ERASMUS UNIVERSITY ROTTERDAM
Erasmus School of Economics

Interpreting machine learning in equities

Bachelor Thesis Quantitative Finance

Author: Ralph Hermeling
Student ID number: 513098

Supervisor: Drs. T. van der Zwan
Second assessor: Prof. Dr. M. van der Wel

Abstract

We conduct a comparative analysis among popular machine learning methods for predicting monthly risk premiums of US equities. Among the methods is XGBoost, a gradient boosting algorithm with countless machine learning competitions wins. Moreover, we use the latest model-agnostic[1] tools, such as SHapley Additive exPlanations (SHAP) and Permutation Feature Importance (PFI), to interpret XGBoost's predictions and better understand the behavior of risk premiums. Our results show that we can not replicate the performance of Gu, Kelly, and Xiu (2020) by using a subset of their data that includes the most important predictors, which are variations on momentum, liquidity, and volatility. This suggests that larger datasets are essential to deep learning's success. XGBoost had the best predictive accuracy in our research and SHAP indicates that stock fundamentals are relatively more important for larger stocks than for smaller stocks and that the opposite is true for momentum.

July 3, 2021

[1] Seperating the explanations from the machine learning model

Contents

1 Introduction

Gu, Kelly, and Xiu (2020) demonstrated large economic gains to investors who use complex machine learning methods for measuring asset risk premiums.[2] However, one large disadvantage of these complex machine learning methods is that they do not explain their predictions. Understanding why those predictions were made can be as crucial as creating accurate forecasts. Not only does it provide insights as to how we can improve our models, but it can get us a step closer to understanding the behavior of risk premiums. This raises the dilemma of sacrificing predictive accuracy for more interpretability or is there a method we can use to interpret these sophisticated models?

Recently S. Lundberg and Lee (2017) proposed SHapley Additive exPlanations (SHAP), which is a framework for interpreting complex machine learning methods. SHAP provides each feature an importance value for a particular prediction. It is based on Shapley values (Shapley, 1997) from conditional game theory and has desirable properties. Besides SHAP, we can also estimate a feature's importance by measuring the decrease in out-of-sample R^2 when omitting it from the test data, this is referred to as Permutation Feature Importance (PFI). Gu et al. (2020) followed this approach. We apply both techniques to interpret machine learning.

Our comparative analysis of machine learning methods for measuring asset risk premiums is different in two ways to that of Gu et al. (2020). First, we use a subset of their data due to a lack of computing resources. The selected subset results from the outer product between 20 of their most important firm characteristics and 9 macroeconomic variables (including a constant), adding up to 180 stock-level covariates. The sample remains the same. It begins in 1957, ends in 2016, and consists of approximately 30,000 equities. Second, we add XGBoost from Chen and Guestrin (2016) to the comparative analysis, which is an accurate and scalable implementation of the gradient boosting algorithm. Many cutting-edge industry applications use XGBoost.

From our research we conclude that we cannot replicate the same performance as Gu et al. (2020) by solely relying on variations resulting from the outer product of the 20 most important firm characteristics and 9 macroeconomic variables. Since all our models heavily under perform (negative and small values for R^2). Neural networks, which were the winners in their research, actually produced negative values for R^2. This confirms the notion that a larger dataset contributes to deep learning's success. Our best model was XGBoost, which had the highest R^2 and was one of the few models that actually produced positive R^2. Note that its values for R^2 were still a lot smaller than that of Gu et al. (2020).

[2] Our aim is to measure conditional expected stock returns in excess of the risk-free rate. In academic finance this is traditionally referred to as the "risk premium" due to its close connection with equilibrium compensation for bearing equity investment risk. We use the terms "expected return" and "risk premium" interchangeably.

By using SHAP and PFI for XGBoost we discovered that variations on momentum are relatively more important for smaller sized stocks than for larger sized stocks. Whereas variations on stock fundamentals are relatively more important for larger size stocks than for smaller size stocks.

2 Data

We use the data set by Gu, Kelly, and Xiu (2020), which contains firm characteristics of around 30,000 US equities. The monthly returns for these equities are collected by using WRDS. Moreover, we obtain the Treasury-bill rate to proxy for the risk-free rate when calculating individual excess returns.[3] The sample starts in March 1957 and ends in December 2016 adding up to 60 years. Due to lack of computing power we opted for a subset of the available firm characteristics. To decide which firm characteristics to keep we researched the variable importance figure in (Gu et al., 2020, p. 31) and picked the 20 most important firm characteristics among the neural networks. Since, these models had the best monthly forecast performance. The 74 industry dummies defined by Gu et al. (2020) do not appear in any of the variable importance figures and are omitted for this reason. The resulting 20 most important firm characteristics are listed in Appendix B.[4] Furthermore, we also include the following eight macroeconomic explanatory variables from Welch and Goyal (2008): dividend-price ratio (dp), earnings-price ratio (ep), book-to-market ratio (bm), net equity expansion (ntis), Treasury-bill rate (tbl), term spread (tms), default spread (dfy), and stock variance (svar).[5]

For pre-processing we map the firm characteristics and macroeconomic variables cross-sectionally and period-by-period into the [-1,1] interval by using the MinMaxScaler method from Pedregosa et al. (2011). This is done to achieve optimal machine learning performance and prevent data leakage. Moreover, we perform the same outer product as Gu et al. (2020) between the firm characteristics and the macroeconomic variables. The resulting stock-level covariates $z_{i,t}$ are defined as follows:

$$z_{i,t} = x_t \otimes c_{i,t}, \tag{1}$$

where x_t is a $P_x \times 1$ vector of macroeconomic variables including a constant and $c_{i,t}$ is a $P_c \times 1$ matrix of firm characteristics for each stock i at time t. For predicting an asset's excess return we use $z_{i,t}$, which is a $(P_c P_x = P) \times 1$ vector containing interactions between stock-level characteristics and macroeconomic state variables. The final number of covariates P is equal to

[3] We picked the monthly Treasury-bill rate from Amit Goyal's website. Note that this rate is annualized. Hence one should convert it into a monthly rate by taking the power of $\frac{1}{12}$

[4] The monthly data are available from Shihao Gu's website.

[5] This monthly data are available from Amit Goyal's website.

$20 \times (8 + 1) = 180$.

The motivation behind the structure of $z_{i,t}$ according to Gu et al. (2020) is the standard beta-pricing representation of the asset pricing conditional Euler equation, which is defined as follows:

$$E_t(r_{i,t+1}) = \beta_{i,t}^{'} \lambda_t. \tag{2}$$

The design of $z_{i,t}$ allows for two things. First pure stock level interactions are able to enter the expected returns via $c_{i,t}$ in a similar way as $\beta_{i,t}^{'}$ does and it also enables aggregate economic conditions to enter which is in analogy with λ_t. We borrow the example from Gu et al. (2020) to show the analogy, if $\beta_{i,t} = \theta_1 c_{i,t}$ and $\lambda_t = \theta_2 x_t$, for some constant parameter matrices θ_1 $(K \times P_c)$ and θ_2 $(K \times P_x)$, then the beta-pricing model in (2) becomes

$$g^*(z_{i,t}) = E_t(r_{i,t+1}) = \beta_{i,t}' \lambda_t = c_{i,t}' \theta_1' \theta_2 x_t = (x_t \otimes c_{i,t})' vec(\theta_1' \theta_2) =: z_{i,t}' \theta, \tag{3}$$

where $\theta = vec(\theta_1' \theta_2)$. The broader model is more general since $g^*(.)$ also allows for nonlinearity. Moreover, nonlinear $g^*(.)$ formulations enlarge the predictor set with functional transformations of the baseline predictor set $z_{i,t}$.

3 Methodology

This section describes the methods we use to predict an asset's excess return and to interpret predictions. We follow Gu, Kelly, and Xiu (2020) and view an asset's excess return as an additive prediction error model:

$$r_{i,t+1} = E_t(r_{i,t+1}) + \epsilon_{i,t+1}, \tag{4}$$

where

$$E_t(r_{i,t+1}) = g^*(z_{i,t}). \tag{5}$$

Stocks are represented by the index $i = 1, ..., N_t$ and months by $t = 1, ..., T$. If we perform research on a yearly basis, then t corresponds to years. Our aim is to find a function of predictor variables, which maximizes the out-of-sample explanatory power for realized $r_{i,t+1}$, and which can be used as an proxy for $E_t(r_{i,t+1})$. The P-dimensional vector $z_{i,t}$ in equation (5) represents the predictor variables. The $g^*(.)$ function corresponds to the conditional expected return and depends neither on i or t. By retaining the same form across time and different stocks, it allows the model to use information from the entire panel. Gu, Kelly, and Xiu (2020) state that this provides extra stability to estimates of risk premia for any individual asset. Lastly, $g^*(.)$ depends only on z through $z_{i,t}$. This means that we make predictions without the knowledge of history prior to t, or of any individual stock that is not the i^{th}.

3.1 Sample Splitting and Hyperparameter Optimization

We perform the same sample splitting strategy as Gu et al. (2020). Every year from the sample period of 1987-2016 is considered as an out-of-sample year and has its own training, validation and testing set. The size of the validation and testing set remain the same across all out-of-sample years and equal 12 years and 1 year, respectively. The size of the training set depends on the out-of-sample year since it increases with every refit. To explain this refitting process, let 1987 denote the first out-of-sample year, then the training period is 18 years (1957-1974) and the validation period is 12 years (1975-1986). For the next out-of-sample year 1988 we keep the validation period the same size but roll it over by one year, and we increase the training set by one year. This results in a training period of 19 years (1957-1975) and a validation period of 12 years (1976-1987).

Each set fulfills the following purpose. The training set is used for estimating the model given a set of hyperparameters[6], the validation set for tuning these hyperparameters, and the testing set for evaluating the model's forecast performance. Our approach of finding the best performing configuration of hyperparameters differs from (Gu et al., 2020) since we use Bayesian Optimization (Bergstra, Yamins, & Cox, 2013) or Hyperband (Li, Jamieson, DeSalvo, Rostamizadeh, & Talwalkar, 2017) for automatic hyperparameter optimization. These two algorithms are described in the Appendix.

3.2 Dimension Reduction: Principal Components Regression and Partial Least Squares

Principal Components Regression (PCR) and Partial Least Squares (PLS) are both popular dimension reduction techniques. PCR can be divided into two steps. In the first step Principal Component Analysis (PCA) is performed in order to find components, which capture the covariance structure among the explanatory variables the best. Then in the second step components, that explain the most variance, are used for prediction in a linear regression type fashion. This results in the omission of components with low variance, which causes regularization of the prediction problem.

One of PCR's disadvantages is that it does not take into account the ultimate statistical objective when selecting components, since it picks components that explain the most variance among the explanatory variables. This may result in the omission of significant components that explain little variance.

[6]We define hyperparameters as parameters of a machine learning model that control its learning process and complexity.

PLS however does incorporate the objective function when computing its components, because it selects them based on their covariance with the target variable. PLS regression works in the following way. Compute the univariate return prediction coefficient for each variable j using OLS. The resulting coefficient is denoted by ϕ_j and corresponds to the 'partial' sensitivity of the returns to every variable j. Thereafter, create an aggregate component that consists of all the predictors weighed proportional to their respective ϕ_j. Thus, larger ϕ_j gets a larger weight. Additional components are created from the orthogonalization of the target and all explanatory variables to the previously made components. This process is repeated untill the desired number of PLS components is achieved. We use the same implementation of the models as Gu et al. (2020)

$$R = Z\theta + E, \tag{6}$$

where R is the $NT \times 1$ vector of $r_{i,t+1}$, Z is the $NT \times P$ matrix of stacked predictors $z_{i,t}$, and E is a $NT \times 1$ vector of residuals $\epsilon_{i,t+1}$. For both methods we define the forecast model as follows

$$R = (Z\Omega_K)\theta_K + \tilde{E}, \tag{7}$$

where Ω_K is a $P \times K$ matrix with columns $w_1, w_2, ..., w_k$. The weights used to create the j^{th} component are denoted by w_j. Thus by definition $Z\Omega_k$ represents the dimension-reduced version of the full explanatory variable set. Thereby, θ_K is a $K \times 1$ vector instead of a $P \times 1$ one. The objective function of the two models differ. PCR selects the combination weights Ω_K in a recursive way and to get the j^{th} component we solve

$$w_j = \arg\max_{w} \text{Var}(Zw), \text{ s.t. } w'w = 1, \text{ Cov}(Zw, Zw_l) = 0, \, l = 1, 2, ..., j-1. \tag{8}$$

For the j^{th} PLS component we solve

$$w_j = \arg\max_{w} \text{Cov}^2(R, Zw), \text{ s.t. } w'w = 1, \text{ Cov}(Zw, Zw_l) = 0, \, l = 1, 2, ..., j-1. \tag{9}$$

To implement these methods we use the PCR and PLS class from Pedregosa et al. (2011) in Python. The only tune-able parameter for both of these methods is the number of components K.

3.3 XGBoost and Random Forest

To implement a gradient boosted regression trees (GBRT) algorithm as noted by Gu et al. (2020) we use XGBoost (XGB) from Chen and Guestrin (2016), which provides scalability in all scenarios and runs 10 times faster than existing popular solutions on a single machine. XGB owes its scalability to several system and algorithmic improvements. Here are the following

improvements: a tree algorithm which handles sparse data and a quantile sketch procedure which enables handling instance weights in approximate tree learning. XGB also offers distributed and parallel computing which decreases the learning time such that more time is available for model exploration. Moreover, XGB provides a cache-aware block structure [7] for out-of-core tree learning. XGB uses K additive functions to predict. We follow Chen and Guestrin (2016) for the notation of this model. The predictions are made as follows

$$g(z_{i,t}, K) = \sum_{k=1}^{K} f_k(z_{i,t}),\ f_k \in \mathcal{F}, \tag{10}$$

where $\mathcal{F} = \{f(z_{i,t}) = w_{q(z_{i,t})}\}(q : \mathbb{R}^P \rightarrow T, w \in \mathbb{R}^T)$ denotes the space of regression trees. Variable q corresponds to the structure of each tree that maps a sample to their respective leaf index. Each tree is represented by f_k and has its own independent tree structure q and leaf weights w. The leaves of every regression tree have a continuous score, where w_i corresponds to the score of leaf i. The total number of leaves in each tree is denoted by T. The tree structure q also consists of the decision rules thus given a sample we can classify it into one of the leaves and get the prediction by summing up the score in the corresponding leaves. The functions $f_k(.)$ in equation (10) use the following regularized objective function to learn

$$\mathcal{L}(g) = \sum_i l(\hat{r}_{i,t}, r_{i,t}) + \sum_k \Omega(f_k), \tag{11}$$
$$\text{where } \Omega(f_k) = \gamma T + \alpha||w|| + \frac{1}{2}\lambda||w||^2.$$

The loss function l is differentiable and convex. Regularization is introduced through Ω, which includes l_1 and l_2 penalization on the leaf weights denoted by α and λ, respectively. Moreover, the depth of a tree is penalized by a factor γ. This design punishes the model's complexity and reduces overfitting. The tree model provided in Equation (11) has functions as parameters. As a result it can not be solved by using standard optimization methods in Euclidean space. Hence, the model is trained in an additive way. Let $r_{i,t}^{(s)}$ be the prediction of the i^{th} instance at iteration s and time t. To minimize the objective function we add f_s and it becomes

$$\mathcal{L}^{(s)} = \sum_{i=1}^{n} l(r_{i,t}, \hat{r}_{i,t}^{(s-1)} + f_s(z_{i,t})) + \Omega(f_s). \tag{12}$$

[7] A cache-aware block model is created to solve the drawback of RAM, which is that it considers only one memory and is not an approximation of the actual hardware (Angrish & Garg, 2011). The cache-aware model is a two-level hierarchy model with block transfer.

This implies that we greedily add f_s such that it improves our loss score. To quickly optimize the objective function we use second-order Taylor approximation.

$$\mathcal{L}^{(s)} \simeq \sum_{i=1}^{n} [l(r_{i,t}, \hat{r}_{i,t}^{(s-1)}) + g_i f_s(z_{i,t}) + \frac{1}{2} h_i f_s^2(z_{i,t})] + \Omega(f_s), \tag{13}$$

where $g_i = \delta_{\hat{r}^{s-1}} l(r_{i,t}, \hat{r}_{i,t}^{(s-1)})$ and $h_i = \delta_{\hat{r}^{s-1}}^2 l(r_{i,t}, \hat{r}_{i,t}^{(s-1)})$ are first and second order gradient statistics on the loss function, respectively. This approximation enables us to estimate the optimal weight of a leaf and its corresponding value. The details of XGB's algorithm are described in the Appendix, but it begins with initialization of a shallow tree. Thereafter, the gradient and second derivative of the regularized loss function as seen in Equation (13) are computed in order to fit a new decision tree to the residuals from the previous iteration.[8] Next, its prediction gets scaled by a factor η and added to the total. This process is repeated until the ensemble consists of K trees. Besides the scaling of predictions, there are three other parameters we use to reduce overfitting when building trees: max depth L which corresponds to the longest path from a root to a leaf and controls the tree's complexity, ϑ corresponds to the percentage of randomly chosen features to be available when constructing a tree, and ζ which is the minimum sum of weights in a child. As a result we have an additive model of shallow trees with the following hyperparameters (L, η, K, γ, α, λ, ϑ, ζ) which are tuned using Bayesian Optimization. We implement the model in Python using the XGBRegressor class from Chen and Guestrin (2016).

Random Forest (RF) is just as XGB an ensemble forecasting technique, but the ensembles are created in a different way. RF builds each tree from a bootstrapped sample of the training set. Moreover, RF constructs each tree from a random subset of the features. The size of this random subset is denoted by ι. The reason behind these two measures, which create randomness, is to reduce the variance of the forest estimator. Decision trees typically experience high variance and also tend to overfit. As Pedregosa et al. (2011) state by introducing randomness in the construction of decision trees we decouple the prediction of the trees and thereby the forecast errors. As a result, when averaging the predictions some errors may cancel out. Thus RF reduces variance by creating ensembles of diverse trees, which in turn might lead to a small increase of bias. However, the decrease in variance is in most cases significant. Thus these two measures yield a better overall model.

When constructing trees we use a maximum number of features L. Furthermore, we use the Histogram-based Algorithm (Ke et al., 2017) in combination, with the Mean Squared Error loss as split criterion. In total RF has K trees in its ensemble and the prediction is computed by

[8] To find the best split inside each base learner we use the Histogram-based Algorithm from Ke et al. (2017), which is also described in the Appendix.

taking the average of the ensemble

$$g(z_{i,t}, K) = \frac{1}{K} \sum_{k=1}^{K} f_k(z_{i,t}),\ f_k \in \mathcal{F}. \tag{14}$$

For RF we choose the following hyperparameters to tune (L,K,ι), which is done by Bayesian Optimization. Moreover, we implement the GPU-accelerated version of RF in Python with the help of cuML's RandomForestRegressor class from Raschka, Patterson, and Nolet (2020).

3.4 Neural Networks

Our Feed Forward Neural Networks consist of the following parts: an "input layer" which is a row from our data set, one or multiple hidden layers that are connected and together alter the predictors' values, and an "output layer" which combines the hidden layers' information into a prediction. Gu et al. (2020) conclude that Neural Networks with fewer layers and nodes often perform better in smaller data sets. Our data set represents approximately 20% of the original data set. Thus the Neural Networks we construct have one or two layers, which are called NN1 and NN2, respectively. The number of neurons in each layer is chosen to be a tunable hyperparameter. Both networks are fully connected thus every unit in a layer gets input from every unit of the previous layer. Moreover, each unit uses the same nonlinear 'activation function' f when aggregating the unit weights from the previous layer. For the activation function we follow Gu et al. (2020) and use the rectified linear unit (ReLU) from Nair and Hinton (2010). ReLU stimulates sparsity in active neurons and enables faster evaluation of derivatives. It is defined as follows

$$\text{ReLU} = \begin{cases} 0 & \text{if } x < 0 \\ \text{x} & \text{otherwise.} \end{cases} \tag{15}$$

For notation of our Neural Network models we also follow Gu et al. (2020). Let $K^{(l)}$ correspond to the number of neurons in each layer $l = 1, 2, .., L$. The output of each neuron k in layer l is denoted as $x_k(l)$. Moreover, the vector of outputs (including a constant) of this layer is represented by $x^{(l)} = (1, x_1^{(l)}, ..., x_{K^{(l)}}^{(l)})'$. When initializing the network, the input layer corresponds to the raw predictors, and is denoted by $x^{(0)} = (1, z_1, ..., z_N)'$. Thus the recursive output formula for the network for each neuron in layer $l > 0$ becomes

$$x_k^{(l)} = \text{ReLU}(x^{(l-1)'} \theta_k^{(l-1)}) \tag{16}$$

with predicted output

$$g(z, \theta) = x^{(L-1)'} \theta^{(L-1)}. \tag{17}$$

Lastly, each hidden layer l has $K^{(l)}(1 + K^{(l-1)})$ weight parameters, and the final output layer has $1 + K^{(L-1)}$ weights.

The configuration of our neural networks includes the use of the Mean Squared Error with l_1 penalization as the loss function, the Adam solver (Kingma & Ba, 2014) in order to find the best weights, early stopping for regularization, and batch normalization (Ioffe & Szegedy, 2015) which standardizes the input layer for each mini-batch in order to make the network faster and more stable. We omitted the ensemble technique mentioned by Gu et al. (2020) due to the lack of computing resources. The Neural Networks are implemented with Tensorflow (Abadi et al., 2015) and Keras (Chollet, 2015) in Python. An overview of the hyperparameters is given in the Appendix.

3.5 Performance Evaluation

We follow the approach of Gu, Kelly, and Xiu (2020) to assess forecast performance for individual excess stock returns forecasts, out-of-sample R^2 is calculated as follows

$$R^2_{oos} = 1 - \frac{\sum_{(i,t)\in\mathcal{T}_3}(r_{i,t+1} - \hat{r}_{i,t+1})^2}{\sum_{(i,t)\in\mathcal{T}_3} r^2_{i,t+1}}, \tag{18}$$

where $\mathcal{T}_3$ corresponds to the testing subsample. This corresponds to the data of the fits, which never entered model estimation or tuning and are truly out-of-sample. Equation (18) shows that R^2_{oos} pools prediction errors across firms and time. The higher R^2_{oos} is the better the forecast performance of a model. We set the denominator of R^2_{oos} equal to the sum of squared returns *without demeaning*. By comparing predictions against a forecast value of zero instead of historical mean returns we avoid artificially lowering the bar for "good" forecasting performance (Gu, Kelly, & Xiu, 2020).

In order to compute the annual R^2_{oos} we first add one to the (predicted) returns such that we are able to take the product of those months and compute the annualized return. Thus we redefine $r_{i,t+1}$ of equation (18) as $r_{i,t+1} = r_{i,t+1,january} \times r_{i,t+1,february} \times ... \times r_{i,t+1,december} - 1$, where $r_{i,t+1,month}$ corresponds to the return of stock i at year $t + 1$ and month $month$. The predicted return $\hat{r}_{i,t+1}$ is defined in the same manner. We repeat this process for every stock i in the panel and for every year t in the testing subsample. With the redefined (predicted) returns we compute R^2_{oos} the same way as described in (18).

In order to calculate the annual R^2_{oos} for the top 1,000 companies we gather the largest companies monthly and add their respective (predicted) returns to a vector $r'_{month,t+1}$, where $month$ is the month, and $t + 1$ is the year. Furthermore, we add one to annualized (predicted) returns for the same reason as mentioned above. By definition $r_{month,t+1,i}$ is the i^{th} company in

month *month* and year $t+1$. Then to compute the annualized return we take the elementwise product of this group for every month. This operation corresponds to $r_{t+1} = r'_{january,t+1} \odot r'_{february,t+1} \odot .... \odot r'_{december,t+1} - 1'$. Then we can substitute r_{t+1} and $\hat{r}_{i,t+1}$ in equation (18) with their annualized counterparts and compute the annual R^2_{oos}. The same process is repeated for the bottom 1,000 companies, but we collect the respective 1,000 smallest companies by value instead.

We compare the out-of-sample predictive accuracy between two models by using the adjusted Diebold-Mario statistic from Gu, Kelly, and Xiu (2020), which takes the strong dependence in the cross section into account. Thus the test statistic $DM_{12} = \bar{d}_{12}/\hat{\sigma}_{\bar{d}_{12}}$ is more likely to satisfy the mild regularity conditions needed for asymptotic normality and provides appropriate p-values. It tests the predictive accuracy of method (1) against method (2)

$$d_{12,t+1} = \frac{1}{n_{3,t+1}} \sum_{i=1}^{n_3} \big((\hat{e}^{(1)}_{i,t+1})^2 - (\hat{e}^{(2)}_{i,t+1})^2 \big), \tag{19}$$

where $\hat{e}^{(1)}_{i,t+1}$ and $\hat{e}^{(2)}_{i,t+1}$ correspond to the prediction error for stock return i at time t using each method, and $n_{3,t+1}$ represents the number of stocks in the testing sample at time $t+1$. Moreover, $\bar{d_{12}}$ and $\hat{\sigma}_{\bar{d_{12}}}$ correspond to the mean and Newey-West standard error of $d_{12,t}$ over the testing sample, respectively.

3.6 Interpreting Results

Gu, Kelly, and Xiu (2020) have shown that neural networks and tree ensembles have the best forecast performance. However, the parameters of these methods do not provide the same straightforward interpretation as linear models or decision trees. We combine the variable importance research of Gu et al. (2020) with SHapley Additive exPlanations in order to estimate feature importance regardless of the model's complexity.

3.6.1 Permutation Feature Importance

Gu et al. (2020) perform Permutation Feature Importance (PFI) to estimate variable importance. PFI measures the change in panel predictive R^2 of a model after setting all values of variable j to zero, while keeping the rest fixed. This implies that every stock interaction with variable j is also set to zero.

We extend the research of variable importance by adding SHapley Additive exPlanations in order to estimate variable importance.

3.6.2 SHapley Additive exPlanations

Molnar (2019) states that game theory can be used to estimate feature importance in Machine Learning methods. The game's payout is the prediction and the players are the feature values used for this prediction. Then Shapley values, which is a method from conditional game theory (Shapley, 2016), can tell us how to allocate the payout fairly among the features. Shapley values have desirable properties such as efficiency, symmetry, additivity, and dummy. These properties are defined in Appendix A.1.

SHapley Additive exPlanations (SHAP) by S. Lundberg and Lee (2017) is a method to explain individual predictions and is based on Shapley values. Thereby inheriting its properties. Whereas popular feature attribution methods are inconsistent, meaning that they can attribute a lower importance to features when in reality the impact of those features actually increase, SHAP provides unique consistent and locally accurate attribution values (S. M. Lundberg, Erion, & Lee, 2018). Thus, in addition to the properties from Shapley values SHAP has these 3 desirable properties: local accuracy, missingness, and consistency. These are described in Appendix A.2. The theoretical implication of the consistency property is that the SHAP value changes depending on whether the marginal contribution of a feature value also changes. This also applies if the marginal contribution stays the same. Because of this property SHAP also satisfies other Shapley properties such as Linearity, Dummy and Symmetry. This is shown in the Appendix of S. Lundberg and Lee (2017).

SHAP provides an efficient way to estimate Shapley values for tree models such as XGBoost. SHAP has a different approach to the Shapley value expectation since they represent it as an additive feature attribution method $h(q') = \phi_0 + \sum_{j=1}^{N} \phi_j q'_j$, where h denotes the explanation model, $q' \in \{0,1\}^N$ the coalition vector, M corresponds to the maximum coalition size, and $\phi_j \in \mathbb{R}$ is the j^{th} feature contribution. The j^{th} element of the coalition vector q' equals one if feature j is part of the coalition and zero otherwise. S. M. Lundberg et al. (2018) state that one must define a mapping h_z that maps between a vector q' that represents which features are missing and the original function's domain. This mapping enables us to evaluate $f(h_z(q'))$ and thereby measure the effect of observing a feature or not (by setting $q'_i = 1$ or $q'_i = 0$). We follow the notation of S. M. Lundberg et al. (2018) and define $f_z(S) = f(h_z(q')) = E[f(z)|z_S]$, where S corresponds to the set of non-zero indices in q', and $E[f(z)|z_S]$ is the expected value of the function conditional on the subset S of the predictors. Shapley values are combined with the conditional expectations from SHAP in order to compute the j^{th} feature contribution ϕ_j

$$\phi_j = \sum_{S \subseteq \{z_1,\ldots,z_N\} \setminus \{z_j\}} \frac{|S|!\,(N - |S| - 1)!}{N!} \left[f_z\left(S \cup \{z_j\}\right) - f_z(S) \right], \tag{20}$$

where N is the number of input features.

The properties of the above defined SHAP values are desirable, but two problems hinder us in using them. The first is the challenge of estimating $E[f(z)|z_S]$ efficiently, and the second problem is the exponential complexity of Equation (20). That is why we use the fast SHAP estimation method for trees and ensembles of trees called Tree SHAP from S. M. Lundberg et al. (2018). This algorithm reduces the complexity of computing the exact SHAP values from $O(TL2^N)$ to $O(TLD^2)$, where T corresponds to the number of trees, L is the maximum number of leaves in any tree, N is the number of predictors, and D is the maximum depth of any tree. The exact algorithm is given in the Appendix, since it is a page long. The intuition behind the algorithm is as follows, the tree-structure is leveraged in order to compute the conditional expectation recursively and at the same time the subset of relevant features in that branch is monitored. This monitoring allows us to compute the weights of the contribution of each subset. One disadvantage of using the conditional expectation instead of the marginal marginal expectation is that features who have no contribution to the prediction can get a feature value other than zero (Janzing, Minorics, & Blöbaum, 2020). The non-zero feature value occurs when the feature is correlated with a feature that actually contributes to the prediction. Thus, SHAP discovers seemingly important features and thereafter we have to differentiate the truly contributing predictors from the correlated ones by using our knowledge of the dataset.

4 Results

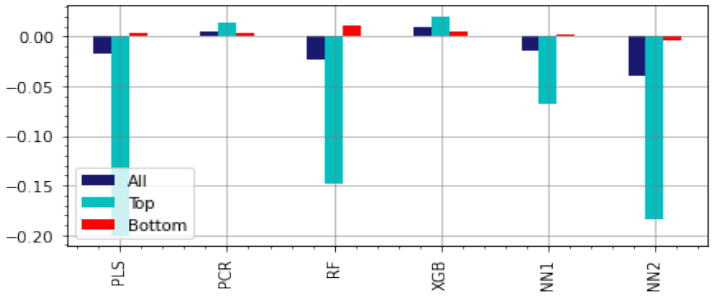
The results section is divided into two parts. The first part is the result of reproducing the research of Gu et al. (2020) on a smaller dataset that is described in section 2. It consists of a comparison between the models of their annual and monthly R^2_{oos}. Thereafter, part 2 starts with variable importance by using the Permutation Feature Importance from section 3.6.1 and continues with an overview of which variables are the most important across the models and within each model. Next, we compute SHAP values for the entire panel, the top 1,000 and bottom 1,000 stocks in order to interpret the predictions. Then, we zoom in on 2010, since that is our best year according to monthly R^2_{oos}, and provide figures to interpret the best and worst predictions from XGB.

4.1 Empirical asset pricing via machine learning

Gu et al. (2020) displayed that almost all models, listed in table 1, have a positive monthly R^2_{oos} for every group. However, table 1 shows that only PCR and XGB have a positive R^2_{oos} for all groups. This suggests that we can not replicate the same performance as Gu et al. (2020) by

Table 1: Monthly Out-of-sample Stock-level Prediction Performance (Percentage R^2_{oos})

	PLS	PCR	RF	XGB	NN1	NN2
All	-0.017	0.005	-0.023	0.010	-0.014	-0.040
Top	-0.200	0.013	-0.149	0.020	-0.069	-0.185
Bottom	0.003	0.004	0.011	0.005	0.002	-0.005



Note: In this table, we report monthly R^2_{oos} for the entire panel of stocks using Partial Least Squares (PLS), Principal Components Regression (PCR), Random Forest (RF), XGBoost (XGB), and Neural Networks with 1 to 2 layers NN1-NN2. We also report these R^2_{oos} for samples that consists only of the top 1,000 or bottom 1,000 stocks by market value. The histogram below the table provides a visual comparison of the R^2. Lastly, we omitted the results of neural networks with more than 2 layers since their R^2 values were very negative and needed a logarithmic scale to get them to fit in the histogram with the other models.

solely relying on variations of momentum, liquidity or volatility. Moreover, the discrepancy in performance can be caused by the size of our data set, which is one fifth the size of theirs. Note that we use 180 predictors, whereas they use 920. Especially Neural Networks, which were the winners in their research, suffer from the decrease in sample size. Friedman, Hastie, Tibshirani, et al. (2001) state that model complexity should only grow with parameter size, and neural networks (deep learning) are fairly complex, thereby implying that it will not operate optimally for smaller datasets. This appears to be true for our case.

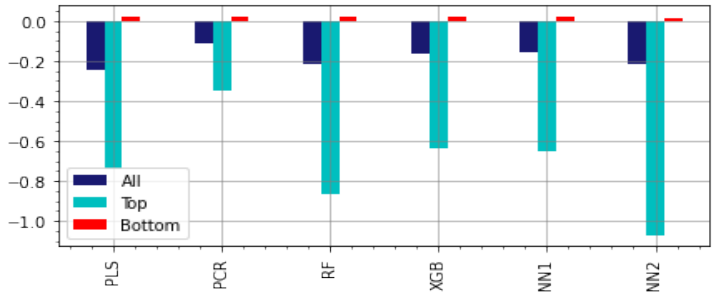
XGB has the best monthly R^2_{oos}, which suggests that it captures significant non linear relations that linear methods such as PLS can not do. Moreover, it also achieves better results than RF. This is due to its sequential approach of building decision trees, which repetitively leverages the patterns in residuals, whereas RF builds independent trees and aggregates the predictions. Table 1 indicates that all models except the NN2 have a positive R^2_{oos} when predicting the monthly return for the bottom 1,000 companies, but it can be debated whether their positive R^2_{oos} is significantly different from zero.

Table 2 shows that all models only have a positive annual R^2_{oos} for the bottom 1000 companies, which can be implied from the positive monthly R^2_{oos}. Even the NN2 model, which does not have a positive monthly R^2_{oos}, has a positive annual R^2_{oos}. XGB has the best annual R^2_{oos} when predicting returns for the bottom 1,000 stocks.

Gu et al. (2020) state that positive coefficients in the column indicate whether the row model is significantly worse than the column model, but this is not the proper use of the Diebold Mariano

Table 2: Annual Out-of-sample Stock-level Prediction Performance (Percentage R^2_{oos})

	PLS	PCR	RF	XGB	NN1	NN2
All	-0.244	-0.111	-0.218	-0.164	-0.156	-0.217
Top	-0.739	-0.352	-0.863	-0.640	-0.648	-1.072
Bottom	0.023	0.021	0.020	0.024	0.023	0.014



Note: Annual return forecasting R^2_{oos} (see table 1).

Table 3: Comparison of Monthly Out-of-Sample Prediction using Diebold-Mariano Tests

	PCR	RF	XGB	NN1	NN2
PLS	**6.19**	**-5.58**	**8.42**	0.66	**-6.39**
PCR		**-14.85**	**4.34**	**-10.38**	**-21.73**
RF			**16.21**	**7.97**	-0.50
XGB				**-12.78**	**-22.05**
NN1					**-11.10**

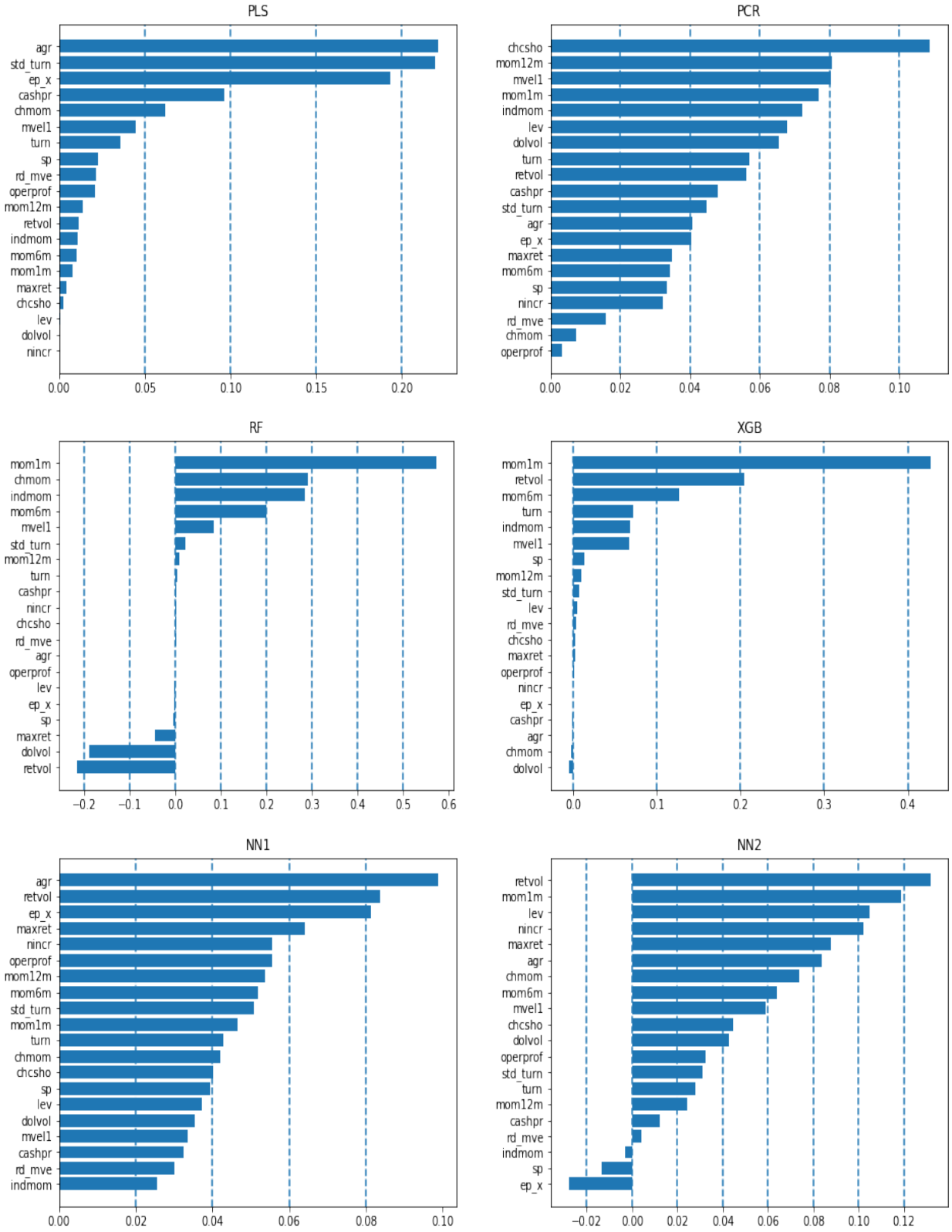
Note: This table presents the pairwise Diebold-Mariano test statistics comparing the out-of-sample stock-level forecast performance among 6 models. Bold font indicates that the difference is significant at a 5% level or better for individual tests.

statistic. It can only be used to determine whether two models have the same predictive accuracy. Thus, table 3 shows that the predictive accuracy of XGB is significantly different from the rest of the models.

4.2 Interpreting machine learning

Table 4 presents the variable importance structure of each method. Tree ensembles have a dominant set of important predictors, whereas neural networks and PCR have a more diverse set of important variables. This is due to limiting the depth of every tree to 6, which forces it to use a small set of variables and reduces overfitting. However, For RF and NN2 the figures in table 4 show that omitting certain variables actually increases the R^2_{oos}. This suggests that these models tend to overfit. By omitting these variables, we get better generalized predictions and therefore an increase in R^2_{oos}. Moreover, table 4 indicates that variables which include variations on momentum are the most important for tree ensemble methods. This is in contrast with the neural networks, where variations of momentum are not as important and importance is shared

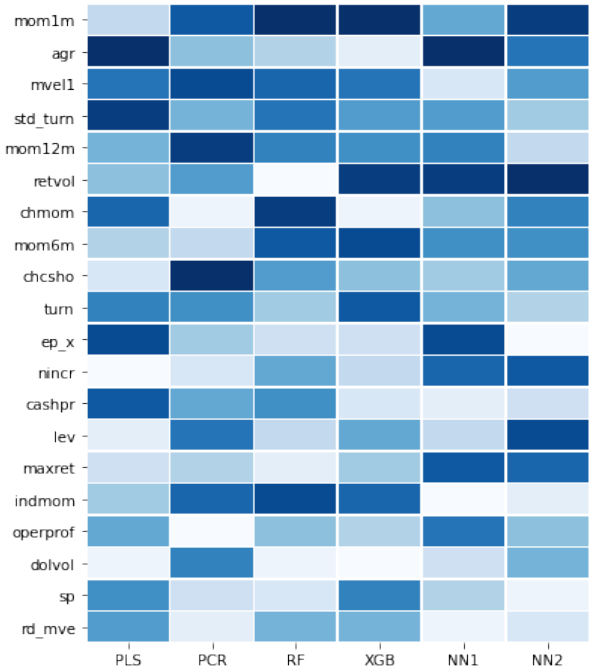
Table 4: Variable Importance By Model



Note: Variable importance of all firm characteristics in each model according to the permutation method described in section 3.6.1. Variable importance is average over all testing samples. Variable importances within each model are normalized to sum to one.

more evenly across predictors.

Figure 1: Characteristic Importance



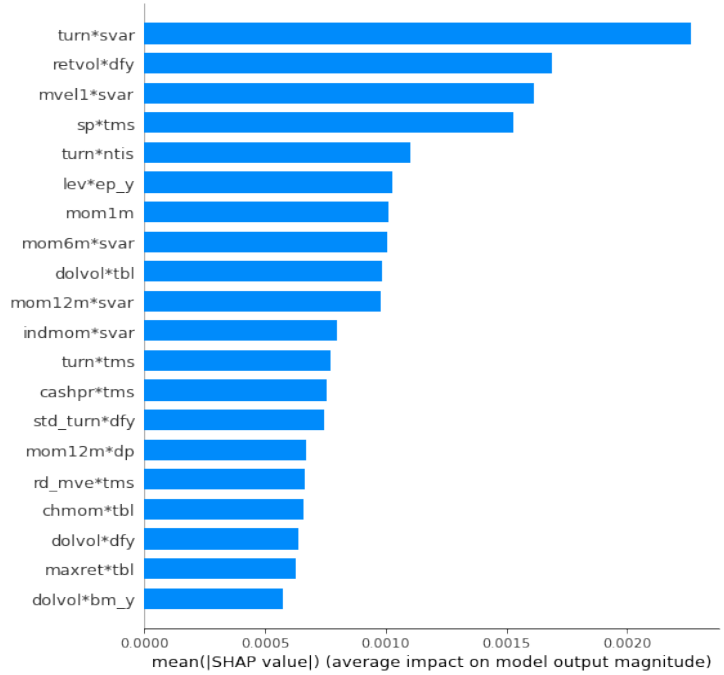
Note: This figure provides a graphical illustration of variable importance across and within models. The variables on the y-axis are sorted by Permutation Feature Importance and the darker the color of a cell in a column the more important the variable is for the model corresponding to that column.

If we combine table 4 and figure 1 we see that the most important variables are those that concern momentum and liquidity, which is in accordance with the research of Gu et al. (2020). Moreover, the property that the most important variables, across each model, share is that they are updated monthly. This is also reflected in the results of Gu et al. (2020).

Figure 2 indicates that the top 2 features are $turn{*}svar$ and $retvol{*}dfy$, which correspond to the product of *Share turnover* $\times$ *Stock variance*, and *Return volatility* $\times$ *Default yield spread*. These two products contain predictors that indicate stock liquidity (*turn* and *retvol*). Furthermore, figure 2 shows that two other most important variables $mvel1(size) \times svar$ and $sp \times tms =$ *Sales to price* $\times$ *term spread* are related to the firm's fundamentals.

Figure 3 points out that the 2 most important variables for the top group are $turn * svar$ and $mvel1 * svar$, which includes interactions between stock fundamentals and volatility. For the bottom group the 2 most important variables are $retvol{*}dfy$ and $mom1m$(1 month momentum), which are variations on momentum and volatility. This suggests that momentum contributes more to the prediction for smaller stocks than for larger stocks, which can be explained by the fact that smaller stocks are more volatile and in turn momentum plays more of an important role. The results also show that for larger stocks firm characteristics are more important and this is supported by decades of finance literature.

Figure 2: Variable Importance for XGBoost using SHAP

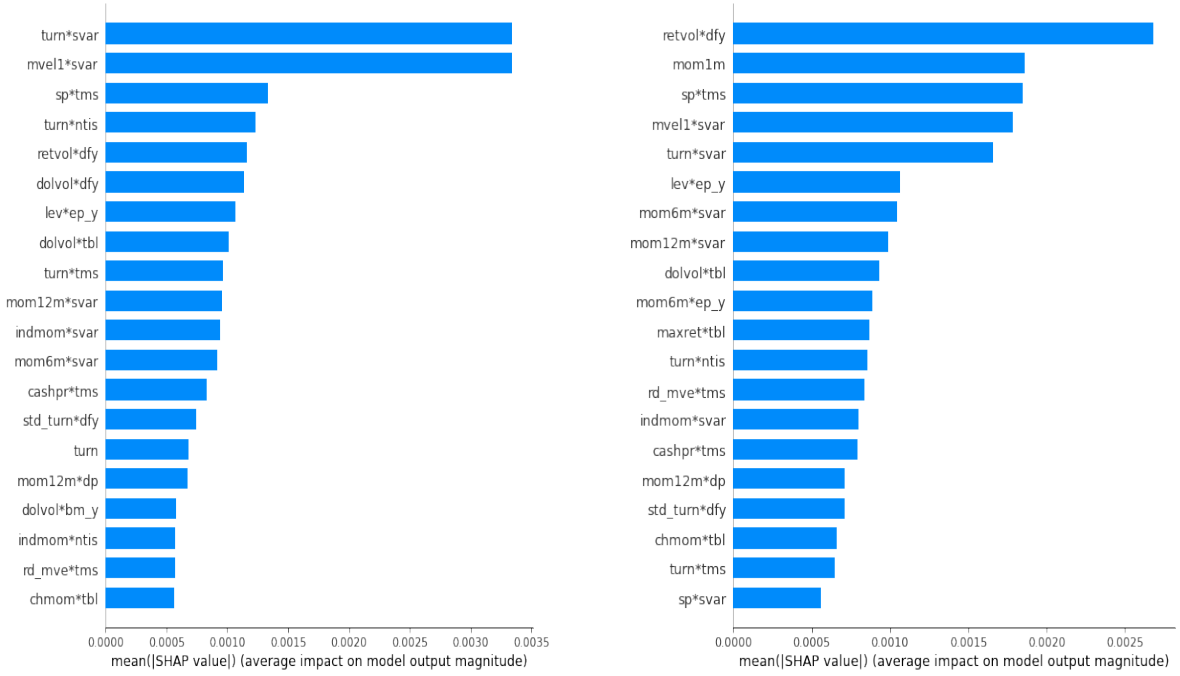


Note: SHAP values explain the difference between the average prediction and the prediction. In this plot we present the absolute mean SHAP values of the top 20 features by value for the entire panel.

Figure 3: Variable Importance for XGB using SHAP when predicting Top and Bottom 1,000 stocks by size.

(a) Top

(b) Bottom

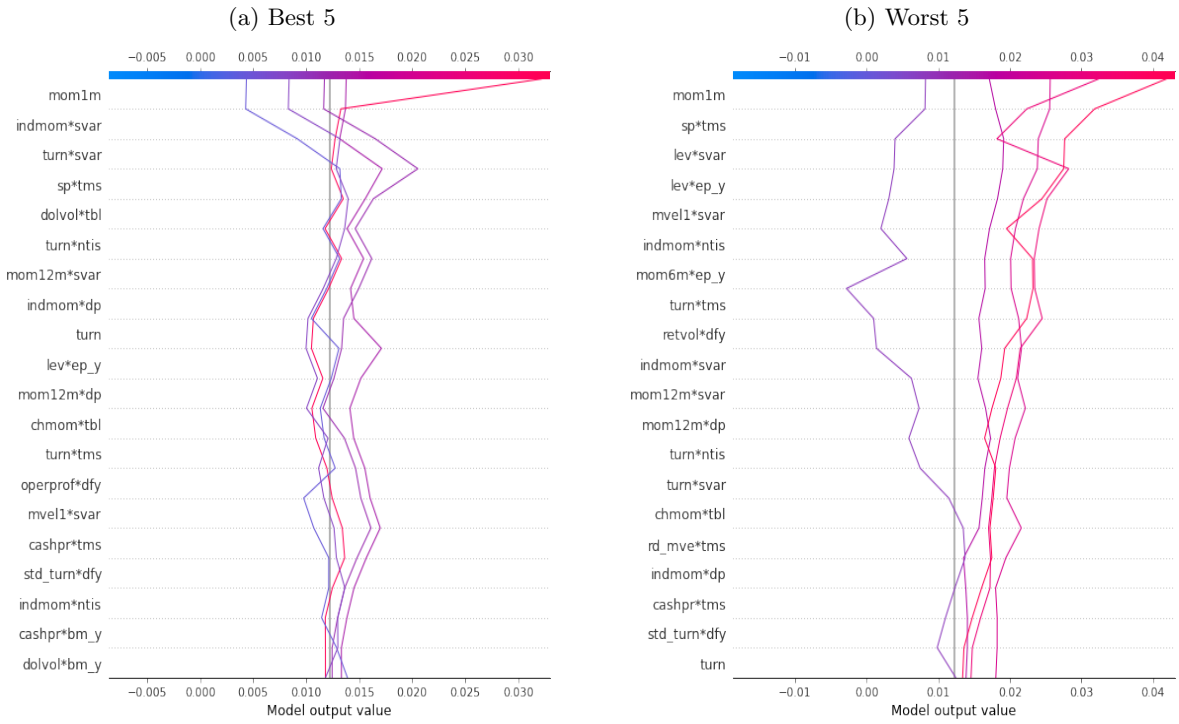


Note: See figure 2. We report SHAP values for samples that consist of top 1,000 or bottom 1,000 stocks by size.

It is inherently more valuable to acquire insights about which variables are important in a good working model than in a bad one. Thus, for the following plots we zoom in on 2010,

since XGBoost has the best monthly R^2_{oos} for that period. Decision plot 4.a displays that

Figure 4: Decision plot of the best and worst 5 predictions of monthly returns in 2010



Note: Decision plot shows how each prediction (plotted line) arrives to its value. At the top of the plot, each line strikes the x-axis at its corresponding observation's predicted value. Moving from the bottom to the top, SHAP values are added to the model's base value (mean of predicted returns).

the best predictions stay centered around 0, and only start moving when entering the area of the most important variables. This indicates a relatively large contribution of the most important variables to the prediction. Whereas decision plot 4.b shows that the predictions deviate relatively more from 0. Furthermore, the purple plotted lines do not have any direction change when they visit the most important variables. This suggests that the most important predictors have relatively less contribution to the prediction.

5 Conclusion

Our results show that XGBoost and PCR are the only models that achieve positive monthly R^2_{oos} for all groups. This is in contrast with the research by Gu et al. (2020), where almost all models listed in this research have positive R^2_{oos}. In addition, XGB and PCR have a lot smaller values for R^2_{oos}. The difference in performance can be explained by our data set, which is one fifth the size of theirs. This suggests that merely including the most important variables in the research of Gu et al. (2020) is not sufficient for achieving the same performance, which confirms the notion that you need a relatively large dataset for deep learning success.

Our variable importance research suggests that the most important variables are variations

on momentum and volatility, which is in accordance with the research of Gu et al. (2020). Moreover, if we take a closer look at our best performing model XGB we conclude that features, which incorporate variations on a stock's fundamentals, are relatively more important for larger stocks than for smaller stocks. Whereas, momentum is relatively more important for smaller stocks than for larger stocks. This is due to the larger volatility of smaller stocks, which in turn makes momentum a more important signal.

To conclude, our research implemented the latest model agnostic tools on machine learning algorithms and successfully inferred economic insights regardless of the models' complexity. Further research should implement the same model agnostic tools on neural networks, which leverage the full dataset from Gu et al. (2020), to discover more meaningful relations of risk premiums.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Retrieved from `https://www.tensorflow.org/` (Software available from tensorflow.org)

Angrish, R., & Garg, D. (2011). Efficient string sorting algorithms: Cache-aware and cache-oblivious. *International Journal of Soft Computing and Engineering (IJSCE)*, *1*(2), 12–16.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (nips 2011)* (Vol. 24).

Bergstra, J., Yamins, D., & Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning* (pp. 115–123).

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

Chollet, F. (2015). *Keras.* `https://keras.io`.

Friedman, J., Hastie, T., Tibshirani, R., et al. (2001). *The elements of statistical learning* (Vol. 1) (No. 10). Springer series in statistics New York.

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

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456).

Janzing, D., Minorics, L., & Blöbaum, P. (2020). Feature relevance quantification in explainable ai: A causal problem. In *International conference on artificial intelligence and statistics* (pp. 2907–2916).

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, *30*, 3146–3154.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, *18*(1), 6765–6816.

Lundberg, S., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*.

Lundberg, S. M., Erion, G. G., & Lee, S.-I. (2018). Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*.

Molnar, C. (2019). *Interpretable machine learning.* (`https://christophm.github.io/interpretable-ml-book/`)

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml.*

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*.

Shapley, L. S. (1997). A value for n-person games. *Classics in game theory*, *69*.

Shapley, L. S. (2016). *17. a value for n-person games.* Princeton University Press.

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.

Appendices

A Interpretation Methods

A.1 Shapley values

As Molnar (2019) states game theory can be used in order to estimate feature importance in Machine Learning methods. The game's payout is the prediction and the players are the feature values used for this prediction. Then Shapley values, which is a method from conditional game theory (Shapley, 2016), tells us how to allocate the payout fairly among the features.

We follow these key terms from Molnar (2019) to explain Shapley values. The feature value corresponds to the numerical or categorical value of a feature and sample, the Shapley value is the feature contribution to the prediction, and the value function corresponds to the payout function for coalition of players.

Shapley values are computed by measuring the contribution it has to the prediction over all possible coalitions of feature values, and thereafter aggregate the weighted contributions. We calculate the Shapley value for feature j as follows:

$$\phi_j(val) = \sum_{S \subseteq \{z_1,\ldots,z_N\} \setminus \{z_j\}} \frac{|S|!\,(N-|S|-1)!}{N!} \left(val\left(S \cup \{z_j\}\right) - val(S)\right), \tag{21}$$

where z_j represents the vector of feature j, S corresponds to the subset of feature values excluding j, and N the number of features. The function $val_z(S)$ outputs the prediction for feature values in set S, which are marginalized over the feature values that are not in set S

$$val_z(S) = \int g(z_1, \ldots, z_p) d\mathbb{P}_{z \notin S} - E_Z(g(Z)). \tag{22}$$

One of main the advantages of Shapley values are its desirable properties. It is the only attribution method which satisfies efficiency, symmetry, additivity, and dummy. The combination of these properties makes the Shapley value per definition a fair payout (Molnar, 2019).

Efficiency requires that feature contributions sum up to the difference between the average and the prediction for z

$$\sum_{j=1}^{N} \phi_j = g(z) - E_Z(g(Z)). \tag{23}$$

Symmetry requires that feature values i and j have the same value if and only if they have the same contribution to all possible coaltions

$$val(S \cup \{z_i\}) = val(S \cup \{z_j\}). \tag{24}$$

Additivity ensures that in a game of combined payouts val and val^+ their respective values

can be added

$$\phi_j + \phi_j^+. \tag{25}$$

To clarify the value of additivy we provide an example. Given a Random Forest model, we can compute the Shapley feature values for each individual tree and thereafter average them to get the feature values for the Random Forest.

Dummy ensures that a feature j, which does not contribute in all possible coalitions of feature values, has a Shapley value of 0. Thus

$$\text{if } val(S \cup \{x_j\}) = val(S),\ \forall S \subseteq \{x_1, \ldots, x_p\} \tag{26}$$
$$\text{then } \phi_j = 0.$$

For computation of the exact Shapley values we need to evaluate all possible coalitions sets with and without the feature value j. As you can expect this is computationally very expensive. Hence, in the following subsection we describe SHapley Additive exPlanations (SHAP) by S. Lundberg and Lee (2017), which is a faster approach to estimating Shapley values, and it provides many global interpretation methods which are based on aggregated Shapley Values. Since these methods are based on Shapley values they possess the same properties.

A.2 SHapley Additive exPlanations

Local Accuracy

$$f(z) = h(q') = \phi_0 + \sum_{j=1}^{N} \phi_j q'_j \tag{27}$$

To derive the efficiency property from Local Accuracy, we set all elements of q' to one and define $\phi_0 = E_Z(g(Z))$. This results in

$$f(z) = \phi_0 + \sum_{j=1}^{N} \phi_j q'_j = E_Z(g(Z)) + \sum_{j=1}^{N} \phi_j. \tag{28}$$

Missingness If $q'_j = 0 \Rightarrow \phi_j = 0$. Thus missingness enforces that an absent feature does not have any contribution.

Consistency We follow the definition from Molnar (2019). Let $q'_{\setminus j}$ denote that $q'_j = 0$ and $f_z(q') = f(h_z(q'))$. For any two models f and f that satisfy

$$f'_z(q') - f'_z(q'_{\setminus j}) \geq f_x(q') - f_z(q'_{\setminus j}), \tag{29}$$

for all inputs $q' \in \{0,1\}^N$, then

$$\phi_j(f', z) \geq \phi_j(f, z). \tag{30}$$

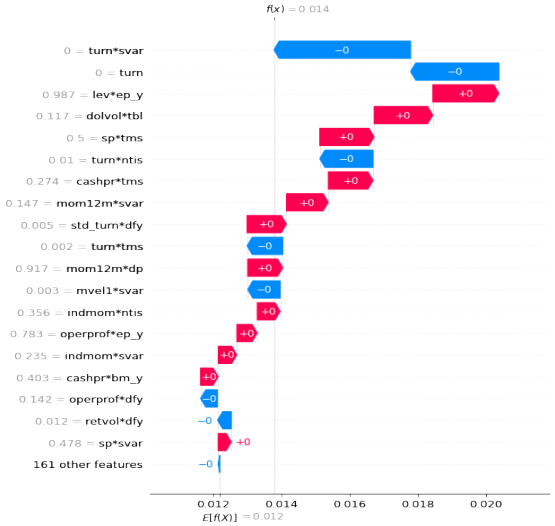
B Data

C Results

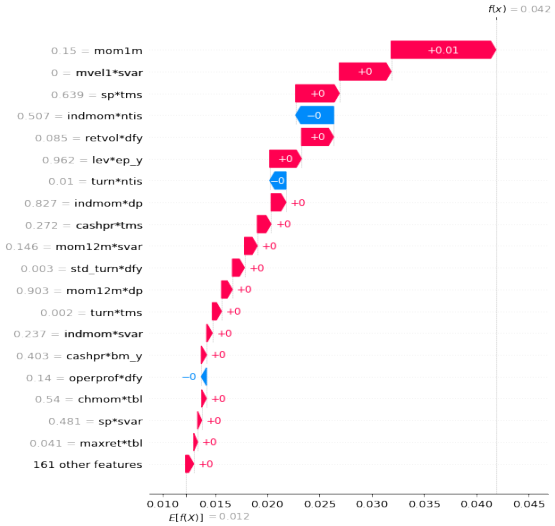
C.1 Waterfall Plot

Figure 5: Waterfall plot of the best and worst prediction by monthly R^2_{oos} in 2010

(a) Best



(b) Worst



Note: The bottom of the waterfall represents the expected value of the model output, and then every row shows how the contribution, which is positive (red) or negative (blue), moves the value from the expected model output over the background data set (conditional expectation) to the prediction of this model (top).

The waterfall plot in figure 5 shows that $turn * svar$ has a relatively large negative contribution to the predicted return, this indicates that the higher the interaction between liquidty and volatility the lower the expected return.

D Parameter Hyperoptimization

D.1 Hyperparameters

D.2 Bayesian Optimization

As noted in (Bergstra et al., 2013) Bayesian Optimization (BO) aims to find the global minimum of a function with the help of Bayes Theorem. It works by constructing a surrogate function, which corresponds to a probabilistic model of the objective function, and that is searched with an acquisition function before samples are to be evaluated with the real objective function. In order to further describe how BO works we define some key terms used in global function optimization:

- **Samples**: Are examples from the domain and represented by a vector
- **Search Space**: Set through which algorithm searches

Table 5: Details of the Characteristics

No.	Acronym	Firm characteristic	Paper's author(s)	Year	Journal	Data Source	Frequency
1	agr	Asset growth	Cooper, Gulen & Schill	2008	JF	Compustat	Annual
2	cashpr	Cash productivity	Chandrashekar & Rao	2009	WP	Compustat	Annual
3	chcsho	Changes in shares outstanding	Pontiff & Woodgate	2008	JF	Compustat	Annual
4	chmom	Change in 6-month momentum	Gettleman & Marks	2006	WP	CRSP	Monthly
5	dolvol	Dollar trading volume	Chordia, Subrahmanyam & Anshuman	2007	JFE	CRSP	Monthly
6	ep	Earnings to price	Basu	1977	JF	Compustat	Annual
7	indmom	Industry momentum	Moskowitz & Grinblatt	1999	JF	CRSP	Monthly
8	lev	Leverage	Bhandari	1988	JF	Compustat	Annual
9	maxret	Maximum daily return	Bali, Cakici & Whitelaw	2011	JFE	CRSP	Monthly
10	mom12m	12-month momentum	Jagadeesh	1990	JF	CRSP	Monthly
11	mom1m	1-month momentum	Jagadeesh & Titman	1993	JF	CRSP	Monthly
12	mom6m	6-month momentum	Jagadeesh & Titman	1993	JF	CRSP	Monthly
13	mvel1	Size	Banz	2005	RAS	Compustat	Quarterly
14	nincr	Number of earning increases	Barth, Elliot & Finn	1999	JAR	Compustat	Quarterly
15	operprof	Operating profitability	Fama & French	2015	JFE	Compustat	Annual
16	rd_mve	R&D to market capitalization	Guo, Lev & Shi	2006	JBFA	Compustat	Annual
17	retvol	Return volatility	Ang, Hodrick, Xing & Zhang	2006	JF	CRSP	Monthly
18	sp	Sales to price	Barbee, Mukherji & Raines	1996	FAJ	Compustat	Annual
19	std_turn	Volatility of liquidity (share turnover)	Chordia, Subrahmanyarn & Anshuman	2001	JFE	CRSP	Monthly
20	turn	Share turnover	Datar, Naik & Radcliffe	1998	JFM	CRSP	Monthly

Table 6: Hyperparameters For All Methods

PLS	PCR	RF	XGBoost	NN1-NN2
K	K	Depth $= 1 \sim 6$	Depth $= 1 \sim 6$	l_1 penalty
		# Trees $= 100$	# Trees $K \in (1000, 10000)$	$\lambda_1 \in (e^{-5}, e^{-3})$
		# Features in each split	Learning rate η	Learning rate LR
		$\in \{3, 5, 10, 20, 50, ...\}$	$\eta \in (0.1, 0.3)$	$\in (e^{-5}, e^{-2})$
			Depth penalty	Batch Size $= 10000$
			$\gamma \in (1, 9)$	Epochs $= 50$
			l_1 penalty	Patience $= 5$
			$\alpha \in (40, 180)$	Adam Para. $=$ Default
			l_2 penalty	Neurons per layer
			$\lambda \in (0, 1)$	$s \in (5, 150)$
			Colsample by three	
			$\vartheta \in (0.5, 1)$	
			min child weight	
			$\zeta \in (0, 10)$	

- **Objective Function**: Function which has input a sample and as output a cost
- **Cost**: Quantitative score for a sample computed with the Objective Function.

First we provide Bayes Theorem such that we can understand how it applies to Hyperparameter Optimization. Let A and B denote events then Bayes Theorem allows us to compute the conditional probability of event $P(A|B)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{31}$$

For BO we are not interested in calculating the conditional probability but instead we want to optimize a proportional quantity. To get this proportional quantity we remove the normalizing value of $P(B)$, which transforms Equation (31) into the following equation: $P(A|B) = P(B|A)P(A)$. This transformed equation is in analogy with the equation:

$$posterior = likelihood \times prior. \tag{32}$$

Given this framework we can use samples from the domain and their respective cost in order to quantify the beliefs of an unknown objective function. In other words we can use Bayes Theorem in combination with samples and their cost to approximate an computationally expensive objective function.

Thus we create specific samples $x_1, x_2, ..., x_n$ and compute their cost using the real objective function $f(.)$. Then we group the samples and their outcome as pairs and consider that as our data $D = \big\{\{x_1, f(x_2)\}, \{x_2, f(x_2)\}, ..., \{x_n, f(x_n)\}\big\}$, which we use to define the prior. The probability of observing the data D given the function f is denoted by $P(D|f)$ and is in analogy with the likelihood in equation (32). Since D is subject to change when more observations are

added, the likelihood will also change. The posterior is denoted by $P(f|D)$ and corresponds to an approximation of the objective function. Thus $P(f|D)$ enables us to evaluate candidate samples at a relatively low cost.

In BO $P(f|D)$ is referred to as the *surrogate* function and it can be created in multiple ways. We consider a regression model where the input is represented by D and the output by the score. The most used model used for this regression is a Gaussian Process (GP), which is a model that builds a joint probability distribution over the variables based on the assumption that they are distributed in a multivariate Gaussian way. GP inherently ensures that estimates come from a distribution with a mean and standard deviation, which is useful for later use in the *acquisition* function.

Thus we use the surrogate function to test a couple of samples in the domain. Thereafter a search algorithm algorithm determines which area of the sample space is most likely to pay off and the acquisition function is used to decide whether a given sample is worthy of evaluating with the real objective function. The acquisition function can use the surrogate score to estimate the worthiness of a sample but given that the surrogate function summarizes the conditional probability $p(f|D)$ in a probabilistic way we can also use the probabilistic information from this model. Here again there are multiple methods we use the method of Expected Improvement (EI).

To summarize the algorithm of BO we provide the following flow. First pick a sample that results from optimizing the acquisition function. After a sample is picked it is evaluated and added to D, thereafter the posterior and surrogate function is updated. This process is repeated until extrema of the objective function are discovered, the resources are depleted, or a good enough sample is found.

The search and surrogate function we use is from Bergstra, Bardenet, Bengio, and Kégl (2011) and is called the Tree-structured Parzen Estimator Approach (TPE). This algorithm differs from GP since it estimates $P(D|f)$ and $P(f)$ instead of directly estimating $P(f|D)$. We implemented it by using the Hyperopt Python package from Bergstra et al. (2013).

Tree SHAP

Algorithm 2 Tree SHAP

procedure TS(x, $tree = \{v, a, b, t, r, d\}$)
  ϕ = array of $len(x)$ zeros
  procedure RECURSE(j, m, p_z, p_o, p_i)
    m = EXTEND(m, p_z, p_o, p_i)
    if $v_j \neq internal$ **then**
      for $i \leftarrow 2$ to $len(m)$ **do**
        $w = sum(\text{UNWIND}(m, i).w)$
        $\phi_{m_i} = \phi_{m_i} + w(m_i.o - m_i.z)v_j$
      end for
    else
      $h, c = x_{d_j} \leq t_j \ ? \ (a_j, b_j) : (b_j, a_j)$
      $i_z = i_o = 1$
      k = FINDFIRST($m.d, d_j$)
      if $k \neq$ nothing **then**
        $i_z, i_o = (m_k.z, m_k.o)$
        m = UNWIND(m, k)
      end if
      RECURSE(h, m, $i_z r_h / r_j$, i_o, d_j)
      RECURSE(c, m, $i_z r_c / r_j$, 0, d_j)
    end if
  end procedure
  procedure EXTEND(m, p_z, p_o, p_i)
    $l = len(m)$
    $m = copy(m)$
    $m_{l+1}.(d, z, o, w) = (p_i, p_z, p_o, l = 0 \ ? \ 1 : 0)$
    for $i \leftarrow l - 1$ to 1 **do**
      $m_{i+1}.w = m_{i+1}.w + p_o m_i.w(i/l)$
      $m_i.w = p_z m_i.w[(l - i)/l]$
    end for
    return m
  end procedure
  procedure UNWIND(m, i)
    $l = len(m)$
    $n = m_l.w$
    $m = copy(m_{1 \ldots l-1})$
    for $j \leftarrow l - 1$ to 1 **do**
      if $m_i.o \neq 0$ **then**
        $t = m_j.w$
        $m_j.w = n \cdot l/(j \cdot m_i.o)$
        $n = t - m_j.w \cdot m_i.z((l - j)/l)$
      else
        $m_j.w = (m_j.w \cdot l)/(m_i.z(l - j))$
      end if
    end for
    for $j \leftarrow i$ to $l - 1$ **do**
      $m_j.(d, z, o) = m_{j+1}.(d, z, o)$
    end for
    return m
  end procedure
  RECURSE(1, [], 1, 1, 0)
  return ϕ
end procedure

T is the number of trees and M is the number of features.

D.3 Hyperband Optimization

Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.

input : R, η (default $\eta = 3$)
initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$

for $s \in \{s_{\max}, s_{\max} - 1, \ldots, 0\}$ **do**
  $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$, $r = R\eta^{-s}$
  // begin SUCCESSIVEHALVING with (n, r) inner loop
  T =get_hyperparameter_configuration(n)
  for $i \in \{0, \ldots, s\}$ **do**
    $n_i = \lfloor n\eta^{-i} \rfloor$
    $r_i = r\eta^i$
    $L = \{$run_then_return_val_loss$(t, r_i) : t \in T\}$
    T =top_k$(T, L, \lfloor n_i/\eta \rfloor)$
  end
end
return *Configuration with the smallest intermediate loss seen so far.*

E Algorithms

E.1 Histogram Algorithm

Algorithm 1: Histogram-based Algorithm

Input: I: training data, d: max depth
Input: m: feature dimension
$nodeSet \leftarrow \{0\}$ ▷ tree nodes in current level
$rowSet \leftarrow \{\{0, 1, 2, ...\}\}$ ▷ data indices in tree nodes
for $i = 1$ ***to*** d **do**
  for $node$ ***in*** $nodeSet$ **do**
    usedRows $\leftarrow rowSet[node]$
    for $k = 1$ ***to*** m **do**
      $H \leftarrow$ new Histogram()
      ▷ Build histogram
      for j ***in*** *usedRows* **do**
        bin $\leftarrow I$.f[k][j].bin
        H[bin].y $\leftarrow H$[bin].y + I.y[j]
        H[bin].n $\leftarrow H$[bin].n + 1
      Find the best split on histogram H.
      ...
  Update $rowSet$ and $nodeSet$ according to the best split points.
  ...

E.2 XGBoost Algorithm

Model input: dataset D := $\{(x_1, y_1), ..., (x_N, y_N), y_i \in \mathbb{R}\}$

Model output: Final regressor $f_{m(x)}$

Steps:

(1) Initialization $f_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} Loss(y_i, \gamma)$

(2) **for** $m = 1$ **to** M **do**

Compute the gradient:

$$g_i = \frac{\delta Loss(y_i, f_{m-1}(x_i))}{f_{m-1}(x)},\ i = 1, 2, ..., N$$

Compute the second derivative:

$$g_i = \frac{\delta^2 Loss(y_i, f_{m-1}(x_i))}{f_{m-1}(x)},\ i = 1, 2, ..., N$$

Fit a new decision tree by minimizing the loss function with regularization term:

New tree: $G_m(x_i) = \sum_{j=1}^{J} b_j * \mathbb{I}(x_i \in R_j)$

Find the best tree structure : $\{R_j\}_{j=1}^{J} = \underset{\{R_j\}_{j=1}^{J}}{\operatorname{argmin}} \sum_{j=1}^{J} (G_j * b_j + \frac{1}{2}(H_j + \lambda) * b_j^2)) + \gamma * J$,

where $G_j = \sum_{x_i \in R_j},\ H_j = \sum_{x_i \in R_j} h_i$

Best predicted value of tree node: $b_j^* = -\frac{G_j}{H_j + \lambda}$

Minimal final loss $Loss(y, f_m(x)) = \sum_{j=1}^{J} (-\frac{1}{2} * \frac{G_j^2}{H_j + \lambda}) + \gamma * J$

Update the function $f_m(x)$:

$f_m(x) = f_{m-1}(x) + G_m(x)$

(3) The final output model $f_M(x)$ becomes:

$f_M(x) = f_0(x) + \sum_{m=1}^{M} G_m(x)$