Project Data Science in Finance

Presented to the Erasmus School of Economics

In Partial Fulfilment of the Requirements for the Seminar:

Data Science in Finance

Lecturer: **Prof. Amar Soebhag**

Submitted in February 2024 by:

Marco Bergamin, Felix Thierbach, Lara Cerqueira,
Hidde van de Burgwal & Antonius von Oertzen

Contents

1 Introduction

This research project aims to replicate and extend the work conducted by Gu et al. (2020) in their paper *'Empirical asset pricing via machine learning'*. The focal point of this study lies in the empirical usage of machine learning (ML) methodologies in asset pricing, particularly in predicting excess returns of the S&P 500 index and tracing predictive gains.

Following the methodological framework by Gu et al. (2020), our research aims to replicate their time-series analysis while expanding it to include three additional ML models. Specifically, we seek to predict the excess returns of the S&P 500 index by constructing a portfolio comprised of the top 500 companies by market capitalization on a monthly basis. To predict asset risk premia, a comprehensive set of firm-specific characteristics and macroeconomic factors are incorporated as regressors.

We adopt the methodology employed by Gu et al. (2020), which includes Ordinary Least Squares (OLS), OLS-3, Partial Least Squares (PLS), Principal Component Regression (PCR), Elastic Net (ENet), Generalized Linear Models (GLM), Random Forest (RF), Gradient Boosted Regression Trees with Huber loss (GBRT), and various neural network architectures ranging from NN1 to NN5. Furthermore, we extend the comparative analysis by introducing the concept of Bagging (BAG) and Extreme Boosting (XGB) in regression tree models as well as Bayesian Additive Regression Trees (BART).

The overarching objectives of this study are two: to see if our results align with Gu et al. (2020) regarding their findings related to neural networks offering better results in prediction (with the highest out-of-sample predictive R-squared among all the methods) and the outperformance of the strategy created (using ML techniques) when compared to buy-and-hold investor and other regression-based strategies. The Sharpe ratio measures the latter and translates itself into economic gains, which are the ultimate goal of asset pricing.

2 Data

We obtain monthly total individual equity returns from CRSP for all firms listed in the NYSE, AMEX, and NASDAQ. Our sample begins in March 1957 (the start date of the S&P 500) and ends in December 2016, totaling 60 years. The 94 firm characteristics (61 of which are updated annually, 13 are updated quarterly, and 20 are updated monthly) were downloaded from the author's website (which were taken from Jeremiah Green's Website). Eight macroeconomic factors were obtained from Amit Goyal's website.

The eight variables were computed following the definition of Welch and Goyal (2008) and include dividend-price ratio (dp, given by the difference between the log of dividends and the log of prices), earnings-price ratio (ep, difference between the log of earnings and the log of prices), book-to-market ratio (bm), net equity expansion (ntis), Treasury-bill rate (tbl), term spread (tms, which is the difference between the long term yield on government bonds (lty) and the Treasury-bill (tbl)), default spread (dfy, the difference between BAA and AAA-rated corporate bond yields), and stock variance (svar).

After gathering firm characteristics, macroeconomic predictors, and returns data, our primary aim is to construct a portfolio capable of replicating the returns of the S&P 500 index. Given that the S&P 500 tracks the stock performance of the top 500 companies by market capitalization listed on U.S. stock exchanges, we adopted a similar methodology. Specifically, for each month, we identified the 500 largest companies by market value (measured by mvel1), using the Permno (a company-specific identification code) of these top 500 companies to retrieve their corresponding returns. We filtered out companies with no returns, resulting in a total of 643 remaining, representing only 0.18% of the total number of stocks initially in the portfolio, which we then removed from our data set. The weight of each stock in the constructed portfolio is computed for every point in time, denoted as $w_{i,t}$, based on the market capitalization of each firm. These calculated weights will subsequently determine portfolio returns.

The firm characteristic "SIC2" corresponds to the Standard Industrial Classification (SIC) code, and we have 65 different SIC values in our portfolio. These were transformed into 65 dummy variables, less than the 74 dummies used in Gu et al. (2020), as we reduced our data to only 500 companies per month. We also scaled all predictors to values between -1 and 1 with a mean of 0. Our overarching methodological approach therefore contains a model which includes the asset's return in excess of the risk-free rate as the dependent variable, with 94 firm characteristics, 65 industry dummies, and the interaction of the 94 characteristics with each of the 8 macroeconomic factors as covariates. This results in a total of 911 independent variables.

With the computed weights, we derived the weighted returns, which served as the basis for evaluating the accuracy of our portfolio returns compared to the actual returns of the S&P 500 index. The analysis revealed a correlation of 99.54% between the two, with average returns differing by only 0.0003 from those of the S&P 500 index. Furthermore, the variance exhibited a negligible difference, while the skewness and kurtosis deviated by 0.0639 and 0.0575, respectively.

The dataset spans from March 1957 to December 2016 and is divided into three segments for analysis: training, validation, and out-of-sample testing data. Initially, the training data comprises the period from 1957 to 1974, validation data spans from 1975 to 1986, and testing data includes observations from 1987 onwards. This split changes over the 30 iterations of testing done per model as described in section 3.1.

All 15 methods used aim to approximate the excess return ($r_{i,t+1}$) through the following empirical asset pricing model:

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

where

$$E_t(r_{i,t+1}) = g^*(z_{i,t}) \tag{2}$$

Each stock is indexed as $i = 1, ..., N_t$ and months by $t = 1, ..., T$. Our objective is to isolate a representation of $E_t(r_{i,t+1})$ as a function of predictors, represented by the P-dimensional vector $z_{i,t}$, that maximizes the out-of-sample explanatory power for realized $r_{i,t+1}$.

3 Methodology

The following section provides a detailed outline of the implementation of all machine learning models discussed in this paper. As stated previously, we follow the methodology of Gu et al. (2020) as closely as possible for all models covered by their paper. Furthermore, we illustrate our implementation of three additional tree-based models, namely, Bagging (BAG), Extreme Gradient Boosting (XGB) and Bayesian Additive Regression Trees (BART). We discuss our implementation of each of these models individually and in great detail to facilitate a potential replication of these methods.

Before taking the above-mentioned deep dive into the statistical details of each model, we explain two overarching components of our methodological approach, i.e. the way we utilize our dataset splits to measure performance and the Huber loss function, in subsections 3.1 and 3.2, respectively. Although the exact implementation of these components may differ slightly across models, the general concept stays the same and is crucial for a correct recreation. Any model specific changes or additions to these concepts will be highlighted in the relevant model's subsection throughout the methodology. Finally, we also illustrate how we transform the out-of-sample R-squared values we obtain from running our models into Sharpe ratio improvements.

3.1 Model Training, Tuning & Testing

As mentioned in section 2 of this paper, we split our data into three subsequent subsets, namely, the training set, the validation set and the test set. We perform the split in a way that maintains the temporal order of the data. After splitting the data, our general approach was to train our models on the training set. Next, we use the validation set to tune the model specific hyperparameters, thereby identifying the optimal parameter values. Finally we test our trained model with the optimal hyperparameters found via validation by predicting the monthly excess returns of the first year of the test set and computing the out-of-sample R-squared of our predictions to measure their performance. We then repeat this process for every year of the test set using an expanding window approach.

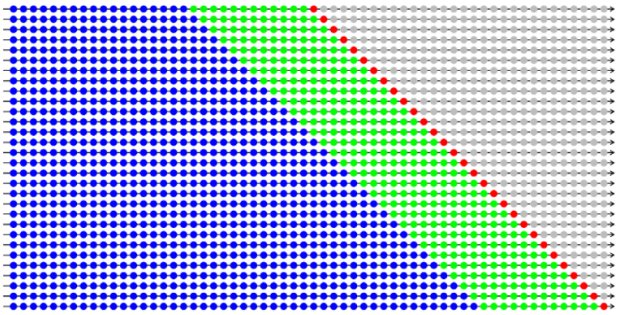

Figure 1: Illustration of Training, Tuning & Testing Methodology

We start our first iteration of training, validating and predicting with the following dataset splits: a training set containing the first 18 years, a validation set containing the subsequent 12 years and a test set containing the last 30 years. We then expand the training set by one (temporally subsequent) year for the next iteration and move forward the test set by one year while keeping the validation set at 12 years. This concept is illustrated in Figure 1, in which the blue dots represent training years, the green dots represent validation years, and the red dots represent the year tested for each iteration. Completing all iterations yields 30 predictions including a final predictions for 2016 based on 47 training years.

This method provides the most accurate performance measures of machine learning models as the use of a validation set ensures that no additional tuning occurs in the test set predictions. The implementation of 30 test years also provides us with more prediction accuracy as we can account for potential variation amongst years when computing the final out-of-sample R-squared performance measure. We compute the out-of-sample R-squared by modifying the stock level out-of-sample R-squared formula of Gu et al. (2020) for the portfolio level returns. On stock-level predictions, the out-of-sample R-squared can be calculated as follows:

$$R^2_{OOS,s} = 1 - \frac{\sum_{(i,t)\in T_3}(r_{i,t+1} - \hat{r}_{i,t+1})^2}{\sum_{(i,t)\in T_3}(r_{i,t+1})^2} \tag{3}$$

where T_3 indicates that fitted models are tested only on the test subsample. It is important to note that the denominator of the function includes returns without demeaning. This means we assume historical returns to be 0 in order to avoid model underperformance due to very noisy historical average excess returns, as shown by Gu et al. (2020).

We modify formula 3 to find the S&P 500 portfolio out-of-sample R-squared by multiplying every individual excess stock return monthly prediction, $\hat{r}_{i,t}$, by the weight of that stock within the S&P 500 at that point in time, $w_{i,t}$, in the following way:

$$\hat{r}^p_{i,t+1} = \sum_{i=1}^{n} w^p_{i,t} \times \hat{r}_{i,t+1} \tag{4}$$

Thus, the formula for the portfolio out-of-sample R-squared corresponds to:

$$R^2_{OOS,p} = 1 - \frac{\sum_{(i,t)\in T_3}(r^p_{i,t+1} - \hat{r}^p_{i,t+1})^2}{\sum_{(i,t)\in T_3}(r^p_{i,t+1})^2} \tag{5}$$

3.2 The Huber Loss Function

To counter the effect of heavy-tailed observations we apply the Huber loss function in some of our machine learning models. Huber loss is defined as follows:

$$H(x;\xi) = \begin{cases} x^2, & \text{if } |x| \leq \xi; \\ 2\xi|x| - \xi^2, & \text{if } |x| > \xi. \end{cases} \tag{6}$$

The Huber loss function, denoted as $H(\cdot)$, incorporates the features of squared loss for smaller errors and absolute loss for larger errors. This incorporation is controlled by a tuning parameter ξ, which can be optimized based on the data and is set to the 99.9% quantile in our study.

We apply Huber loss to some of our models as stock returns are known to produce heavy-tailed observations especially with stock-level predictor variables. Although the concept of Huber loss replacing the simple mean squared error loss is applicable to most of our models, the implementation is not as straight forward. Therefore, we apply Huber loss only to those functions that were reported by Gu et al. (2020) to have benefited from its implementation, except for GLM, due to computational challenges that we encountered during the process of running GLM.

3.3 Simple Linear (OLS)

Our most basic model is the simple linear predictive OLS regression. We do not expect this model to do well considering its inability to take into account a large number of dimensions and non-linear relationships. Therefore, we simplify the model to contain only 3 firm specific characteristics, namely size, book-to-market ratio and momentum, where momentum is made up of four momentum variables: short-term reversal (mom1m), mid-term reversal (mom6m), stock momentum (mom12m), and long-term reversal (mom36m)(Gu et al., 2020). We call this model OLS-3 and let it serve as our benchmark for assessing predictive performance.

We base our estimation on standard least squares, or "l_2" objective function as seen below:

$$L(\theta) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}(r_{i,t+1} - g(z_{i,t};\theta))^2 \tag{7}$$

However, in order to control for heavy-tailed observations we also run the OLS-3 model with the Huber loss function. As returns are generally quite volatile, we expect the OLS-3 +H to give a slightly better performance result than when we fit the model based on the mean squared error. It is also important to note that the OLS regression model is the only model in our analysis that has no hyperparameters and therefore does not have to be tuned using the validation set. Instead, we simply fit the model on the training sets and test it for each test year.

3.4 Penalized Linear (Elastic Net)

The OLS loses performance in the presence of many predictors and starts to overfit noise rather than extracting signals. Therefore, we analyse linear penalisation methods and look specifically, in our case,

at the elastic net. Penalized methods differ by appending a penalty to the loss function. Through this process, the machine learning method assigns greater importance to significant regressors while assigning less importance to less significant ones. Consequently, it mitigates the risk of incorporating noise into the model. There are several choices of penalty functions we focus on the elastic net penalty function:

$$\phi(\theta; \lambda, \rho) = \lambda(1-\rho)\sum_{j=1}^{P}|\theta_j| + \frac{1}{2}\lambda\rho\sum_{j=1}^{P}\theta_j^2. \tag{8}$$

The elastic net's penalty function is governed by two hyperparameters, λ and ρ, where ρ determines the weighting between the l1 and l2 parameter penalization. When $\rho = 0$, it mirrors the lasso approach, employing an absolute value, or "l1," parameter penalization. This characteristic of the lasso leads to the precise setting of coefficients for certain covariates to zero. Conversely, when $\rho = 1$, it corresponds to a ridge regression, which utilizes an l2 parameter penalization. Ridge regression pulls all coefficient estimates towards zero but doesn't enforce exact zeros for any coefficients. The tuning parameters are tuned with the use of the validation sample. We tune the elastic net for the parameter values $\rho = 0, 0.5, 1$ and $\lambda \in (1\times10^{-4}, 1\times10^{-1})$. Additionally, we compare our results with those obtained using the elastic net integrated with the Huber loss function.

3.5 Dimension Reduction (PLS & PCR)

When dealing with a large number of regressors, especially comprising macroeconomic factors and firm characteristics, it is expected to encounter highly dependent predictors. To address this issue, dimension reduction methods are commonly employed, such as Partial Least Squares (PLS) and Principal Components Regression (PCR).

PCR approach involves extracting M principal components, denoted as $Z_1, ..., Z_M$. This entails identifying a limited set of principal components that can adequately explain most of the variability in the dataset. Subsequently, these components are utilized as predictors in a linear regression model fitted using least squares. However, PCR does not take the response variables into account when selecting the components, making it an unsupervised model .

In contrast, PLS follows a similar principle but does so in a supervised manner, incorporating the dependent variables in the identification of the components. This ensures that the components not only capture the variance in the regressors but also consider the response variables. For each predictor j, a coefficient φ_j is derived by regressing the dependent variable on the predictor j. The first component is a linear combination of all the regressors and their respective coefficients. This method assigns greater weight to regressors with more significant partial effects on the regressand. Subsequent components are computed similarly to the first one but on the orthogonalized data concerning the previously constructed components (James et al., 2013).

By employing PLS or PCR, instead of utilizing P predictors directly, we reduce the dimensionality to K principal components. These methods then forecast the K θs using an Ordinary Least Squares (OLS) regression. K is a hyperparameter determined through the validation sample. Since specific values were not disclosed, we set $K \in (1, 3, 5, 7, 10, 20, 40, 50)$ for PCR, and $K \in (1, 5, 20, 40, 50)$ for PLS.

3.6 Generalized Linear (GLM)

Generalized linear model (GLM) is a flexible generalization of ordinary linear regression. In this model, we advance towards a nonparametric approach by integrating a K-term spline series expansion of the predictors.

$$g(z; \theta, p(.)) = \sum_{j=1}^{P} p(z_j)'\theta_j \tag{9}$$

Here, the p(.) function represents a spline series of order two:

$$p(.) = (p_1(.), p_2(.), ..., p_k(.) = (1, z, (z - c_1^2), (z - c_2^2), ..., (z - c_{K-2}^2) \tag{10}$$

where $c_1, c_2, \ldots, c_{k-2}$ denote knots, and the parameters now form a K×N matrix $\theta = (\theta_1, \theta_2, ..., \theta_n)$.

As per Gu et al. (2020), given the presence of 3 knots, we set K=3 as one of our tuning parameters. The model adjusts the traditional linear form by integrating a spline series expansion, thereby enhancing flexibility in capturing the relationship between predictors and the outcome variable.

The usage of splines leads to a great increase in the number of model parameters. To control degrees of freedom we use the group lasso as penalization function. Group Lasso selects either all K spline terms associated with a given characteristic or none of them. Additionally, we either use least squares or Huber robust objective function. We tune our model with $\lambda \in (1 \times 10^{-4}, 1 \times 10^{-1})$ and K=3.

3.7 Regression Trees

Regression trees are a type of decision tree used for predicting continuous target variables. They recursively partition the feature space, selecting features and thresholds at each node to minimize the variance of the target variable within each subset. This process continues until a stopping criterion is met, resulting in leaf nodes where predictions are made based on the mean (or median) of the target variable. Regression trees are interpretable and can capture complex nonlinear relationships and interactions between features, but they are prone to overfitting without proper constraints. More formally, the prediction of a tree, T , with K "leaves" (terminal nodes), and depth L, can be written as

$$g(z_i, t; 0, K, L) = \sum_{k=1}^{K} \theta_k \mathbf{1}_{\{(z_i,t) \in C_k(L)\}}. \tag{11}$$

where $C_k(L)$ represents one of the K partitions of the dataset. Each partition is a product of up to L indicator functions of the predictors. The constant associated with partition k, denoted as θ_k, is defined to be the sample average of outcomes within the partition. Trees are among the prediction methods most prone to overfitting, and therefore must be heavily regularized.

3.7.1 Light Gradient Boosting (LGBM)

The first of our regression tree models is the regularization technique known as "boosting," which combines forecasts from oversimplified trees iteratively. This approach, rooted in boosting theory, aggregates predictions from weak learners to form a single "strong learner" ensemble with improved stability. In our implementation, instead of utilizing traditional Gradient Boosted Regression Trees (GBRT), we opted for Light Gradient Boosting Machine (LGBM) due to its advantages in computational power. LGBM employs a leaf-wise tree growth strategy and efficient histogram-based algorithms, making it faster and more scalable for large-scale datasets.

The process starts by fitting a shallow decision tree, typically with a small depth (e.g., $L = 1$). This initial tree serves as a weak predictor, exhibiting substantial bias within the training dataset. Subsequently, another simple tree of the same shallow depth captures the prediction residuals from the initial tree. The predictions from these two trees are combined into an ensemble prediction. To mitigate overfitting, the contribution of the second tree is scaled down by a factor ν (ν ranges between 0 and 1).

At each iteration b, a new shallow tree is fitted to the residuals from the model comprising $b - 1$ trees. The residual prediction from this new tree is integrated into the overall prediction with a shrinkage weight

ν. This iterative process continues until the ensemble consists of B trees. The final output of the LGBM model is an additive combination of shallow trees characterized by three tuning parameters: the maximum depth (L), the shrinkage factor (ν), and the total number of trees (B). These parameters are adaptively selected during the validation phase to optimize the model's performance. In our analysis, we again look at the same tuning parameter values as the underlying paper with $L = \{1, 2\}$, $B \in \{1, 1000\}$ and $\nu \in \{0.01, 0.1\}$.

3.7.2 Bagging (BAG)

To extend Gu et al. (2020)'s comparative analysis, we introduced three additional methods to enhance the breadth of our investigation. The first method we incorporated is the ensemble learning technique known as "bagging," which stands for Bootstrap Aggregating. We included bagging due to its capability to improve prediction stability and accuracy by aggregating multiple predictors generated from bootstrapped samples of the training data.

In our implementation, we employ the Bagging algorithm (BAG) to construct an ensemble of predictors. The process begins by randomly selecting subsets of the training data with replacement, forming multiple bootstrap samples. Each bootstrap sample is then used to train a base learner, often a decision tree, on the respective subset of data. Subsequently, predictions from each base learner are aggregated to form the ensemble prediction. Unlike boosting, which prioritizes learning from misclassified instances, bagging treats all observations equally during training.

At each step of the bagging process, a new bootstrap sample is drawn, and a new base learner is trained. This process continues iteratively until a predefined number of base learners, denoted as B, have been trained. The final output of the Bagging model is obtained by averaging the predictions from all base learners. This ensemble prediction tends to have reduced variance compared to individual base learners, leading to improved generalization performance.

Similar to GBRT, the Bagging model also involves tuning parameters to optimize its performance. These parameters typically include the the maximum depth of individual trees (L), the total number of trees (B). Here we test for the values $L \in \{3, 6\}$ and $B = 300$.

3.7.3 Random Forest (RF)

Random forests present an advancement over bagged trees, implementing a minor modification that removes the trees' correlations. Analogous to bagging, multiple decision trees are crafted based on bootstrapped training samples. However, during tree construction, whenever a split is being considered, a random set of m predictors is selected from the total p predictors as candidates for splitting. A single predictor from this subset is then allowed for the split. A fresh set of m predictors is sampled at each split, usually with m being around $\sqrt{p}$, meaning the number of predictors considered per split is approximately the square root of the total predictors, leading to a structure like the one in Figure 2.

In essence, when constructing a random forest, the algorithm is deliberately prohibited from considering the majority of available predictors at each split. Although this might sound unconventional, it's backed by a clever rationale. For instance, if there exists a particularly dominant predictor alongside several moderately influential predictors in the dataset, the majority of the bagged trees will use this dominant predictor in their top splits. Consequently, all bagged trees will bear significant resemblance to one another, resulting in highly correlated predictions. However, averaging these highly correlated predictions doesn't lead to as pronounced a reduction in variance as averaging uncorrelated ones. To address this, random forests limit each split to consider only a subset of the predictors. Thus, on average, $\frac{p-m}{p}$ of the splits will not factor in the dominant predictor, allowing other predictors to have more influence. This process can be seen as "decorrelating" the trees, which makes the average of the trees less variable and therefore more dependable (James et al., 2013).

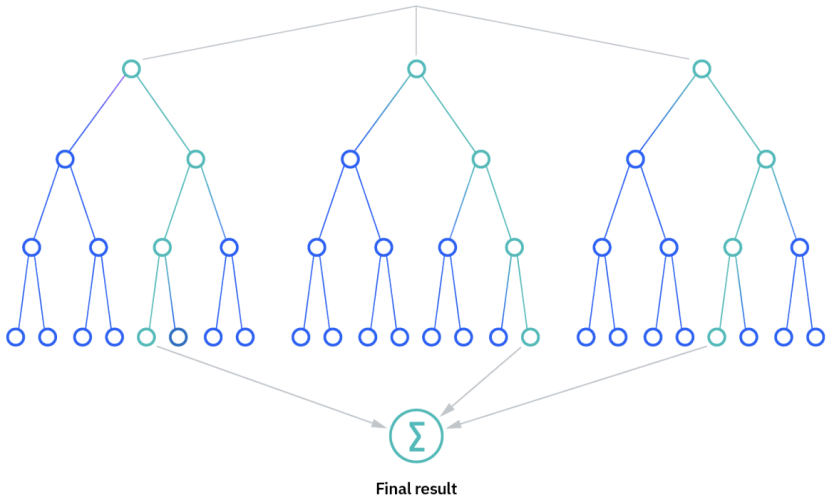

Figure 2: Graphical representation of random forest

The random forest algorithm requires tuning parameters to optimize its performance. The main tuning parameters for random forest include *n. estimators*,[1], which specifies the number of trees in the forest, *max depth*, which sets the maximum depth of each tree and *max features*, which defines the maximum number of features considered for splitting at each node. In the paper, *n. estimators* is set to 300, *max depth* ranges from 1 to 6, and *max features* spans [3, 5, 10, 30, 50, 100,...], offering a wide range of options. However, for practical reasons, including computational limitations, we decided to reduce our tuning parameter options. We used *n. estimators*: [300], *max depth*: [3, 6], and *max features*: [30, 50, 100], which still provide a diverse enough set of parameters for tuning RF while being more computationally efficient. This choice allows us to approximate the performance of the models described in the paper while adapting to the constraints of our computational resources.

3.7.4 Extreme Gradient Boosting (XGBoost)

The second newly introduced model is the ensemble learning technique known as "Extreme Gradient Boosting" (XGBoost), which we incorporated to augment our analysis of machine learning techniques. XGBoost is renowned for its robustness and scalability as an implementation of gradient boosting machines.

In XGBoost, decision trees are sequentially added to the ensemble, with each tree aiming to correct the errors made by the previous ones. Unlike bagging, which treats all observations equally, XGBoost prioritizes learning from the residuals of the previous models, focusing on instances where the model performs poorly. This targeted learning strategy enables XGBoost to effectively address challenging patterns within the data.

Each iteration of the XGBoost process involves training a new decision tree, referred to as a weak learner, to fit the residual errors of the ensemble. These trees are typically shallow to prevent overfitting and to maintain computational efficiency. The final output of the XGBoost model is obtained by summing the predictions from all individual trees in the ensemble, leading to improved generalization performance and robust predictions.

Similar to Bagging, tuning parameters such as the maximum depth of individual trees (L) and the total number of trees (B) are crucial for optimizing XGBoost's performance. In our analysis, we explore a range of parameter values, including $L \in \{1, 2\}$ and $B \in \{500, 1000\}$, to comprehensively assess the model's performance across different configurations. By including XGBoost alongside the previous methods, we aim to

[1]The notation used to describe the hypeparameters is the same implemented in the Python notebook, to facilitate the comparison.

incorporate a high performing ML model into our comparative analysis that has the potential to rival neural networks in terms of prediction accuracy.

3.7.5 Bayesian Additive Regression Trees (BART)

Our last newly introduced model is the ensemble learning technique known as "Bayesian Additive Regression Trees" (BART), which is a Bayesian approach to fitting additive regression models using decision trees. In BART, each decision tree is treated as a component of a flexible regression model, and the model is constructed by summing the predictions from an ensemble of trees. Unlike traditional bagging or boosting methods, BART uses a Bayesian framework to estimate the parameters of the model, allowing for uncertainty quantification and regularization.

The BART algorithm starts with an initial model and iteratively updates it by adding new trees, each time sampling from the posterior distribution of the model parameters. This process results in a posterior distribution over the space of regression functions, providing a measure of uncertainty for each prediction. The final prediction in BART is obtained by averaging predictions from multiple samples of the posterior distribution, resulting in an ensemble prediction that captures both model uncertainty and variability due to the data.

Similar to other ensemble methods, BART also involves tuning parameters to optimize its performance, such as the number of trees in the ensemble and the prior distributions on the model parameters. In our analysis, we explore a range of parameter values, including the number of trees (num_trees) set to 100, 200, and 300, the burn-in period (burnin) set to 50, 150, and 200, and the maximum number of stages (max_stages) set to 500, 1000, and 2000. By varying these parameters, we aim to identify the optimal configuration for the BART model, balancing model complexity and predictive performance.

3.8 Neural Networks (NN1 - NN5)

The last non-linear model used is the neural network. This deep-learning method is arguably the most powerful machine-learning method currently used. However, its high performance and flexibility come at the cost of low interpretability. A neural network is often called a black box because we can not entirely see what is happening in the neural network. It also requires a big training sample to be effective and not overfit. Furthermore, there are a lot of parameters and structural choices we have to define for a neural network to work. Choices as the number of hidden layers, the number of neurons in each layer, and which units are connected. For example, in Figure 3, there are three hidden layers, each of five neurons connected to the next layer. The input size is also five, and the output layer has three neurons. Our parameters and structure differ from Figure 3 but correspond with the parameters chosen by Gu et al. (2020).

In this paper, we examine the predictive power of traditional feed-forward neural networks. These networks draw inspiration from biological brains, consisting of interconnected layers that process and transform information. The input layer consists of raw predictors, while one or more 'hidden layers' interact with these predictors nonlinearly (Gu et al., 2020). Ultimately, the 'output layer' combines the hidden layers into a single prediction. More hidden layers do not necessarily lead to better performance, as the 'universal approximation theorem' suggests that a single hidden layer is sufficient. However, literature has shown that deeper networks can achieve the same accuracy with fewer parameters. Simple networks with fewer layers and nodes often perform better in small data sets. As finding the 'perfect' Neural Network using cross-validation is unrealistic, Gu et al. (2020) have chosen a fixed structure and certain parameters, ex-ante.

Five neural networks are created with an added hidden layer in each new neural network in which the neuron count per player is determined using the 'geometric pyramid rule', and all networks are fully connected. Table 1 shows an overview of the neural networks, the number of hidden layers, and the number of neurons per layer.

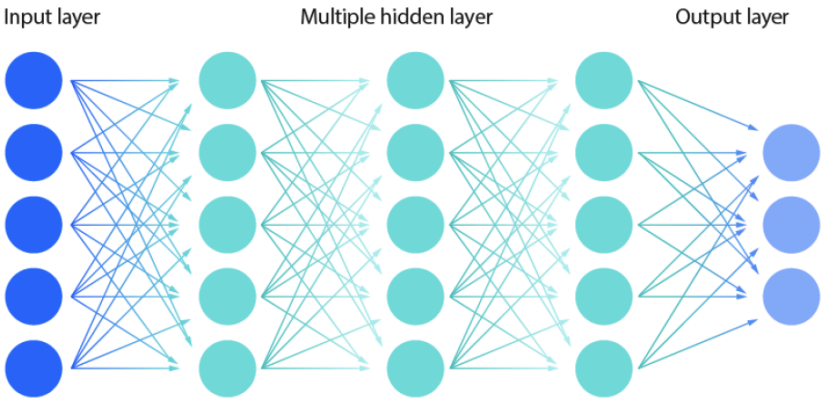

Figure 3: Graphical representation of a neural network with hidden layers

Hidden layers	1	2	3	4	5
NN1	32				
NN2	32	16			
NN3	32	16	8		
NN4	32	16	8	4	
NN5	32	16	8	4	2

Table 1: Number of layers and neurons in Neural Network

The nonlinear activation function used at all nodes is the rectified linear unit(ReLU), defined as

$$\text{ReLU(x)} = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{otherwise,} \end{cases} \tag{12}$$

Our neural network has a layered structure where $K^{(l)}$ represents the number of neurons within layer $'l'$. Each neuron's output is denoted as $x_k^{(l)}$. The input layer, $x^{(0)}$ is directly formed from the raw predictors. The network recursively updates itself. At each layer $(l > 0)$, individual neurons calculate output as follows:

$$x_k^{(l)} = \text{ReLU}(x^{(l-1)'}\theta_k^{(l-1)}) \tag{13}$$

with final output

$$g(z;\theta) = x^{(L-1)'}\theta^{(L-1)}) \tag{14}$$

Each hidden layer has $K^{(l)}(1 + K(l-1)$ weight parameters, with an additional $1 + K^{L-1}$ weights for the output layer.

Objetive Function and Optimization

We find the best neural network by minimizing the penalized l_2 objective function of prediction errors. Unlike tree-based algorithms, neural networks update all their parameters simultaneously, giving them a significant advantage. However, this flexibility comes at a cost: nonlinearity, complex structure, and many parameters make direct optimization difficult.

We use stochastic gradient descent (SGD) to address this. SGD approximates the true gradient using only a small, random subset of the data per step, making it much faster. We use the adaptive moment estimation algorithm (Adam), a more efficient version of SGD. In addition, we use l_1 penalization of the weight parameters with values $\lambda_1 \in (10^{-5}, 10^{-3})$. Due to the network's complexity, we layer several regularization techniques alongside SGD.

- **Learning Rate Shrinkage**: We gradually reduce the step size the algorithm can take as it approaches the optimum. This prevents noise from overwhelming the search process. The learning rate used is LR $\in \{0.001, 0.01\}$.
- **Early Stopping**: We start with certain parameter guesses, and at each new guess, predictions are constructed for the validation sample. The optimization is terminated when the validation sample errors start to increase. The patience we use is 5, which means that after five trials of no improvement, the model stops.
- **Batch Normalization**: We control for the variability of predictors across different network regions. The algorithm cross-sectionally demeans and variance standardizes the batch inputs for each hidden unit in each training step to prevent instability caused by changing distributions during training.
- **Ensemble**: We train multiple networks and predict by averaging all network forecasts to reduce prediction variance. We use an ensemble level of 5, which means we train 5 similar networks, which is different from Gu et al. (2020), who use an ensemble of 10. We used a lower ensemble because we had a different trade-off between computational power and accuracy. With our data set, a higher ensemble level might lead to overfitting.

The batch size, which determines how many samples the network processes before updating its weights, is set to 10,000. Larger batches lead to slower iterations and tend to have better generalization but might get stuck in local minima, which can lead to smoother convergence, however, the training might be slower.

The number of Epochs determines the number of training iterations. Each epoch consists of multiple smaller training steps where batches of data are fed into the network. In line with Gu et al. (2020), we chose to use 100 epochs. The network will not have exposure to the data to learn the complex patterns, leading to underfitting if there are too few epochs. Too many epochs might lead to the network memorizing the specific details and noise in the training set, leading to overfitting and poor performance on unseen data.

3.9 Sharpe Ratio Improvements

In order to quantify the extent to which an investor following an active ML prediction strategy may improve their portfolio performance, Gu et al. (2020) convert the out-of-sample R-squared values into Sharpe ratio estimates using the following formula:

$$SR^* = \sqrt{\frac{SR^2 + R^2}{1 - R^2}} \tag{15}$$

where SR^* denotes the Sharpe ratio of an active investment strategy based on ML predictions and SR denotes the Sharpe ratio of following a buy-and-hold strategy. We then subtract the Sharpe ratio of a buy-and-hold strategy (i.e. 0.51) from the Sharpe ratio of an active ML strategy: $\Delta SR = SR^* - SR$. This provides us with an annualized change in Sharpe ratio per ML model that an investor can expect to obtain by following an active investment strategy based on the predictions (i.e. R-squared) of that specific model.

The Sharpe ratio improvements obtained by Gu et al. (2020) can be found in Table 4 of the appendix. Most of our results are negative, as will be discussed in the following section. Therefore, we will not report Sharpe ratios since they indicate that there are more benefits in following a buy-and-hold strategy.

4 Results and Analysis

Table 2 shows the out-of-sample R-squared values we obtained for each of the implemented models. In contrast to the results found by Gu et al. (2020) in Appendix 1, it shows that all of our models seem to have a negative prediction accuracy which indicates that machine learning models are not an adequate method to forecast S&P 500 returns. As our findings are contradictory to prior research, we critically analyze each of our models' out of sample R-squared separately in this section of the paper and attempt to trouble shoot our methodological approach.

OLS-3 + H	**PLS**	**PCR**	**ENet**	**GLM**	**RF**	**LGBM**
-0.27	-0.07	-0.18	-0.02	0.09*	-0.01	-0.01
XG Boost	**BART**	**NN1**	**NN2**	**NN3**	**NN4**	**NN5**
-0.42	0.13*	-18.86	-3.61	-1.89	-3.46	-68.89

Table 2: Monthly Portfolio-level Out-of-Sample Predictive R^2

4.1 Analysis of Linear Models: OLS-3, ENet, PCR, PLS & GLM

Gu et al. (2020) reported that regularized linear models exhibit inferior predictive performance compared to other methodologies when applied to forecasting returns of the S&P 500. LS-3 and OLS-3 with Huber robust objective function are regarded as the simplest approaches, both anticipated to yield notably inadequate results. The outcomes of OLS are omitted in the paper due to their highly negative values. When applying Huber in the referred model, the results land in a monthly out-of-sample predictive R-squared of -0.22.

Our analysis of OLS-3 resulted in a monthly out-of-sample predictive R-squared of -0.67. When employing the Huber objective function the results raise to -0.27, which very close to the value found by the paper and reassures the predictive benefits of adding Huber.

Dimension reduction techniques, such as Partial Least Squares (PLS) and Principal Component Regression (PCR), are among the worst predictor methods reported in the referenced paper with out-of-sample R-squared of -0.86 and -1.55, respectively. Despite similar observations indicating inadequate predictive capacity, our analysis demonstrates comparatively better outcomes, with values of -0.068 and -0.181 for PLS and PCR, respectively, ranking among our more successful predictive methodologies.

Employing Elastic Net resulted in exacerbated negative results when integrating the Huber loss function, leading to its exclusion from this report. Conversely, utilizing the same method without loss functions yielded significantly improved results, with an out-of-sample predictive R_2 of -0.02 when setting ρ=0.5 and -0.10 when setting $\rho \in \{0, 0.5, 1\}$ to be predicted through validation and allowing for lasso ($\rho = 0$) and ridge ($\rho = 1$) as shrinkage methods.

Regarding Generalized Linear Models (GLM), we encountered numerous challenges. We were unable to fully replicate this method due to our inability to integrate Huber loss. Additionally, we had to reduce the number of iterations from 30 to 1 because GLM proved to be particularly demanding in terms of computational

power, requiring over an hour to complete a single iteration. Despite this, on a single iteration, GLM yielded an out-of-sample predictive R-square of 0.09. It is, however, not advised to compare these results with those reported in the paper or with other values, due to the afore-mentioned reasons.

4.2 Analysis of Tree-Based Models: Random Forest (RF), LGBM, Bagging, XGBoost, & BART

We expanded upon the tree-based models utilized by Gu et al. (2020) by incorporating additional methods such as XGBoost, Bagging, and BART. Additionally, we opted for implementing LGBM instead of GBRT due to its computational advantages.

LGBM demonstrated a significant computational advantage over GBRT, achieving an R^2 of -0.01 with Huber Loss. In contrast, XGBoost (Chen and Guestrin, 2016), known for its computational efficiency and suitability for large datasets, only yielded an R^2 of -0.42. We also investigated the performance of the newly added methods, BART and Bagging, which are further explained in James et al. (2013). However, computational constraints limited our ability to analyze results comprehensively. To address this issue, we reduced the number of iterations from 30 to 15. With limited iterations BART achieved an out-of-sample R^2 of 0.13 which is the best R^2 we observe but also more likely to be due to luck based on the law of large numbers. For Random Forest (RF), we obtained an out-of-sample R^2 of -0.01 which is in line with the other tree-based methods.

Overall, our analysis highlights the computational advantages of LGBM over traditional GBRT, as well as the potential of BART despite computational challenges. Further exploration is needed to fully assess the performance of Bagging in our study.

4.3 Analysis of Neural Networks: NN1 - NN5

While Gu et al. (2020) found neural networks to be their best-performing models, with NN3 having an out-of-sample predictive R^2 of 1.80, our results demonstrate a contrasting outcome. Our neural networks proved to be the least effective method with our best model (NN3) yielding an out-of-sample predictive R^2 of -1.89. In line with the results of Gu et al. (2020), adding more layers reduces the model's performance after the neural network with three hidden layers. NN4 and NN5 have an out-of-sample predictive R^2 of -3.46 and -68.89, respectively.

One potential explanation for the underperformance of neural networks and regression trees is overfitting. Neural networks and regression trees need a big training sample and a high level of signal-to-noise. Gu et al. (2020) also have a low signal-to-noise level but potentially training on a broader selection of stocks beyond those included in the S&P 500. This detail is not clarified in their paper. As we only replicate a part of the original work, it is possible that Gu et al. (2020) trained their models on all stocks used in their cross-sectional section and subsequently tested this on the S&P 500. In our case, using the same parameters with a smaller training set could have exacerbated overfitting (Chollet, 2021). A smaller batch size, fewer epochs, and a lower ensemble size would be more appropriate to mitigate this.

5 Conclusion and Limitations

To summarize, we replicated the time-series methodology employed by Gu et al. (2020) in an effort to forecast the S&P 500 using machine learning models. This comparative analysis covers all 12 models mentioned by Gu et al. (2020) and three additional models that can be found in most higher level education data science textbooks. The models are the following: OLS, ENet, PLS, PCR, GLM, LGBM, BAG, RF, XGB, BART and NN1-5.

In their paper, Gu et al. (2020) find that ML models offer a clear advantage over more traditional models in predicting stock returns. They also find that neural networks perform best out of all studied models while

shallow networks outperform deep ones. Lastly, they show that implementing an active investment strategy based on predictions of the studied ML models results in fairly substantial improvements in Sharpe ratio over a traditional buy and hold strategy.

In comparison, our findings yield a rather surprising result as none of the ML models we tested show any predictive power in forecasting excess returns. Hence, we are also not able to show any improvements in Sharpe ratio by implementing active investment strategies based on ML predictions. One take-away that we were able to replicate on the other hand was that shallow networks perform "less badly" than deeper networks with NN3 providing the best result amongst the neural networks.

Considering that our results are not in line with those found by Gu et al. (2020) it is difficult to draw conclusions as it is unclear whether our results are accurate or not. Therefore, we choose to elaborate on the limitations and difficulties we faced during the research process with the aim to facilitate future replications of this methodological approach. The results obtained from future replications my shed more light on this matter.

Our primary limitation in this research project was a lack of computing power [2]. In our experience it is not uncommon to run a code only to find that it is incorrect because it does not yield the results one is looking for. However, when a code takes multiple days to run, this sort of error can result in a major setback. We encountered this issue with various tree-based methods and neural networks for which the longest run-times consisted of multiple days.

In addition to this, we encountered difficulties in data pre-processing and model implementation due to a lack of clear information. This may be partly due to the fact that the methodology of our reference paper is not tailored specifically to the time-series analysis but also incorporates the cross-sectional data and methodology. One major consequence of this was that the authors did not explicitly specify whether they trained their models on the complete dataset of 30,000 firms or solely on the 500 firms that were part of the S&P 500 every month. Especially in the context of regression trees and deep learning, such a change in training data sample size is crucial for a successful implementation. In hindsight, we assume that their models were trained on data for all 30,000 firms and simply tested on the S&P 500 portfolio which would explain our deviation from their findings. However, we would like to highlight that including such a large training sample of 30,000 firms as mentioned above, would result in even more excessive run-times which should be accounted for in the time management of future replications.

[2]We want to acknowledge the generous support of the Erasmus Center for Data Analytics, and especially Jos, Farshad, and Zaman for graciously lending us access to their equipment. This allowed us to significantly reduce the running time of some of the models.

References

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794.

Chollet, F. (2021). *Deep learning with python*. Simon; Schuster.

Gu, S., Kelly, B., & Xiu, D. (2020). Empirical asset pricing via machine learning. *The Review of Financial Studies*, *33*(5), 2223–2273.

James, G., Witten, D., Hastie, T., Tibshirani, R., et al. (2013). *An introduction to statistical learning* (Vol. 112). Springer.

Welch, I., & Goyal, A. (2008). A comprehensive look at the empirical performance of equity premium prediction. *The Review of Financial Studies*, *21*(4), 1455–1508.

A Appendix

OLS-3 + H	PLS	PCR	ENet + H	GLM + H	RF
-0.22	-0.86	-1.55	0.75	0.71	1.37

GBRT + H	NN1	NN2	NN3	NN4	NN5
1.40	1.08	1.13	1.80	1.63	1.17

Table 3: Authors' Monthly Portfolio-level Out-of-Sample Predictive R^2 - (Gu et al., 2020)

OLS-3 + H	PLS	PCR	ENet + H	GLM + H	RF
-	-	-	0.08	0.08	0.14

GBRT + H	NN1	NN2	NN3	NN4	NN5
0.15	0.11	0.12	0.20	0.17	0.12

Table 4: Authors' Implied Sharpe Ratios Improvements - (Gu et al., 2020)