

Empirical Asset Pricing via Machine Learning

Ze Yu Zhong

Monash University

Abstract

Empirical Asset Pricing via Machine Learning

Empirical Asset Pricing via Machine Learning

Introduction

Motivations

The ambiguity surrounding the true functional forms of how the predictors interact and influence the risk premia is another problem which machine learning is well suited to dealing with. Machine learning methods are highly flexible, able to capture highly non-linear patterns, but with the availability of parameter penalization and validation sets can still avoid overfitting and discovering non-existent patterns.

Literature

Main Findings

Limitations of Machine Learning

Machine learning excels at prediction problems, namely estimating $E(r_{i,t+1}|\mathcal{F}_t)$, where $r_{i,t+1}$ is an asset's excess return over the risk free rate, and \mathcal{F}_t is the set of all information (including unobservable) available to market participants in this context.

This means that machine learning algorithms do not, nor do they aim to, explain how the market works in terms of underlying dynamics and equilibria. Though a machine learning algorithm may be able to identify patterns that otherwise cannot be easily found, an economist is still required to analyse these patterns to construct and hypothesize economic theory.

Methodology

Overall Design

Each model will be presented and explained so that a reader without any machine learning background can understand the basic idea behind the model. Computational methods such as algorithms however, will only have their general principles and background explained in the appendix. This is because there are many variations of

probably fo

more on sin

lation aspec

pending

pending Lit

ture Review

Pending

algorithms available, and more importantly, specific understanding of how the algorithm works is not necessary.

All asset excess returns are modelled as an additive prediction error model:

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

where

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

Sample Splitting

Imperative to any machine learning technique is the establishment of how the dataset is to be split into training, validation and test sets. The training set is used to initially build the model and provide initial estimates of parameters, whereas the validation set is used to tune model parameters to optimise out of sample performance, thus preventing overfitting. Essentially, the validation set acts as a simulation of out of sample testing. The test set is used only for evaluation, and is thus truly out of sample.

There are three main approaches to splitting temporal data (such as financial data).

The first is to decide arbitrarily on a single training, validation and test set. This method is straightforward and the least computationally intensive, but is limited and inflexible in evaluating how models perform when more recent data is provided for training.

The second method is a "rolling window" method, where a fixed size or "window" for the training and validation set is first chosen. This window then incrementally move forwards in time to include more recent data, with a set of forecasts for the test sets made for all possible windows.

The third is a "recursive" method, which is the same as the rolling window method, but different in that the training set always contains previous data, with only the validation set staying fixed in size and "rolling" forwards. Hence, it is like a rolling

window approach that has a growing training sample.

Both the rolling window and recursive schemes are very computationally intensive. Therefore, a hybrid of the methods where the training, validation and test samples are split, and hence models refit once each year. The training set is increased by one year each year, the validation set remains one year in length but moves forward by one year, and forecasts are only made using that model for the subsequent year. Cross validation was crucially not done to maintain the temporal ordering of the data.

Classification and Regression Trees

Classification and regression trees are fully non-parametric models that can capture complex multi-way interactions. A tree "grows" in a series of iterations. With each iteration, a split ("branch") is made along one predictor such that it is the best split available at that stage (in terms of lowering the loss function). These steps are continued until either the stopping criterion is met (such as via regularization) or each observation is its own node. The eventual model slices the predictor space into rectangular partitions, and predicts the unknown function $g^*(.)$ with the average value of the outcome variable in each partition.

The prediction of a tree, \mathcal{T} , with K "leaves" (terminal nodes), and depth L is

$$g(z_{i,t}; \theta, K, L) = \sum_{k=1}^K \theta_k \mathbf{1}_{z_{i,t} \in C_k(L)} \quad (3)$$

For this study, only recursive binary trees (the most common and easy to implement) are considered. The popular l_2 impurity was also chosen as the loss function (conceptually similar to mean squared error):

$$H(\theta, C) = \frac{1}{|C|} \sum_{z_{i,t} \in C} (r_{i,t+1} - \theta)^2 \quad (4)$$

where $|C|$ denotes the number of observations in set C (partition). Given C , it is clear that the optimal choice for minimising the loss function is simply

$$\theta = \frac{1}{|C|} \sum_{z_{i,t} \in C} r_{i,t+1} \text{ i.e. the average of the partition.}$$

Trees, grown to a deep enough level, are highly unbiased and flexible. The tradeoff of course, is their high variance and instability. Thus, an ensemble method called "Random Forest" was used to regularize trees by combining many different trees into a single prediction.

Random Forests

Random Forests are an extension of trees that attempt to address some of their problems. A random forest algorithm creates B different bootstrap samples from the training dataset, fits an overfit regression tree to each using only a random subset m size from all available predictors (also known as dropout), and then averages their forecasts. The overfit trees means that the underlying trees has low bias, and the dropout procedure means that they have low correlation. Thus, averaging these low bias, uncorrelated trees results in a low bias, yet stable model. Specific details of the random forest algorithm used are detailed in the appendix.

Neural Networks

Neural networks are arguably the most complex type of model available, able to capture several non-linear interactions through many layers, hence its other name "deep learning." On the flipside, their high flexibility often means that they are among the most parameterized and least interpretable models, earning them the reputation as a black box model.

The scope of this paper is limited to traditional "feed-forward" networks. The feed forward network consists of an "input layer" of scaled data inputs, one or more "hidden layers" which interact and non-linearly transform the inputs, and finally an output layer that aggregates the hidden layers and transform them a final time for the final output.

Neural networks with up to 5 hidden layers were considered, each named NNX where X represents the number of hidden layers. The number of neurons in each layer was chosen according to the geometric pyramid rule, i.e. NN1 has 32 neurons, NN2 has 32 and 16 neurons in the first and second hidden layers respectively, NN3 has 32, 16, and 8 neurons, NN4 has 32, 16, 8, and 4 neurons, and NN5 has 32, 16, 8, 4, 2 neurons

respectively. All units are fully connected; that is, each neurons receives input from all neurons the layer before it.

The ReLU activation function was used for all hidden layers owing to its high computational speed, and hence popularity within recent literature. Other potential choices for activation functions such as sigmoid, softmax, tanh etc. were not used due to computational cost.

$$ReLU(x) = \max(0, x) \quad (5)$$

The neural networks detailed in this paper have the following general formula. Let $K^{(l)}$ denote the number of neurons in each layer $l = 1, \dots, L$. Define the output of neuron k in layer l as $x_k^{(l)}$. Next, define the vector of outputs for this layer as $x^{(l)} = (1, x_1^{(l)}, \dots, x_{K^{(l)}}^{(l)})'$. The input layer is defined using predictors, $x^{(0)} = (1, z_1, \dots, z_N)'$. The recursive output formula for the neural network at each neuron in layer $l > 0$ is then:

$$x_k^{(l)} = ReLU(x^{(l-1)'} \theta_k^{l-1}), \quad (6)$$

with the final output

$$g(z; \theta) = x^{(L-1)'} \theta^{L-1} \quad (7)$$

The neural network's weight and bias parameters are estimated by minimizing the penalized l_2 objective function of prediction errors.

Due to the highly complex nature of the loss function, and the high dimensionality caused by the large number of parameters, this optimisation problem is very computationally intensive to solve, generally done via algorithms such as Gauss-Newton steps and is called "gradient descent." A common solution is to use "stochastic gradient descent" (SGD) where instead optimising the loss function with respect to the entire training sample, only a small, random subset of the data (mini batches) is used at each optimisation step. This sacrifices some accuracy for a dramatic improvement in computational speed.

Due the noisiness (randomness) introduced by SGD, the learning rate (step size of each descent) needs to be shrunk towards zero as the gradient approaches zero to avoid the noisiness of the mini batch causing an "overshoot" of the optimum. A learning rate shrinkage algorithm was therefore employed.

"Batch normalization" is a technique for addressing a phenomenon known as internal covariate shift, a particularly prevalent problem in training deep, complex neural networks. Internal covariate shift is where the distributions of each layer's inputs change as the parameters of the previous layer change, resulting in the need for much slower learning rates and more careful initialization of parameters. By normalizing (de-meaning and variance standardizing) each training step (batch) input, the representation power of each unit is restored. Additionally, significant gains in computational speeds may also be achieved.

Finally, multiple random seeds (initializations) were used in training neural networks and the resulting predictions were averaged in an ensemble like fashion. This allows for regularization for the variance associated with the initial random seed.

Simulation

Performance Evaluation

Predictive performance for individual excess stock returns were assessed using the out of sample R^2 :

$$R_{OOS}^2 = 1 - \frac{\sum_{(i,t) \in \mathcal{T}_3} (r_{i,t+1} - \hat{r}_{i,t+1})^2}{\sum_{(i,t) \in \mathcal{T}_3} r_{i,t+1}^2} \quad (8)$$

where \mathcal{T}_3 indicates that the fits are only assessed on the test subsample, which is never used for training or tuning.

Variable Importance

Study

Data

Model

All machine learning methods are designed to approximate the empirical model $E_t(r_{i,t+1}) = g * (z_{i,t})$ defined in equation (2). The baseline set of stock-level covariates $z_{i,t}$ as:

$$z_{i,t} = x_t \otimes c_{i,t} \tag{9}$$

where $c_{i,t}$ is a $P_c \times 1$ matrix of characteristics for each stock i , and x_t is a $P_x \times 1$ vector of macroeconomic predictors (and are this common to all stocks, including a constant). $z_{i,t}$ is a $P \times 1$ vector of features for predicting individual stock returns ($P = P_c P_x$) and includes interactions between individual characteristics and macroeconomic characteristics.

Trevor Hastie, 2013

References

Trevor Hastie, J. F., Robert Tibshirani. (2013). *The elements of statistical learning*.

Appendix

Algorithms

Trees.

Algorithm 1: Classification and Regression Tree

Initialize ;

for d from 1 to L **do** **for** i in $C_l(d-1), l = 1, \dots, 2^{d-1}$ **do** For each feature $j = 1, 2, \dots, P$, and each threshold level α , define a split as $s = (j, \alpha)$ which divides $C_l(d-1)$ into C_{left} and C_{right} :

$$C_{left}s = \{z_j \leq \alpha\} \cap C_l(d-1); C_{right}s = \{z_j > \alpha\} \cap C_l(d-1)$$

Define the impurity function:

$$\mathcal{L}(C, C_{left}, C_{right}) = \frac{|C_{left}|}{|C|} H(C_{left}) + \frac{|C_{right}|}{|C|} H(C_{right})$$

where

$$H(C) = \frac{1}{|C|} \sum_{z_{i,t} \in C} (r_{i,t+1} - \theta)^2, \theta = \frac{1}{|C|} \sum_{z_{i,t} \in C} r_{i,t+1}$$

 and $|C|$ denotes the number of observations in set C

Find the optimal split

$$s^* \leftarrow \underset{s}{\operatorname{argmin}} \mathcal{L}(C(s), C_{left}(s), C_{right}(s))$$

Update nodes (partition the data):

$$C_{2l-1}(d) \leftarrow C_{left}(s^*), C_{2l}(d) \leftarrow C_{right}(s^*)$$

end**end****Result:** The prediction of a regression tree is:

$$g(z_{i,t}; \theta, L) = \sum_{k=1}^{2^L} \theta_k \mathbf{1}_{z_{i,t} \in C_k(L)}; \theta_k = \frac{1}{|C_k(L)|} \sum_{z_{i,t} \in C_k(L)} r_{i,t+1}$$

Random Forest.

Algorithm 2: Random Forest

for b from 1 to B **do**

Draw bootstrap samples $(z_{i,t}, r_{i,t+1}), (i, t) \in \text{Bootstrap}(b)$ from the dataset
 Grow a tree T_b using Algorithm, using only a random subsample, say \sqrt{P} of all
 features Denote the resulting *bth* tree as

$$\hat{g}_b(z_{i,t}, \hat{\theta}_b, L) = \sum_{k=1}^{2^L} \theta_b^k \mathbf{1}_{z_{i,t} \in C_k(L)}$$

end**Result:** The final random forest prediction is given by the output of all trees:

$$\hat{g}_b(z_{i,t}; L, B) = \frac{1}{B} \sum_{b=1}^B \hat{g}_b(z_{i,t}, \hat{\theta}_b, L)$$

Neural Networks. There are numerous stochastic gradient descent algorithmsavailable.

Algorithm 3: Early stopping via validation

Initialize $j = 0$, $\epsilon = \infty$ and select the patience parameter p (max iterations)**while** $j < p$ **do**

Update θ using the training algorithm Calculate the prediction error from the
 validation sample, denoted as ϵ'

if $\epsilon' < \epsilon$ **then**| $j \leftarrow 0$ | $\epsilon \leftarrow \epsilon'$ | $\theta' \leftarrow \theta$ **else**| $j \leftarrow j + 1$ **end****end****Result:** θ' is the final parameter estimate

ch differ-
 pes of al-
 ms. Paper
 ADAM

Algorithm 4: Batch Normalization for one activation over one batch

Input: Values of x for each activation over a batch $\mathcal{B} = x_1, x_2, \dots, x_N$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{N} \sum_{i=1}^N x_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{N} \sum_{i=1}^N (x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta := BN_{\gamma, \beta}(x_i)$$

Result: $y_i = BN_{\gamma, \beta}(x_i) : i = 1, 2, \dots, N$
