below. In addition to the common R^2 Out of sample, that measures the proportion of the variance in the actual returns that is explained by our predictors, as Fieberg [47] and Cakici et. al. [66] i will deploy a common metric to measure the capability of a model to predict the magnitude of the returns, the out-of-sample (pseudo) R^2, which is defined as:

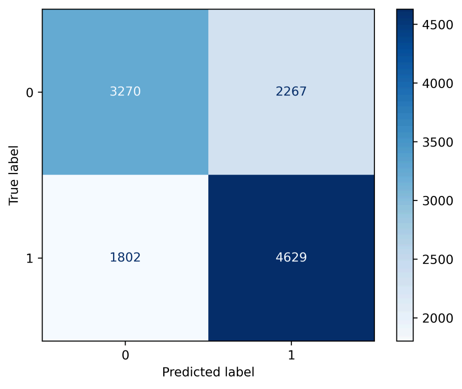
$$R^2_{\text{Pseudo OOS}} = 100 \cdot \left(1 - \frac{\sum_t \sum_i (\hat{r}_{i,t} - r_{i,t})^2}{\sum_t \sum_i r^2_{i,t}}\right)$$



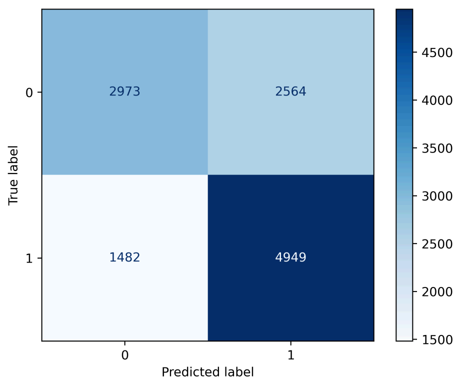
(a) Random Forest



(b) Gradient Boosting Regressor



(c) Neural Network 1



(d) Neural Network 2

Figure 3.1: Confusion matrices. The matrices are constructed by comparing the target monthly return and the predictions of each of the machine learning models in order to show the exact number of correct and incorrect predictions

This approach of assessing predictive accuracy compares predicting errors between different models to a benchmark prediction of zero. This strategy is consistent with the methodology employed by GKX. The use of zero-prediction as a benchmark rather than the historical mean commonly utilized in Out of Sample R-squared calculations is justified by zero-prediction's greater success at forecasting stock returns. The R^2OOS metric spans from $-\infty$ to 100, with negative values indicating a model that performs worse than the zero-prediction benchmark. A number of zero indicates that the two forecasts have identical precision, however positive values show that the model surpasses the zero prediction, with a maximum score of 100 reflecting a perfect forecast.
Other two statistical metrics used to complete the analysis are Mean Squared Error (MSE) and Mean Absolute Error (MAE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

The use of both measures allows us to examine the impact that outlier predictions have on model performance. MSE squares the errors before averaging, making it more sensitive to big deviations, which can severely penalize models with a few substantial errors. In contrast, MAE computes the average of the absolute disparities between actual and predicted values, assigning equal weight to all errors regardless of magnitude. As a result, MAE is a more robust measure of average prediction accuracy, especially in the presence of outliers. The results are stored in table 3.2.

From the point of view of forecast errors, the analysis of MAE and MSE shows that all models behave similarly. Surpisingly, ensemble methodologies, Stacking regressor and Voting regressor, are the ones that exhibit higher forecast errors, albeit slightly, under the point of view of both metrics. It is also crucial to emphasise that for all models the MSE is lower than the MAE: this indicates that the magnitude of most errors is relatively small, which is most likely due to the fact that there are no significant outliers or large deviations in the predictions. This can in part be explained, for Random forest and Gradient Boosting Regressor, by the fact that the predictions generated by the models exhibit relatively low standard deviation, as shown by the table A.4 available in APPENDIX A.

In line with the results achieved by [48] from the analysis of R^2OOS metrics is it possible to infere that ML algorithms perform quite bad in explaining monthly returns of the stocks contained in the major European index. Among all, `NN1` and `NN2` have the highest R^2OOS and $pseudo - R^2OOS$ values, with `NN1` recording the highest values of R^2, respectively of 0.26% and 0.29%, suggesting that it is able to explain some level of variance in the out-of-sample predictions and that again, neural based models are able to outperform tree-based models. Even for these models, however, the relatively low positive R^2 values obtained imply poor generalization performance, if we consider the financial metrics under observation. All the other models exhibit negative R^2, meaning they perform worse than naive predictions of zero return. Although ensembles models might be useful to decrease error via model averaging, in this case they are not capturing the general, fundamental relationships across the data well, exhibiting the lowest R^2. The findings of the current study go against the ones obtained by [30] and [64]. The researchers discovered modestly positive R^2 values for most deployed models, including RF, GBR, and NN. They employed more predictors, which is most likely why the disparities exist. Nevertheless, the values of R^2 for neural network models is relatively similar.

These findings highlight the need of including complex predictor interactions, which are inherent in neural network models but tree-model based methodologies. The results also reveal that in the monthly return situation, the upsides of "deep" learning are constrained. The two layer model fail to improve over the single layer one.

Metric	RF	GBR	NN1	NN2	SR	VR
MSE (%)	1.16	1.15	1.10	1.11	1.14	1.18
MAE (%)	6.90	6.86	6.72	6.74	6.85	7.05
R^2 OOS (%)	-5.25	-4.29	0.26	0.14	-3.38	-6.40
Pseudo R^2 OOS (%)	-5.23	-4.26	0.29	0.17	-3.35	-6.38

Table 3.2: Statistical analysis metrics for each of the machine learning model

However, comparing these indicators separately does not allow us to determine whether one model is considerably superior to another. Following Fieberg [47] and Otto [64], i use the popular Diebold and Mariano (DM) test [67] to compare the prediction accuracy of two models. The null hypothesis for equal prediction errors is $H_0 : E[dt] = 0$, where dt represents the loss differential between two forecasts at time t. The test statistic is defined as following:

$$DM = \frac{\bar{d}_{1,2}}{\hat{\sigma}_{\bar{d}}} \sim \mathcal{N}(0,1), \quad \text{with}$$

$$\bar{d}_{1,2} = \frac{1}{T}\sum_{t=1}^{T} \hat{d}_{1,2,t}$$

$$\hat{d}_{1,2,t} = \frac{1}{N}\sum_{i=1}^{N} \left(e_{1,i,t}^2 - e_{2,i,t}^2\right)$$

$$\hat{\sigma}_{\bar{d}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T} \left(\hat{d}_t - \bar{d}\right)^2}$$

Table 3.3 presents the test statistic for the DM test on the full sample of stocks. * , **, *** represents respectively statistical significance at 10%, 5% and 1% level. The models listed in the rows are compared against the models listed in the columns. A positive value suggests that the model in the column performs better than the model in the row, on average, according to the DM test.

	RF	GB	NN1	NN2	SR	VR
RF	–	-1.81*	4.22***	4.17***	0.95	-2.47**
GB		–	4.02***	3.94***	1.43	-0.47
NN1			–	-1.2	-4.15***	-4.50***
NN2				–	-40.3***	-4.53***
SR					–	-0.8
VR						–

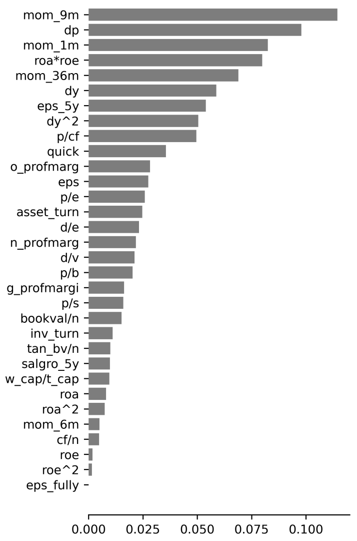
Table 3.3: T-test of pairwise Diebold-Mariano (DM) tests. This table reports t-test of pairwise Diebold-Mariano (DM) tests comparing stock-level return forecasts of various models against the null hypothesis of equal predictive accuracy. Positive numbers show that the column model outperforms the row model. *** measures the levels of statistical significance, 10%,5% and 1%.

The DM test validates the empirical conclusions generated from the R^2 values shown in Table 3.2. The first conclusion to draw is that neural network models, according to the DM test, can outperform tree-based and enseble models at 1% statistical significance, with `NN1` remaining the highest performing model. In the realm of tree-based models, `RF` appears to be the most valuable, outperforming `GRB` at the 10% statistical significance level and the stacking regressor model at the 5% level. For other pairwise comparisons, the null hypothesis of identical prediction accuracy cannot be rejected

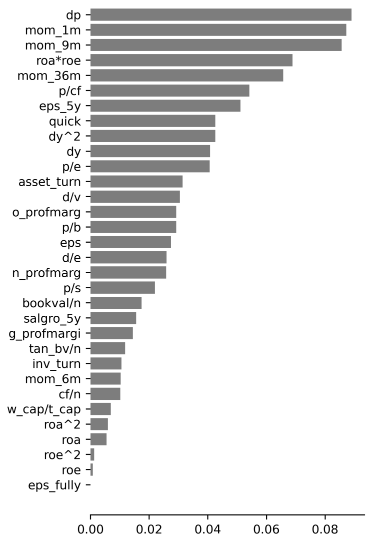
3.2 Variable Importance

Subsequently, although the various forecast models exhibit similar levels of complexity (at least within each model family), it is important to explore whether different techniques highlight distinct predictors as most significant for predicting future returns. To assess the importance of specific variables, i adopt the methodology introduced by Kelly et al. [65], which entails computing the variable importance matrix for each model in two stages. Initially, the absolute variable importance is determined by measuring the decrease in R^2 when a particular predictor's values are all set to zero in the training dataset. Subsequently, i standardize the absolute variable importance values to ensure they sum up to one, indicating the relative contribution of each variable to the model. Figure 3.2 depicts the relative variable importance measures for each forecast model separately.

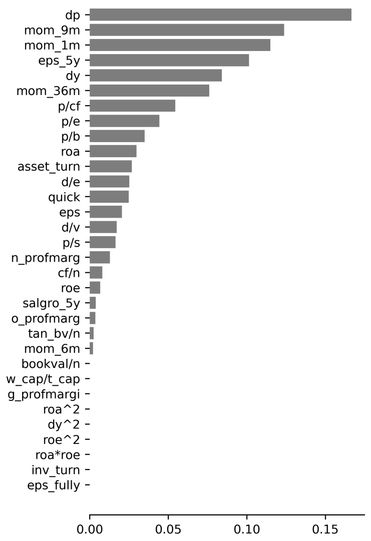
On average, in line with [66] and [64] i find that the various forecast models identify similar variables as the most valuable (the momentum features, dp, dy, eps_5y, etc.). However, some models concentrate on a small number of factors. In aversion to [64] i find that neural network models give more importance to less number of predictors, opposed to tree based models. Furthermore, the same sort of models have a very close pattern of varying relevance, in terms of which predictors and the behavior of their concentration. For what concerns momentum features, my findings resembles the ones of [66] on the European stock market and the ones of [30] on the US market. For example short-term-reversal (mom_1m) scored second and first respectively in the above mentioned researches, while in the present studies it ranks third, as it is exhibited in 3.3. Apart from this, many other financial indicators do not seem to have much relevance for the models considered. This may be surprising as, for example, related to ROE, financial evaluation models of companies have been developed based mainly on this measure and how it changes over time. A similar argument can be made for the operating profit margin or the gross profit margin, indicators that are often taken into account to calculate the true value of a company and consequently its stock price. The reason why they are not particularly important may be the type of data used (monthly) or the time frame considered. Also worth mentioning is the importance of the dividend payout ratio and dividend yield, which have been shown (by means of linear regression studies) to be important for investors and thus able to explain stock returns, as mentioned above in the Literature review section



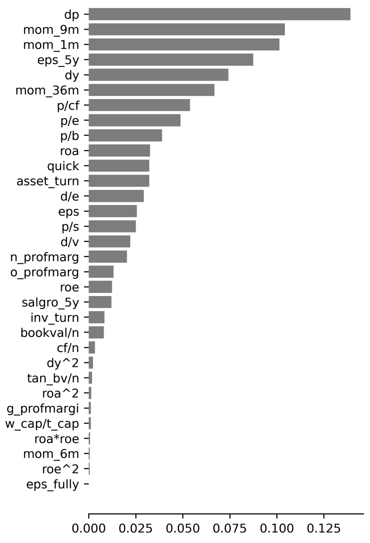
(a) Random forest



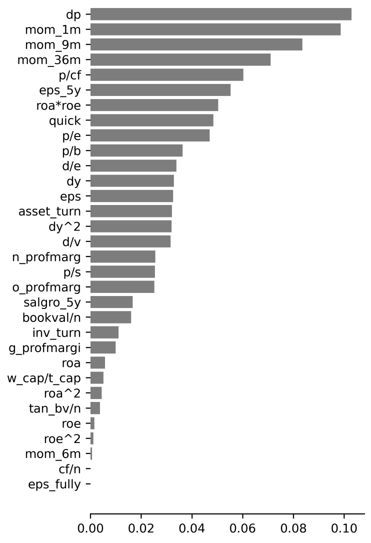
(b) Gradient Boosting



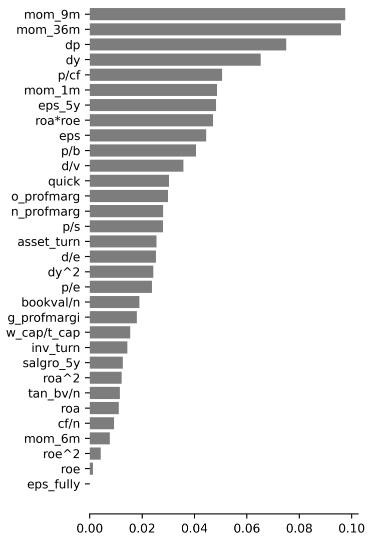
(c) Neural network 1



(d) Neural network 2



(e) Stacking regressor



(f) Voting regressor

Figure 3.2: Model-specific relative variable importance. This figure depicts the time series average of relative variable importance indicators of each machine learning model

To improve inter-model comparability, i rank the average relative contribution of each variable within a single model and add the ranks across all models to get an overall rank (greater variable importance = higher rank). Figure 3.3 depicts a heat map showing relative variable relevance rankings. The rows are arranged in descending order according to overall rank. As a result, the higher a variable is ranked, the more significant it is overall. Darker cell colors indicate that the associated variable is more important to the specific model.

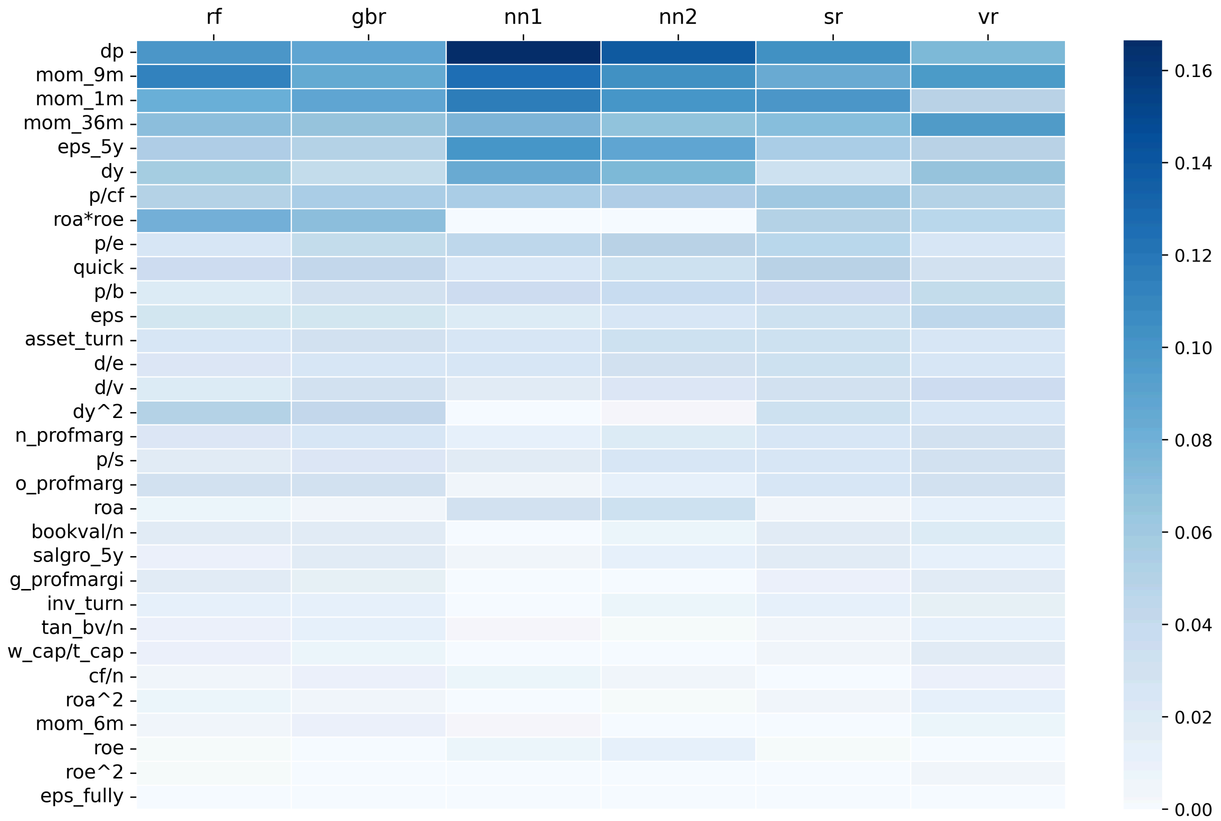


Figure 3.3: Variable importance heatmap. This figure depicts a heat map of the average relative variable importance ranks of each machine learning model described in the "Forecast models" section. The rankings are obtained by ranking the relative contribution of each variable within a specific model and summing the ranks across all models to obtain an overall rank. The rows are arranged in descending order according to overall rank. Darker cell colors indicate that the associated variable is more important to the model.

3.3 Portfolio Performances

Statistical performance is of secondary relevance to an investor. According to Leitch and Tanner [68], statistical measurements are only loosely related to the economic potential gain of a forecast. As a result, it is useful to determine if differences in statistical predictive performance transfer into differences in predictive capacity from an economic standpoint for a realistic trading strategy.
In order to do so, i assess the economic performance of our models by portfolio sorts, designing a new set of portfolios to directly, exploit machine learning forecasts. For each stock in our dataset, machine learning models generate predictions of future returns on a monthly basis.

Using these predictions, i sort stocks into deciles. At the end of each month, i sort stocks into deciles based on their predicted returns from the machine learning models. The stocks in decile 1 (Low) are those with the lowest predicted returns, while the stocks in decile 10 (High) are those with the highest predicted returns. Each decile represents a portfolio of stocks ranked by their expected performance for the following month, as forecasted by the models. Once the stocks are sorted into deciles, i calculate average return, realized return, standard deviation and sharpe ratio, on a monthly basis. These metrics help to evaluate how well the machine learning models' predictions align with the actual stock returns. In addition to evaluating the individual deciles, a High-Low (H-L) strategy is constructed. This strategy involves creating a long-short portfolio by going long on the stocks in decile 10 (those predicted to have the highest returns) and short on the stocks in decile 1 (those predicted to have the lowest returns). The results are display in table 3.4

Out-of-sample portfolio performance is remarkably similar to the previously disclosed results on machine learning forecast accuracy. Machine learning forecasts from all methods grows in proportion with realized returns monotonically with expection of decile 8 of NN2, decile 4, 5,7 of VR, decile 6 and 8 of SR, from 3 to 6 og GRB and 4, 7 and 8 of RF. Neural network methods once again outperform tree-based techniques and ensemble techniques due to the similarity of decile predictions with realized returns. In addition, for all except the most extreme deciles, the quantitative fit between anticipated returns and average realized returns employing neural networks is quite close, in particular for NN1. The greatest High - Low strategy is from NN1, which returns on average 2.56% every month. Its monthly volatility is 13.10% resulting in an monthly out-of-sample Sharpe ratio of 0.39%. Another remarakable high - low strategy is the one from NN2, with a montlhy average returns of 2.55%, a monthly standard deviation of 13.05% and a Sharpe ratio of 0.39%. The discussed techniques captures cross-sectional diversity in realized excess returns, although their ability to provide effective trading signals varies significantly.

	NN1				**RF**				**GBR**			
	Pred	Avg	SD	SR	Pred	Avg	SD	SR	Pred	Avg	SD	SR
1 (low)	-2.00	-0.82	6.36	-0.14	-3.65	-0.29	7.15	-0.05	-3.16	-0.26	7.13	-0.05
2	-0.70	-0.56	6.16	-0.11	-1.71	0.15	6.61	0.01	-1.23	-0.07	6.36	-0.03
3	-0.13	-0.03	6.23	-0.02	-1.03	0.30	6.38	0.03	-0.65	0.50	6.17	0.07
4	0.24	0.07	6.18	-0.01	-0.54	0.23	6.36	0.02	-0.26	0.33	6.46	0.04
5	0.53	0.41	6.14	0.05	-0.14	0.32	6.10	0.04	0.07	0.25	6.30	0.02
6	0.78	0.43	5.85	0.06	0.21	0.65	6.07	0.09	0.34	0.21	6.30	0.02
7	1.02	0.94	6.26	0.13	0.54	0.35	5.73	0.04	0.59	0.42	5.80	0.06
8	1.26	1.03	6.34	0.15	0.89	0.44	6.00	0.06	0.83	0.70	6.08	0.10
9	1.57	1.23	5.93	0.19	1.31	0.80	6.12	0.11	1.12	1.02	5.97	0.15
10 (high)	2.35	1.75	6.74	0.24	2.23	1.43	6.47	0.21	1.84	1.31	6.49	0.19
H-L	4.34	2.56	13.10	0.39	5.89	1.72	13.62	0.26	5.00	1.57	13.62	0.24

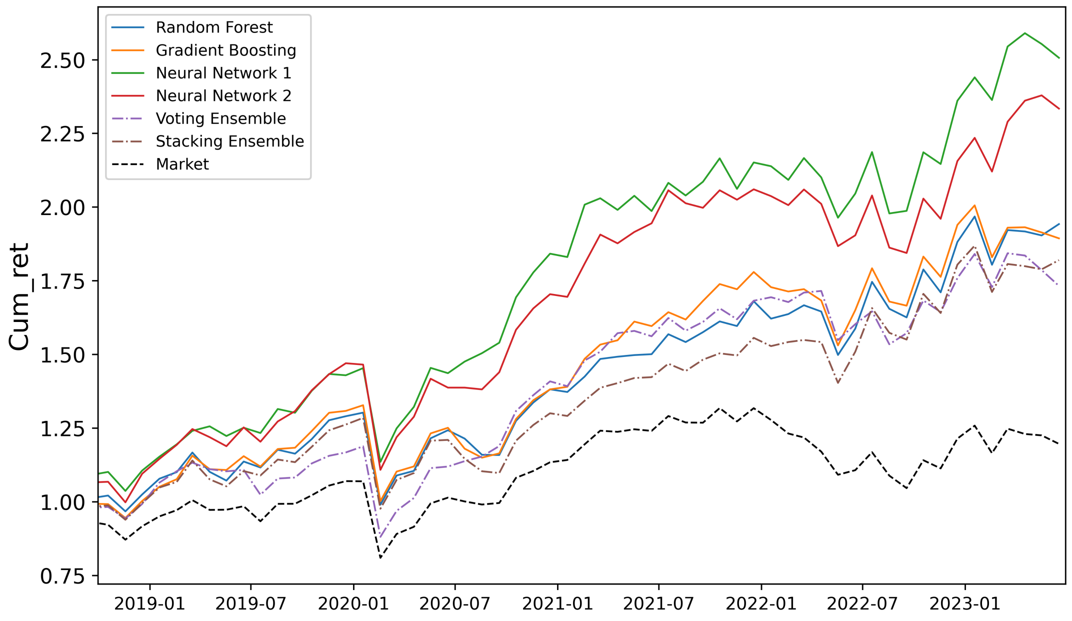
	NN2				VR				SR			
	Pred	Avg	SD	SR	Pred	Avg	SD	SR	Pred	Avg	SD	SR
1 (low)	-2.11	-0.86	6.49	-0.15	-6.31	0.17	6.85	0.01	-3.91	-0.36	7.24	-0.06
2	-0.85	-0.54	6.11	-0.11	-3.22	0.28	6.69	0.03	-1.63	0.23	6.43	0.02
3	-0.29	-0.38	6.31	-0.08	-1.99	0.36	6.28	0.04	-0.87	0.26	6.42	0.03
4	0.07	0.32	6.17	0.04	-1.03	0.08	6.16	-0.00	-0.33	0.32	6.45	0.03
5	0.36	0.53	5.76	0.07	-0.20	0.09	6.23	-0.00	0.09	0.40	6.08	0.05
6	0.62	0.68	6.08	0.10	0.58	0.57	6.31	0.07	0.46	0.31	6.02	0.04
7	0.89	0.93	6.25	0.13	1.31	0.43	6.03	0.05	0.82	0.50	5.73	0.07
8	1.22	0.83	6.40	0.11	2.17	0.63	6.33	0.08	1.18	0.47	5.90	0.06
9	1.66	1.24	6.43	0.18	3.35	0.70	6.25	0.10	1.61	0.97	6.44	0.13
10 (high)	2.68	1.68	6.56	0.24	6.16	1.09	6.00	0.16	2.58	1.30	6.29	0.19
H-L	4.80	2.55	13.05	0.39	12.47	0.92	12.86	0.15	6.49	1.66	13.52	0.25

Table 3.4: Decile Portfolios based on predicted expected returns from each model. This table presents the time series averages of predicted and realized returns of decile portfolios based on the expected returns obtained from each machine learning model

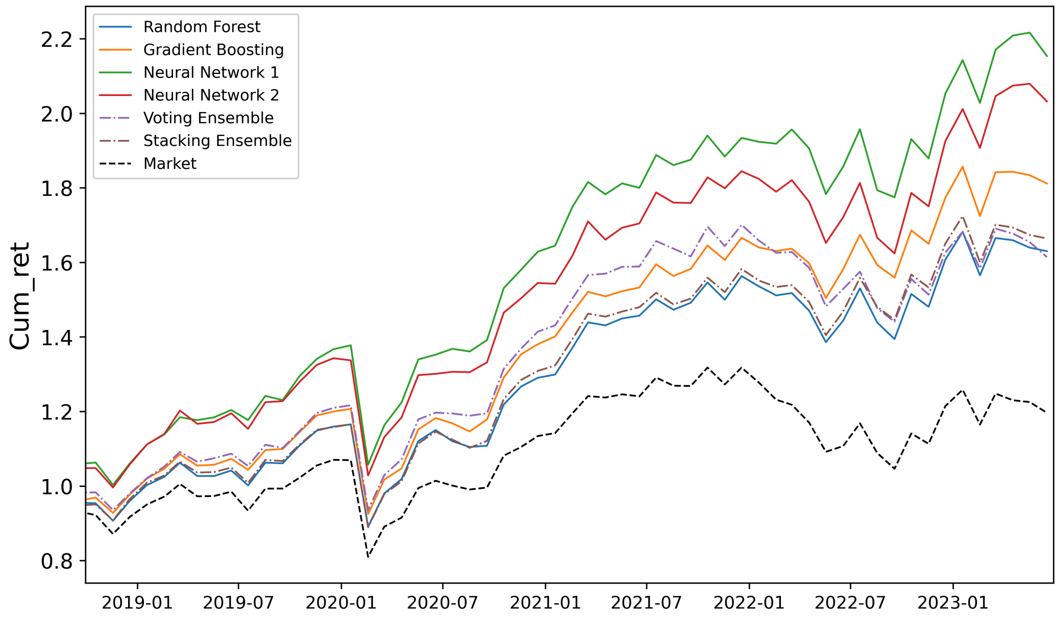
I will deploy four different ways to assess the economic significance of the machine learning models used. First, will be create two 'long only' portfolios. The originated portfolios take long positions in equities that the model predicts will have the greatest returns, ignoring low deciles. Two portfolios will be created using solely long positions and an equal weight criterion: the top 10% portfolio, which selects the top 10% of the best performing stocks, and the top 30% portfolio, which selects the top 30% of the best performing stocks. This twofold strategy allows to see whether there are economic gains even when evaluating a bigger universe of stocks to invest in (i.e. the top 30%). If the universe of equity in which to invest is limited, high-capital trading techniques may face liquidity issues and huge bid-ask spreads, rendering the strategy unprofitable. Table 3.5 summarizes the performance metrics for the two long methods.
Figure 3.4 shows the results of table 3.5. It plots the cumulative performance of expected return-sorted portfolios with monthly restructuring of the market portfolio and of each of the portfolios construced using the machine learning models described in the section Empirical Analysis, for both the top 30% and top 10% long only strategy.

Metrics	Top 10%						Top 30%						MKT
	RF	GBR	NN1	NN2	VR	SR	RF	GBR	NN1	NN2	VR	SR	MKT
Cumulative Return	1.94	1.89	2.51	2.33	1.73	1.82	1.63	1.81	2.15	2.03	1.62	1.66	1.23
Annual Return	0.14	0.13	0.18	0.17	0.11	1.12	0.10	0.12	0.15	0.14	0.10	0.10	0.05
Annual Standard Deviation	0.19	0.19	0.18	0.18	0.18	0.18	0.17	0.16	0.17	0.17	0.16	0.17	0.17
Annual Sharpe Ratio	0.73	0.70	1.02	0.91	0.63	0.67	0.60	0.73	0.89	0.82	0.60	0.62	0.27

Table 3.5: Risk and return characteristics of top 10% and top 30% long strategy



(a) Top 10%



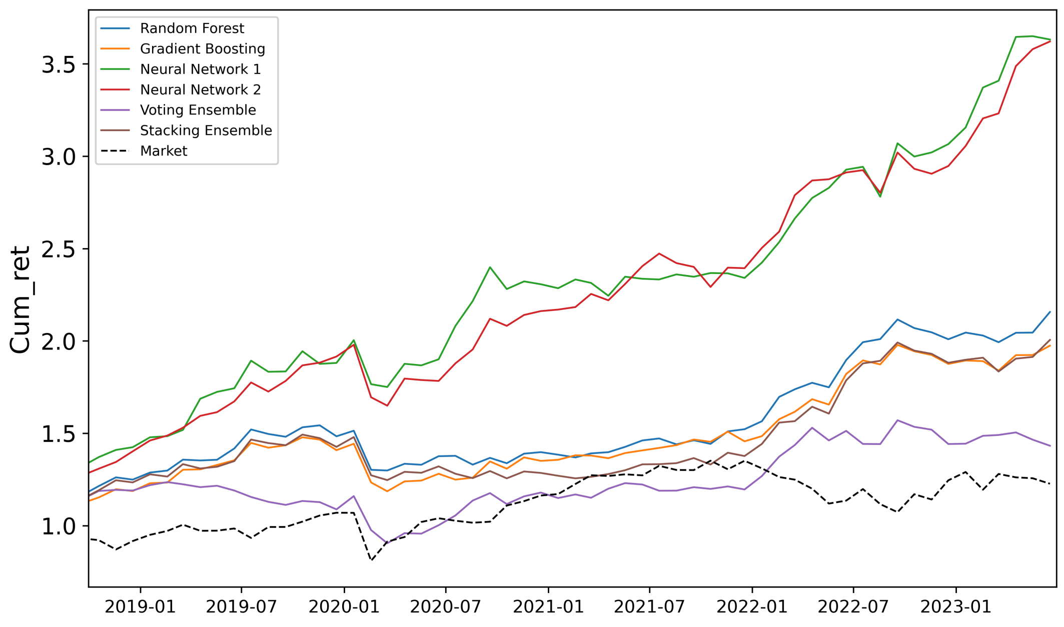
(b) top 30%

Figure 3.4: Cumulative performance of expected return-sorted portfolios with monthly restructuring - Long only strategy. This figure shows the performances of the top 10% and top 30% strategy going long on the best performing stocks. Each month the equal weights portfolios are rebalanced to match the best performing deciles.

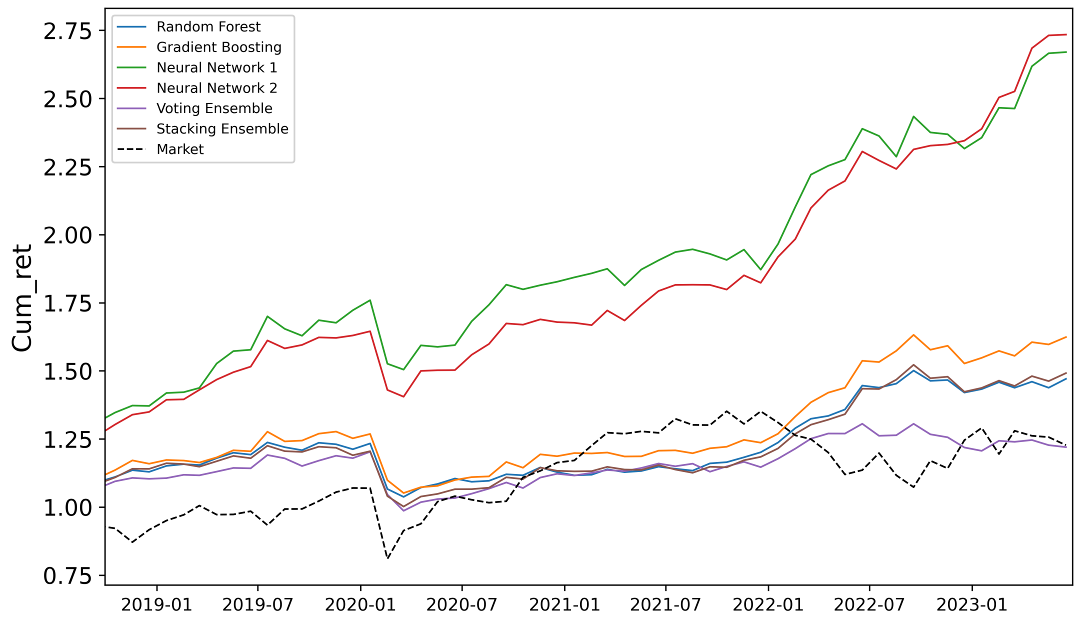
In line with Leitch and Tanner [68], is it observed that the poor statistical performance achieved in the preceding sections are not always a sign of poor economic performance of the machine learning models under consideration. It can be seen that all models in both strategies outperform the market portfolio. Again, neural models is proved superior to tree-based models and ensemble techniques, as also anticipated by the decile construction. In particular, given the greater proximity to realised returns, the neural models are able to more accurately predict which stocks to select each month as top performers, both in the case of the top 10% and the top 30% strategy. NN1 has the highest cumulative return of 2.51, followed by NN2 with 2.33, which is considerably larger than the market's cumulative return of 1.20. Furthermore, the Sharpe ratios of these models beat that of the market: 1.02 for NN1 and 0.91 for NN2, compared to the market Sharpe ratio of only 0.27. All models had an annual volatility of roughly 0.18, which is lower than the market volatility, indicating that these models do not introduce undue risk while providing higher returns. The outperformance shown here highlights the power of machine learning techniques and, in particular, neural networks in picking up more sophisticated patterns in the data than traditional models or market strategies. It is worth mentioning that even tree based models, had a rather good performance, too, with cumulative returns of 1.94 and 1.89, respectively. Even in the context of economic performances analysis, the ensemble techniques seems to perform the worst with cumulative returns of and 1.73 and 1.83, respectively, but still above the market return. When the algorithm selects the top 3 deciles, with the op 30% strategy, returns drop, but still remain above the market portfolio. Again, the neural models prove superior, with NN1 achieving a cumulative return of 2.15 and a sharpe ratio of 0.82. Overall the standard deviation levels remain the same, resulting in lower sharpe ratio levels for all the models considering the lower returns achieved. It is interesting to note that when increasing the investment pool from 10% to 30%, machine learning models seems to converge in performance. In this context, the cumulative returns of Random Forest, Gradient Boosting, and Neural Networks are much closer to each other. This suggests that with more stocks in the investment pool, the predictiveness of the machine learning models becomes somewhat watered down.

In addition to the two long only strategies, there are two mirror strategies that allow investors to take short positions. The main reasons for distinguishing between these two types of strategies are as follows:

- It is possible that some investors are not in a position to take short positions on equities: therefore, the aim of this analysis is also to test whether the use of these models is valid for an average investor with limited access to economic resources and trading strategies.
- It's crucial to test the quality of the predictions made by the models in the event of a bear market. If the cumulative return on the use of short positions is lower than the corresponding strategy when the only option is to go long, this means that the model is not capable of correctly predicting downward stock movements, but is more likely to correctly predict a return when the latter is positive.



(a) Top 10%



(b) top 30%

Figure 3.5: Cumulative performance of expected return-sorted portfolios with monthly restructuring - Long short strategy. This figure shows the performances of the top 10% and top 30% strategy going long on the best performing stocks and short on the worst performing stocks. Each month the equal weights portfolios are rebalanced to match the best and the worst performing deciles.

Two distinct portfolios will be constructed for the long-short strategy. The first will be a top 10% portfolio, comprising the top 10% of the best-performing stocks and the top 10% of the worst-performing stocks. The second will be a top 30% portfolio, comprising the top 30% of the best-performing stocks and the top 30% of the worst-performing stocks. It is evident that the strategy will entail going long in the top percentage and short in the worst. The results for each model are summarised in Table 3.6, with particular focus on the contribution that the short position, labelled 'Annual short return', makes to the total cumulative return of the portfolio. Figure 3.5 shows the results of table 3.6. It plots the cumulative performance of expected return-sorted portfolios with monthly restructuring of the market portfolio and of each of the portfolios constructed using the machine learning models described in the section Empirical Analysis, for both the top 30% and top 10% long and short strategy.

Metrics	**Top 10%**						**Top 30%**						**MKT**
	RF	GBR	NN1	NN2	SR	VR	RF	GBR	NN1	NN2	SR	VR	MKT
Cumulative Return	2.16	1.97	3.63	3.62	2.01	1.43	1.47	1.62	2.67	2.73	1.49	1.22	1.23
Annual Return	0.14	0.13	0.24	0.24	0.13	0.07	0.07	0.09	0.18	0.18	0.08	0.08	0.05
Annual Standard Deviation	0.11	0.12	0.14	0.12	0.12	0.14	0.09	0.09	0.11	0.10	0.09	0.08	0.18
Annual Sharpe Ratio	1.27	1.08	1.77	1.96	1.07	0.54	0.85	0.72	1.63	1.92	0.83	0.46	0.27
Annual Long Return	0.14	0.13	0.18	0.17	0.12	0.11	0.10	0.12	0.15	0.14	0.11	0.10	-
Annual Short Return	-0.01	0.00	-0.06	-0.07	-0.01	0.04	0.03	0.03	-0.03	-0.04	0.03	0.06	-

Table 3.6: Risk and return characteristics of top 10% and top 30% long short strategy

In the top 10% strategy shown in 3.5, neural networks models are again performing significantly better than the market and other machine learning models. The cumulative return for NN1 peaked 3.63, 3.62 for NN2. These returns are significantly higher than the market benchmark of 1.23, indicating that the capabilities to select top performers and correctly short underperformers. The annual short returns, (that are the contribution to the cumulative perfomance of going shorts on the selected stocks) for NN1 and NN2 are -0.06 and -0.07, respectively, signaling that the models are able to identify the worst stock performers and, hence, generate profits from shorting. This clearly outperforms the RF and GBR models, which have near-zero annual short returns of -0.01 and 0.00, respectively. The latter models were more conservative in their shorting strategy by opening positions with limited downside but also limiting the potential gain from short positions. This due to their lack of ability to predict returns accurately in line with the realized ones, as it was shown in table 3.4 The Sharpe ratios demonstrate the improvements with respect to the long only strategy. NN1 has a Sharpe ratio of 1.77, which is significantly greater than the 1.02 value achieved in the long-only strategy and far above the market's Sharpe ratio of 0.27. This increase in the Sharpe ratio supports the concept that short positions boost the risk-adjusted return of neural networks through higher returns as well as risk diversification. Ensemble models, such as the Stacking and Voting ensemble models, perform noticeably worse, with cumulative returns of 2.01 and 1.43, respectively, and lower, but stil high Sharpe ratios. This underperformance might partly be because of their generally weaker shorting ability; the Voting Ensemble has a positive short return of 0.04, indicating incorrect selection of the 10% worst performing stocks. Figure 3.5 shows that expanding the investment horizon to the top 30%, reduces the cumulative returns for most of the models, though neural networks still ranked first among all models. The annual short returns remain overall negative, with the models still apparently effective in their choice of stocks to short against. NN1 and NN2 report short returns of -0.03 and -0.04, respectively, which again shows that even within a wider universe of stocks, the neural networks are able to select those stocks that would ultimately will depicts negative returns. On the other hand, the short returns for both Random Forest and Gradient Boosting are slightly positive at 0.03, which suggests they are incorrectly shorting the stocks that instead have an increase in stock price. This weakness on the short side explains their relatively lower cumulative returns of 1.62 and 1.67, respectively, compared to the neural networks. The worst-performing model in this case is the Voting Regressor, which, due to an annual short return contribution of 0.06, underperforms the market portfolio by 0.01%, but with a lower standard deviation of 0.1%.

3.3.1 Transaction costs

Transaction expenses reduce the profitability of any investment. Investing strategies with high turnover rates may not outperform in terms of returns. This study aims to determine whether machine learning models outperform OLS due to higher turnover rates. Realistic transaction costs can be difficult to obtain, time-consuming, and expensive (Lesmond et al. [69]). Collins and Fabozzi [70] argue that genuine transaction costs are intrinsically unobservable. Therefore, estimating transaction costs is likely to be erroneous. The present study considers the average monthly turnover of the top 10% and top 30% of the investment strategies, both in the long-only and long-short strategy. The turnover is calculated as follows:

$$\text{Turnover} = \frac{1}{T}\sum_{t=1}^{T}\left(\sum_{i}\left|w_{i,t+1} - \frac{w_{i,t}(1+r_{i,t+1})}{\sum_j w_{j,t}(1+r_{j,t+1})}\right|\right), \tag{3.1}$$

where $w_{i,t}$ is the weight of stock i in the portfolio at time t.
Furthermore, in accordance with the methodology proposed by Cakici et al. [66], i also calculated the breakeven trading costs as the mean portfolio return divided by the mean turnover. Breakeven trading costs are those that refer to the amount beyond which a trading strategy remains profitable. This is extremely valuable in strategies with heavy rebalancing or high turnover, wherein modest transaction costs can shrink net returns considerably. Comparing breakeven trading costs with the actual or estimated trading costs in the market allows an investor or the portfolio manager to see how robust and practically applicable the trading strategy may be beyond its theoretically computed returns.
The results are showed in table 3.7 for both the long only and the long-short strategy:
In the L-Top 10% approach, neural networks have a considerably lower turnover of about 59% for NN1 and 60% for NN2, compared to others such as RF and GBR, which have turnovers of around 74-76%. Lower levels of such turnover would indicate that the conducted neural networks had more stable trading activity than the other models, supporting prior assessments that found greater performance and greater cumulative returns. These models also have higher break-even transaction costs (2.52% for NN1 and 2.33% for NN2), allowing them to absorb more transaction fees without losing profitability. This is because to their higher returns, which allow them to better absorb the impact of trading expenses than models with significantly lower returns, such as RF and GBR. The LS-Top 10% long-short strategy is more

Metrics	**L-Top 10%**						**L-Top 30%**					
	RF	**GBR**	**NN1**	**NN2**	**VR**	**SR**	**RF**	**GBR**	**NN1**	**NN2**	**VR**	**SR**
Avg monthly Turnover (%)	76	74	59	60	61	75	67	66	56	57	58	66
BE Transaction Costs (%)	1.49	1.47	2.52	2.33	1.55	1.338	1.26	1.51	2.21	2.05	1.41	1.33

Metrics	**LS-Top 10%**						**LS-Top 30%**					
	RF	**GBR**	**NN1**	**NN2**	**VR**	**SR**	**RF**	**GBR**	**NN1**	**NN2**	**VR**	**SR**
Avg monthly Turnover (%)	144	142	118	121	124	143	132	130	112	115	116	130
BE transaction Costs (%)	0.82	0.74	1.69	1.62	0.49	0.76	0.45	0.58	1.34	1.33	0.28	0.48

Table 3.7: Turnover and Break even trading cost for the long only strategies and long short strategies.

complex because it manages both long and short positions; hence, all turnover increases. For neural networks, turnover rises to 118-121%, and for Random Forest and Gradient Boosting Regressor, it equals 144-142%, respectively. In oppsition to [66] and in line with [47], neural network model show the lowest scores, while tree based models the highest ones. Despite the high turnovers, the break-even transaction costs remain high for neural networks: 1.69% for NN1 and 1.62% for NN2, confirming their dominance in overall performance. Models that exhibited difficulties in handling short positions, such as the Voting Ensemble, show very low break-even transaction costs of 0.49%, underlining its vulnerability to transaction costs in high-turnover environments. The top 30% strategies show similar trends: the overall turnover is lower, but still higher for the long-short version with respect to the long-only. Neural networks continue to highly outperform due to stronger returns, while tree based models and ensemble techniques remain more cost-sensitive.

Conclusion

This research has demonstrated that machine learning models, especially neural networks, hold significant advantages in predicting stock returns within the European equity market. It is presented that suitably tuned ML models outperform basic market-wide heuristics portfolios on long only and long-short strategies, even if a limited number of predictors is considered. The results point to the possibility that machine learning can successfully uncover complex nonlinear relationships in financial data, which would ensure enhanced predictive efficiency and better performance in portfolios creation. The study hereby provides evidence that machine learning models can scan achieve notable risk-adjusted outperformance in European markets, with a relatively low statistical significance from the out-of-sample tests. Given the relatively small sample of firms, the constrained time span, and the limited set of predictors in number and nature, this research contributes to the existing body of literature related to asset pricing using machine learning in European markets. In all the aspects of the analysis, neural network methods outperformed consistently compared with Ensemble techniques and tree-based methods. The scope of this research could be extended considering the following concepts:

- implementation of transaction costs for portfolios construction;
- consideration of stock liquidity and size in portfolios construction;
- rolling or expanding window datasplit, while considering a limited sample size and number of predictors;

Appendix A

Additional Tables and Figures

A.1 Empirical Analysis

NACE Code	**Description**
A	Agriculture, Forestry and Fishing
B	Mining and Quarrying
C	Manufacturing
D	Electricity, Gas, Steam and Air Conditioning Supply
E	Water Supply; Sewerage, Waste Management and Remediation Activities
F	Construction
G	Wholesale and Retail Trade; Repair of Motor Vehicles and Motorcycles
H	Transportation and Storage
I	Accommodation and Food Service Activities
J	Information and Communication
K	Financial and Insurance Activities
L	Real Estate Activities
M	Professional, Scientific and Technical Activities
N	Administrative and Support Service Activities
O	Public Administration and Defence; Compulsory Social Security
P	Education
Q	Human Health and Social Work Activities
R	Arts, Entertainment and Recreation
S	Other Service Activities
T	Activities of Households as Employers; Undifferentiated Goods and Services
U	Activities of Extraterritorial Organisations and Bodies

Table A.1: NACE Codes and Descriptions

Predictor	**Corr /w return**
roa	0.035049
o_profmargi	0.033561
eps_basic	0.028434
eps_fully	0.028329
p/s	0.024785
eps_5y	0.024633
asset_tunr	0.019406
roe	0.017956
quick	0.017420
roic	0.016616
p/e	0.015837
current	0.015732
g_profmargi	0.014576
salgro_5y	0.007344
w_cap/t_cap	0.006884
inv_turn	0.006158
p/cf	0.004585
p/b	0.004499
cf/n	0.002114
d/v	-0.001920
n_profmargi	-0.005147
d/e	-0.007664
tan_bv/n	-0.008129
bookval/n	-0.011999
eps	-0.013412
dp	-0.025744
dy	-0.167749

Table A.2: Predictors time-series correlations with returns

Predictor	**VIF**
bookval/n	4.964133
roa	3.861362
tan_bv/n	2.702279
cf/n	2.512495
roe	2.334957
eps_fully	2.278092
p/b	1.804407
asset_turn	1.577246
p/s	1.459381
g_profmargi	1.394601
w_cap/t_cap	1.268088
dy	1.259769
o_profmargi	1.247353
p/cf	1.231761
dp	1.206332
d/e	1.168680
eps_5y	1.162011
d/v	1.106803
eps	1.068407
salgro_5y	1.049787
n_profmargi	1.044080
p/e	1.043180
monthly_returns	1.033329
inv_turn	1.002257

Table A.3: Variable importance factors. This table is constructed by the studying the correllation with returns of the factors that are still in the dataset after the elimination based on the correlation matrix. Since there are no values higher than 5 these are the predictors that will be deployed in the analysis

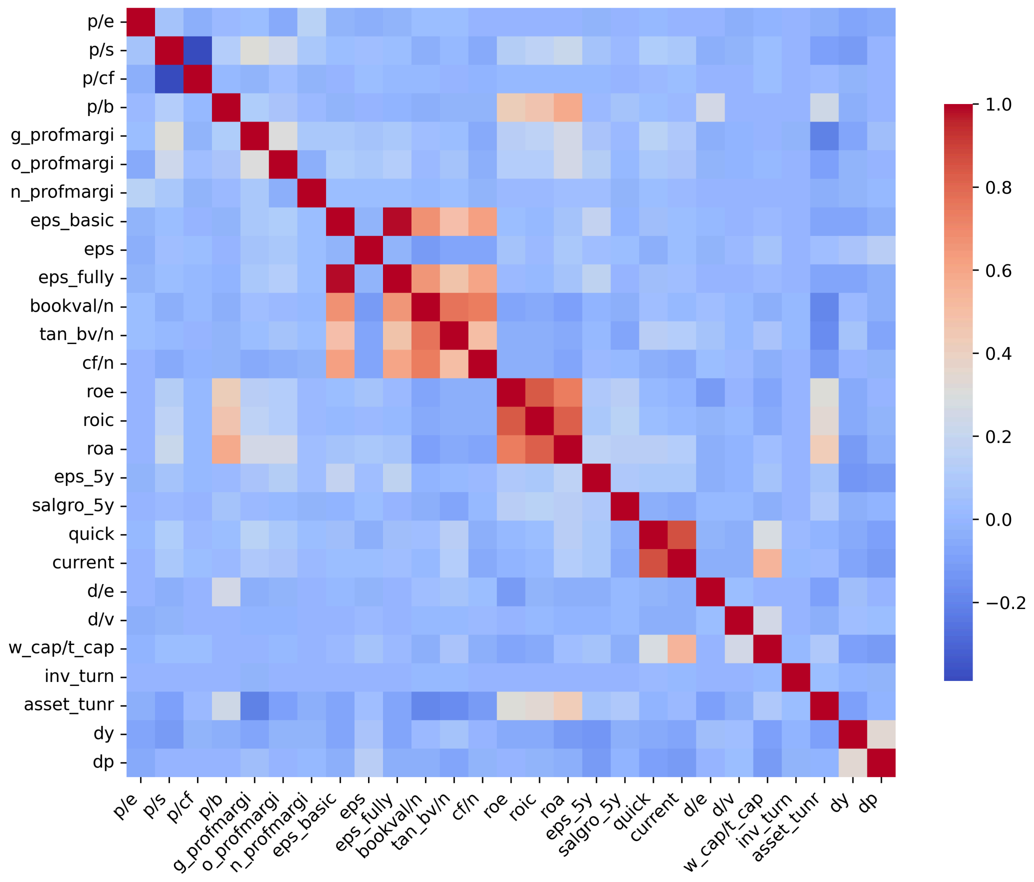


Figure A.1: Predictors correlation matrix

Metric	**Actual (%)**	**RF (%)**	**GBR (%)**	**NN 1 (%)**	**NN 2 (%)**
Mean Return (Monthly)	0.44	0.20	0.16	0.77	0.43
Std Dev (Monthly)	8.34	3.43	3.22	5.92	5.94
Sharpe Ratio (Monthly)	4.06	2.84	1.99	11.37	5.47
Mean Return (Annual)	5.27	2.37	1.97	9.29	5.10
Std Dev (Annual)	28.90	11.87	11.15	20.52	20.57
Sharpe Ratio (Annual)	14.07	9.85	6.89	39.40	18.96

Table A.4: Summary statistics for Market, Random Forest, Gradient Boosting, and Neural Networks

Appendix B

Python Code

The code was implemented using Jupyter Notebook. The code below is structured as follows:

1. `Data cleaning and predictors modeling`
2. `Data splitting and models preparation`
3. `Training, tuning and testing the ML models`
4. `Classification analysis`
5. `Statistical Analysis`
6. `Variable Importance`
7. `Portfolio Performances`

1 Data cleaning and predictors modeling

```
[ ]: import pandas as pd
     import openpyxl as op
     import matplotlib.pyplot as plt
     import numpy as np
     from scipy import stats
     import seaborn as sns
     import matplotlib.pyplot as plt
     import os
     from statsmodels.stats.outliers_influence import variance_inflation_factor
     from scipy.stats.mstats import winsorize
     from IPython.display import display
     from sklearn.impute import KNNImputer
     from sklearn.model_selection import train_test_split, TimeSeriesSplit
     from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
     from sklearn.model_selection import GridSearchCV
     from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
     from scikeras.wrappers import KerasRegressor
     from keras.models import Sequential
     from keras.layers import Dense, Input
     from keras.optimizers import Adam
     from keras.callbacks import EarlyStopping
     from keras import regularizers
     from tensorflow.keras.callbacks import ReduceLROnPlateau
     import tensorflow as tf
     from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
      ↪f1_score
     import statsmodels.api as sm
     from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
     from sklearn.ensemble import StackingRegressor
     from sklearn.linear_model import LinearRegression
```

```
[ ]: #Load the Excel file into a DataFrame
     df = pd.read_excel('/Users/samueletroiano/Desktop/code/European_stocks_panel2.
      ↪xlsx')
     dimensions = df.shape

     # Function to rename the first 229 observations for a given company
     def rename_first_half(df, company, new_name):
         company_indices = df[df['Companies'] == company].index
         df.loc[company_indices[:229], 'Companies'] = new_name

     # Rename the first 229 observations for SAN, AIR, and SHEL
```

```
rename_first_half(df, 'SAN', 'SAN1')
rename_first_half(df, 'AIR', 'AIR2')
rename_first_half(df, 'SHEL', 'SHEL2')
```

```
[ ]: #Now since there are missing values in our dataset we need to clear it. in order␣
      ↪to do that we replace the
     #crossectional median at each time t for each predictor.

     placeholder_values = ['', '$$ER: 4540,NO DATA VALUES FOUND', '$$ER: 2376,NO DATA␣
      ↪AVAILABLE']

     def replace_with_cross_section_median(group):
         for column in group.columns[3:]:
             group[column] = group[column].replace(placeholder_values, pd.NA)
             cross_section_median = group[column].median()
             group[column].fillna(cross_section_median, inplace=True)
         return group
     df_filled = df.groupby('Time', group_keys=False).
      ↪apply(replace_with_cross_section_median)
     #dividing columm by 100 since some ratio are not in percentages
     columns_to_divide = ['GROSS PROFIT MARGIN', 'OPERATING PROFIT MARGIN', 'NET␣
      ↪PROFIT MARGIN ','ROE', 'ROA','SALES GROWTH - 5 YR',
            'Debt / CE ', 'Debt/TOTAL CAPITAL/STD', 'work capital / tot␣
      ↪capital','DIVIDEND YIELD',
            'DIVIDEND PAYOUT (% EARNINGS)']
     df[columns_to_divide] = df[columns_to_divide]/100
```

```
[ ]: #calculating returns
     df['monthly_returns'] = np.log(df['Price'] / df['Price'].shift(1))
     print("The dimensions of the dataset are:", df.shape)
     target_column = df['monthly_returns']

     #eliminate the returns on the first month for each stocks since they are␣
      ↪computed using prices of the company before
     df['Time'] = pd.to_datetime(df['Time'])
     specific_date = pd.to_datetime('2004-06-18')
     df = df[df['Time'] != specific_date]
     print("The dimensions of the dataset are:", df.shape)
```

```
[ ]: #correlation matrix and elimination of too much correlated feature

     firm_char = df.iloc[:, 4:31]
     corr_matrix = firm_char.corr()
     plt.figure(figsize=(10, 8))
     sns.heatmap(corr_matrix, cmap='coolwarm', annot=False, square=True,␣
      ↪cbar_kws={"shrink": .75})
     plt.xticks(rotation=45, ha='right')
     plt.yticks(rotation=0)
     plt.title('Feature Correlation Matrix', fontsize=16)
     plt.show()

     corr_matrix = firm_char.corr()
     upper_triangle = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).
      ↪astype(bool))
     high_corr = [(column, row) for column in upper_triangle.columns for row in␣
      ↪upper_triangle.index if np.abs(upper_triangle.loc[row, column]) > 0.8]

     print("Highly correlated pairs:")
     print(high_corr)

     #now we drop the variables with correllation > 0.8
     df = df.drop(['EPS - BASIC ', 'ROIC', ' CURRENT RATIO'], axis=1)
     print("The dimensions of the new dataset are:", df.shape
```

2 Data splitting and models preparation

[ ]:
```
#continue of correlation analysis - VIF

P = df.iloc[:,4:29]
P = sm.add_constant(P)
vif_data = pd.DataFrame()
vif_data['Predictor'] = P.columns
vif_data['VIF'] = [variance_inflation_factor(P.values, i) for i in range(P.
 ↪shape[1])]
vif_data = vif_data[vif_data['Predictor'] != 'const']
vif_data = vif_data.sort_values(by='VIF', ascending=False)

#winsorization
columns_to_winsorize = df.iloc[:, 4:37]
df.iloc[:, 4:37] = columns_to_winsorize.apply(lambda x: pd.Series(winsorize(x.
 ↪dropna(), limits=[0.01, 0.01]), index=x.dropna().index).reindex_like(x))

firm_char2 = df.iloc[:, 4:28]
monthly_returns = df['monthly_returns']
correlations = firm_char.corrwith(monthly_returns)
print(correlations.sort_values(ascending=False))

#creation of polynomial feature
selected_features = df[['ROA', 'DIVIDEND YIELD', 'ROE']]
squared_features = selected_features ** 2
squared_features.columns = [f"{col}^2" for col in squared_features.columns]
df = pd.concat([df, squared_features], axis=1)
df['ROA*ROE'] = df['ROA'] * df['ROE']
```

```
[ ]: #construct momentum feature

df['mom_1m'] = df.groupby('Companies')['Price'].transform(lambda x: np.log(x / x.
 ↪shift(1)))
df['mom_6m'] = df.groupby('Companies')['Price'].transform(lambda x: np.log(x / x.
 ↪shift(6)))
df['mom_9m'] = df.groupby('Companies')['Price'].transform(lambda x: np.log(x / x.
 ↪shift(9)))
df['mom_36m'] = df.groupby('Companies')['Price'].transform(lambda x: np.log(x /␣
 ↪x.shift(36)))

def apply_spline_interpolation(group, column_name):
    return group[column_name].interpolate(method='spline', order=2)

df['mom_1m'] = df.groupby('Companies', group_keys=False).apply(lambda group:␣
 ↪apply_spline_interpolation(group, 'mom_1m'))
df['mom_6m'] = df.groupby('Companies', group_keys=False).apply(lambda group:␣
 ↪apply_spline_interpolation(group, 'mom_6m'))
df['mom_9m'] = df.groupby('Companies', group_keys=False).apply(lambda group:␣
 ↪apply_spline_interpolation(group, 'mom_9m'))
df['mom_36m'] = df.groupby('Companies', group_keys=False).apply(lambda group:␣
 ↪apply_spline_interpolation(group, 'mom_36m'))

imputer = KNNImputer(n_neighbors=5)
# Apply KNN imputation to the columns with remaining NaNs
df[['mom_1m', 'mom_6m','mom_9m','mom_36m']] = df.groupby('Companies',␣
 ↪group_keys=False).apply(
    lambda group: pd.DataFrame(imputer.fit_transform(group[['mom_3m', 'mom_6m',␣
 ↪'mom_9m','mom_36m']]),
                               columns=['mom_\m', 'mom_6m', 'mom_9m','mom_36m'],
                               index=group.index)
)

df.insert(36, 'monthly_returns', df.pop('monthly_returns'))

#summary statistics
selected_columns = df.iloc[:, 4:37]
avg = selected_columns.mean()
std = selected_columns.std()

summary_df = pd.DataFrame({
    'Avg': avg,
    'Std': std,
})
#ranking and normalization of variable using Kelly et al. (2019) method
```

```
def rank_normalize_kelly(series):
    ranks = series.rank(method='average')
    n = len(series)
    normalized_ranks = 2 * ranks / (n + 1) - 1
    return normalized_ranks
df = df.copy()

for col in df.columns[4:32]:
    df[col] = df.groupby('Time')[col].transform(rank_normalize_kelly)
```

[ ]:
```
#after having 'cleaned' a bit the data and creating all the features we were␣
↪missing we now can start
#first we need to create our target variable. that is the next month return
df['monthly_target_return'] = df['monthly_returns'].shift(-1)
#now we set up a target
df['target'] = (df['monthly_target_return'] > 0).astype(int)
df['monthly_target_return'].to_csv('monthly_target_return.txt', index=False,␣
↪header=False)

#cross sectional mean
grouped = df.groupby('Time')
cross_sectional_mean = grouped['monthly_target_return'].mean()
annual_mean = cross_sectional_mean.resample('Y').mean()
plt.figure(figsize=(6, 3))
plt.plot(annual_mean.index.year, annual_mean, marker='o', linestyle='-',␣
↪color='r')
plt.xlabel('')
plt.ylabel('')
plt.title('Cross-Sectional Mean Return by Year')
plt.grid(True)
years = annual_mean.index.year
plt.xticks(ticks=years[::3], labels=years[::3])
plt.show()
```

3 Training, tuning and testing the ML models

[ ]:
```
#Data splititng
train_size = 100 # 8.5 years of monthly data
val_size = 60     # 5 years of monthly data
test_size = 68    # 5.5 years of monthly data

def fixed_window_split(df, train_size, val_size, test_size):
    splits = {}

    for company in df['Companies'].unique():
        company_df = df[df['Companies'] == company]
        if len(company_df) >= (train_size + val_size + test_size):
            # Training data
            X_train = company_df.iloc[:train_size, 4:36]  # Firm characteristics␣
 ↪(columns 5 to 36)
            y_train_reg = company_df.iloc[:train_size, 37]  # Next month's␣
 ↪return target (column 38)
            y_train_class = company_df.iloc[:train_size, 38]  # Binary indicator␣
 ↪(column 39)

            # Validation data
            X_val = company_df.iloc[train_size:train_size + val_size, 4:36]
            y_val_reg = company_df.iloc[train_size:train_size + val_size, 37]
            y_val_class = company_df.iloc[train_size:train_size + val_size, 38]

            # Test data
            X_test = company_df.iloc[train_size + val_size:train_size + val_size␣
 ↪+ test_size, 4:36]
            y_test_reg = company_df.iloc[train_size + val_size:train_size +␣
 ↪val_size + test_size, 37]
            y_test_class = company_df.iloc[train_size + val_size:train_size +␣
 ↪val_size + test_size, 38]

            splits[company] = {
                'X_train': X_train,
                'y_train_reg': y_train_reg,
                'y_train_class': y_train_class,
                'X_val': X_val,
                'y_val_reg': y_val_reg,
                'y_val_class': y_val_class,
                'X_test': X_test,
                'y_test_reg': y_test_reg,
                'y_test_class': y_test_class
            }

    return splits

fixed_splits = fixed_window_split(df, train_size, val_size, test_size)

# Combine the training, validation, and test sets across all companies
X_train_combined = pd.concat([fixed_splits[company]['X_train'] for company in␣
 ↪df['Companies'].unique()], axis=0)
y_train_combined = pd.concat([fixed_splits[company]['y_train_reg'] for company␣
 ↪in df['Companies'].unique()], axis=0)
```

```
X_val_combined = pd.concat([fixed_splits[company]['X_val'] for company in␣
↪df['Companies'].unique()], axis=0)
y_val_combined = pd.concat([fixed_splits[company]['y_val_reg'] for company in␣
↪df['Companies'].unique()], axis=0)

X_test_combined = pd.concat([fixed_splits[company]['X_test'] for company in␣
↪df['Companies'].unique()], axis=0)
y_test_combined = pd.concat([fixed_splits[company]['y_test_reg'] for company in␣
↪df['Companies'].unique()], axis=0)
```

```
[ ]: # Function to create the first neural network model with 1 hidden layer
def create_nn_1model(learning_rate=0.001, l2_reg=0.001):
    model = Sequential()
    model.add(Input(shape=(X_train_combined.shape[1],)))
    model.add(Dense(32, activation='relu', kernel_regularizer=regularizers.
↪l2(l2_reg)))
    model.add(Dense(1, activation='linear'))
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model

# Function to create the second neural network model with 2 hidden layers
def create_nn_2model(learning_rate=0.001, l2_reg=0.001):
    model = Sequential()
    model.add(Input(shape=(X_train_combined.shape[1],)))
    model.add(Dense(64, activation='relu', kernel_regularizer=regularizers.
↪l2(l2_reg)))
    model.add(Dense(32, activation='relu', kernel_regularizer=regularizers.
↪l2(l2_reg)))
    model.add(Dense(1, activation='linear'))
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='mean_squared_error')

    return model
early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
↪restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss' , mode = 'min', factor=0.5,␣
↪patience=5, min_lr=0.0001)

# Initialize models
rf_model = RandomForestRegressor(random_state=42)
gbr_model = GradientBoostingRegressor(random_state=42)
nn1_model = KerasRegressor(model=create_nn_1model, verbose=0,␣
↪callbacks=[early_stopping, reduce_lr])
nn2_model = KerasRegressor(model=create_nn_1model, verbose=0,␣
↪callbacks=[early_stopping, reduce_lr])
```

```
# Grid Search Parameters

nn_param_grid_1 = {
    'learning_rate_init': [0.001,0.003,0.005],
    'batch_size': [8,16,32,64,128,256],
    'epochs': [250,500,750,1000,1250],
    'l2_reg': [0.001,0.003,0.005],
}


nn_param_grid_2 = {
    'learning_rate_init': [0.001,0.003,0.005],
    'batch_size': [8,16,32,64,128,256],
    'epochs': [250,500,750,1000,1250],
    'l2_reg': [0.001,0.003,0.005],
}

rf_param_grid = {
    'n_estimators': [300,400,600,800],
    'max_depth': [25,30,35,45],
    'max_features': ['sqrt','log2',0.6],
    'min_samples_leaf': [2,5,10,15],
    'bootstrap': [True],

}

gbr_param_grid = {
    'n_estimators': [350,500,600,750],
    'learning_rate': [0.001,0.003,0.005],
    'max_depth': [10,15,20],
    'subsample': [0.5,0.7,1],
    'min_samples_split': [10,15,20]

}
```

```
[ ]: def rmse(y_true, y_pred):
         return tf.sqrt(tf.reduce_mean(tf.square(y_true - y_pred)))

     rf_grid, gbr_grid, nn1_grid, nn2_grid = None, None, None, None

     best_models = {}

     models_and_grids = [
         ("NeuralNetwork_1HL", nn1_model, nn_param_grid_1),
         ("NeuralNetwork_2HL", nn2_model, nn_param_grid_2),
         ("RandomForest", rf_model, rf_param_grid),
         ("GradientBoosting", gbr_model, gbr_param_grid)

     ]

     results = []

     for model_name, model, param_grid in models_and_grids:
         print(f"Training and tuning {model_name}...")

         grid = GridSearchCV(estimator=model, param_grid=param_grid,␣
     ↪scoring='neg_mean_squared_error', cv=3, verbose=2)

         if "NeuralNetwork" in model_name:
             grid_result = grid.fit(X_train_combined, y_train_combined,␣
     ↪validation_data=(X_val_combined, y_val_combined), callbacks=[early_stopping])
             if model_name == "NeuralNetwork_1HL":
                 nn1_grid = grid_result
             elif model_name == "NeuralNetwork_2HL":
                 nn2_grid = grid_result
         else:
             grid_result = grid.fit(X_train_combined, y_train_combined)
             if model_name == "RandomForest":
                 rf_grid = grid_result
             elif model_name == "GradientBoosting":
                 gbr_grid = grid_result

         best_model = grid_result.best_estimator_
         best_params = grid_result.best_params_

         best_models[model_name] = best_model

         val_predictions = best_model.predict(X_val_combined)
         val_rmse = rmse(y_val_combined, val_predictions)

         print(f"{model_name} Best Parameters: {best_params}")
         print(f"{model_name} Validation RMSE: {val_rmse}")
         results.append({
             'Model': model_name,
             'Best Params': best_params,
             'Validation RMSE': val_rmse
         })

     best_rf = best_models["RandomForest"]
     best_gbr = best_models["GradientBoosting"]
     best_nn_1 = best_models["NeuralNetwork_1HL"]
     best_nn_2 = best_models["NeuralNetwork_2HL"]
```

```
# Predictions on the test set
rf_test_predictions = best_rf.predict(X_test_combined)
gbr_test_predictions = best_gbr.predict(X_test_combined)
nn_1_test_predictions = best_nn_1.predict(X_test_combined)
nn_2_test_predictions = best_nn_2.predict(X_test_combined)
```

[ ]:
```
#Batch size graph
if nn1_grid is not None:
    batch_sizes_nn1 = nn1_grid.cv_results_['param_batch_size']
    rmse_values_nn1 = np.sqrt(-nn1_grid.cv_results_['mean_test_score'])

    results_df_nn1 = pd.DataFrame({
        'batch_size': batch_sizes_nn1,
        'rmse': rmse_values_nn1
    })

    mean_rmse_per_batch_nn1 = results_df_nn1.groupby('batch_size')['rmse'].mean()

    # Plot the mean RMSE for each batch size for NN1
    plt.figure(figsize=(6, 4))
    plt.plot(mean_rmse_per_batch_nn1.index, mean_rmse_per_batch_nn1.values,␣
 ↪marker='o', color='blue', label='Neural Network 1')
    plt.xlabel('Batch size')
    plt.ylabel('Mean RMSE')
    plt.title('Mean RMSE for different batch sizes (Neural Network 1)')
    plt.grid(True)
    plt.legend()
    plt.show()

if nn2_grid is not None:
    batch_sizes_nn2 = nn2_grid.cv_results_['param_batch_size']
    rmse_values_nn2 = np.sqrt(-nn2_grid.cv_results_['mean_test_score'])

    results_df_nn2 = pd.DataFrame({
        'batch_size': batch_sizes_nn2,
        'rmse': rmse_values_nn2
    })

    mean_rmse_per_batch_nn2 = results_df_nn2.groupby('batch_size')['rmse'].mean()

    # Plot the mean RMSE for each batch size for NN2
    plt.figure(figsize=(6, 4))
    plt.plot(mean_rmse_per_batch_nn2.index, mean_rmse_per_batch_nn2.values,␣
 ↪marker='o', color='green', label='Neural Network 2')
    plt.xlabel('Batch size')
    plt.ylabel('Mean RMSE')
    plt.title('Mean RMSE for different batch sizes (Neural Network 2)')
    plt.grid(True)
    plt.legend()
    plt.show()
```

```
[ ]: #epochs graph
if nn1_grid is not None:
    batch_sizes_nn1 = nn1_grid.cv_results_['param_epochs']
    rmse_values_nn1 = np.sqrt(-nn1_grid.cv_results_['mean_test_score'])

    results_df_nn1 = pd.DataFrame({
        'epochs': epochs_nn1,
        'rmse': rmse_values_nn1
    })

    mean_rmse_per_batch_nn1 = results_df_nn1.groupby('epochs')['rmse'].mean()

    # Plot the mean RMSE for each epochs for NN1
    plt.figure(figsize=(6, 4))
    plt.plot(mean_rmse_per_batch_nn1.index, mean_rmse_per_batch_nn1.values,␣
 ↪marker='o', color='blue', label='Neural Network 1')
    plt.xlabel('epochs')
    plt.ylabel('Mean RMSE')
    plt.title('')
    plt.grid(True)
    plt.legend()
    plt.show()

if nn2_grid is not None:
    batch_sizes_nn2 = nn2_grid.cv_results_['param_epochs']
    rmse_values_nn2 = np.sqrt(-nn2_grid.cv_results_['mean_test_score'])

    results_df_nn2 = pd.DataFrame({
        'epochs': epochs_nn2,
        'rmse': rmse_values_nn2
    })

    mean_rmse_per_batch_nn2 = results_df_nn2.groupby('epochs')['rmse'].mean()

    # Plot the mean RMSE for each batch size for NN2
    plt.figure(figsize=(6, 4))
    plt.plot(mean_rmse_per_batch_nn2.index, mean_rmse_per_batch_nn2.values,␣
 ↪marker='o', color='green', label='Neural Network 2')
    plt.xlabel('epochs')
    plt.ylabel('Mean RMSE')
    plt.title('')
    plt.grid(True)
    plt.legend()
    plt.show()
```

```
[ ]: #Summary statistics
     def calculate_statistics(returns, risk_free_rate=2.0):
         """
         Calculate the mean, standard deviation, and Sharpe ratio of a return series.

         Parameters:
         - returns: Array-like, the series of returns.
         - risk_free_rate: Float, the risk-free rate (default is 0).

         Returns:
         - mean_return: The mean return.
         - std_dev: The standard deviation of returns.
         - sharpe_ratio: The Sharpe ratio of the returns.
         """
         mean_return = np.nanmean(returns)
         std_dev = np.nanstd(returns)

         # Handle the case where std_dev is 0 to avoid division by zero
         if std_dev == 0:
             sharpe_ratio = np.nan  # Set Sharpe ratio to NaN or another placeholder
         else:
             sharpe_ratio = (mean_return - risk_free_rate) / std_dev

         return mean_return, std_dev, sharpe_ratio

     actual_returns = np.array(y_test_combined)
     rf_predictions = np.array(rf_test_predictions)
     gbr_predictions = np.array(gbr_test_predictions)
     nn_1_predictions = np.array(nn_1_test_predictions)
     nn_2_predictions = np.array(nn_2_test_predictions)

     actual_mean, actual_std, actual_sharpe = calculate_statistics(actual_returns)
     rf_mean, rf_std, rf_sharpe = calculate_statistics(rf_predictions)
     gbr_mean, gbr_std, gbr_sharpe = calculate_statistics(gbr_predictions)
     nn_1_mean, nn_1_std, nn_1_sharpe = calculate_statistics(nn_1_predictions)
     nn_2_mean, nn_2_std, nn_2_sharpe = calculate_statistics(nn_2_predictions)

     annual_actual_mean = actual_mean * 12
     annual_actual_std = actual_std * np.sqrt(12)
     annual_actual_sharpe = actual_sharpe * np.sqrt(12)

     annual_rf_mean = rf_mean * 12
     annual_rf_std = rf_std * np.sqrt(12)
     annual_rf_sharpe = rf_sharpe * np.sqrt(12)
```

```
annual_gbr_mean = gbr_mean * 12
annual_gbr_std = gbr_std * np.sqrt(12)
annual_gbr_sharpe = gbr_sharpe * np.sqrt(12)

annual_nn1_mean = nn_1_mean * 12
annual_nn1_std = nn_1_std * np.sqrt(12)
annual_nn1_sharpe = nn_1_sharpe * np.sqrt(12)

annual_nn2_mean = nn_2_mean * 12
annual_nn2_std = nn_2_std * np.sqrt(12)
annual_nn2_sharpe = nn_2_sharpe * np.sqrt(12)
```

4 Classification analysis

[ ]:
```
#Classification Analysis

rf_predictions_binary = (rf_predictions > 0).astype(int)
gbr_predictions_binary = (gbr_predictions > 0).astype(int)
nn1_predictions_binary = (nn_1_predictions > 0).astype(int)
nn2_predictions_binary = (nn_2_predictions > 0).astype(int)

actual_binary = (actual_returns > 0).astype(int)

# Random Forest metrics
rf_accuracy = accuracy_score(actual_binary, rf_predictions_binary)
rf_precision = precision_score(actual_binary, rf_predictions_binary)
rf_recall = recall_score(actual_binary, rf_predictions_binary)
rf_f1 = f1_score(actual_binary, rf_predictions_binary)

# Gradient Boosting metrics
gbr_accuracy = accuracy_score(actual_binary, gbr_predictions_binary)
gbr_precision = precision_score(actual_binary, gbr_predictions_binary)
gbr_recall = recall_score(actual_binary, gbr_predictions_binary)
gbr_f1 = f1_score(actual_binary, gbr_predictions_binary)

# Neural Network 1 metrics
nn1_accuracy = accuracy_score(actual_binary, nn1_predictions_binary)
nn1_precision = precision_score(actual_binary, nn1_predictions_binary)
nn1_recall = recall_score(actual_binary, nn1_predictions_binary)
nn1_f1 = f1_score(actual_binary, nn1_predictions_binary)

# Neural Network 2 metrics
nn2_accuracy = accuracy_score(actual_binary, nn2_predictions_binary)
nn2_precision = precision_score(actual_binary, nn2_predictions_binary)
nn2_recall = recall_score(actual_binary, nn2_predictions_binary)
nn2_f1 = f1_score(actual_binary, nn2_predictions_binary)
```

```
#confusion matrix
def plot_confusion_matrix(y_true, y_pred, title="Confusion Matrix"):
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(cmap=plt.cm.Blues)
    plt.title(title)
    plt.show()

plot_confusion_matrix(actual_binary, rf_predictions_binary, title="Random Forest␣
 ↪Confusion Matrix")
plot_confusion_matrix(actual_binary, gbr_predictions_binary, title="Gradient␣
 ↪Boosting Confusion Matrix")
plot_confusion_matrix(actual_binary, nn1_predictions_binary, title="Neural␣
 ↪Network 1 Confusion Matrix")
plot_confusion_matrix(actual_binary, nn2_predictions_binary, title="Neural␣
 ↪Network 2 Confusion Matrix")
```

```
[ ]: #Stacking and Voting Regressor - Ensemble
y_test_combined = pd.concat([split['y_test_reg'] for split in fixed_splits.
 ↪values()], axis=0)

best_nn_1_no_callbacks = KerasRegressor(model=best_nn_1.model, verbose=0)
best_nn_2_no_callbacks = KerasRegressor(model=best_nn_2.model, verbose=0)

stacking_regressor = StackingRegressor(
    estimators=[
        ('rf', best_rf),
        ('gbr', best_gbr),
        ('nn1', best_nn_1_no_callbacks),
        ('nn2', best_nn_2_no_callbacks)

    ],
    final_estimator=LinearRegression()
)

X_train_combined.fillna(X_train_combined.mean(), inplace=True)
y_train_combined.fillna(y_train_combined.mean(), inplace=True)
X_test_combined.fillna(X_test_combined.mean(), inplace=True)
y_test_combined.fillna(y_test_combined.mean(), inplace=True)
X_train_combined = pd.concat([split['X_train'] for split in fixed_splits.
 ↪values()])
y_train_combined = pd.concat([split['y_train_reg'] for split in fixed_splits.
 ↪values()])

stacking_regressor.fit(X_train_combined, y_train_combined)
```

```
voting_regressor = VotingRegressor(
    estimators=[
        ('rf', best_rf),
        ('gbr', best_gbr),
        ('nn1', best_nn_1_no_callbacks),
        ('nn2', best_nn_2_no_callbacks)
    ]
)

voting_regressor.fit(X_train_combined, y_train_combined)

stacking_predictions = stacking_regressor.predict(X_test_combined)
stacking_mse = mean_squared_error(y_test_combined, stacking_predictions)
voting_predictions = voting_regressor.predict(X_test_combined)
voting_mse = mean_squared_error(y_test_combined, voting_predictions)
```

[ ]:
```
#Classification Analysis of combined models
y_test_binary = (y_test_combined > 0).astype(int)

stacking_predictions_binary = (stacking_predictions > 0).astype(int)
voting_predictions_binary = (voting_predictions > 0).astype(int)

stacking_accuracy = accuracy_score(y_test_binary, stacking_predictions_binary)
stacking_precision = precision_score(y_test_binary, stacking_predictions_binary)
stacking_recall = recall_score(y_test_binary, stacking_predictions_binary)
stacking_f1 = f1_score(y_test_binary, stacking_predictions_binary)

voting_accuracy = accuracy_score(y_test_binary, voting_predictions_binary)
voting_precision = precision_score(y_test_binary, voting_predictions_binary)
voting_recall = recall_score(y_test_binary, voting_predictions_binary)
voting_f1 = f1_score(y_test_binary, voting_predictions_binary)

metrics_data = {
    'Metric': ['Accuracy', 'Precision', 'Recall', 'F1 Score'],
    'Stacking Regressor': [stacking_accuracy, stacking_precision,␣
 ↪stacking_recall, stacking_f1],
    'Voting Regressor': [voting_accuracy, voting_precision, voting_recall,␣
 ↪voting_f1]
}

metrics_df = pd.DataFrame(metrics_data)

y_test_binary = (y_test_combined > 0).astype(int)

stacking_predictions_binary = (stacking_predictions > 0).astype(int)
voting_predictions_binary = (voting_predictions > 0).astype(int)
```

```
stacking_accuracy = accuracy_score(y_test_binary, stacking_predictions_binary)
stacking_precision = precision_score(y_test_binary, stacking_predictions_binary)
stacking_recall = recall_score(y_test_binary, stacking_predictions_binary)
stacking_f1 = f1_score(y_test_binary, stacking_predictions_binary)

voting_accuracy = accuracy_score(y_test_binary, voting_predictions_binary)
voting_precision = precision_score(y_test_binary, voting_predictions_binary)
voting_recall = recall_score(y_test_binary, voting_predictions_binary)
voting_f1 = f1_score(y_test_binary, voting_predictions_binary)

metrics_data = {
    'Metric': ['Accuracy', 'Precision', 'Recall', 'F1 Score'],
    'Stacking Regressor': [stacking_accuracy, stacking_precision,␣
 ↪stacking_recall, stacking_f1],
    'Voting Regressor': [voting_accuracy, voting_precision, voting_recall,␣
 ↪voting_f1]
}

metrics_df = pd.DataFrame(metrics_data)

plot_confusion_matrix(y_test_binary, stacking_predictions_binary,␣
 ↪title="Stacking Regressor Confusion Matrix")
plot_confusion_matrix(y_test_binary, voting_predictions_binary, title="Voting␣
 ↪Regressor Confusion Matrix")
```

5 Statistical Analysis

[ ]:
```
#Statistical Analysis

num_stocks = len(df['Companies'].unique())
mask = ~np.isnan(y_test_combined)

def pseudo_r2_oos(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum(y_true ** 2)
    return 1 - ss_res / ss_tot

def newey_west_standard_error(diffs, lag=1):

    diffs = np.asarray(diffs)
    T = len(diffs)
    gamma_0 = np.sum(diffs ** 2) / T
    gamma_lags = np.array([np.sum(diffs[:T-l] * diffs[l:]) / T for l in range(1,␣
 ↪lag+1)])
    nw_var = gamma_0 + 2 * np.sum(gamma_lags)
    nw_std_error = np.sqrt(nw_var / T)
    return nw_std_error

def diebold_mariano_test_crossectional(y_true, y_pred1, y_pred2, num_stocks):

    e1 = (y_true - y_pred1).to_numpy()
    e2 = (y_true - y_pred2).to_numpy()
    diffs = e1 ** 2 - e2 ** 2
    d_t = np.mean((e1 ** 2 - e2 ** 2).reshape(-1, num_stocks), axis=1)
    d_mean = np.mean(d_t)
    d_var_nw = newey_west_standard_error(d_t)
    dm_stat = d_mean / d_var_nw
    p_value = 2 * (1 - stats.norm.cdf(np.abs(dm_stat)))

    return dm_stat, p_value

model_names = ["Random Forest", "Gradient Boosting", "Neural Network 1", "Neural␣
 ↪Network 2","Stacking Regressor", "Voting Regressor"]
model_predictions = [
    rf_predictions,
    gbr_predictions,
    nn_1_predictions,
    nn_2_predictions,
    stacking_predictions,
    voting_predictions
]
metrics = {
    "Model": [],
    "MSE": [],
    "MAE": [],
    "R^2 OOS": [],
    "Pseudo R^2 OOS": []
}

for name, predictions in zip(model_names, model_predictions):
    mse = mean_squared_error(y_test_combined[mask], predictions[mask])
    mae = mean_absolute_error(y_test_combined[mask], predictions[mask])
    r2 = r2_score(y_test_combined[mask], predictions[mask])
    pseudo_r2 = pseudo_r2_oos(y_test_combined[mask], predictions[mask])

    metrics["Model"].append(name)
    metrics["MSE"].append(mse)
    metrics["MAE"].append(mae)
    metrics["R^2 OOS"].append(r2)
    metrics["Pseudo R^2 OOS"].append(pseudo_r2)

metrics_df = pd.DataFrame(metrics)
```

```
dm_test_results = []

for i, name1 in enumerate(model_names):
    for j, name2 in enumerate(model_names):
        if i < j:  # Compare each pair once
            dm_stat, p_value =␣
↪diebold_mariano_test_crossectional(y_test_combined, model_predictions[i],␣
↪model_predictions[j], num_stocks)
            dm_test_results.append({
                "Model 1": name1,
                "Model 2": name2,
                "DM Statistic": dm_stat,
                "p-value": p_value
            })

dm_test_results_df = pd.DataFrame(dm_test_results)
dm_matrix = np.full((len(model_names), len(model_names)), "", dtype=object)

def format_stat_with_significance(stat, p_value):
    if p_value < 0.01:
        return f"{stat:.2f}***"
    elif p_value < 0.05:
        return f"{stat:.2f}**"
    elif p_value < 0.1:
        return f"{stat:.2f}*"
    else:
        return f"{stat:.2f}"

for result in dm_test_results_df.itertuples():
    model1_idx = model_names.index(result._1)
    model2_idx = model_names.index(result._2)
    dm_matrix[model1_idx, model2_idx] = format_stat_with_significance(result._3,␣
↪result._4)

dm_df = pd.DataFrame(dm_matrix, index=model_names, columns=model_names)
print(dm_df.to_string(index=True))
```

```
[ ]: residuals_rf = y_test_combined - rf_predictions
residuals_gbr = y_test_combined - gbr_predictions
residuals_nn1 = y_test_combined - nn_1_predictions
residuals_nn2 = y_test_combined - nn_2_predictions
residuals_stacking = y_test_combined - stacking_predictions
residuals_voting = y_test_combined - voting_predictions

residuals_data = [residuals_rf, residuals_gbr, residuals_nn1, residuals_nn2,␣
↪residuals_stacking, residuals_voting]
model_names = ["RF", "GBR", "NN1", "NN 2", "SR", "VR"]
```

```
plt.figure(figsize=(5, 3))
sns.boxplot(data=residuals_data)
plt.xticks(range(len(model_names)), model_names, rotation=45)
plt.title('Residuals Distribution by Model')
plt.ylabel('')
plt.xlabel('')
plt.xticks(rotation=0)
plt.show()
```

[ ]:
```
predictors_names = X_test_combined.columns

if isinstance(X_test_combined, np.ndarray):
    X_test = pd.DataFrame(X_test_combined, columns=predictors_names)
if isinstance(X_train_combined, pd.DataFrame):
    X_test = pd.DataFrame(X_test_combined, columns=X_train_combined.columns)

nan_count_in_column = df['monthly_target_return'].isna().sum()
```

6 Variable Importance

[ ]:
```
#variable importance

models = {
    'Random Forest': best_rf,
    'Gradient Boosting': best_gbr,
    'Neural Network 1': best_nn_1,
    'Neural Network 2': best_nn_2,
    'Stacking Regressor': stacking_regressor,
    'Voting Regressor': voting_regressor
}

mask = ~np.isnan(y_test_combined)

model_importances = {}
for model_name, model in models.items():

    reductions = []
    y_pred = model.predict(X_test_combined)  # Assuming X_test is a DataFrame
    baseline_r2 = r2_score(y_test_combined[mask], y_pred[mask])

    for j in range(X_test.shape[1]):
        X_test_zeroed = X_test_combined.copy()
        X_test_zeroed.iloc[:, j] = 0
        y_pred_zeroed = model.predict(X_test_zeroed)
        r2_zeroed = r2_score(y_test_combined[mask], y_pred_zeroed[mask])
        reduction = baseline_r2 - r2_zeroed
        reductions.append(reduction)

    reductions = np.array(reductions)
    positive_reductions = reductions[reductions > 0]
    variable_importance_positive = positive_reductions / np.
 ↪sum(positive_reductions)

    variable_importance = np.zeros_like(reductions)
    variable_importance[reductions > 0] = variable_importance_positive
    model_importances[model_name] = variable_importance

    importance_df = pd.DataFrame({
    'predictor': predictors_names,
    'importance': variable_importance
    }).sort_values(by='importance', ascending=False)

    # Plotting
    plt.figure(figsize=(3, 4))
    plt.barh(importance_df['predictor'], importance_df['importance'],␣
 ↪color='gray')
    plt.gca().spines['top'].set_visible(False)
    plt.gca().spines['right'].set_visible(False)
    plt.gca().spines['left'].set_visible(False)
    plt.gca().spines['bottom'].set_visible(True)
    plt.title(model_name, fontsize=10)
    plt.xlabel('Importance', fontsize=10)
    plt.ylabel('Predictor', fontsize=8)
    plt.xticks(fontsize=7)
    plt.yticks(fontsize=7)
    plt.gca().invert_yaxis()
    plt.show()
```

[ ]:
```
#Heatmap of variable importance
models = {
    'Random Forest': best_rf,
    'Gradient Boosting': best_gbr,
    'Neural Network 1': best_nn_1,
    'Neural Network 2': best_nn_2,
}

importance_df = pd.DataFrame(index=predictors_names)

for model_name, importance in model_importances.items():
    if len(importance) == len(predictors_names):
        importance_df[model_name] = importance
    else:
        print(f"Mismatch in length for model {model_name}: "
              f"Expected {len(predictors_names)} but got {len(importance)}")

importance_df['total_importance'] = importance_df.sum(axis=1)
importance_df = importance_df.sort_values(by='total_importance', ascending=False)
importance_df = importance_df.drop(columns=['total_importance'])
plt.figure(figsize=(10, 6))
ax=sns.heatmap(importance_df, annot=False, cmap='Blues', linewidths=0.5)
custom_model_names = ['RF', 'GBR', 'NN1', 'NN2', 'SR', 'VR']
plt.xticks(ticks=np.arange(len(custom_model_names)) + 0.5,␣
 ↪labels=custom_model_names, rotation=0, fontsize=10)
plt.yticks(fontsize=6)
plt.title('Heatmap of Variable Importance Across Models')
plt.xlabel('')
plt.ylabel('')
ax.xaxis.set_ticks_position('top')
plt.show()
```

[ ]:
```
#preparation for NACE analysis
test_start_date = '2018-09-18'
test_end_date = '2023-06-18'
columns_to_keep = ['Time', 'Companies', 'NACE IC','monthly_target_return']
industry_df = df[(df['Time'] >= test_start_date) & (df['Time'] <=␣
 ↪test_end_date)].copy()
industry_df = industry_df[columns_to_keep]

nace_df = pd.concat([industry_df, X_test_combined], axis=1)

models = {
    'Random Forest': best_rf,
    'Gradient Boosting': best_gbr,
    'Neural Network 1': best_nn_1,
    'Neural Network 2': best_nn_2,
    'Stacking Regressor': stacking_regressor,
    'Voting Regressor': voting_regressor
}
nace_importances = {}
for nace_ic, group in nace_df.groupby('NACE IC'):
    X_group = group.drop(columns=['Time', 'Companies', 'NACE␣
 ↪IC','monthly_target_return'])
    y_group = group['monthly_target_return'].values

    mask = ~np.isnan(y_group)
    cumulative_reductions = np.zeros(X_group.shape[1])

    for model_name, model in models.items():
        y_pred = model.predict(X_group)
        baseline_r2 = r2_score(y_group[mask], y_pred[mask])

        reductions = []
        for j in range(X_group.shape[1]):
            X_group_zeroed = X_group.copy()
            X_group_zeroed.iloc[:, j] = 0
            y_pred_zeroed = model.predict(X_group_zeroed)
            r2_zeroed = r2_score(y_group[mask], y_pred_zeroed[mask])
            reduction = baseline_r2 - r2_zeroed
            reductions.append(reduction)
        reductions = np.array(reductions)
        cumulative_reductions += reductions
    average_reductions = cumulative_reductions / len(models)

    positive_reductions = average_reductions[average_reductions > 0]
    variable_importance_positive = positive_reductions / np.
 ↪sum(positive_reductions)
    variable_importance = np.zeros_like(average_reductions)
    variable_importance[average_reductions > 0] = variable_importance_positive
    top_5_indices = np.argsort(variable_importance)[-5:][::-1]
    top_5_importance = variable_importance[top_5_indices]
    top_5_predictors = X_group.columns[top_5_indices]

    nace_importances[nace_ic] = pd.DataFrame({
        'predictor': top_5_predictors,
        'importance': top_5_importance
    })

for nace_ic, importance_df in nace_importances.items():
    print(f"\nTop 5 Predictors for NACE IC Code: {nace_ic}")
    print(importance_df)
```

7 Portfolio Performances

```
[ ]: #preparation for portfolio construction
test_start_date = '2017-11-18'
test_end_date = '2023-06-18'
columns_to_keep = ['Time', 'Companies', 'monthly_target_return']

test_period_df = df[(df['Time'] >= test_start_date) & (df['Time'] <=␣
 ↪test_end_date)].copy()
test_period_df = test_period_df[columns_to_keep]

test_period_df['rf_predictions'] = rf_predictions
test_period_df['gbr_predictions'] = gbr_predictions
test_period_df['nn_1_predictions'] = nn_1_predictions
```

```
test_period_df['nn_2_predictions'] = nn_2_predictions
test_period_df['stacking_predictions'] = stacking_predictions
test_period_df['voting_predictions'] = voting_predictions
test_period_df.head()
```

```
[ ]: #decile construction

def calculate_decile_metrics(test_period_df, prediction_column):
    unique_times = test_period_df['Time'].unique()
    decile_results = {i: {'Pred': [], 'Avg': [], 'SD': []} for i in range(1, 11)}

    for time in unique_times:
        period_data = test_period_df[test_period_df['Time'] == time].copy()
        period_data.loc[:, 'Decile'] = pd.qcut(period_data[prediction_column],␣
 ↪10, labels=False, duplicates='drop') + 1
            for decile in range(1, 11):
            decile_data = period_data[period_data['Decile'] == decile]
            if len(decile_data) > 0:
                pred_mean = decile_data[prediction_column].mean()
                realized_mean = decile_data['monthly_target_return'].mean()
                realized_sd = decile_data['monthly_target_return'].std()

                decile_results[decile]['Pred'].append(pred_mean)
                decile_results[decile]['Avg'].append(realized_mean)
                decile_results[decile]['SD'].append(realized_sd)

    for decile in decile_results.keys():
        decile_results[decile]['Pred'] = np.mean(decile_results[decile]['Pred'])␣
 ↪* 100
        decile_results[decile]['Avg'] = np.mean(decile_results[decile]['Avg']) *␣
 ↪100
        decile_results[decile]['SD'] = np.mean(decile_results[decile]['SD']) *␣
 ↪100
        decile_results[decile]['SR'] = decile_results[decile]['Avg'] /␣
 ↪decile_results[decile]['SD'] if decile_results[decile]['SD'] != 0 else np.nan

    return decile_results

def create_performance_table(decile_results):
    performance_table = pd.DataFrame({
        'Decile': [f'Decile {i}' for i in range(1, 11)],
        'Pred': [decile_results[i]['Pred'] for i in range(1, 11)],
        'Avg': [decile_results[i]['Avg'] for i in range(1, 11)],
        'SD': [decile_results[i]['SD'] for i in range(1, 11)],
        'SR': [decile_results[i]['SR'] for i in range(1, 11)],
    })
    hl_row = pd.DataFrame({
        'Decile': ['H-L'],
        'Pred': [performance_table['Pred'].iloc[-1] - performance_table['Pred'].
 ↪iloc[0]],
        'Avg': [performance_table['Avg'].iloc[-1] - performance_table['Avg'].
 ↪iloc[0]],
        'SD': [performance_table['SD'].iloc[-1] - performance_table['SD'].
 ↪iloc[0]],
        'SR': [performance_table['SR'].iloc[-1] - performance_table['SR'].
 ↪iloc[0]],
    })
    performance_table = pd.concat([performance_table, hl_row], ignore_index=True)

    return performance_table
model_predictions = {
    'Random Forest': 'rf_predictions',
    'Gradient Boosting': 'gbr_predictions',
    'Neural Network 1': 'nn_1_predictions',
    'Neural Network 2': 'nn_2_predictions',
    'Voting Ensemble': 'voting_predictions',
    'Stacking Ensemble': 'stacking_predictions'
}

performance_tables = {}

for model_name, prediction_column in model_predictions.items():
    decile_metrics = calculate_decile_metrics(test_period_df, prediction_column)
    performance_table = create_performance_table(decile_metrics)
    performance_tables[model_name] = performance_table
    print(f"\nPerformance Table for {model_name}:")
    print(performance_table)
```

```
[ ]: #EQUAL WEIGHT, NO TRANSACTION COSTS, LONG ONLY

test_period_df['Time'] = pd.to_datetime(test_period_df['Time'])
market_returns = test_period_df.groupby('Time')['monthly_target_return'].mean()
market_cumulative_returns = (1 + market_returns).cumprod()

def calculate_portfolio_returns(test_period_df, prediction_column, top_percent):
    unique_times = test_period_df['Time'].unique()
    portfolio_returns = []

    for time in unique_times:
        period_data = test_period_df[test_period_df['Time'] == time]
        period_data = period_data.sort_values(by=prediction_column,␣
 ↪ascending=False)
        top_n = int(len(period_data) * top_percent)
        selected_companies = period_data.head(top_n)
        avg_return = selected_companies['monthly_target_return'].mean()
        portfolio_returns.append(avg_return)

    portfolio_returns = np.array(portfolio_returns)
    cumulative_returns = (1 + portfolio_returns).cumprod()

    return cumulative_returns, portfolio_returns

strategies = {
    'Top 10%': 0.10,
    'Top 30%': 0.30,
}

model_predictions = {
    'Random Forest': 'rf_predictions',
    'Gradient Boosting': 'gbr_predictions',
    'Neural Network 1': 'nn_1_predictions',
    'Neural Network 2': 'nn_2_predictions',
    'Voting Ensemble': 'voting_predictions',
    'Stacking Ensemble': 'stacking_predictions'
}

cumulative_returns = {strategy: {} for strategy in strategies}
performance_metrics = {strategy: {} for strategy in strategies}

def calculate_annual_metrics(monthly_returns):
    if len(monthly_returns) == 0 or np.all(monthly_returns == 0):
        return np.nan, np.nan, np.nan
    mean_monthly_return = np.mean(monthly_returns)
    std_dev_monthly = np.std(monthly_returns)
    mean_annual_return = mean_monthly_return * 12
    std_dev_annual = std_dev_monthly * np.sqrt(12)
    sharpe_ratio_annual = mean_annual_return / std_dev_annual if std_dev_annual !
 ↪= 0 else np.nan

    return mean_annual_return, std_dev_annual, sharpe_ratio_annual

for strategy_name, top_percent in strategies.items():
    strategy_metrics = {}

    for model_name, prediction_column in model_predictions.items():
        cum_returns, portfolio_returns =␣
 ↪calculate_portfolio_returns(test_period_df, prediction_column, top_percent)
        cumulative_returns[strategy_name][model_name] = cum_returns
        mean_annual_return, std_dev_annual, sharpe_ratio_annual =␣
 ↪calculate_annual_metrics(portfolio_returns)
```

```
        final_cumulative_return = cum_returns[-1]

        strategy_metrics[model_name] = {
            'Cumulative Return': final_cumulative_return,
            'Annual Return': mean_annual_return,
            'Annual Standard Deviation': std_dev_annual,
            'Annual Sharpe Ratio': sharpe_ratio_annual
        }
    mean_annual_return_market, std_dev_annual_market, sharpe_ratio_annual_market␣
 ↪= calculate_annual_metrics(market_returns)
    final_cumulative_return_market = market_cumulative_returns.iloc[-1]

    strategy_metrics['Market'] = {
        'Cumulative Return': final_cumulative_return_market,
        'Annual Return': mean_annual_return_market,
        'Annual Standard Deviation': std_dev_annual_market,
        'Annual Sharpe Ratio': sharpe_ratio_annual_market
    }

    performance_metrics[strategy_name] = strategy_metrics

performance_metrics_dfs = {
    strategy: pd.DataFrame.from_dict(metrics, orient='index').T
    for strategy, metrics in performance_metrics.items()
}
for strategy_name, df in performance_metrics_dfs.items():
    print(f"\nPerformance Metrics for {strategy_name}:")
    print(df)
def plot_strategy_results(strategy_name):
    plt.figure(figsize=(10, 6))

    line_styles = {
        'Random Forest': ('-', 'tab:blue'),
        'Gradient Boosting': ('-', 'tab:orange'),
        'Neural Network 1': ('-', 'tab:green'),
        'Neural Network 2': ('-', 'tab:red'),
        'Voting Ensemble': ('-.', 'tab:purple'),
        'Stacking Ensemble': ('-.', 'tab:brown'),
        'Market': ('--', 'black')
    }
    cumulative_returns[strategy_name]['Market'] = market_cumulative_returns.
 ↪values
    label_refs = []

    for model_name, cum_returns in cumulative_returns[strategy_name].items():
        linestyle, color = line_styles.get(model_name, ('-', 'black'))
        line, = plt.plot(test_period_df['Time'].unique(), cum_returns,␣
↪linestyle, color=color, linewidth=1, label=model_name)
        label_refs.append(line)
    plt.gca().spines['top'].set_visible(False)
    plt.gca().spines['right'].set_visible(False)
    plt.gca().spines['left'].set_visible(True)
    plt.gca().spines['bottom'].set_visible(True)
    plt.title(f'Cumulative Returns Comparison - {strategy_name}')
    plt.xlabel('Time', fontsize=15)
    plt.ylabel('Cumulative Return', fontsize=15)
    plt.xticks(fontsize=13)
    plt.yticks(fontsize=13)
    plt.xlim(pd.to_datetime('2017-11-18'), pd.to_datetime('2023-06-30'))
    plt.legend(handles=label_refs, loc='upper left', bbox_to_anchor=(1, 1),␣
↪ncol=1, fontsize=8)
    plt.show()

for strategy_name in strategies:
    plot_strategy_results(strategy_name)
```

```
[ ]: #EQUAL WEIGHT, TRANSACTION COSTS, LONG ONLY

test_period_df = test_period_df.sort_values(by='Time')
def calculate_turnover(test_period_df, portfolio_weights, unique_times):
    turnover_values = []

    for t in range(len(unique_times) - 1):
        current_time = unique_times[t]
        next_time = unique_times[t + 1]
        current_period_data = test_period_df[test_period_df['Time'] ==␣
↪current_time]
        next_period_data = test_period_df[test_period_df['Time'] == next_time]
        w_current = portfolio_weights[current_time]
        w_next = portfolio_weights[next_time]
        returns_next = {row['Companies']: row['monthly_target_return'] for _,␣
↪row in next_period_data.iterrows()}
        rebalanced_weights = {}
        total_weighted_returns = sum([w_current[company] * (1 + returns_next.
↪get(company, 0)) for company in w_current])

        for company in w_current:
            rebalanced_weights[company] = (w_current[company] * (1 +␣
↪returns_next.get(company, 0))) / (1 + total_weighted_returns)
        turnover = sum([abs(w_next.get(company, 0) - rebalanced_weights.
↪get(company, 0)) for company in w_next])
        turnover_values.append(turnover)
    return np.mean(turnover_values)

def calculate_break_even_transaction_cost(portfolio_returns, turnover_values):
    avg_portfolio_return = np.mean(portfolio_returns) * 100
    avg_turnover = np.mean(turnover_values)

    if avg_turnover == 0:
        return np.nan
    break_even_cost = avg_portfolio_return / avg_turnover
    return break_even_cost

def run_strategy_and_calculate_turnover():
    unique_times = test_period_df['Time'].unique()

    strategies = {
        'Top 10%': 0.10,
        'Top 30%': 0.30,
    }

    model_predictions = {
        'Random Forest': 'rf_predictions',
        'Gradient Boosting': 'gbr_predictions',
        'Neural Network 1': 'nn_1_predictions',
        'Neural Network 2': 'nn_2_predictions',
        'Voting Ensemble': 'voting_predictions',
        'Stacking Ensemble': 'stacking_predictions'
    }

    turnover_break_even_results = {strategy: {} for strategy in strategies}

    for strategy_name, top_percent in strategies.items():
        for model_name, prediction_column in model_predictions.items():
            cum_returns, portfolio_returns =␣
↪calculate_portfolio_returns(test_period_df, prediction_column, top_percent)
            portfolio_weights = {}
            for time in unique_times:
                period_data = test_period_df[test_period_df['Time'] == time].
↪copy()
                period_data = period_data.sort_values(by=prediction_column,␣
↪ascending=False)
                top_n = int(len(period_data) * top_percent)
                selected_companies = period_data.head(top_n)
                equal_weight = 1 / len(selected_companies)
                portfolio_weights[time] = {row['Companies']: equal_weight for␣
↪idx, row in selected_companies.iterrows()}
            turnover_values = calculate_turnover(test_period_df,␣
↪portfolio_weights, unique_times)
```

```
            break_even_cost =␣
↪calculate_break_even_transaction_cost(portfolio_returns, turnover_values)
            turnover_break_even_results[strategy_name][model_name] = {
                'Turnover': turnover_values,
                'Break-even Transaction Costs': break_even_cost
            }
    return turnover_break_even_results
def display_turnover_break_even_costs(results):
    for strategy_name, model_results in results.items():
        print(f"\nTurnover and Break-even Transaction Costs for {strategy_name}:
↪")
        model_names = list(model_results.keys())
        header = "{:<25}".format("") + "".join([f"{model:<20}" for model in␣
↪model_names])
        print(header)
        turnover_row = "{:<25}".format("Average Turnover (Monthly)") + "".
↪join([f"{metrics['Turnover']:<20.6f}" for metrics in model_results.values()])
        print(turnover_row)
        breakeven_row = "{:<25}".format("Break-even Transaction Costs") + "".
↪join([f"{metrics['Break-even Transaction Costs']:<20.6f}" for metrics in␣
↪model_results.values()])
        print(breakeven_row)

results = run_strategy_and_calculate_turnover()
display_turnover_break_even_costs(results)
```

```
[ ]: # EQUAL WEIGHT, NO TRANSACTION COSTS, LONG-SHORT STRATEGY

test_period_df['Time'] = pd.to_datetime(test_period_df['Time'])

market_returns = test_period_df.groupby('Time')['monthly_target_return'].mean()
market_cumulative_returns = (1 + market_returns).cumprod()

def calculate_portfolio_returns(test_period_df, prediction_column, top_percent):
    unique_times = test_period_df['Time'].unique()
    long_returns = []
    short_returns = []

    for time in unique_times:
        period_data = test_period_df[test_period_df['Time'] == time]
        period_data = period_data.sort_values(by=prediction_column,␣
↪ascending=False)
        top_n = int(len(period_data) * top_percent)
        selected_long_companies = period_data.head(top_n)
        selected_short_companies = period_data.tail(top_n)
        long_return = selected_long_companies['monthly_target_return'].mean()
        short_return = selected_short_companies['monthly_target_return'].mean()
        long_returns.append(long_return)
        short_returns.append(short_return)

    long_returns = np.array(long_returns)
    short_returns = np.array(short_returns)
    portfolio_returns = long_returns - short_returns
    cumulative_returns = (1 + portfolio_returns).cumprod()

    return cumulative_returns, portfolio_returns, long_returns, short_returns

strategies = {
    'Top 10%': 0.10,
    'Top 30%': 0.30,
}

model_predictions = {
    'Random Forest': 'rf_predictions',
    'Gradient Boosting': 'gbr_predictions',
    'Neural Network 1': 'nn_1_predictions',
    'Neural Network 2': 'nn_2_predictions',
    'Voting Ensemble': 'voting_predictions',
    'Stacking Ensemble': 'stacking_predictions'
}

cumulative_returns = {strategy: {} for strategy in strategies}
performance_metrics = {strategy: {} for strategy in strategies}

for strategy_name, top_percent in strategies.items():
    strategy_metrics = {}

    for model_name, prediction_column in model_predictions.items():
        cum_returns, portfolio_returns, long_returns, short_returns =␣
 ↪calculate_portfolio_returns(test_period_df, prediction_column, top_percent)
        cumulative_returns[strategy_name][model_name] = cum_returns

        mean_annual_return, std_dev_annual, sharpe_ratio_annual =␣
 ↪calculate_annual_metrics(portfolio_returns)
        mean_annual_long_return, _, _ = calculate_annual_metrics(long_returns)
        mean_annual_short_return, _, _ = calculate_annual_metrics(short_returns)
        final_cumulative_return = cum_returns[-1]

        strategy_metrics[model_name] = {
            'Cumulative Return': final_cumulative_return,
            'Annual Return': mean_annual_return,
            'Annual Standard Deviation': std_dev_annual,
            'Annual Sharpe Ratio': sharpe_ratio_annual,
            'Annual Long Return': mean_annual_long_return,
            'Annual Short Return': mean_annual_short_return
        }

    mean_annual_return_market, std_dev_annual_market, sharpe_ratio_annual_market␣
↪= calculate_annual_metrics(market_returns)
    final_cumulative_return_market = market_cumulative_returns.iloc[-1]

    strategy_metrics['Market'] = {
        'Cumulative Return': final_cumulative_return_market,
        'Annual Return': mean_annual_return_market,
        'Annual Standard Deviation': std_dev_annual_market,
        'Annual Sharpe Ratio': sharpe_ratio_annual_market
    }

    performance_metrics[strategy_name] = strategy_metrics

performance_metrics_dfs = {
    strategy: pd.DataFrame.from_dict(metrics, orient='index').T
    for strategy, metrics in performance_metrics.items()
}

for strategy_name, df in performance_metrics_dfs.items():
    print(f"\nPerformance Metrics for {strategy_name}:")
    print(df)

def plot_strategy_results(strategy_name):
    plt.figure(figsize=(10, 6))

    line_styles = {
        'Random Forest': ('-', 'tab:blue'),
        'Gradient Boosting': ('-', 'tab:orange'),
        'Neural Network 1': ('-', 'tab:green'),
        'Neural Network 2': ('-', 'tab:red'),
        'Voting Ensemble': ('-', 'tab:purple'),
        'Stacking Ensemble': ('-', 'tab:brown'),
        'Market': ('--', 'black')
    }
    cumulative_returns[strategy_name]['Market'] = market_cumulative_returns.
↪values
    label_refs = []

    for model_name, cum_returns in cumulative_returns[strategy_name].items():
        linestyle, color = line_styles.get(model_name, ('-', 'black'))
        line, = plt.plot(test_period_df['Time'].unique(), cum_returns,␣
↪linestyle, color=color, linewidth=1, label=model_name)
        label_refs.append(line)
    plt.title('')
    plt.xlabel('')
    plt.ylabel('Cum_ret',fontsize=15)
    plt.xticks(fontsize=10)
    plt.yticks(fontsize=13)
    plt.xlim(pd.to_datetime('2018-09-30'), pd.to_datetime('2023-06-30'))
    plt.legend(handles=label_refs, loc='upper left', fontsize=8)
    plt.show()

for strategy_name in strategies:
    plot_strategy_results(strategy_name)
```

```
[ ]: # EQUAL WEIGHT, TRANSACTION COSTS, LONG-SHORT STRATEGY

def calculate_turnover_long_short(test_period_df, portfolio_weights_long,␣
 ↪portfolio_weights_short, unique_times):
    turnover_values = []

    for t in range(len(unique_times) - 1):
        current_time = unique_times[t]
        next_time = unique_times[t + 1]
        current_period_data = test_period_df[test_period_df['Time'] ==␣
 ↪current_time]
        next_period_data = test_period_df[test_period_df['Time'] == next_time]
        w_current_long = portfolio_weights_long[current_time]
        w_next_long = portfolio_weights_long[next_time]
        w_current_short = portfolio_weights_short[current_time]
        w_next_short = portfolio_weights_short[next_time]
        returns_next = {row['Companies']: row['monthly_target_return'] for _,␣
 ↪row in next_period_data.iterrows()}

        rebalanced_weights_long = {}
        total_weighted_returns_long = sum([w_current_long[company] * (1 +␣
 ↪returns_next.get(company, 0)) for company in w_current_long])

        rebalanced_weights_short = {}
        total_weighted_returns_short = sum([w_current_short[company] * (1 +␣
 ↪returns_next.get(company, 0)) for company in w_current_short])

        for company in w_current_long:
            rebalanced_weights_long[company] = (w_current_long[company] * (1 +␣
 ↪returns_next.get(company, 0))) / (1 + total_weighted_returns_long)

        for company in w_current_short:
            rebalanced_weights_short[company] = (w_current_short[company] * (1 +␣
 ↪returns_next.get(company, 0))) / (1 + total_weighted_returns_short)
```

```
        turnover_long = sum([abs(w_next_long.get(company, 0) -␣
↪rebalanced_weights_long.get(company, 0)) for company in w_next_long])
        turnover_short = sum([abs(w_next_short.get(company, 0) -␣
↪rebalanced_weights_short.get(company, 0)) for company in w_next_short])
        total_turnover = turnover_long + turnover_short

        turnover_values.append(total_turnover)
    return np.mean(turnover_values)

def run_long_short_strategy_and_calculate_turnover():
    unique_times = test_period_df['Time'].unique()
    turnover_break_even_results = {strategy: {} for strategy in strategies}

    for strategy_name, top_percent in strategies.items():
        for model_name, prediction_column in model_predictions.items():
            cum_returns, portfolio_returns, long_returns, short_returns =␣
↪calculate_portfolio_returns(test_period_df, prediction_column, top_percent)

            portfolio_weights_long = {}
            portfolio_weights_short = {}
            for time in unique_times:
                period_data = test_period_df[test_period_df['Time'] == time].
↪copy()
                period_data = period_data.sort_values(by=prediction_column,␣
↪ascending=False)
                top_n = int(len(period_data) * top_percent)
                selected_long_companies = period_data.head(top_n)
                selected_short_companies = period_data.tail(top_n)
                equal_weight_long = 1 / len(selected_long_companies)
                equal_weight_short = 1 / len(selected_short_companies)
                portfolio_weights_long[time] = {row['Companies']:␣
↪equal_weight_long for idx, row in selected_long_companies.iterrows()}
                portfolio_weights_short[time] = {row['Companies']:␣
↪equal_weight_short for idx, row in selected_short_companies.iterrows()}

            turnover_values = calculate_turnover_long_short(test_period_df,␣
↪portfolio_weights_long, portfolio_weights_short, unique_times)
            break_even_cost =␣
↪calculate_break_even_transaction_cost_long_short(portfolio_returns,␣
↪turnover_values)
            turnover_break_even_results[strategy_name][model_name] = {
                'Turnover': turnover_values,
                'Break-even Transaction Costs': break_even_cost
            }

    return turnover_break_even_results
```

```
long_short_results = run_long_short_strategy_and_calculate_turnover()

def display_turnover_break_even_costs_long_short(results):
    for strategy_name, model_results in results.items():
        print(f"\nTurnover and Break-even Transaction Costs for {strategy_name}:
 ↪")
        model_names = list(model_results.keys())
        header = "{:<25}".format("") + "".join([f"{model:<20}" for model in␣
 ↪model_names])
        print(header)
        turnover_row = "{:<25}".format("Average Turnover (Monthly)") + "".
 ↪join([f"{metrics['Turnover']:<20.6f}" for metrics in model_results.values()])
        print(turnover_row)
        breakeven_row = "{:<25}".format("Break-even Transaction Costs") + "".
 ↪join([f"{metrics['Break-even Transaction Costs']:<20.6f}" for metrics in␣
 ↪model_results.values()])
        print(breakeven_row)
display_turnover_break_even_costs_long_short(long_short_results)
```

References

[1] E. F. Fama. “Efficient capital markets”. In: *Journal of finance* 25.2 (1970), pp. 383–417.

[2] P. A. Samuelson. “Proof that properly discounted present values of assets vibrate randomly”. In: *The Bell Journal of Economics and Management Science* (1973), pp. 369–374.

[3] M. G. Kendall. “The Analysis of Economic Time Series—Part 1: Prices”. In: *Journal of the Royal Statistical Society. Series A (General)* 116.1 (1953), pp. 11–34.

[4] A. Cowles. “A Revision of Previous Conclusions Regarding Stock Price Behavior”. In: *Econometrica* 28.4 (1960), pp. 909–915.

[5] M. F. Osborne. “Brownian motion in the stock market”. In: *Operations research* 7.2 (1959), pp. 145–173.

[6] E. F. Fama. “The behavior of stock-market prices”. In: *The journal of Business* 38.1 (1965), pp. 34–105.

[7] E. F. Fama. “Efficient capital markets: II”. In: *The journal of finance* 46.5 (1991), pp. 1575–1617.

[8] E. F. Fama and K. R. French. “Common risk factors in the returns on stocks and bonds”. In: *Journal of financial economics* 33.1 (1993), pp. 3–56.

[9] F. Black, M. C. Jensen, and M. Scholes. “The Capital Asset Pricing Model: Some Empirical Tests”. In: *Studies in the Theory of Capital Markets* (1972). Ed. by M. C. Jensen, pp. 79–121.

[10] M. S. Rozeff. “Dividend yields are equity risk premiums”. In: *Journal of Portfolio management* (1984), pp. 68–75.

[11] J. Y. Campbell and R. J. Shiller. “The dividend-price ratio and expectations of future dividends and discount factors”. In: *The review of financial studies* 1.3 (1988), pp. 195–228.

[12] J. H. Cochrane. “Production-Based Asset Pricing and the Link Between Stock Returns and Economic Fluctuations”. In: *Journal of Finance* 46.1 (1991), pp. 209–237.

[13] W. N. Goetzmann and R. G. Ibbotson. “The Performance of Mutual Funds in the Period 1965-1984”. In: *Journal of Business* 66.2 (1993), pp. 133–156.

[14] R. J. Hodrick. “Dividend Yields and Expected Stock Returns: Alternative Procedures for Inference and Measurement”. In: *Review of Financial Studies* 5.3 (1992), pp. 357–386.

[15] J. Lewellen. “Predicting Returns with Financial Ratios”. In: *Journal of Financial Economics* 74.2 (2004), pp. 209–235.

[16] S. Basu. “Investment Performance of Common Stocks in Relation to Their Price-Earnings Ratios: A Test of the Efficient Market Hypothesis”. In: *Journal of Finance* 32.3 (1977), pp. 663–682.

[17] A. Chattopadhyay, M. R. Lyle, and C. C. Wang. “Accounting data, market values, and the cross section of expected returns worldwide”. In: *Harvard Business School Accounting & Management Unit Working Paper* 15-092 (2016).

[18] S. Kheradyar, I. Ibrahim, and F. M. Nor. “Stock return predictability with financial ratios”. In: *International Journal of Trade, Economics and Finance* 2.5 (2011), p. 391.

[19] X. Chen, K. A. Kim, T. Yao, and T. Yu. “On the predictability of Chinese stock returns”. In: *Pacific-basin finance journal* 18.4 (2010), pp. 403–425.

[20] C. Venkates, M. Tyagi, and L. Ganesh. “Fundamental analysis and stock returns: An indian evidence. Global Advanced Research Journal of Economics”. In: *Accounting and Finance* 1.2 (2012), pp. 033–039.

[21] I. N. Agustin. “The integration of fundamental and technical analysis in predicting the stock price”. In: *Jurnal Manajemen Maranatha* 18.2 (2019), pp. 93–102.

[22] M. B. Khan, S. Gul, S. U. Rehman, N. Razzaq, and A. Kamran. “Financial ratios and stock return predictability (Evidence from Pakistan)”. In: *Research Journal of Finance and Accounting* 3.10 (2012), pp. 1–6.

[23] B. Lev and S. R. Thiagarajan. “Fundamental information analysis”. In: *Journal of Accounting research* 31.2 (1993), pp. 190–215.

[24] U. S. Akbar and N. A. Bhutto. “Predictability of Stock Returns using Financial Ratios: Evidence from Pakistan Stock Exchange”. In: *Pakistan Research Journal of Social Sciences* 2.3 (2023).

[25] M. Dempsey. “The book-to-market equity ratio as a proxy for risk: evidence from Australian markets”. In: *Australian Journal of Management* 35.1 (2010), pp. 7–21.

[26] A. J. Hatta and B. S. Dwiyanto. “The company fundamental factors and systematic risk in increasing stock price”. In: *Journal of Economics, Business, and Accountancy Ventura* 15.2 (2012), pp. 245–256.

[27] D. Martani and R. Khairurizka. “The effect of financial ratios, firm size, and cash flow from operating activities in the interim report to the stock return”. In: *Chinese business review* 8.6 (2009), p. 44.

[28] I. Welch and A. Goyal. “A comprehensive look at the empirical performance of equity premium prediction”. In: *The Review of Financial Studies* 21.4 (2008), pp. 1455–1508.

[29] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, et al. “Recent advances in convolutional neural networks”. In: *Pattern recognition* 77 (2018), pp. 354–377.

[30] S. Gu, B. Kelly, and D. Xiu. “Empirical asset pricing via machine learning”. In: *The Review of Financial Studies* 33.5 (2020), pp. 2223–2273.

[31] E. F. Fama and K. R. French. “Dissecting anomalies”. In: *The journal of finance* 63.4 (2008), pp. 1653–1678.

[32] W. Drobetz and T. Otto. “Empirical Asset Pricing via Machine Learning: Evidence from the European Stock Market”. In: *Journal of Asset Management* 22.7 (2021), pp. 507–538.

[33] O. Tobek and M. Hronec. “Does it pay to follow anomalies research? machine learning approach with international evidence”. In: *Journal of Financial Markets* 56 (2021), p. 100588.

[34] M. Leippold, Q. Wang, and W. Zhou. “Machine learning in the Chinese stock market”. In: *Journal of Financial Economics* 145.2 (2022), pp. 64–82.

[35] R. Chopra and G. D. Sharma. “Application of artificial intelligence in stock market forecasting: a critique, review, and research agenda”. In: *Journal of risk and financial management* 14.11 (2021), p. 526.

[36] A. Rubesam. “Machine learning portfolios with equal risk contributions: Evidence from the Brazilian market”. In: *Emerging Markets Review* 51 (2022), p. 100891.

[37] S. Lalwani and V. Meshram. “Stock Return Prediction in India Using Traditional and Machine Learning Models”. In: *Indian Journal of Finance* 16.5 (2022), pp. 35–52.

[38] M. P. E. Martens and O. Penninga. “Predicting Bond Returns”. In: *Financial Analysts Journal* 77.3 (2021), pp. 133–155.

[39] S. R. Das, M. Kalimipalli, and S. Nayak. “Did CDS Trading Improve the Market for Corporate Bonds?” In: *Journal of Financial Economics* 111.2 (2013), pp. 495–525.

[40] G. A. Karolyi. “Home Bias, an Academic Puzzle”. In: *Review of Finance* 20.6 (2016), pp. 2049–2078.

[41] A. Goyal and S. Wahal. “Is Momentum an Echo?” In: *Journal of Financial and Quantitative Analysis* 50.6 (2015), pp. 1237–1267.

[42] A. G. Woodhouse, R. W. Sias, and H. J. Turtle. “What Drives the Cross-Section of Stock Returns in Global Markets?” In: *Journal of Financial Economics* 126.3 (2017), pp. 620–643.

[43] H. Jacobs and S. Müller. “Anomalies Across the Globe: Once Public, No Longer Existent?” In: *Journal of Financial Economics* 135.1 (2020), pp. 213–230.

[44] C. R. Harvey. “The Scientific Outlook in Financial Economics”. In: *Journal of Finance* 72.4 (2017), pp. 1393–1442.

[45] K. Hou, C. Xue, and L. Zhang. “Replicating Anomalies”. In: *Review of Financial Studies* 31.1 (2018), pp. 129–178.

[46] C. R. Harvey, Y. Liu, and H. Zhu. “... and the Cross-Section of Expected Returns”. In: *Review of Financial Studies* 29.1 (2016), pp. 5–68.

[47] C. Fieberg, D. Metko, T. Poddig, and T. Loy. “Machine Learning Techniques for Cross-Sectional Equity Returns’ Prediction”. In: *OR Spectrum* 45.1 (2023), pp. 289–323.

[48] A. Marsi. “Stock Return Predictability Using Machine Learning: Evidence from the EURO STOXX 50 Index”. In: *Journal of Financial Data Science* 3.2 (2021), pp. 45–62.

[49] T. Masters. *Practical Neural Network Recipes in C++*. Burlington, MA, US: Morgan Kaufmann Publishers, 1993.

[50] M. J. Van der Laan, E. C. Polley, and A. E. Hubbard. “Super Learner”. In: *Statistical Applications in Genetics and Molecular Biology* 6.1 (2007), pp. 1–21.

[51] S. Lessmann, B. Baesens, H. V. Seow, and L. C. Thomas. “Benchmarking State-of-the-Art Classification Algorithms for Credit Scoring: An Update of Research”. In: *European Journal of Operational Research* 247.1 (2015), pp. 124–136.

[52] L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Hoboken, NJ, US: Wiley-Interscience, 2004.

[53] L. Rokach. “Ensemble-based classifiers”. In: *Artificial Intelligence Review* 33.1-2 (2010), pp. 1–39.

[54] J. Patel, S. Shah, P. Thakkar, and K. Kotecha. “Predicting Stock and Stock Price Index Movement Using Trend Deterministic Data Preparation and Machine Learning Techniques”. In: *Expert Systems with Applications* 42.1 (2015), pp. 259–268.

[55] Y. Dong and W. Bao. “Explaining Black-Box Machine Learning Predictions in Finance Using Voting Ensembles”. In: *Journal of Forecasting* 39.6 (2020), pp. 878–892.

[56] N. Jegadeesh and S. Titman. “Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency”. In: *Journal of Finance* 48.1 (1993), pp. 65–91.

[57] M. M. Carhart. “On Persistence in Mutual Fund Performance”. In: *Journal of Finance* 52.1 (1997), pp. 57–82.

[58] B. Han, C. Lee, and M. Worah. “Momentum and Machine Learning: A Framework for Financial Prediction”. In: *Journal of Financial Data Science* 1.3 (2018), pp. 112–130.

[59] J. Nordvig and N. Firoozye. “Momentum Trading and Machine Learning Models in Financial Markets”. In: *Journal of Financial Markets* 45 (2019), pp. 200–219.

[60] Y. Xia, X. Hu, and T. Sargent. “Incorporating Momentum in Machine Learning Models for Financial Predictions”. In: *Journal of Machine Learning in Finance* 4.2 (2020), pp. 154–176.

[61] M. Falk, A. Marx, and T. Gneiting. “Spline Interpolation for Financial Time Series: A Case Study on Missing Value Imputation”. In: *Journal of Applied Statistics* 46.9 (2019), pp. 1567–1583.

[62] S. Beretta and A. Santaniello. “Nearest Neighbor Imputation Algorithms: A Comprehensive Review and Application to Financial Data”. In: *Journal of Big Data* 3.1 (2016), p. 9.

[63] J. Lewellen. “The Cross-Section of Expected Stock Returns”. In: *Critical Finance Review* 4.1 (2015), pp. 1–44.

[64] W. Drobetz and T. Otto. “Empirical asset pricing via machine learning: evidence from the European stock market”. In: *Journal of Asset Management* 22.7 (2021), pp. 507–538.

[65] B. Kelly, S. Pruitt, and Y. Su. “Characteristics are Covariances: A Unified Model of Risk and Return”. In: *Journal of Financial Economics* 134.3 (2019), pp. 501–524.

[66] N. Cakici, C. Fieberg, D. Metko, and A. Zaremba. “Machine learning goes global: Cross-sectional return predictability in international stock markets”. In: *Journal of Economic Dynamics and Control* 155 (2023), p. 104725.

[67] F. Diebold and R. Mariano. “Comparing Predictive Accuracy”. In: *Journal of Business & Economic Statistics* 13.3 (1995), pp. 253–263.

[68] G. Leitch and J. E. Tanner. “Economic Forecast Evaluation: Profits Versus the Conventional Error Measures”. In: *American Economic Review* 81.3 (1991), pp. 580–590.

[69] D. A. Lesmond, J. P. Ogden, and C. A. Trzcinka. “A new estimate of transaction costs”. In: *The review of financial studies* 12.5 (1999), pp. 1113–1141.

[70] B. M. Collins and F. J. Fabozzi. “A methodology for measuring transaction costs”. In: *Financial Analysts Journal* 47.2 (1991), pp. 27–36.