Artificial Intelligence: An empirical asset pricing exercise

Constantine Bardis , 4150084

Professor: Spyros Skouras

Athens, September 2019

Abstract

The purpose of this exercise is to investigate whether a few statistical learning algorithms, both classical and of the machine learning variety, can adequately predict daily index prices of a portfolio of stocks closely resembling the S&P 500 index, while restricted on the number of features available as predictors and on computational power. Under those circumstances, it has been found that more complex algorithms do not necessarily add enough predictive power to justify the extra computational expense, and at times even underperform compared to simpler ones.

More specifically, always regarding the out-of-sample results, we found that the predictability of the daily returns as measured by the coefficient of determination is generally very low, often diving into negative territory; Mean squared errors are generally roughly equal among models, indicating that added complexity does not lead to particularly improved results; and that no single model outperformed the others across the board, thus there is no clear winner. However, the daily Sharpe ratios of every single strategy are positive, albeit still low, with the Convolutional neural network coming on top with 0.1464, much higher than the runner-ups offering ratios just short of 0.05.

Table of Contents

1. Introduction

In the 'Big Data Era', where the abundance of information is unprecedented, and the speed with which it becomes available to interested parties for research or operational or any other kind of use is staggering, models well versed in effectively processing such massive amounts of inputs are rapidly becoming a necessity. The need for such models is also further compounded by the highly nonlinear nature of economic data, as well as the very complex ways the **"***large numbers of predictor variables that the literature has accumulated over five decades***"** (Shihao Gu et al 2018: page 7) often interact, which may not be captured adequately by more classical approaches, such as linear models.

To that end, due to the rapid advances in hardware technology and the increasing sophistication of learning algorithms, the notion of Artificial Intelligence as a tool for dissecting and analyzing various problems of economic interest has emerged, in the form of machine learning, and more recently, deep learning. Those frameworks have been designed to be capable of learning in the face of nonlinearity, such as Decision Trees or Support Vector Machines, even when using models originally developed for other tasks like Recurrent Neural Networks (RNNs) for speech recognition and text generation (see Dean et al. 2012 and Lake et al. 2016).

Financial forecasting problems in particular, a flavor of which to be explored further being the main empirical application focus of this exercise, such as those present in designing and forecasting the prices of financial derivatives (Hutchinson et al., 1994; Yao et al., 2000, among others), automating the construction of portfolios (Heaton et al. 2016) and risk management in various sectors, such as the credit card industry (Butaru et al. 2016) or Mortgage Risk assessment (Sirignano et al 2016) among others, often involve particularly large and complex datasets, which current leading economic models often fall short of adequately specifying. This is where the power of modern machine learning algorithms becomes more apparent by virtue of producing more useful results than standard models, as deep learning is capable of detecting and exploiting all the interactions within the data that *"at least currently, are invisible to any existing financial economic theory*" (J. B. Heaton et al 2016).

However, AI has not been yet established as the mainstream approach in empirical econometric research. What does the literature have to say regarding AI's usefulness in modelling financial problems in the context of asset pricing and of predicting financial returns, and how does it compare with more traditional econometric tools?

Shihao Gu, Bryan Kelly and Dacheng Xiu, in their 2018 paper, utilize several models to attempt to measure and later accurately forecast asset risk premia, including generalized linear models, dimension reduction techniques, boosted regression trees, random forests, and neural networks. They find out that not only that the more advanced methods, particularly densely connected neural networks and decision trees, outperform the more classical methods, due to their increased ability to uncover and model nonlinearity in the data, but also that there exist specific signals that are far more important that others, like stock and industry level momentum, short-term reversal, liquidity and volatility.

J. B. Heaton, N. G. Polson and J. H. Witte (2016) present some state of the art deep learning frameworks like LSTMs and autoencoders, and suggest the idea that they can be used to boost predictive performance significantly. To demonstrate that, they used autoencoders to effectively compress the representation of S&P 500 stock data to a more tractable version for easier processing, which while not perfect, succeeds in being a good approximation of the index.

Yao and Tan (2000) used neural networks for derivative pricing, specifically option price forecasting. They concluded that even though the traditional Black-Scholes model still is useful for pricing at the money options, neural networks tend to outperform it when more volatility is inserted into the 'equation'. Thus, they conclude, that depending on how the data is partitioned, the Blach-Scholes model represents more conservative investors better, and neural networks tend to accommodate more high-risk, high-reward preferences.

Hutchinson, Lo and Poggio (1994) conduct multiple Monte Carlo simulations to test how nonparametric models, like radial basis functions (RBFs) and multilayer perceptron networks (MLPs) compare to their more established, parametric siblings like the Black-Scholes model, when it concerns derivative asset pricing. They concluded that even though they are not unconditionally a strictly better option, they can indeed be more computationally efficient and precise when the underlying dynamics of the assets are fuzzy and dynamic, or when there is no analytical solution to the parametric methods. That is attributed, at least partly, to their ability to let the data itself determine the price dynamics, with very few assumptions on the part of the models, which makes them less prone to specification errors, which often are the bane of parametric models.

Other advantages, the authors note, of these models is that they are more skilled at adapting rapidly to structural changes in the dynamics of the data more effectively than parametric models can, and that even though they can adequately describe a range of instruments, they remain quite simple to to implement in practice. However, they do require quite large amounts of data to be trained properly, which minimizes their usefulness for assets with low availability of historical data, like thinly traded assets or newly created ones, and when the dynamics of the assets are well understood and analytical solutions are well defined, the parametric models tend to perform best in terms of pricing and hedging precision.

There is one another one important drawback of machine learning models, as noted by Moritz and Zimmermann (2016), and one that echoes across the literature: That they do not offer themselves to interpretable predictions, and that they work like black box processes (eg Breiman, 2002). The authors, in their attempt to create portfolios based on various factors like momentum via decision trees, exhibit that they can in fact extract interpretable information from the structure of the forecasts. They also found that only the most recent returns history, that is the past six month ones, are powerful predictors, and that their nonlinear models result in an information ratio up to 3 times higher than the ones achieved by linear frameworks.

An important benefit of decision trees, exemplified on that research, is their ability to effectively process the multitudes of different variables that may be useful for forecasting, taking into account the nonlinear ways they interact, which again,

standard linear models cannot do.This is a significant property, as the literature has not yet conclusively decided which features are fundamentally important for robust, out-of-sample predictions. Also, as found in other papers, decision trees can much more easily accommodate the inclusion of new features and combine them with existing ones to form more informed forecasts, than most standard (parametric) models.

Nayak, Misra, Behera (2017) provide a survey of nature-inspired algorithms used in a very wide variety of applications, and they focus on efficient stock market indices' forecasting by utilizing chemical reaction optimization instead of backpropagation to train the neural networks used. The resulting model, due to its enhanced ability to overcome issues of overfitting, convergence and parameter setting, displayed a significant improvement over the standard MLP approach, tested with 7 index values. Their model has been adapted and trained for predictions over all time periods, namely short, medium and long- term predictions, further adding to its robustness across time horizons.

Henrique, Sobreiro and Kimura (2018) use a support vector machine (SVM) to gauge whether it can deliver higher returns than what the Efficient Market Theory (EMH) would allow, for both large and small cap stocks, in differing markets (Brazilian, American and Chinese) and in the daily and minute data frequency, with the benchmark being a random walk model as proposed by EMH. After extensive cross-validation, they came to the conclusion that the SVR with a linear kernel was delivering better results in the daily frequency, but almost always worse in the minute frequency, particularly when the models were periodically retrained to take advantage of any changes in the trends of the time series. However, those results do not contradict the EMH, as they do not recommend any specific strategies, and the authors note two major limitations that could skew the results, those being the quality of the data and the volume of available data points for effective training.

Menon et al (2018) conducted a study where they compared the performance of 4 types of neural networks, the MLP, RNN, LSTM and CNN (Convolutional Neural Network), in predicting the returns of 5 different companies on NYSE and NSE (National Stock Exchange of India), while only trained on data from a single company from NSE, after normalization was performed. The results indicated that all of the neural networks performed well with the CNN crowned as the best model, and collectively providing an improvement over the standard ARIMA model, which indicated that they were indeed capable of learning the underlying dynamics of both stock markets, unlike the aforementioned linear competitor.The authors bring up the idea that a hybrid neural network could prove to be an interesting next avenue to explore, combining the strengths of LSTMs / RNNs and of the CNNs.

After this short, far from exhaustive literature review into the uses of AI into financial research concerning asset pricing and forecasting of returns, it is time we proceeded to our own mini empirical analysis using a few of the models mentioned, of both classic econometric nature and of the machine learning variety, to perform a short analysis into how these models fare into predicting daily stock prices, closely resembling the S&P 500 index.

The rest of the paper is structured as follows: In the upcoming "Methodology" section the basic theoretical foundations of the algorithms used will be concisely reviewed, along with some of the basic methodological tools and concepts employed to optimize the models' training and tuning; Then, in the "Empirical Results" section the majority of the results will be presented, mainly in the form of plots, starting with exploratory data analysis (EDA) along with the methodology used to gather and prepare the data and concluding in aggregate plots of all the algorithm's performances; In the "Conclusion" section, the most basic insights will be summarized, and exciting new future directions for research will be briefly discussed. In the Appendix, some additional uses of AI in the broader field of finance will be discussed.

2. Methodology

2.1) Model Formulation

In total, excluding benchmarks, a sum of 10 models were trained and evaluated, generally starting with the simpler ones and gradually moving on to newer, and more complex, architectures. In this section, without delving too much onto the mathematical details, the basic theoretical formulations of those models will be presented, as well as some of their advantages and disadvantages. As this is not a theoretical exercise, and for brevity reasons, proofs of the equations to be explained are omitted.

2.1.1) (Multiple) Linear Regression

The most basic model found in economic literature, is also the first to be evaluated and used. Its mathematical implementation is given as follows (in matrix form):

$$\mathrm{Y} = \mathrm{X}\beta + \varepsilon \quad \textbf{(1)}$$
$$\varepsilon \sim \mathrm{N}(0, \sigma^2)$$

Where X: Our feature matrix with the 10 predictor variables, β: The coefficients and ε: Residual errors, supposedly distributed according to normal distribution with mean 0 and constant variance, an assumption which practically does not really apply on our dataset.

This simple equation is then minimized according to the standard Ordinary Least Squares (OLS) method in order to estimate the true parameters β as $\hat{\beta}$, our fitted values, by minimizing the sum of squared residuals:

$$L_{OLS}(\hat{\beta}) = \sum_{i=1}^{n} (y_i - x_i^{'}\hat{\beta})^2 = \left\| y - \mathrm{X}\hat{\beta} \right\|^2 \quad \textbf{(2)}$$

In order to finally (analytically) derive the OLS parameters:

$$\hat{\beta}_{OLS} = (\mathrm{X}'\mathrm{X})^{-1}(\mathrm{X}^{'}\mathrm{Y}) \quad \textbf{(3)}$$

However, as the number of features increases, so does the 'complexity' of the algorithm, that could theoretically result in overfitting (even though practically it is not plausible due to the linear nature of the model), which means it could lose its power to generalize to unseen data well. To that end, we need some form of penalization for increasing the complexity (number of features) to prevent that, leading to the second model utilized:

2.1.2) Elastic Net

The Elastic Net regression effectively combines Lasso regression (L1 regularization) and Ridge regression (L2 regularization) for more effective penalization of added complexity, which avoids completely eliminating all but the most important features, as L1 is prone to, but also working well when the number of features is large, like L2 does. This property of Lasso to zeroing some coefficients can also be used as a sort of feature selection method, like PCA, with the added benefit of being cheap to run given it is also a linear transformation. Therefore, this model's coefficients are produced by the minimization of the following:

$$\hat{\beta} = \arg\min_{\beta}(\|y - X\beta\|^2 + \lambda_1\|\beta\|_1 + \lambda_2\|\beta\|_2) \Leftrightarrow \quad \textbf{(4a)}$$

$$\hat{\beta} = \arg\min_{\beta}\sum_{i}(y_i - \beta^{'}\chi_i)^2 + \lambda_1\sum_{\kappa=1}^{K}|\beta_\kappa| + \lambda_2\sum_{\kappa=1}^{K}\beta_\kappa^2 \quad \textbf{(4b)}$$

The two forms are exactly equal, described here only for clarifying any potential ambiguity.

However, both these models still do not really take into consideration the nature of the data they are trying to learn; Namely they lack any sort of specialized structure which takes advantage of the defining trait of the series, serial autocorrelation. To compensate for this, we proceed to the next model:

2.1.3) Autoregressive – moving-average (ARMA)

This is one of the mainstream models used in financial Econometrics for the task of understanding, and more importantly attempting to predict, various time series of interest, like stock market movements or commodities' future prices. As the name suggests, it consists of two parts: The Autoregressive (AR), which involves regressing the variable on its own lagged values and the Moving Average (MA) part, which models the error term as a linear combination of the previous error terms. Thus it it referred to as **ARMA(p,q)** with **p**: Order of AR (sub)model and **q:** Order of MA (sub)model.

Rigorously, in its general form, it is defined as such:

$$X_t = c + \varepsilon_t + \sum_{i=1}^{p}\varphi_i X_{t-i} + \sum_{i=1}^{q}\theta_i\varepsilon_{t-i} \quad \textbf{(5)}$$

ARMA models by their nature are restricted to handling only one time series for training and prediction, unlike every other model used here; Thus, and to ensure that ours is evaluated in the closest possible manner as the others, it is trained on the unscaled series before any features are computed, as training it on the scaled series could warp its forecasting ability due to its temporal structure, with the same percent split for train and test sets. This does not prevent it from achieving a relatively good performance, as will become apparent in the next section, even though the most potent model identified is a simple AR(2).

2.1.4) Polynomial Regression

Polynomial Regression is the first "quasi" nonlinear algorithm deployed. It is called that here as even though it does not provide a linear line as its set of predictions, it remains a special case of the traditional linear regression, where the terms are expanded to include powers of the predictor variables. For that reason, the same matrix notation applies as was the case with the linear regression, and the same formula for calculating the coefficients, but with X including the expanded feature set:

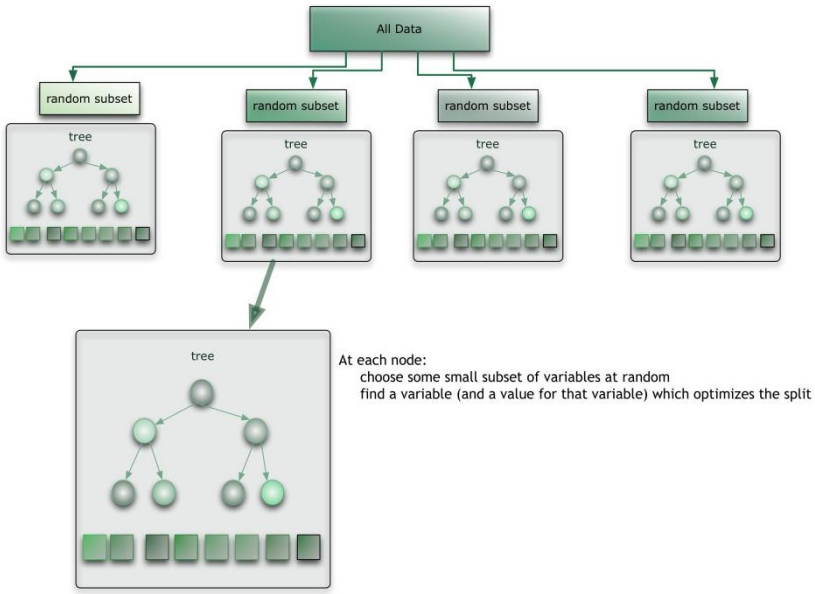
$$\mathrm{Y} = \mathrm{X}\beta + \varepsilon \Leftrightarrow$$
$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} 1\ x_1\ x_1^2 \dots x_1^m \\ 1\ x_2\ x_2^2 \dots x_2^m \\ \dots\dots\ \dots\ \dots\ \dots \\ 1\ x_n\ x_n^2 \dots x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \dots \\ \varepsilon_n \end{bmatrix} \quad \textbf{(6)}$$

The model used here also includes all interaction terms between the variables, and the degrees of the polynomial evaluated are 2nd and 3rd, as degrees higher than that were much more computationally expensive to run and did not offer any out-of-sample improvement over the ones mentioned.

2.1.5) Random Forest

Random Forests are essentially Ensemble algorithms, and in this occasion an ensemble of individual CARTs, decision trees capable of handling both classification and regression tasks. It utilizes multiple such decision trees, each trained on a random subset of the data, via a process called "Bootstrap Aggregation" (or "Bagging") and averages over the predictions to deliver the final result. Due to those techniques, it has the distinctive advantage of being able to reduce variance of individual CARTs, thus eliminating some of the risk of overfitting, but also resulting in more accurate predictions than any individual tree, particularly in classification tasks. Graphically, it looks like this:

Image 1:Random Forest



Source: Dan Benyamin

Without going too deep into the details of how individual CARTs are formulated, which compose the Random Forest, in essence each split is done so that the the greatest reduction in the residual sum of squares is incurred. This is referred to as a reduction to the 'entropy' of the system, which is the disorder of the data, that it aims to minimize. Entropy is defined as

$$\text{Entropy} = -\sum p(X)\log p(X), \qquad \textbf{(7)}$$
$$\text{Where p(X)}: \text{fraction of examples in a given class}$$

Closely related to the concept of entropy is the concept of 'Information Gain', which is loosely defined as the amount of information a feature provides about its potential class, and is based on the decrease in entropy after a dataset is split on an attribute, and it can be used as a criterion with which to minimize the system's entropy. The combination of those two concepts results in the ID3 algorithm, a very basic algorithm for building decision trees. IG can be (rather loosely) be formulated as such:

$$IG(f, sp) = I(parent) - \left(\frac{N_{left}}{N} I(left) + \frac{N_{right}}{N} I(right)\right), \qquad \textbf{(8)}$$
$$\text{f}: \text{feature}, \text{sp}: \text{split-point}, \text{I(.)}: \text{Metric}, \text{N}: \text{Number of samples}$$

Therefore, when it comes to regression trees, which we are interested in given our objective of predicting continuous values, the IG can be defined as such (using Mean Squared Error or MSE):

$$I(node) = MSE(node) = \frac{1}{N_{node}} \sum_{i \in node} (y^{(i)} - \hat{y}_{node})^2, \quad \textbf{(9)}$$

$$\hat{y}_{node} = \frac{1}{N_{node}} \sum_{i \in node} y^{(i)} \quad \textbf{(10)}$$

2.1.6) Support Vector Machine

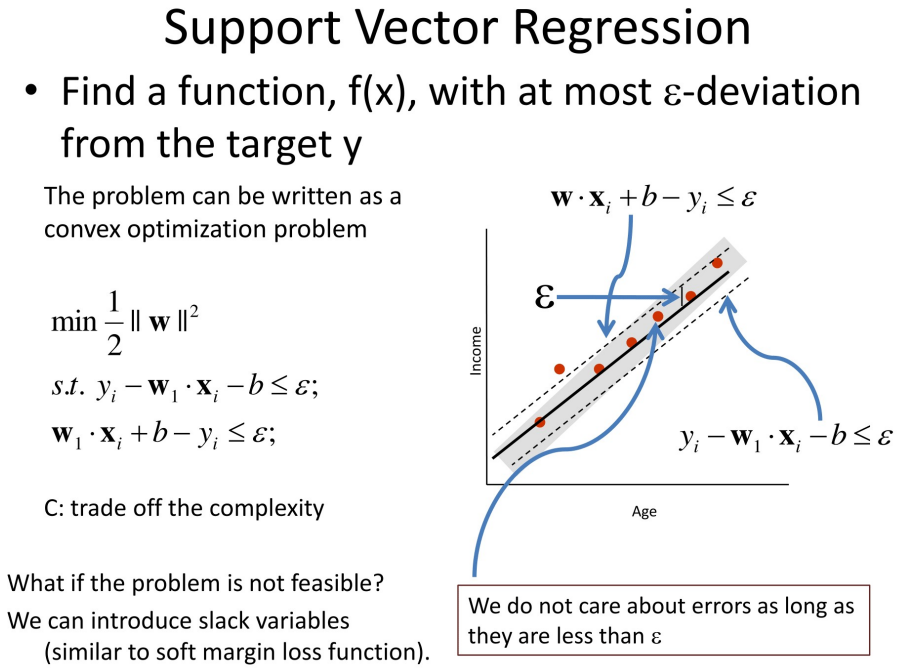
These algorithms produce linear separation boundaries by transforming the initially nonlinear feature space appropriately, and then running a linear regression in that vector space. This linear regression is not the same as the familiar OLS one, and is expressed as:

$$\ell(x) = \sum_i w_i \phi(\chi_i).\phi(\chi) \quad \textbf{(11)}$$

Where χ_i are the observations from the training set, φ(.) is the transform applied to the data, and the dot is the usual matrix dot product.

It is also called the 'maximum margin classifier' as it attempts to find the best linear separation boundary that maximizes the distance between the two clusters of data it trains on, in order to avoid overfitting. Again, we will skip over the convoluted mathematical details, except for one graph which summarizes the objective function it aims to optimize.

Image 2:Support Vector Regression



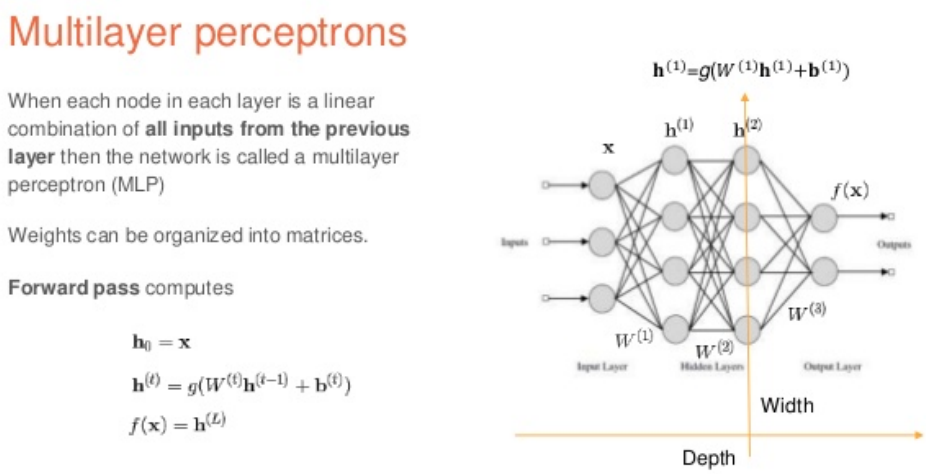
Source: University of Adelaide, Paul Paisitkriangkrai, 2012

2.1.7) Multilayer Perceptron (MLP)

Neural networks are state of the art deep learning methods, used in a very wide variety of fields and contexts, like natural language processing, computer vision, time series prediction, medicinal diagnosis, offering excellent performance even when the data is very complex and highly nonlinear.

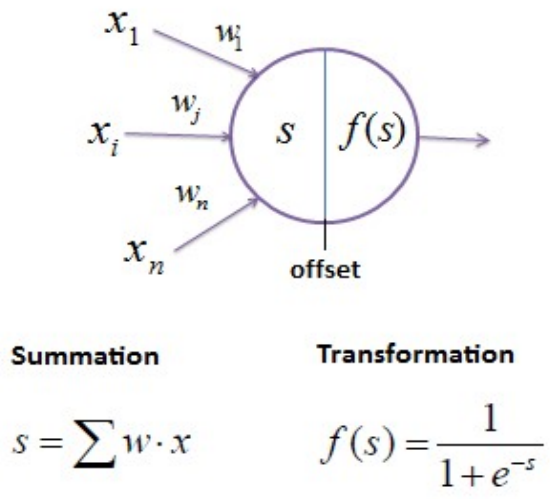
We begin by applying the simplest, but by no means less effective or versatile, architecture of the neural networks, the MLP. While a detailed explanation of how and why it works is outside of the scope of this article, as was the case with the two previous complicated algorithms, some insights into its workings should be provided. Initially, below follow two important graphs which illustrate succinctly how it internally works, the first one providing a bird's eye view, and the second one magnifying an individual neuron's mechanism:

Image 3:MLP Architecture



Source: Elisa Sayrol, Master course, University of Barcelona, Autumn 2017

Image 4:MLP Neuron



Source: Christian Freischlag, digital thinking Apache Spark

As is the case with every other supervised learning algorithm visited so far, the MLP attempts to minimize the output errors, according to some metric specified beforehand, such as MSE, by tuning its weight parameter vectors W and the bias parameters b. The power of these networks lies in the fact that even though they calculate a simple summation for each 'half' of each neuron, that is fed to an activation function f(s) which then applies a nonlinear transformation.

In that way, the network is able to effectively learn nonlinear data, and as more layers and neurons per layer are added, so does its ability to learn increasingly complex data grows. More rigorously, those terms can be expressed as follows:

Let $f_1,...,f_L$ represent the activation functions for each of the L layers. An activation function does the following:

$$f_l^{W,b} = f_l(\sum_{j=1}^{N_l} W_{l_j} X_j + b_l) = f_l(W_l X_l + b_l), 1 \le l \le L \quad \textbf{(12)}$$

There has been much research into what the most effective choice of an activation function is, with no universally accepted consensus as of yet. Sigmoid used to be the most popular choice, but ReLU has gained significant momentum and is currently the most widely adopted choice, particularly in computer vision - related tasks. This can be attributed to a few technical reasons, such as its robustness to the vanishing gradient problem, which essentially means the network stops learning due to very small updates to its weights, its computational efficiency and empirical work which shows it indeed tends to lead to better convergence (for example Krizhevsky et al, 2012).

It does have disadvantages though, such as exploding gradient, as it is not constrained to a certain interval of values like sigmoid, and 'dying ReLU' where if receiving negative values it does not output anything, thus 'dying'. To that end, 'Leaky ReLU' has been proposed as a solution, which will always output something meaningful, however small. More concretely:

$$z = W_l X_l + b_l \quad \textbf{(13)}$$

$$sigmoid : \sigma(z) = \frac{1}{1+e^{-z}} \quad \textbf{(14)}$$

$$\text{ReLU} = \max(0, z) \quad \textbf{(15)}$$

$$\text{Leaky ReLU} = \max(0.1z, z) \quad \textbf{(16)}$$

Where Z: The linear combination fed as input into each of these functions.
The Neural network applies those transformations successively in order to output a final prediction, and the number of times it does depends on how many layers it has. Concretely, denoting $Z^{(l)}$ as the l-th layer (thus $Z^{(0)} = X$) and the final response as Y, 'forward-propagation' is defined as:

$$\begin{aligned} Z^{(1)} &= f^{(1)}(W^{(0)}X + b^{(0)}), \\ Z^{(2)} &= f^{(2)}(W^{(1)}Z^{(1)} + b^{(1)}), \\ &.... \\ Z^{(L)} &= f^{(L)}(W^{(L-1)}Z^{(L-1)} + b^{(L-1)}), \quad \textbf{(17)} \\ Y(X) &= \hat{W}^{(L)}Z^{(L)} + b^{(L)} \quad \textbf{(18)} \\ W_l &\in \Re^{N_l x N_{l-1}} \end{aligned}$$

As any other model, here too the network intends to minimize a given cost function, often with a regularization parameter to prevent overfitting, to which complex models like that are prone to, expressed via the parameter lambda. Usually, the choice is L2 regularization,or L^2-norm, but other norms may also be used, like lasso. Here an example of L2 is provided:

$$\arg\min_{W,b} \frac{1}{T}\sum_{i=1}^{T} L(Y_i, \hat{Y}_{W,b}(X_i)) + \lambda\phi(W,b) \quad \textbf{(19)}$$

$$\phi(W) = \|W\|_2^2 = \sum_{l=1}^{T} W_i^{\mathrm{T}} W_i \quad \textbf{(20)}$$

Finally, in order for the network to be trained, which entails the minimization of the loss function, the first partial derivatives of this cost function must be calculated, so that the weight matrices can be updated in every iteration, via gradient descent, which is a standard first-order, iterative optimization algorithm :

$$\nabla_{W,b} L(Y_i, \hat{Y}_{W,b}(X_i)) \quad \textbf{(21)}$$

This method applied to neural networks specifically is called 'Backpropagation', to contrast with the more straightforward 'forward propagation' and to indicate that the derivatives are being 'propagated back' to the beginning. The resulting backprop equations can be quite complex, so they will not be referenced here.

2.1.8) Convolutional Neural Network (CNN)

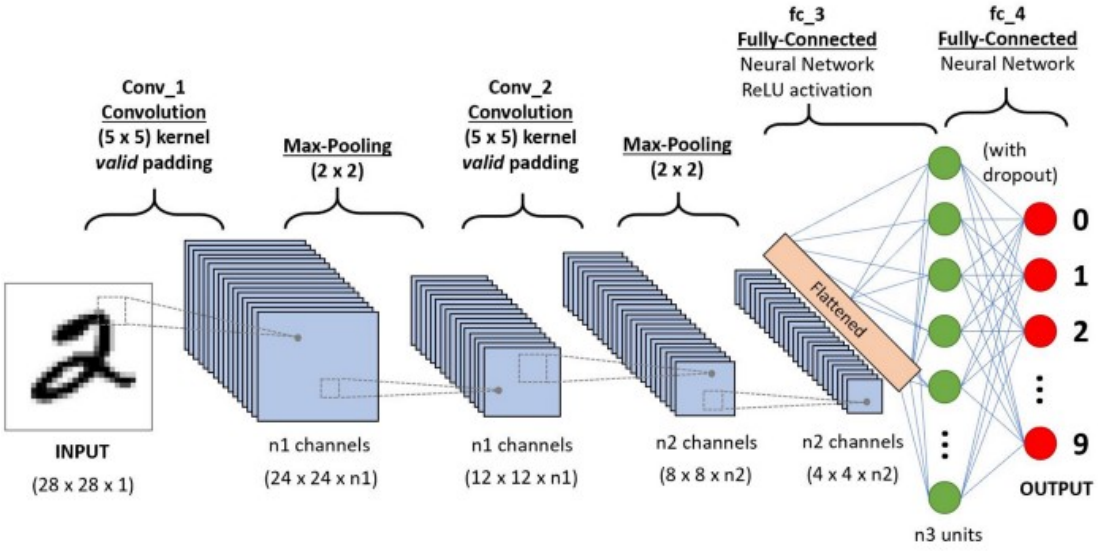
CNNs were originally developed to deal with 2- and 3- dimensional data, like image and video recognition or classification, but have also seen fairly wide use in natural language processing, recommender systems, or even medical image analysis. It is inspired by the mathematical convolution operation, which is an operation on two functions to produce a third one, that expresses how the shape of one is modified by the shape of the other. Rigorously,

$$(f * g)(t) \overset{\Delta}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \quad \textbf{(22)}$$

Where *: is the Convolution Operation
The symbol t does not necessarily represent a time dimension, but the formula can be described as a a weighted average of the function f (τ) at the moment t where the weighting is given by g(–τ) simply shifted by amount t. As t changes, the weighting function emphasizes different parts of the input function.

Without going into further details on how this operation works, it suffices to know that its main purpose is to extract the high level features out of the matrix (such as an image) it is applied to, by essentially reducing its size. An example graph of a CNN used in handwritten digit recognition is provided to provide some additional intuition into how it works from a high level perspective:

Image 5:Convolutional Neural Network - Computer Vision Application



Source: Sumit Saha, Towards Data Science, Medium

Given that CNNs can be particularly deep or complicated in their architecture, like GoogLeNet (2015), with 22 layers or Microsoft ResNet with 152 layers, regularization of some form becomes important. A novel way to do this has been proposed by Hinton et al (2014) called 'Dropout' which essentially assigns some probability at random that some layer's parameters will not be taken into consideration, effectively 'dropped out' during training, but not during testing, and they showed that it does significantly increase performance on various supervised learning tasks, like vision, speech recognition, computational biology and on various benchmark data sets (like MNIST). More specifically, in a simple one layer network, dropout would be inserted as follows, with D denoting a matrix of independent Bernoulli-distributed random variables:

$$D_i^{(l)} \sim Ber(p), \qquad \textbf{(23a)}$$

$$\tilde{Y}_i^{(l)} = D^{(l)} \otimes X^{(l)}, \qquad \textbf{(23b)}$$

$$Y_i^{(l)} = f(Z_i^{(l)}), \qquad \textbf{(23c)}$$

$$Z_i^{(l)} = W_i^{(l)} X^{(l)} + b_i^{(l)} \qquad \textbf{(23d)}$$

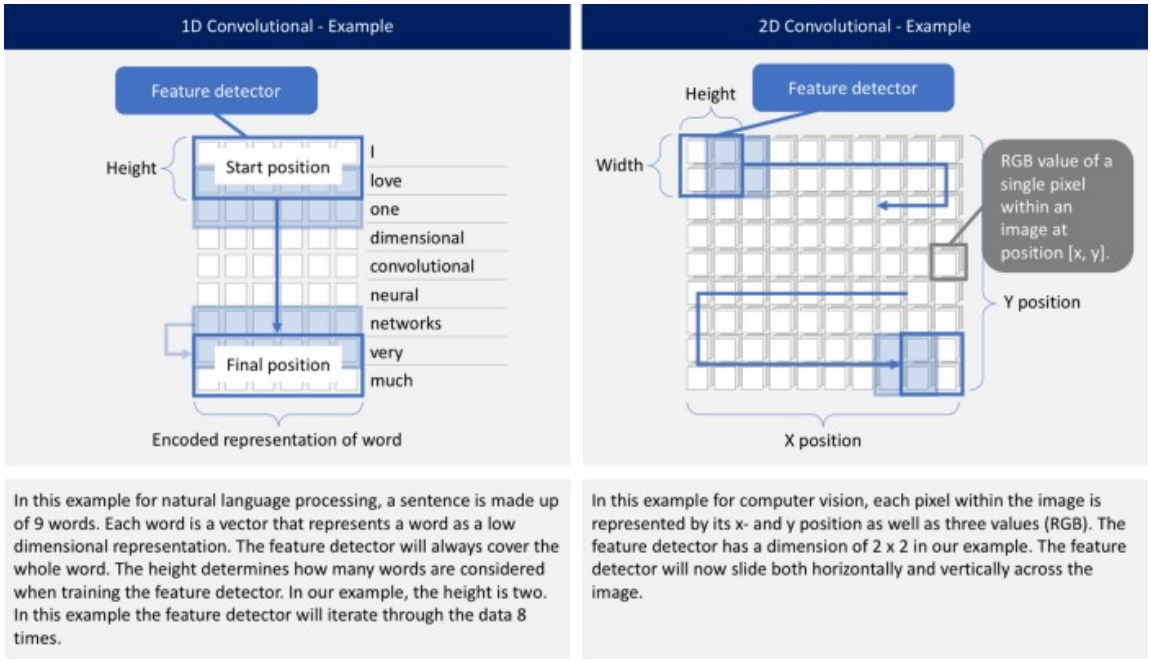
$$\otimes : \text{Hadamard (element - wise) matrix product,}$$

There are several other operations that are available and used specifically in CNNs, but only the Pooling operation will shortly be mentioned, which again is a downsampling operation, intended to extract the most critical features from the matrices it is applied to while further compressing the input size. It also has the added

benefit that it generalizes the results from a Convolutional filter, making the detection of features invariant to scale or orientation changes.

In the same way that CNNs are used for higher dimensional data, they can also be used for 1D data. The difference in an NLP setting is accurately (and intuitively) captured in the following graphic, and the reasoning is exactly the same for the type of data we are interested in, namely time series.

Image 6: 1D VS 2D CNN side-by-side comparison



Source: "1D versus 2D CNN" by Nils Ackermann is licensed under Creative Commons CC BY-ND 4.0

The rest of the operations mentioned, specific to CNNs which deal with higher dimensional data, can also be very comfortably applied to that 1-dimensional architecture as well. The network also has its own gradient descent and forward/backward propagation equations, which will not be mentioned here, but again follow the same pattern as the MLPs.

2.1.9) Long short-term memory Network (LSTM)

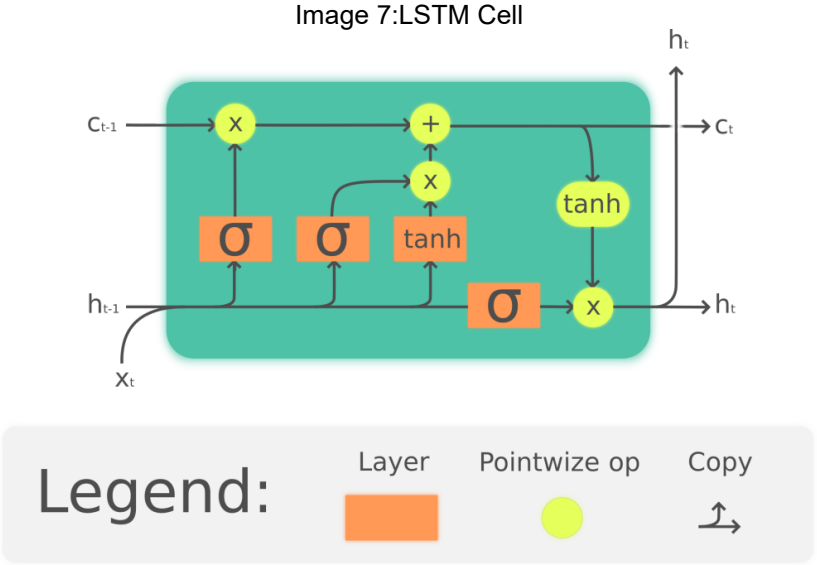
The LSTM network comes as an improvement to standard Recurrent Neural Networks (RNNs), which had been developed specifically to process sequential data. The RNNs, due to their recursive architecture, are able to capture some long-term dependencies within the data, which standard MLPs cannot do. Effectively, this means that the model should be able to share some of its learned parameters across its

different parts. However, as research has shown, like in Bengio et al (1994) or Hochreiter (1991), in practice they may not always succeed.

LSTMs were developed as a solution to this problem, and indeed are a very successful type of model, outperforming other types of RNNs, Hidden Markov Models and other sequence learning models in a very wide variety of applications, such as time series data, natural language processing, smart assistants (like Amazon's Alexa or Google's Allo) and more.

To achieve this state of the art performance, they deploy a sophisticated architecture: Each of their cells consists of three 'gates': The forget gate, which decides what information the network should discard; The input gate, which decides what values of the input to be updated, which often is the value outputted from a previous hidden layer; and the output gate, which determines the output of the cell, by applying a nonlinear activation function, namely tanh. Intuitively, the architecture of each cell can be visualized as such:

Image 7:LSTM Cell



Source: Guillaume Chevalier - Own work, CC BY 4.0

LSTMs can be trained using the same principles as ordinary neural networks, given they are extensions of them, but using more advanced variations of those methods, such as "Backpropagation through time (BPTT)' but a more detailed discussion of how and why LSTMs work again goes far beyond the scope of this exercise, although certainly interesting. However, for completeness, a basic architecture will be presented here:

$$F_t = \sigma(W_f^T[Z_{t-1}, X_t] + b_f), \quad \textbf{(24)}$$

$$I_t = \sigma(W_i^T[Z_{t-1}, X_t] + b_i), \quad \textbf{(25)}$$

$$\bar{C}_t = \tanh(W_c^T[Z_{t-1}, X_t] + b_c), \quad \textbf{(26)}$$

$$C_t = F_t \otimes C_{t-1} + I_t \otimes \bar{C}_t, \quad \textbf{(27)}$$

$$Z_t = O_t \otimes \tanh(C_t) \quad \textbf{(28)}$$

There are other flavors of LSTMs, but we will only briefly mention the 'Bidirectional LSTM' here, as it is used in the empirical section, which is based on the idea that the value at moment t is not only influenced by values before it, like t-1,….t-m, but also by values after it, like t+1, …. t+n. This is more evident in the NLP compartment, where for example a word's meaning can change based on other words present before or after it, as is common in English.

2.1.10) "Hybrid" Neural Network (CNN-LSTM)

The final architecture employed is a combination of the previous two types of neural networks, namely the CNN, for feature extraction at which it admittedly excels, and the LSTM for sequence prediction problems, like video or time series, also a problem it was designed for. This 'Hybrid' naturally aims to take advantage of the power points of both networks, and some of the tricks originally reserved for each architecture, in order to output better predictions.

It is not the first time an architecture like that is used, for example Dohahue et al (2016) have already used it for tasks involving sequences, visual or otherwise, with very positive results outperforming other state of the art approaches, so it remains an interesting question to test it on a financial time series prediction task.

A final important note of practical essence on neural networks: The MLP, as all other neural nets in this exercise, are trained via the Adam ("Adaptive moment estimation") optimizer, which is considered one of the more effective options, and is often used as the default in deep learning applications. It was first presented in the seminal paper "Adam: A Method for Stochastic Optimization" by Kingma and Ba (2015) and essentially it is a combination of two other algorithms, RMSProp and AdaGrad, and takes advantage of the power points of both: It enables the learning rate to adjust dynamically during training, and takes advantage of momentum by using moving averages of the gradients to more smoothly converge to the minimum cost.

In addition, mini-batch gradient decent is used to train all models with the default size per batch of 32 training samples (or 32 time steps) and a learning rate of 0.01. It is also trained with 150 epochs, which means 150 passes over the entire training set. It should however be noted that for the rest of the networks, due to their size and to the fact that their performance, as measured by the MSE metric, plateaued after a while, were trained with less epochs: The CNN and the computationally expensive LSTM

with 120 while the Hybrid model performed just 90. Further testing showed that going above those markers actually seemed to even marginally decrease performance, perhaps due to overfitting.

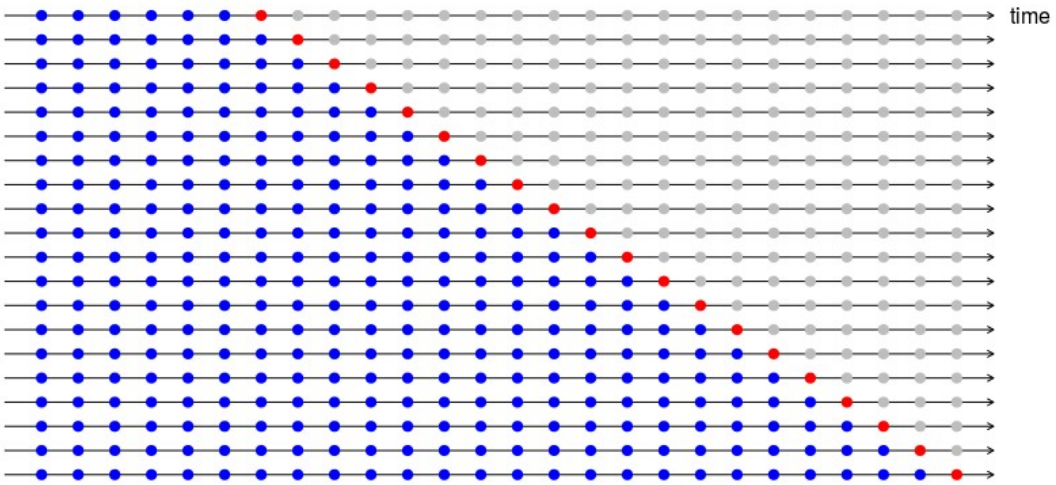
2.2) Cross-Validation

Before we proceed to the empirical section, it is important to clarify the concepts of grid search and of time series cross-validation. Grid search simply means that a 'grid' of hyperparameters for some algorithm is defined, and an exhaustive search is conducted over every possible combination of the candidate values. This means that the asymptotic runtime increases exponentially with each new hyperparameter added, a problem also known as the 'curse of dimensionality', and particularly with more complex, nonlinear models can become very computationally expensive very fast, even though a remedy could be found in parallel computing methods, as the settings evaluated are independent of each other. That is not to say it is the only or indeed the optimal tuning process; Other contenders, like Bayesian optimization or evolutionary algorithms are gaining traction as 'smarter' and often computationally more efficient, informed search techniques.

Cross-validation is a critical concept in machine learning, as it is a process allowing the models to improve their performance on unseen data and thus minimizing the danger of learning noise. In cross-sectional data, K-fold cross-validation is often thought of as the "gold-standard", as it allows for a more thorough estimation of the generalization error, by randomly partitioning the original sample into K equally sized samples, using K-1 for training, the K^{th} for testing, repeating this process K times and then reporting the average test error on all K test subsamples as the final generalization error.

However, as the partitioning process is random, it does not respect temporal dependencies, thus making unsuitable for time series data, of which the defining characteristic is the autocorrelation between successive data points. A more suitable technique, which both respects the temporal dependencies and retains the advantages of vanilla K-fold CV is called 'Forward Chaining', or "time series cross-validation" (abbreviated from now on as tscv), which can be intuitively summarized in the following graphic:

Image 8:Time Series Cross Validation (Forward Chaining)

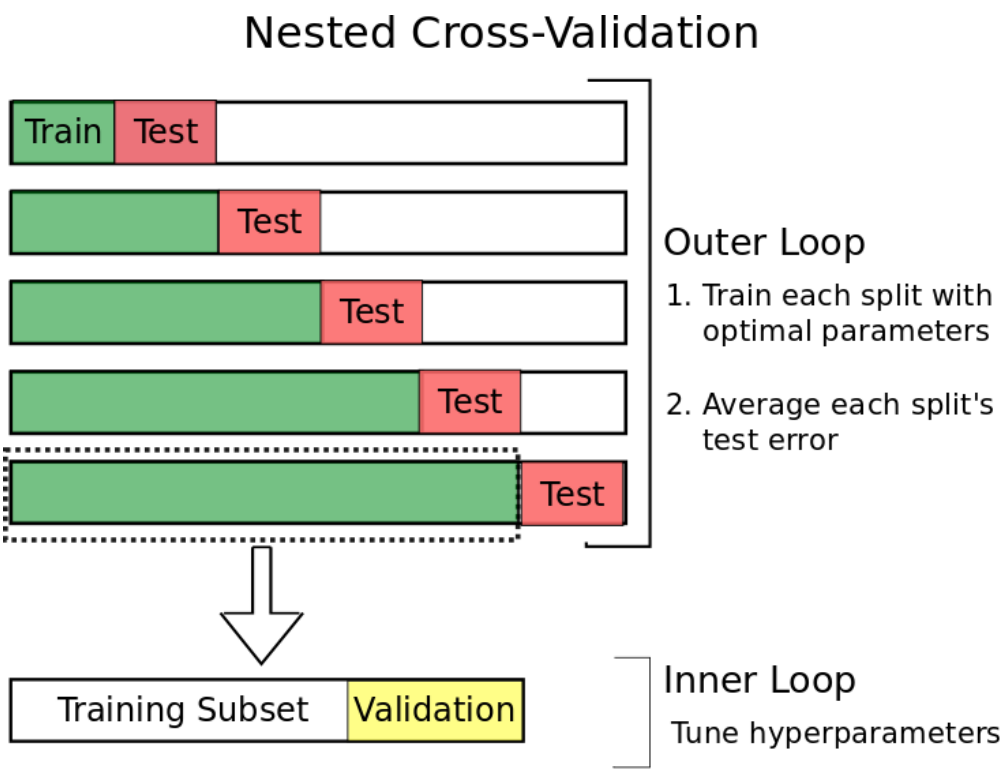


Source: Rob J Hyndman, "Cross-validation for time series"

Essentially, a 'rolling' CV is executed: All data points up to and including time k are selected (blue dots) and used as training data; The datum at time k+1 is used at the test data (red dot), while the rest of the observations are temporarily ignored (white dots); The error is computed on the test datum; The whole process is repeated for i= 1,2, … T-k times, gradually expanding the training set, where T is the total number of observations; Finally the individual errors are used to calculate an aggregate error metric which is our final tscv estimate of the generalization error. The above graphic summarizes one-step rolling forecasts, but can be very easily generalized to multistep ones if need be. Having clarified those terms, it is also set that for the rest of this exercise, each time we use tscv we use it with 5 folds (meaning K=5) and the optimization metric used is the MSE.

In addition, a recap on the algorithm of nested cross-validation is provided, as it describes the process of cross-validation used to define our benchmark 'meta-model'. Simply put, it is a more robust way to conduct more reliable model selection via a double cross-validation, consisting of two loops: The inner loop is used to perform the hyperparameter tuning, which is what we have been using up until now, while an outer loop is used to average the performance of the inner loop k-folds and report a final metric as our result. It can be intuitively illustrated with the following graphic:

Image 9:Nested Time Series Cross Validation



Source: Courtney Cochrane, Toward Data Science, Medium

The algorithm more specifically is the following:

1.Divide the dataset into K cross-validation folds via forward chaining (as shown above, increasing train set size and shifting the test set each turn)
2.For each fold k=1,2,…,K: outer loop for evaluation of the model with selected hyperparameter
2.1 Let test be fold k
2.2 Let trainval be all the data except those in fold k
2.3 Randomly split trainval into L folds
2.4 For each fold l=1,2,...L: inner loop for hyperparameter tuning
2.4.1 Let val be fold l
2.4.2 Let train be all the data except those in test or val
2.4.3 Train with each hyperparameter on train, and evaluate it on val. Keep track of the performance metrics
2.5 For each hyperparameter setting, calculate the average metrics score over the L folds, and choose the best hyperparameter setting.
2.6 Train a model with the best hyperparameter on trainval. Evaluate its performance on test and save the score for fold k.
3. Calculate the mean score over all K folds, and report that as the generalization error.

This algorithm however, even though a statistical improvement over the standard cross-validation, is computationally very expensive and only useful for models with actual hyperparameters able to be tuned. Therefore, for all the models which lack that feature, namely OLS, AR(2) and Polynomial standard 5-fold cross-validation was instead performed. In addition, only very few hyperparameters were used for the

neural networks, due to computational reasons (a single full run for the LSTM/ Hybrid models took several hours for each). Also noted that we utilize 5 folds for both inner and outer loops, and we still utilize the appropriate time series k-fold process as mentioned before, namely forward chaining.

So, how do we use this procedure to generate a 'meta-model'?

What is done is that all models are reapplied to the benchmark data, which essentially is our original data after it has been appropriately engineered in the manner described in the ensuing section, and subjected to (nested) cross-validation in order to ensure a statistically sound way to estimate out-of-sample performance.

Then the model with the best R-squared, chosen instead of MSE used earlier as we are now more interested in explainability for the purpose of establishing a benchmark, will be chosen as our 'meta-model', and it will be used to establish two kinds of benchmarks, one using the Index's history and the other using the Index's history of individual stocks, via a 'bottom -up' approach. The first one is simply going to undergo the same process at all other models, applied to the same training and test data, and used to calculate all the relevant metrics.

The second one, via the 'bottom-up' approach, will be used to individually predict each of the 200 largest corporation's returns, after the feature engineering for each stock is performed, and for every day (where every day is a single time step) the final prediction is going to be the simple average of all 200 predictions. Out of that vector, we will calculate the final metrics to be used as our second meta-model benchmark. More specifically,

$n_f = 10$: Number of features

T : Time-steps after feature engineering

$R_i^{Txn_f}$:Returns of stock i, each of dimension Txn_f

$D^{(Txn_f)x200} = \left[R_1^{Txn_f} \quad ... \quad R_{200}^{Txn_f}\right]$: Matrix containing each of the R_i vectors

Then, we use our meta-model on each entry of the D matrix, generating a $Tx200$ prediction matrix (a prediction for each time-step of each of the 200 vectors); Finally, the average of every row is calculated, leading us to the final vector of predicted values, like we would get by applying any other model. More concretely,

M : Best meta-model based on prior (nested) cross-validation on benchmark data

$$Y_{all\_preds}^{Tx200} = \left[M(R_1^{Txn_f}) \quad ... \quad M(R_{200}^{Txn_f})\right]$$

$$Y_{preds\_mean}^{Tx1} = \begin{bmatrix} (1/200) * \sum_{i=1}^{200} M(R_i^{Txn_f}) \\ ... \\ (1/200) * \sum_{i=1}^{200} M(R_i^{Txn_f}) \end{bmatrix}_{Tx1}$$

With $Y_{preds\_mean}^{Tx1}$ being our final prediction vector, used to calculate the various metrics. This process is repeated for both train and test sets, following the original division and using the same scaling, after all data preprocessing has been done.

Do note that the starting point for both these approaches is the exact same 'meta-model', the best performing algorithm in the R-squared metric after the (nested) cross-validation procedure and essentially only the application approach differs.

3. Empirical Results

3.1) Software

It should be noted that multiple libraries of the Python programming language were used in order to develop those frameworks and conduct the analysis: *statsmodels* was used for the ARMA models and various tests and plots; *sklearn* for the majority of the modelling process and for building all algorithms except the neural networks; *Keras* (with TensorFlow backend) was used to develop the neural networks; *pandas* to import and manipulate the data structures and perform various matrix computations; *matplotlib* and *seaborn* for the various visualizations; *Ta-Lib* for feature creation; and *numpy* for various linear algebra operations. The environment used was Anaconda's *Spyder* scientific IDE (Integrated Development Environment). Mendeley was used to format the bibliography.

3.2) Data and Metrics Preparation

The dataset used in the analysis was taken from Thomson Reuters' Datastream database, and it concerned the 200 largest S&P 500 corporations at 31/12/1979 on the basis of Market Capitalization. The same dataset was also used during the benchmark process, during the training and tuning of our meta-model.

More specifically, regarding the primary dataset, the two categories of data retrieved were the Total Return Index (TRI) of those 200 corporations, which tracks the capital gains of the asset assuming all its distributions are reinvested, thus giving a clearer picture of the asset's performance, and their Market Value (MV), both in daily frequency spanning the time period between 31/12/1979 up to 10/18/2018, for a total of around 38 years worth of daily data observations.

The Datastream data (which also doubles as our index data later) required some elementary engineering before any further computations were to be done. First, it included the "observations" in non-business days, such as New Year's Eve, as duplicated rows, which had to be removed else they would skew the analysis, particularly given the span of the data set.

Then, as is the convention in financial applications for reasons like stationarity or to avoid spurious correlations, the daily TRI prices were converted to daily returns. In addition, in order to avoid 'look ahead bias' before the portfolio is constructed, the MV rows were all shifted by one day so that the following would hold true for the total portfolio return at every given time t:

$$Pt = TRI_t * MV_{t-1} \forall t \quad \textbf{(29)}$$

As it is assumed that at time t would we only know the market value of the previous time point, expressed as t-1. TRI prices will be considered returns for the rest of the article.

In order to insulate the portfolio from any external, random fluctuations and make it develop solely on account of its TRI history, the following formulas were used to recompute an initial MV ratio, and then use its cumulative product with TRIs to build the new, insulated MV data matrix:

$$MV_{i,t-1} = MV_0 * \prod_{j=1}^{t-1}(1 + r_{i,j}) \quad \textbf{(30)}$$

$$MV_{t-1} = \frac{MV_{i,t-1}}{\sum_{i=1}^{N} MV_{i,t-1}} \quad \textbf{(31)}$$

Where i : stock, j : time step, MV_0 : Initial MV value (i.e the first row in the MV set after the preceding operations).

After we obtain our new MV matrix, it is simply a matter of applying equation **(29)** to get the cumulative returns at each time moment, by weighting the contribution of each of the 200 stocks to our portfolio via its market value, creating in essence a market value weighted portfolio, for all time stamps.

With our portfolio now ready, some further computations and feature engineering is conducted to prepare it for supervised learning. First, 90% of the initial portfolio time steps are used to calculate scaling statistics, with which to scale the entire initial, univariate dataset, from which the other features will also be calculated. That is as 90% is chosen to be the size of the training set, with the rest relegated to the test set, in order to avoid data leakage or 'look - ahead' bias by including (unseen) information from the future when scaling present values. More rigorously, the new "Z-values" we get back, and which are used for feature engineering, model training and predictions, are computed as follows:

$$Ztrain = \frac{X_{tr} - \bar{\mu}_{Xtr}}{\sigma(X_{tr})} \quad \textbf{(32)}$$

$$Ztest = \frac{X_{ts} - \bar{\mu}_{Xtr}}{\sigma(X_{tr})} \quad \textbf{(33)}$$

Scaling is performed as some of the algorithms that use euclidean distances in their calculations (like SVMs) or require data to lie in a certain interval for effective training (like neural networks) can perform in an optimal manner. Other benefits of scaling include faster training, for example with gradient descent, minimization of the probability of being stuck in local optima, better error surface and more effective weight decay, particularly helpful for neural networks.

The scaling is only inverted after predictions are made, in order to calculate the metrics with which the models' predictive ability is compared, such as R^2 (coefficient of determination) and Mean Squared Error (MSE), with the values in their original scales. Those metrics are defined as:

$$R^2 = \frac{\sum_{i=1}^{N}(y_i - \hat{y_i})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2} \quad \textbf{(34)}$$

$$MSE = \frac{1}{N} * \sum_{i=1}^{N}(y_i - \hat{y_i})^2 \quad \textbf{(35)}$$

In addition to those metrics, which are used to tune the models, we also utilize two additional metrics to gain a better picture of the algorithms' performance, not computed during training or cross-validation. These two metrics are Pearson's correlation coefficient and the Sharpe ratio, since the bottom line is to find the algorithm which provides the best return on our investment. Sharpe ratio is defined as follows:

$$\text{Sharpe}_{(\text{ratio})} = \frac{\mathrm{R_p} - \mathrm{R_f}}{\sigma_p} \overset{\mathrm{R_f}=0}{\Leftrightarrow} \quad \textbf{(36a)}$$

$$\text{Sharpe}_{(\text{ratio})} = \frac{\mathrm{R_p}}{\sigma_p} \quad \textbf{(36b)}$$

Where p denotes the returns and standard deviation of our portfolio, with the simplifying assumption that the risk-free return is zero.

As Sharpe ratio is a financial metric, often used in practice to compare investing strategies, it can give us a much clearer picture into what it would practically mean to the investor's balance were they to deploy each model into their decision making, thereby making it a very interesting metric to compare our tuned models' predictions with each other.

The non algorithm specific methodology to calculate the ratio, in both train and test sets, after the models have been finalized, is the following:

1. Saved all models' train and test prediction vectors as-is: $Y_{train}^{preds}, Y_{test}^{preds} \forall \text{model}$; Also saved the targets used during modelling process.
2. Create a 'Mask' for each prediction vector:
   For value in prediction vector:
   If value ≥ 0: value = 1
   Else-if value <0: value = -1

For example, if $Y^{pred} = \begin{bmatrix} -0.5 \\ 0.3 \\ 0.02 \end{bmatrix} \overset{Mask}{\Rightarrow} Y_{masked}^{pred} = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$,

Where each element in the vector is a daily prediction.

3. Then perform element-wise multiplication of the masked vectors with the actual targets, in order to translate the predictions to an actual strategy: When the model predicts the portfolio's value will fall (a negative prediction, given it is a regression problem) we want to capitalize on that by selling. Thus, if it does fall, meaning a negative target, the product will be a net positive, thus incurring a profit. Same goes

for when the model predicts a positive value, which is when we want to buy, so that the product between target (positive) and decision again is positive, or profitable. That serves to simulate long and short positions we would take in the actual market, were we to follow those suggestions. For example,

$$Y_{masked}^{pred} = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} \otimes Y^{actual} = \begin{bmatrix} -0.23 \\ -0.5 \\ 0.45 \end{bmatrix} \Rightarrow R_{return}^{realized} = \begin{bmatrix} 0.23 \\ -0.5 \\ 0.45 \end{bmatrix}$$

4. Calculate the mean return (μ) and standard deviation (σ) of each such R vector for each model for both sets (so a total of 20 such vectors).
5. Then calculate the daily realized Sharpe ratio = μ/σ for each model in both sets, and report that as the final metric.

The process for calculating the correlation coefficient was much simpler, as all that was required was to calculate the correlation between each prediction vector with the target vector, again for both sets and for each model. Note that for the plots the MSEs were multiplied by a constant of 1000 for better visualization, and that the graphs have been 'zoomed in' to allow for easy inspection.

Following the scaling, we proceed to enriching our feature space by adding a total of 10 new predictor variables: Two lags of the original series, and the 14, 30, 50 and 200 - day Simple Moving Average (SMA) and Relative Strength Index (RSI), calculated from the original series. Those indicators are defined as:

$$SMA = (1/T) * \sum_{t=1}^{T} TRI_t \quad \textbf{(37)}$$

$$RSI = 100 - \frac{100}{1 + RS} \quad \textbf{(38)}$$

Where RS= Average Gain over "n" period times / Average Loss over "n" period times, where n can be defined from the user (here defined for the aforementioned 4 periods).

Having done those operations, we end up having 10 predictor variables which we feed into the models: Our 2 lags and the 2 technical indicators sampled in 4 different frequencies. Note that the RSIs are also divided by 100 in order to bring them to (roughly) the same range as the others, otherwise they could be given disproportionate weight in some algorithms or slow down the training.

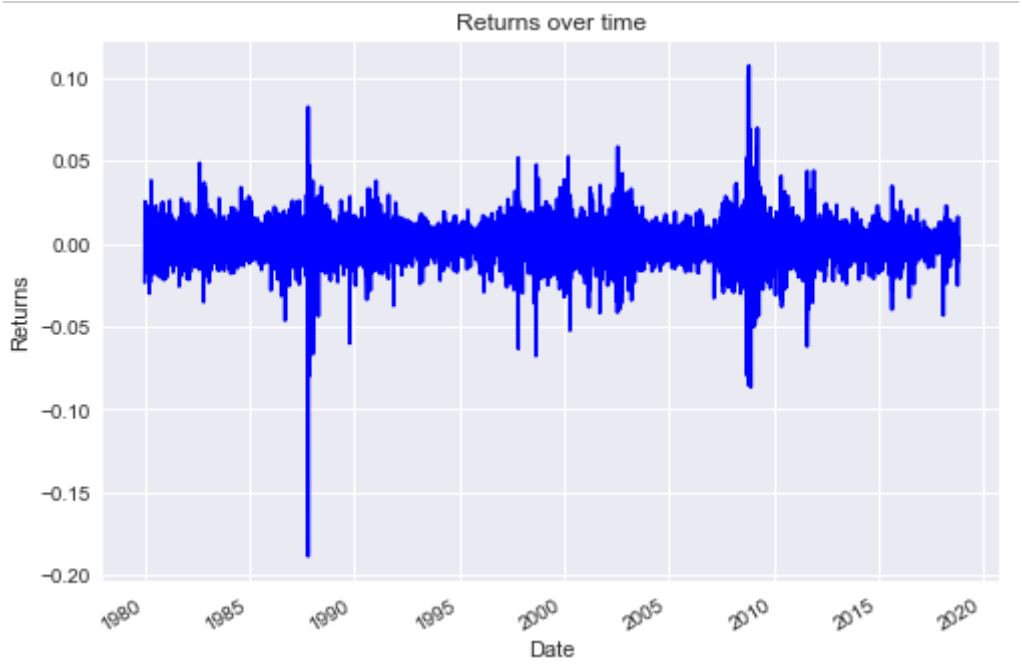
In addition, all NA values created during any stage of the preprocessing pipeline were dropped at each stage prior to the next step, because they do not offer anything to the analysis and to avoid any numerical problems.

3.3) Exploratory Data Analysis (EDA)

We begin with some basic exploratory data analysis (EDA), an integral part of any machine learning project as it allows us a glimpse into the state of our data. First, we begin by simply plotting the original portfolio series, that is before any scaling or feature engineering is conducted, but after it has been modified to only include returns,

since in finance it is customary to only work with returns, as we will also do for the course of the entire discussion.

Figure 1:Portfolio daily returns over time



The differenced series seems to be mostly mean stationary, but with some very noticeable spikes during 1990 and 2010. Variance also seems to not remain consistent over times, but it does appear in clusters, as it is higher around the spikes and lower around calmer times.

However, the graph is mainly useful to provide some intuition about the nature of the data, hence we employ two statistical methodologies to more formally test for stationarity and unit roots, mainly the Augmented Dickey-Fuller and and the KPSS test. The Ho hypothesis for the ADF is that there is a unit root, while the Ho hypothesis for the KPSS is that the data is stationary. With that in mind, here are the tests on our (differenced) series:

Table 1:	
Results of ADF Test	
Test Statistic	-24.700
p-value	0.00
Lags Used	17.00
Critical Value (1%)	-3.43
Critical Value (5%)	-2.86
Critical Value (10%)	-2.56

Table 2:	
Results of KPSS Test	
Test Statistic	0.296
p-value	0.1
Lags Used	39.00
Critical Value (1%)	0.739
Critical Value (5%)	0.463
Critical Value (10%)	0.347

The p-values in ADF test indicate the in our series we reject the Null Hypothesis that there is a unit root, in favor of the alternative. In the KPSS, with a p-value of 10%, we may not reject the Null hypothesis that our data is stationary, in favor of the alternative. Thus, those two tests concur that probably the first order difference has indeed turned our series into a stationary process, making it significantly easier to work with.

Another interesting thing to look at is the distribution of our series, and how close it is to a normal distribution, as it can have important consequences for the modelling process. To that end we will start by employing a simple histogram with 50 bins, and then a kernel density estimate graph (KDE) for a smoother inspection:

Figure 2:Histogram of Portfolio returns

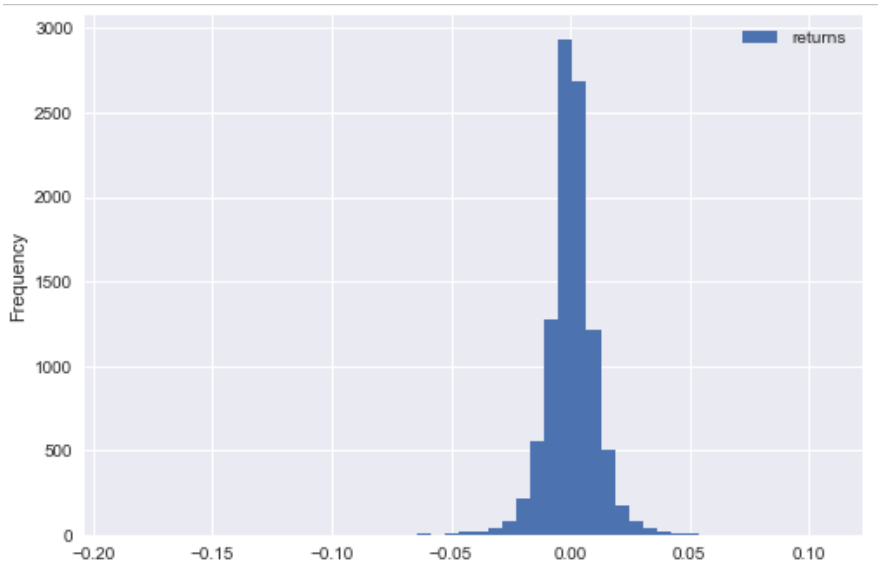
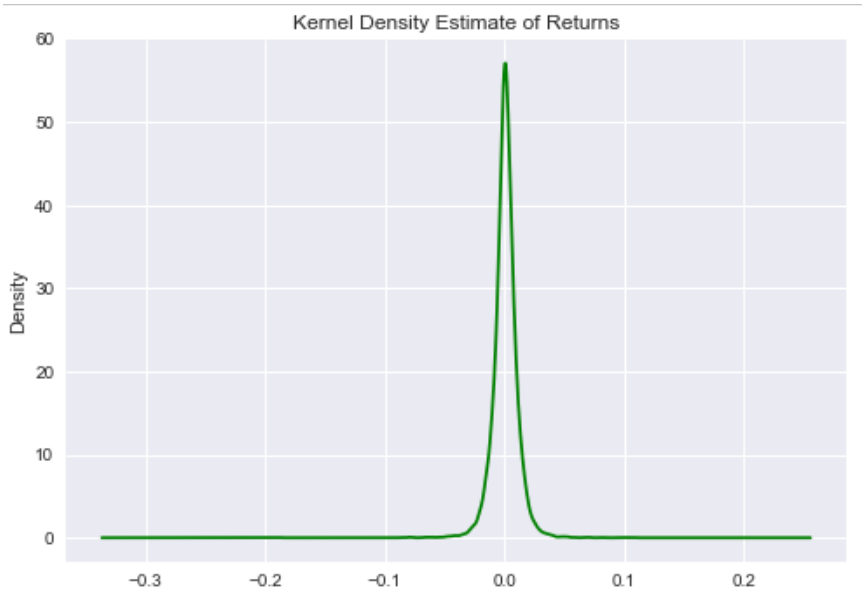


Figure 3:KDE Estimation of Portfolio returns



Both graphs show that our data is strongly centered around a mean of zero, with not too many outliers, resembling a contorted normal distribution around 0 which raises the need for some additional testing. Here we employ the QQ Plot and the Jarque-Bera test (a type of Langrange Multiplier test), which is particularly well suited for large data sets such as ours:

Figure 4:QQ Plot of returns

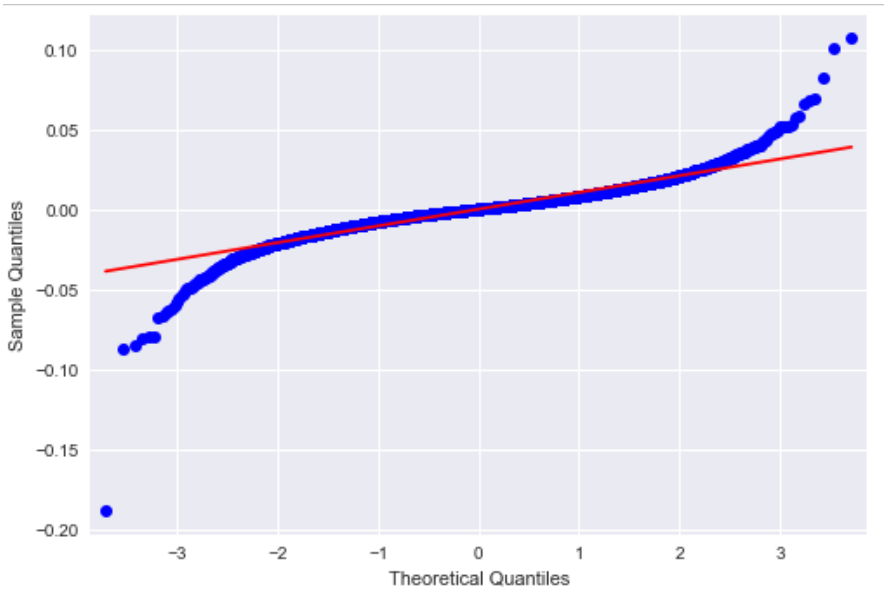


Table 3:
Results of Jarque-Bera Test
Test Statistic: 140763.32,
p-value :0.0,
skewness:-0.69,
kurtosis:21.45

Given the fact that the JB test's Ho hypothesis is that the data is normally distributed, we can easily reject Ho in favor of H1, due to a p-value equal to zero, and a huge t-statistic. That, in conjunction with the QQ plot visual result, indicate that our distribution is not normal, as is often one basic assumption in most standard econometric models, which makes the use of more advanced models imperative.

One other interesting plot, which would describe the effectiveness of the portfolio in the financial sense, is the plotting of the Holding Period Return (HPR) over time. HPR is simply defined as the total return of the instrument over the period it is held, and could be thought of as the sum of the investment's appreciation and its income, like dividends paid, so it offers a comprehensive image of its aggregate performance. The mathematical formulation of HPR is:

$$HPR = \frac{Income + V_n - V_0}{V_0} \Leftrightarrow$$

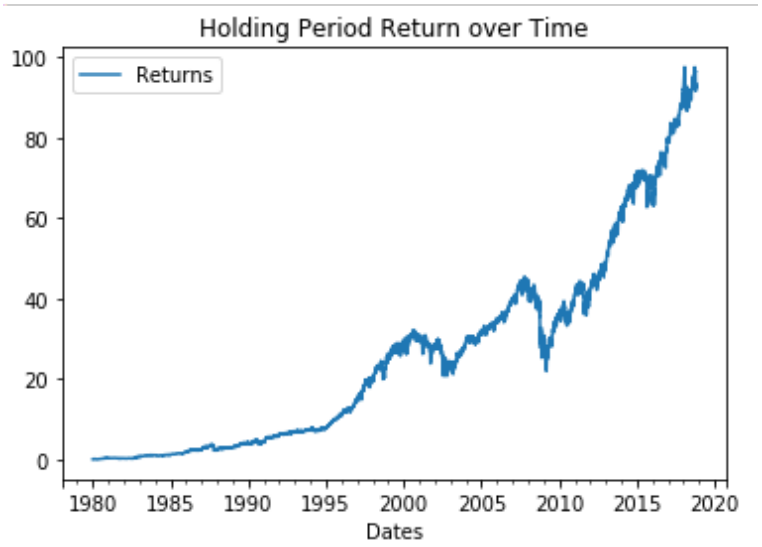
$$HPR_{(2)} = [(1+r_1)(1+r_2)...(1+r_n)] - 1 \quad \textbf{(39)}$$

Where,
Income: The distribution of the proceedings of the asset, like dividends paid;
Vn: The final value of the investment
Vo: The initial value of the investment
r: % return over a given period
n: number of periods

The second formula **(39)** was practically used to create the following plot, which represents the HPR of our market value weighted portfolio of the 200 largest (by market capitalization) stocks of S&P 500 index at 1/1/1980. Empirically, it shows that buying and holding those stocks in that manner for that period of around 40 years was

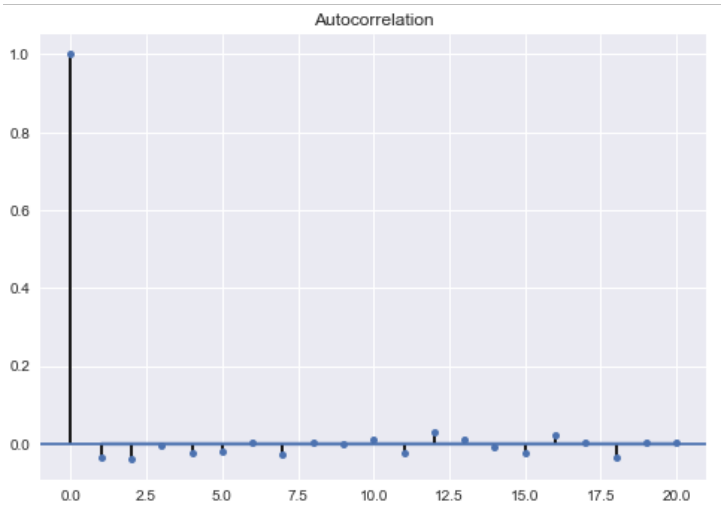
indeed a very profitable investment, earning an HPR of up to 100, with a major fall only around the time the great financial crisis of 2008 occurred, and climbing relatively unimpeded ever since.

Figure 5: Portfolio HPR over time



We will also take a look at the autocorrelation graphs for our portfolio series, which can help to not only gain some intuition on the presence of auto-correlation within the time series, but also indicate what orders in ARMA would be sensible to start with.

Figure 6:Portfolio autocorrelation graph

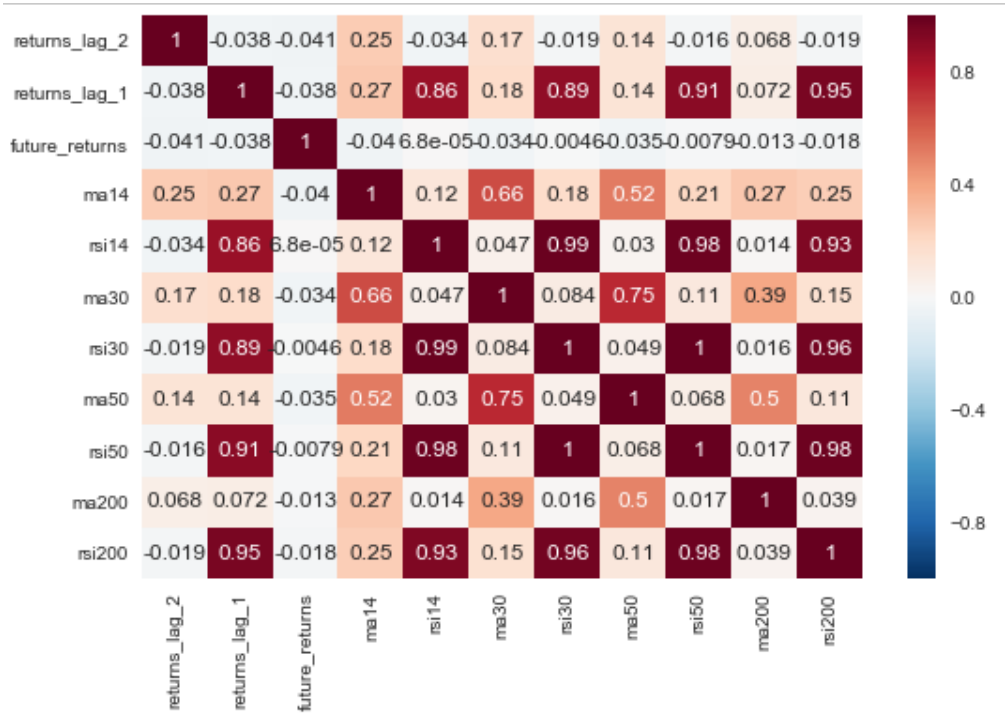


The AC graph indicates that there is no statistically significant relationship of autocorrelation within the portfolio's returns, and implies that the process is more or less a random walk process, hence it does not provide any guidance as to what orders of ARMA would be suitable for forecasting.

After we conduct our feature engineering, that is adding our various new features just prior to utilizing them for predictions, a very useful way to discover (linear)

relationships within the data is to calculate the correlation matrices and visualize them via a tool called a 'heatmap' for easy and quick inspection:

Figure 7: Feature Correlations' Heatmap



'future_returns' are the values we are trying to predict, and its lagged variants are dubbed 'returns_lag_x'. The results are rather predictable, as the non lagged features are slightly to moderately correlated with one another, which is expected given they both represent momentum indicators of differing time scales. It is also apparent that the target variable has almost no correlation with anything else, at the very least linear, which is indicative that the features may indeed be irrelevant and the process inherently unpredictable, or that any relationships are nonlinear, again indicating that only nonlinear models may be able to extract something of value.

3.4) Training and Testing of Predictive Models for a Market Index

3.4.1) Models using the Index's History and History of individual stocks

Initially, we will summarily refer to the model's results mainly in regards to the optimization criterion of R-squared, along with highlighting each algorithm's special side analysis, before concluding with the discussion and comparison along all four metrics used, in conjunction with various benchmarks.

It is important to note that the training and tuning of the models was conducted via MSE, with the intent to minimize it, as we are now predominantly interested in maximizing predictive ability and not so much interpretability. However, we will initially only mention performance as measured by R-squared and postpone the

discussion for all other metrics in the detailed results subsection, as it is a more intuitive metric and offers slightly more interesting results than its counterpart.

We begin the implementation by applying Multiple Linear Regression. To demonstrate the best line fit visually, a scatter plot of the data, both the train (in blue) and the test (red) with the regression line superimposed follows:

Figure 8: Linear Regression: Line of best fit

The distribution of the data seems rather nonsystematic, and it is clear that a linear model cannot possibly get any good forecasts out of that highly nonlinear feature space, which however does not mean that nonlinear models would do significantly better due to the lack of, at least a visible, pattern in the data.

The OLS's weakness is exemplified by the very low score in the metrics of R-squared, which for the in-sample prediction lies around 0.008, and for out-of-sample forecasts is even lower, but still not negative. More specifically, the coefficients (standardized, as the training is done in scaled data) along with their standard errors in parentheses, and the p-values of the coefficients are as follows:

Table 4:

Features	Coefficients
$returns_{t-1}$	-0.2047 (0.038)
$returns_{t-2}$	0.0381 (0.011)
ma14	0.0008 (0.067)
rsi14	9.2034 (5.326)

ma30	0.0269 (0.12)
rsi30	38.06 (33.19)
ma50	-0.1893 (0.149)
rsi50	42.87 (47.12)
ma200	0.1644 (0.20)
rsi200	28.139 (46.23)
constant	-21.081 (15.19)
R-squared (in-sample)	0.008
Obs.	8698
R-Squared (out-sample)	0.002

$Variable$	$P>\|t\|$
$returns_{t-1}$	0.000
$returns_{t-2}$	0.001
$ma14$	0.991
$rsi14$	0.084
$ma30$:	0.822
$rsi30$	0.262
$ma50$	0.205
$rsi50$	0.363
$ma200$	0.428
$rsi200$	0.543

These p-values indicate that only our two lagged return values are statistically significant, and the rest of the features are irrelevant in a linear setting.

It is interesting to see the effect of regularization to linear regression's out of sample performance. To that end, we move on to applying Elastic Regression, which combines L1 and L2 regularization. Elastic regression has two hyperparameters we grid search over, 'alpha' and 'l1_ratio'. Alpha is a constant multiplying the penalty terms, while l1_ratio is the penalty mixing parameter, which are involved in the elastic regression equation as such:

$$\min_w \frac{1}{2N}\sum_{i=1}^{N}\|y - Xw\|_2^2 + \alpha * l1\_ratio\|w\|_1 + 0.5 * \alpha * (1 - l1\_ratio) * \|w\|_2^2 \quad \textbf{(40)}$$

If $l1\,\text{ratio} = 0 \Rightarrow$ then the penalty is L2 only;
Else if $l1\,\text{ratio} = 1 \Rightarrow$ the penalty is L1 only;
Else it is a mix of those two, which is where we will restrict our grid search to.

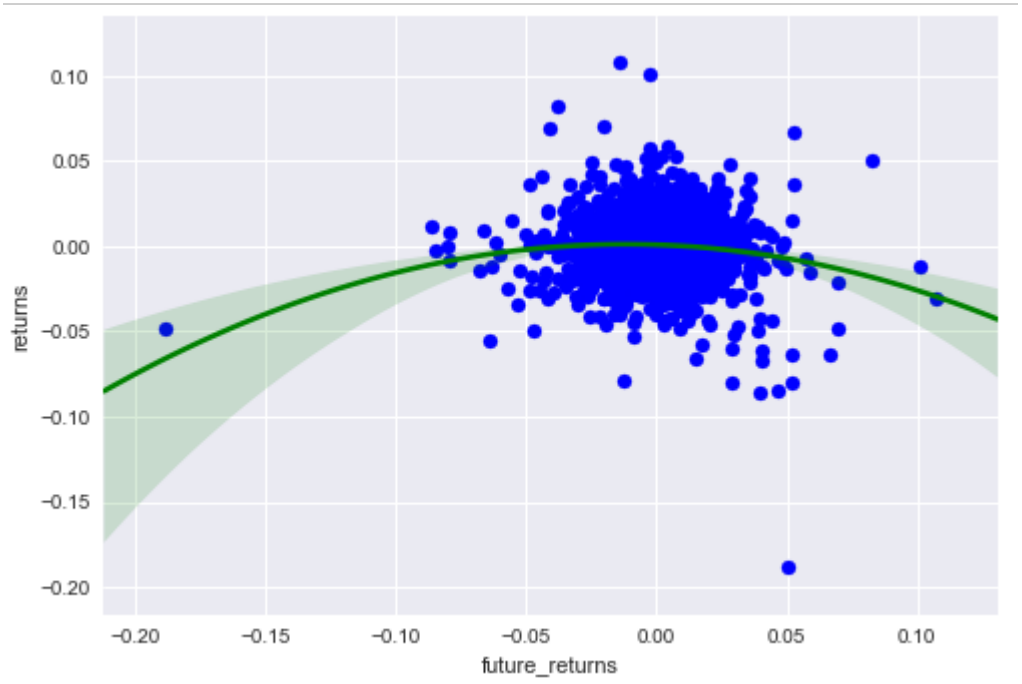
Elastic regression, via tscv grid search of the aforementioned two parameters achieved a worse out-of-sample R-squared than linear regression, and an almost identical value on the training data:

$R^2_{in-sample} \approx 0.00089$

$R^2_{out-sample} \approx 0.0006$

Given those obvious nonlinearities, it is an interesting question how a polynomial regression would fit to the data. We fit polynomials of both the 2nd and the 3rd degree, and find that the second degree one offers much better out-of-sample performance as regards R-squared compared to the third, although still diving into negative territory. More detailed discussion of the performance metrics will follow once the cumulative performance plots are shown, and for now the best 2nd degree fit line is shown:

Figure 9: Polynomial (degree=2) line of best fit



What we can see is that even though it may seem to be more capable at capturing some nonlinearities than its purely linear cousin, the very nature of the data prevents it from being able to effectively capture any existing patterns, resulting in a subpar performance.

What about the performance of a family of models specifically created to deal with time series, our ARMA(p,q) models? After some grid search of ARMA 'hyperparameters', here the p and q values, we come to the conclusion that the AR(2) achieved the best performance by AIC, BIC, R-squared and MSE metrics, closely followed by ARMA(1,1). The coefficients, along with their standard errors are reported below. Note that p-values for both of them and the constant are zero, so they are all statistically significant, a parameter also received into consideration when deciding what model of this family was the best:

$$\hat{returns}_t = \underset{(5.61e-07)}{0.0001} - \underset{(0.005)}{0.0381} returns_{t-1} - \underset{(0.004)}{0.0377} returns_{t-2} \quad \textbf{(41)}$$

$$R^2_{in-sample} \approx 0.0005,$$

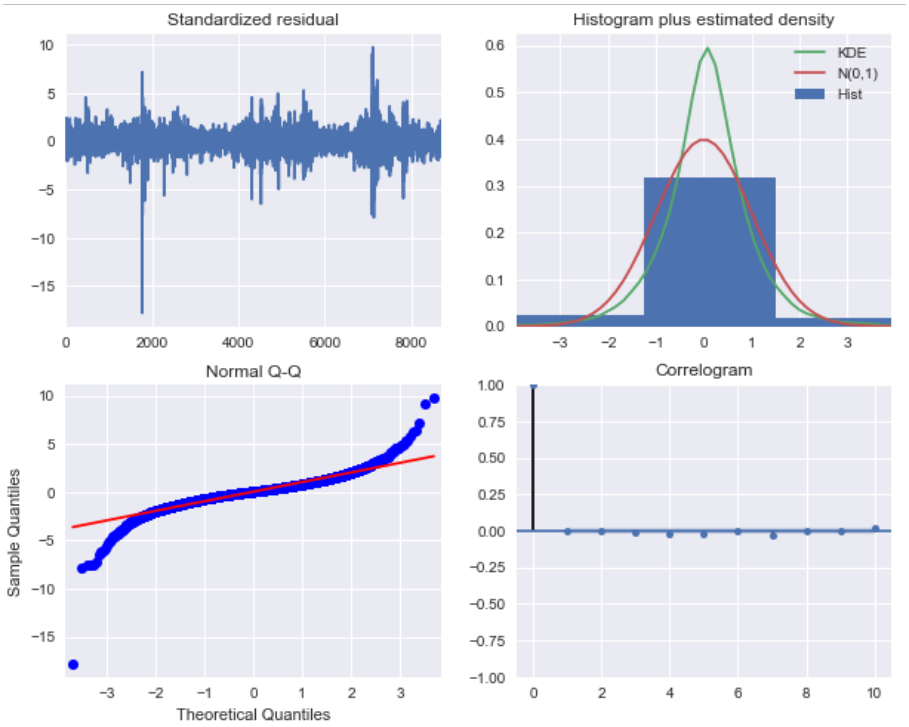
$$R^2_{out-sample} \approx 0.00016$$

Note that for the very long out-of-sample forecasting period that was conducted so that the model would be comparably measured against the other models, consisting of 967 data points, eventually the forecast converged to the long-term mean of the series, which is evident of those linear models' inability to handle long prediction horizons. Note how in the prediction graph for the last 30 in-sample predictions and first 10 out-of sample ones how quickly the prediction line flattens:

Figure 10: AR(2) line of best fit, in- and out- of sample

While even during in-sample prediction it is evident it cannot keep up with the daily, seemingly random, fluctuations of the returns. That inability to fully model the data is also made evident by the diagnostics of the fitted model:

Figure 11: Diagnostics Graphs: Residuals, Histogram, QQ plot and ACF



Where both the QQ Plot and the KDE plots indicate a less than perfect fit: The former implies the samples do not follow the standard normal trend, indicated by the red line, only to be confirmed by the latter as the residuals do not align with the theoretical standard normal distribution, indicating there could be a better fit, perhaps with a more sophisticated model.

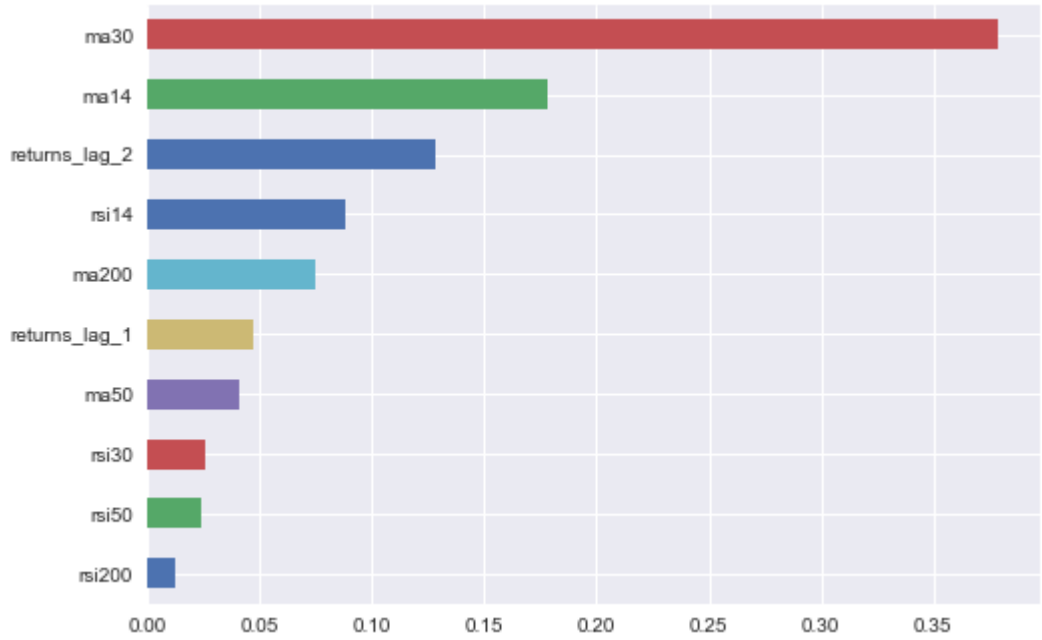
The next algorithm, and one with good empirical performance in dealing with nonlinear data, is the Random Forest. Essentially, it is an ensemble of individual decision trees (CARTs) applied on various subsamples of the training data, with their predictions' averaged to simultaneously improve forecasting accuracy and control overfitting.

The hyperparameter grid that was specified here for tscv consisted of the following three parameters: the number of estimators, or the number of individual trees used in every iteration; The minimum samples required to be at any leaf node for the split to be allowed; The maximum depth of the tree, which should be monitored to avoid detrimental overfitting. One other thing that should be mentioned is that tree based algorithms, due to the way they are built, do not care much for feature scaling and will deliver the same results regardless of any feature engineering done, which is another key benefit of them, compared to say SVMs or Neural Networks which can suffer a dramatic loss of predictive ability if faced with not appropriately designed data.

Decision trees, and by extend random forests, can enable us to measure the relative importance of each feature during prediction, which can be though of as how much the tree nodes use a particular feature to reduce impurity (or maximize information

gain). The higher that weight is, the more useful its feature was while building the tree, and the higher its relative importance compared to the others. Here we create one such bar chart, sorted in descending order:

Figure 12:Random Forest feature importances



The graph is interesting as it depicts the short term moving average indicators as by far the most important features, indicating that short-term momentum can indeed play a part in forecasting. It is also interesting as it gives more weight to the second lagged return than the more recent one. It also seems as most medium to long term metrics were not considered especially important as well.

Random forest performs rather well compared to the previous two models, but with still very low R-squared values:

$$R^2_{out-sample} \approx R^2_{in-sample} \approx 0.002$$

Moving on to the SVM, here too we execute a hyperparameter grid search, with three components: The Kernel (linear or Radial Basis Function); Gamma, the Kernel coefficient; C, the penalty parameter. The last two are only useful when utilizing the Gaussian RBF Kernel, defined as

$$K(x_i, x_j) = \exp(-\gamma \left\| x_i - x_j \right\|^2), \quad \textbf{(42)}$$

$$\gamma = \frac{1}{2\sigma^2} > 0$$

Thus gamma is a free parameter, which essentially decides how much influence the support vector x_i has in the classification of the training sample x_j. A large gamma can be thought to lead to low bias - high variance models, and vice versa. The penalty parameter C essentially is used to decide how smooth the decision surface should be,

with smaller values leading to smoother decision planes, which means smaller penalization of the slack variables introduced to simplify the initial optimization problem, or allowing higher leniency to errors in order to avoid over complicated decision boundaries.

After the grid search is performed, we end up favoring the RBF kernel, yet still get very low values for the R-squared metric, falling into slightly negative territory for out-of-sample predictions:

$$R^2_{in-sample} \approx 0.028$$

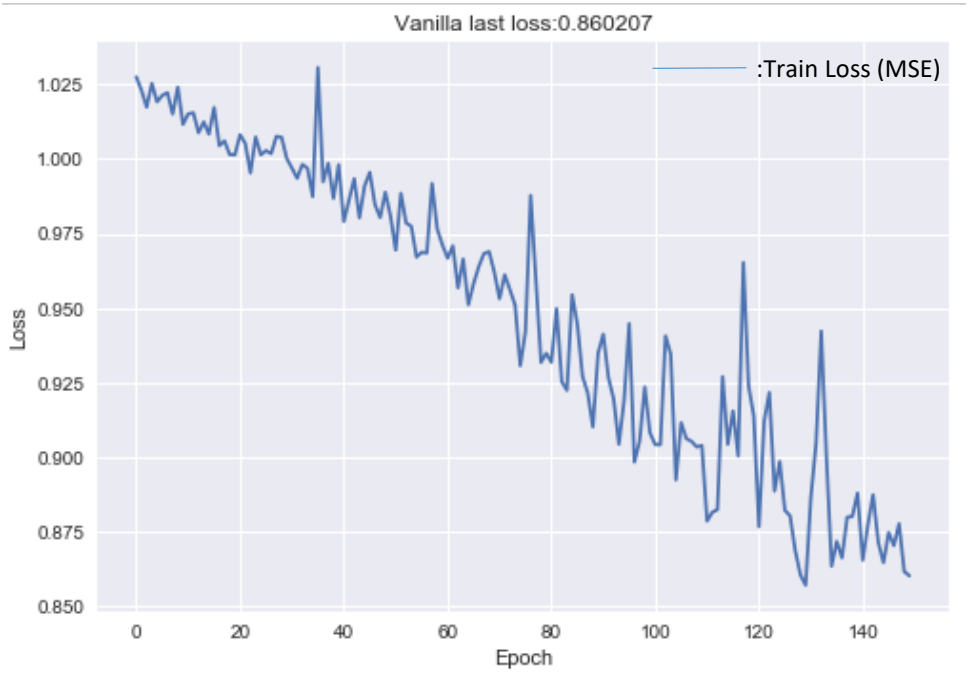
$$R^2_{out-sample} \approx -0.01$$

Finally, the more complicated, oftentimes criticized as black box processes, neural networks are tested. We first evaluate the MLP. The architecture simply consists of the densely connected layers for the learning part, along with two dropout layers for efficient regularization:

Layer (type)	Param #	**(Table 5: MLP)**
Dense (n=100)	1100	
Dropout (dr=0.3)	0	
Dense (n=150)	15150	
Dense (n=50)	7550	
Dropout (dr=0.1)	0	
Dense (n=1)	51	
Total params: 23,851, n:neurons, dr=dropout rate		

Judging by the loss graph, where loss here is equivalent to the MSE metric, we can see that although it constantly declines as the training goes it, it does so in a very noisy manner and with wide fluctuations, which could signal that either a better choice of hyperparameters is needed, thus much more extensive tuning, or that the data itself resembles a random walk thus is it impossible for any algorithm to effectively learn its structure.

Figure 13:MLP Loss History (Training)



The metrics also reveal that indeed its predictive ability is not particularly high, with a still negative out-of-sample R-squared:

$$R^2_{in-sample} \approx 0.00355$$
$$R^2_{out-sample} \approx -0.0951$$

The next variant applied is the 1D (1 Dimensional) Convolutional Neural network. Its exercise specific architecture can be summed up in the following board:

Layer (type)	Param #	**(Table 6: CNN)**
Conv1D (f=64)	1344	
MaxPooling	0	
Dropout (dr=0.25)	0	
Conv1D (f=64)	8256	
Dropout (dr=0.15)	0	
Flatten	0	
Dense (n=100)	6500	
Dense (n=100)	5050	
Dense	51	

Total params: 21,201, n:neurons, dr=dropout rate, f=filters

It uses the 1D convolutional layer, the 1D max pooling for feature extraction, the dropout for regularization, and the densely connected layers as per customary at the end for outputting the final predictions. The loss graph is very similar to the previous one, indicating the aforementioned two potential problems and the same progress during training.

Figure 14:CNN Loss History (Training)

What is interesting though is the very large disparity between the in- and out- of sample coefficients of determination: While the out-of-sample one lies in negative territory, the in-sample one is very quite high, particularly compared with that of all preceding models:

$R^2_{in-sample} \approx 0.2606$

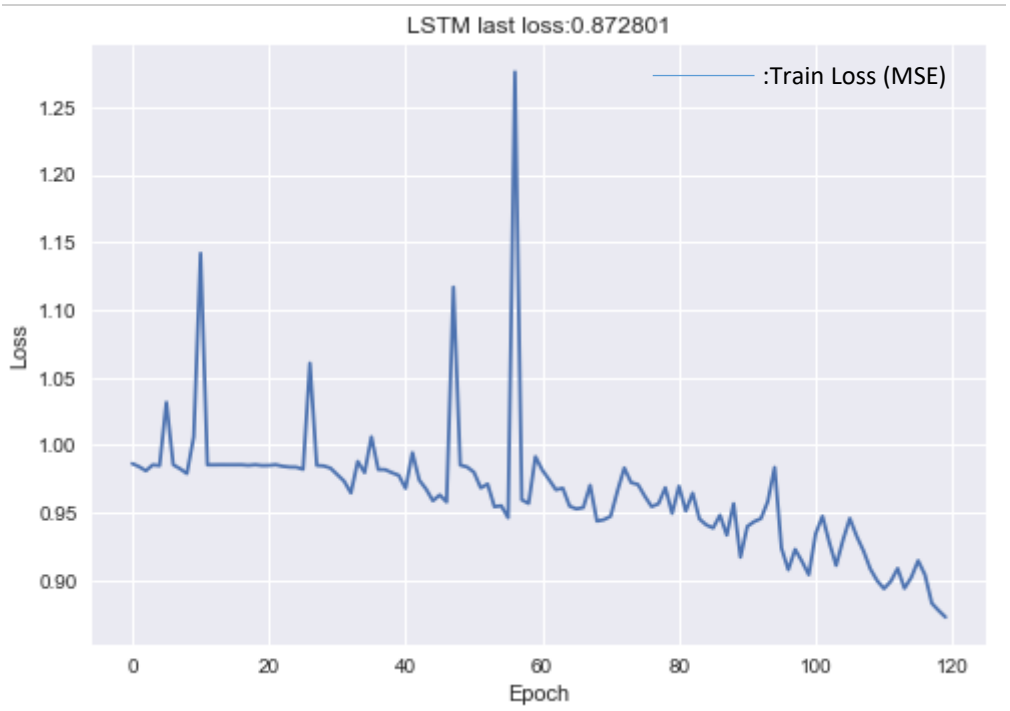
$R^2_{out-sample} \approx -0.1162$

The penultimate model is the LSTM, and its architecture is summarized as follows:

Layer (type)	Param #	**(Table 7: LSTM)**
Bidirectional LSTM (n=50)	24400	
LSTM (n=100)	80400	
Dropout (dr=0.25)	0	
LSTM (n=50)	30200	
Dense (n=100)	5100	
Dropout (dr=0.25)	0	
Dense (n=1)	101	
Total params: 140,201, n:neurons, dr=dropout rate		

The first layer utilizes a Bidirectional LSTM cell, in order to take advantage of any interrelationships both forward and backward in time, while the other layers are a combination of vanilla LSTM cells, densely connected layers and dropout layers. Below follows the graph of its loss history, which also is very similar to the previous ones, perhaps indicative of the inherent randomness of the data preventing a smooth convergence:

Figure 15:LSTM Loss History (Training)



This randomness is particularly evident here given the very high loss spikes at seemingly random moments, when one would expect the error would simply drop off in a rather predictable fashion. Despite this, the RNN achieved a positive out-of-sample R-squared, and a rather high in-sample on too:

$$R^2_{in-sample} \approx 0.1266$$

$$R^2_{out-sample} \approx 0.00386$$

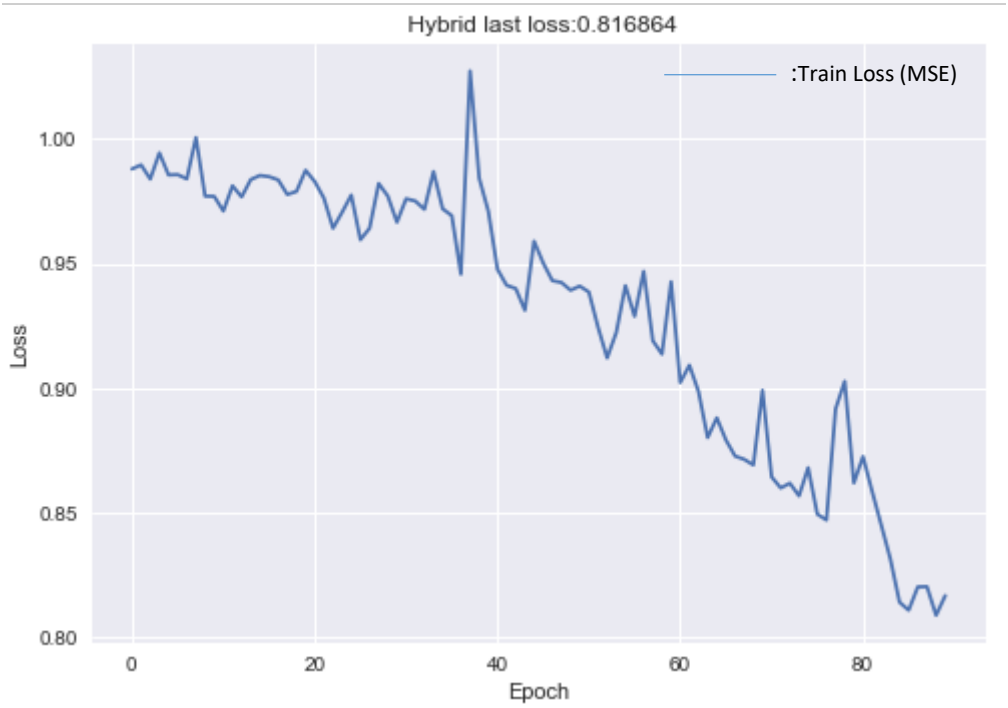
The final model is our RNN-CNN Hybrid. It is a computationally expensive model, due to its complexity, thus it is trained with the least amount of epochs of all neural networks. Notice the jump in the number of (trainable) parameters from the LSTM, the second most extensive network, is over two-fold:

Layer (type)	Param #	(**Table 8: LSTM-CNN)**
Conv1D (f=256)	5376	
MaxPooling	0	
Dropout (dr=0.1)	0	
LSTM (n=100)	142800	
LSTM (n=100)	80400	
LSTM (n=100)	80400	
Dropout (dr=0.15)	0	
Dense	101	

Total params: 309,077, n:neurons, dr=dropout rate, f=filters

This architecture first utilizes the CNN for feature extraction, in conjunction with the MaxPooling operation, the output of which is directly then fed to a 3-layer deep LSTM network for dissection, with a densely connected layer to output the final prediction, and dropout layers for regularization.

Figure 16:Hybrid Loss History (Training)



The hybrid nature of this network is also evident by its loss graph, which combines the obvious, rapid downward movement of the loss observed in the graph of the CNN, while also suffering from the occasional loss spikes, observed in the LSTM graph, albeit less extreme and frequent. Nevertheless, at least judging by R-squared, it does

not achieve as high performance out-of-sample as the vanilla LSTM network, while achieving a higher in-sample result than LSTM, but lower than the CNN. In short, its performance lies somewhere between the average of its constituent members, at least regarding that metric.

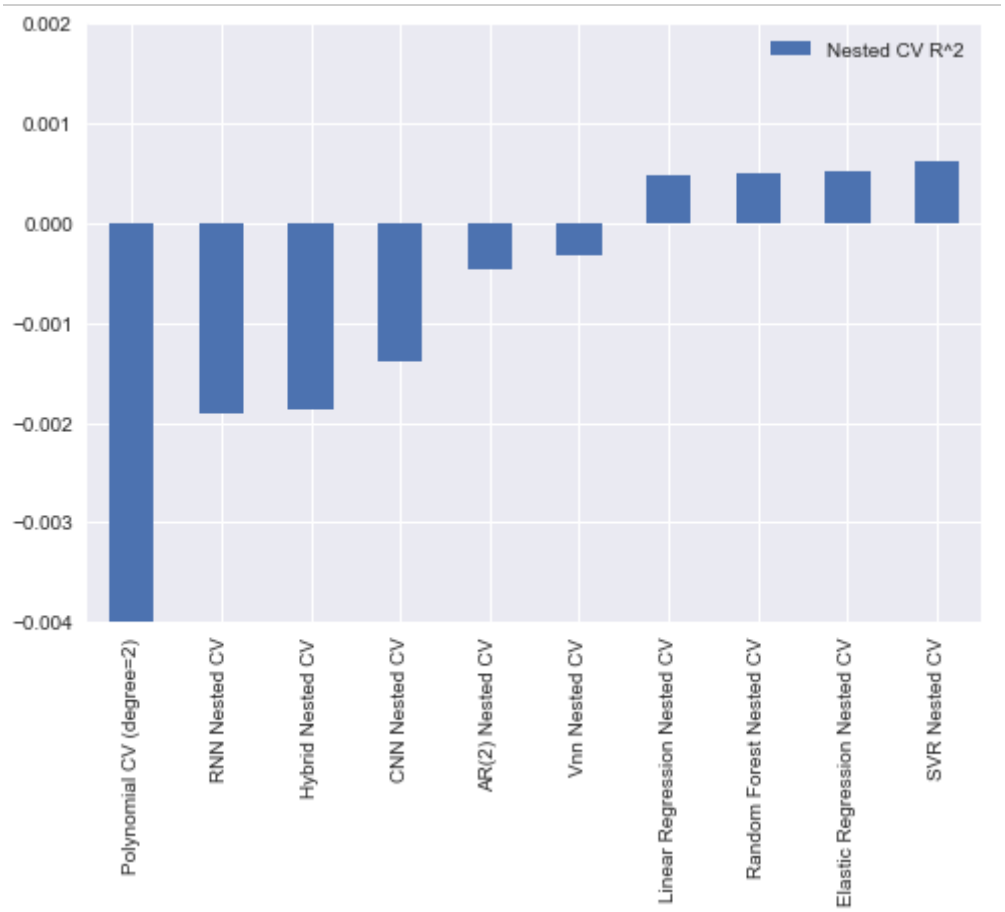
$R^2_{in-sample} \approx 0.18959$

$R^2_{out-sample} \approx$ -0.0572

Nevertheless, R-squared is not a particularly suitable metric for nonlinear models, nor does it necessarily tells us if the underlying model is actually good at what it does. In addition, it is a metric mostly suited for rating model's ability to explain the data and not so much about its ability to predict, which is what we are principally interested in.

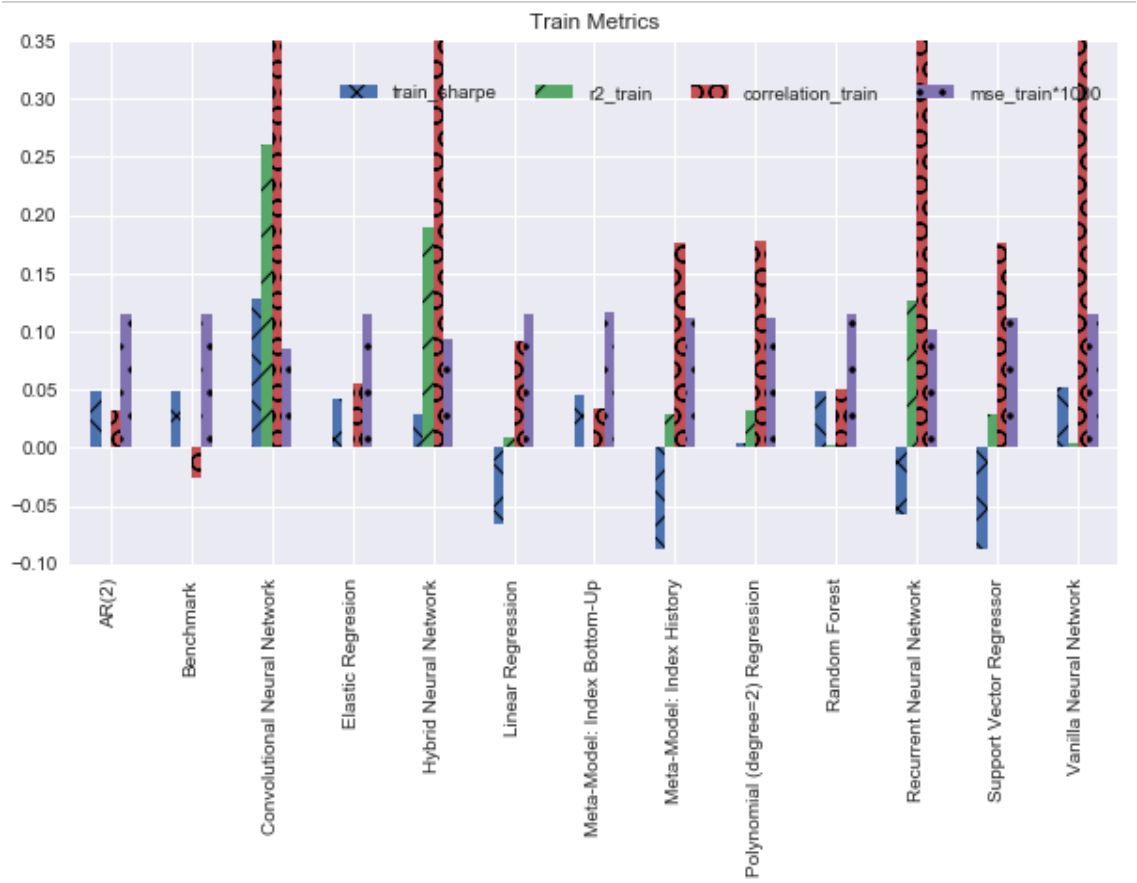
In addition,the empirical analysis will have to conclude by attempting to respond to the question of what exactly, at the end of the day, constitutes a 'good' model? To answer that, the establishment of a benchmark is necessary, with which the rest of our working models can be measured with. Usually, an 'index portfolio' could be used to that end, which simply is the index itself. However, we go further and establish two kinds of benchmarks: A basic AR(0), essentially a constant, 'always-long' model, to establish a quick baseline; And our 'meta-model' mentioned earlier, employed both directly to the Index and to the Index's history of individual stocks. The selection process crowns the SVM as the winner, achieving a slightly higher score:

Figure 17: Meta-Model Selection results



Therefore, to alleviate both aforementioned concerns, the following two graphs will plot not only the R-squared of all our models, but also 3 other useful metrics: MSE, Correlation and daily Sharpe ratio, which have been computed as described in the preceding 'Data Preparation' section, including the benchmark and the meta-models:

Figure 18: Train set results



What does that graph reveal?

First, regarding our primary tuning metric, the MSE, we observe that almost every single model outputs roughly the same number, with the CNN being slightly superior. This may be a confirmation of the suspicion we had since the EDA, namely of the rather random nature of the data, which disallows any model to effectively learn it, thus added complexity should not be equated with improved forecasting ability. This is further confirmed by the fact that all models' MSE is very close to that of all our three benchmark models, indicating that they in fact do fail to do any better.

Regarding Pearson's correlation, an interesting pattern emerges: The more complicated models, here the neural networks, all seem to have a very high coefficient, especially relevant to the benchmarks. That, in conjunction with the rather low MSEs and given the high R-squared values of the CNN, RNN and Hybrid networks could imply they overfitted the data, in spite of the dropout regularization they were subject to and their relatively mild size. If that is true, then those model's performance would drop quite substantially when employed on out-of sample data.

The story the meta-models let on, regarding that metric, is also interesting: It seems like predicting the Index directly is a much more effective approach than attempting to individually predict its constituents. This could indicate that the autocorrelation of the values of the index is a stronger one that its relation to its comprising stocks across that time period.

When it comes to Sharpe rations, we see that they are generally close to zero or negative, with the notable exception of the CNN, which has an impressively high Sharpe compared to its competitors and the benchmarks, indicating that it may be a promising model. The rest of the models however hover around the benchmark's ratio, indicating again that they do not in fact offer much value.

Again, the meta-models offer themselves for a very interesting story. Here we see the reverse pattern that what was the case with correlation, where following the history of the index results in a significantly poorer financial outcome that taking the average of its constituents. It could be owing to the fact that individual stocks are quicker to respond to price movements, thus when modelling on their level, it is easier to capture the right direction, thus earning higher Sharpe ratios, though that is not definitive.

It is an oddity that the RNN, even though it seems to do well in its predictions as measured by R-squared and correlation, has a very bad Sharpe ratio and is included among the three worse models in that regard (the two other being linear regression and SVM). An explanation for this could be that when the RNN outputs a value in the correct direction of the target (for example a positive value when the target is positive) it is closer in distance to the target, thus increasing R-squared, yet often fails to predict in the correct direction, thus suggesting bad tactics. Below follows the breakdown of results:

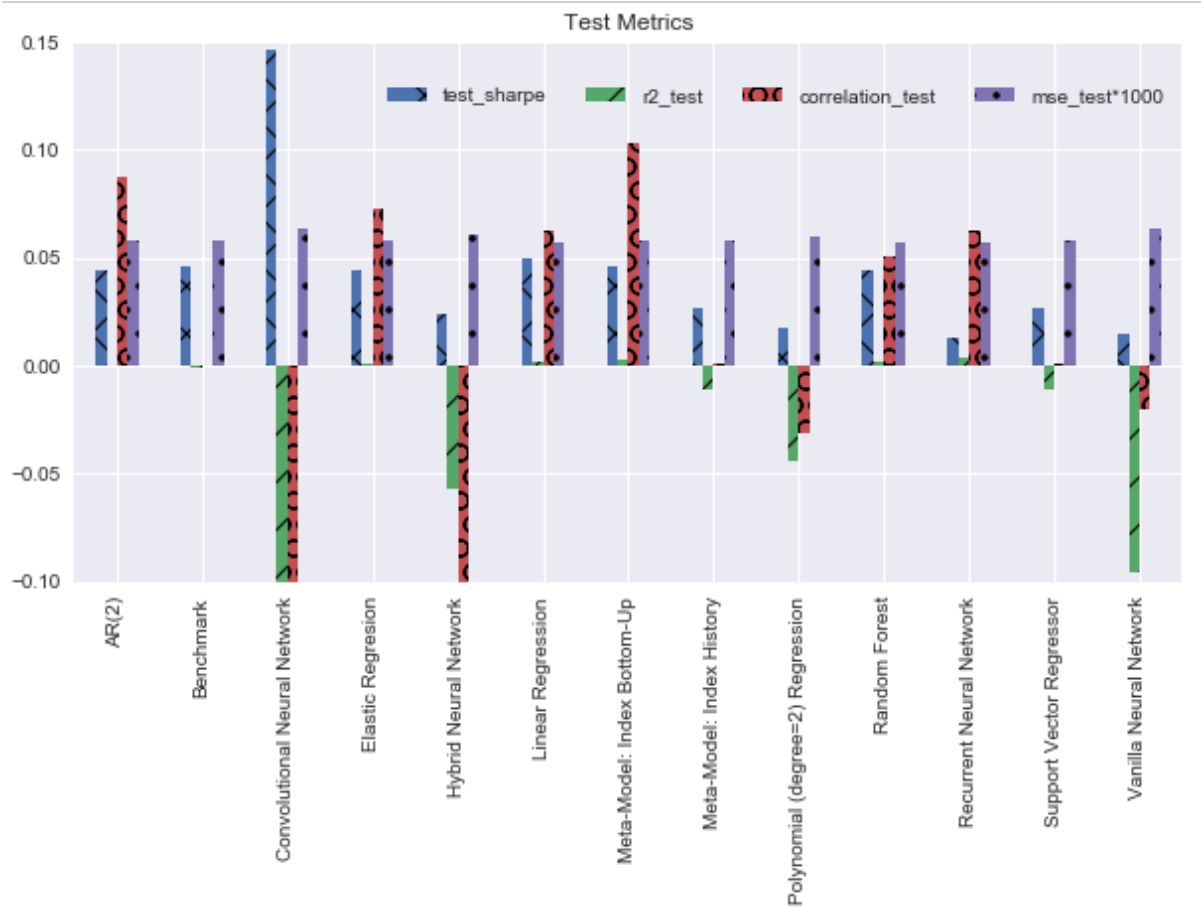
Table 9:

TRAIN	Sharpe	R-squared	Correlation	MSE*1000
AR(2)	0.0484	0.0005	0.0325	0.1155
Benchmark	0.0484	0.0000	-0.0257	0.1155
Convolutional Neural Network	0.1275	0.2606	0.5179	0.0854
Elastic Regression	0.0424	0.0009	0.0559	0.1154
Hybrid Neural Network	0.0295	0.1896	0.4398	0.0937
Linear Regression	-0.0651	0.0083	0.0913	0.1146
Meta-Model: Index Bottom-Up	0.0450	0.0000	0.0331	0.1165
Meta-Model: Index History	-0.0873	0.0288	0.1755	0.1122
Polynomial	0.0042	0.0313	0.1770	0.1119

(degree=2) Regression				
Random Forest	0.0482	0.0020	0.0505	0.1153
Recurrent Neural Network	-0.0568	0.1266	0.3591	0.1009
Support Vector Regression	-0.0873	0.0288	0.1755	0.1122
Vanilla Neural Network	0.0514	0.0036	0.4322	0.1151

Moving on to the test set results, we also have interesting insights to gleam, especially in comparison to the train set results. Here is where theoretically a more unbiased estimate of our algorithms' performance can be formed:

Figure 19:Test set results



Initially, we observe the same pattern regarding MSE: It is very similar across our models and fluctuates very little. This stability across models and working sets could be yet another indication that added model complexity, does not actually help bring the error down; That could be due to a variety of reasons, perhaps the biggest one

being the semi random nature of the data itself, as predicted by economic theory. Here, the benchmark model also has around the same error as the other models, lending more weight to that perspective.

We also observe that indeed, as regards to R-squared and correlation coefficients metrics, the performance of the best train set neural networks plummets to solidly negative territory, indicating redundant complexity. This does not impede them from achieving positive returns, and the CNN seems to be the by far best model in that regard, easily topping the charts. Aside from the neural networks, we can observe generally good correlation scores for our various models, much higher than what achieved by the benchmark.

It can also be seen that even the very simple linear and elastic regression models result in favorable financial outcomes, comparable to the random forest and support vector machines, again favoring the hypothesis that when it comes to financial time series modeled without a particularly rich feature set, and in the daily frequency, complex is not necessarily better. Nonetheless, they for the most part still fail to surpass the simplistic benchmark set.

It is intriguing that all Sharpe values in the test set are positive, in contrast to the much more disappointing train set values, as one would expect the opposite i.e the out-of-sample results would be worse. A reason for this could be that the train set is simply harder to model: It covers a long period beginning from 1980 all the way up to 2015, which includes many crises and rises. On the other hand, the test set covers the period from 2015 up to later 2018, a period when the US stock market rallied, so pretty much any strategy would work well, justifying the high Sharpe values across all models.

In the same vein, the meta-model based on the individual stocks seems to do much better than the one directly predicting the index. This again could be attributed to it being more sensitive to individual stocks' upwards movement, rather than just relying on a more modest variation of the entire index, thus better able to capture those fluctuations, like in the training set.

Another reason for higher Sharpe ratios could simply be that the complexity of the test set is lower, as for the most part there is just one big upward trend, and no dilution with recessions and other unforeseen or 'black swan' events, like there probably were quite a few in the preceding, long 30- year period. For completeness reasons, the breakdown of the results follows:

Table 10:

TEST	Sharpe	R-squared	Correlation	MSE*1000
AR(2)	0.0443	0.0002	0.0876	0.0577
Benchmark	0.0459	-0.0006	0.0000	0.0577
Convolutional Neural Network	0.1464	-0.1163	-0.1979	0.0640
Elastic Regression	0.0443	0.0006	0.0731	0.0576

Hybrid Neural Network	0.0244	-0.0572	-0.1144	0.0606
Linear Regression	0.0499	0.0021	0.0626	0.0576
Meta-Model: Index Bottom-Up	0.0464	0.0028	0.1031	0.0579
Meta-Model: Index History	0.0267	-0.0108	0.0006	0.0583
Polynomial (degree=2) Regression	0.0173	-0.0444	-0.0310	0.0602
Random Forest	0.0443	0.0021	0.0509	0.0576
Recurrent Neural Network	0.0127	0.0039	0.0628	0.0571
Support Vector Regression	0.0267	-0.0108	0.0006	0.0583
Vanilla Neural Network	0.0146	-0.0951	-0.0201	0.0632

3.4.2) Summary of evidence on training and testing of predictive models for a market index

The purpose of the meta-models, is to find a systematic way to be able to foresee in advance what model among all available options would be best in forecasting abilities over a given time horizon. Some would argue that more complex models are better, due to their supposed ability to capture nonlinearities more easily; Others, might accept the fact that the stock market is unpredictable and opt for the simpler, more computationally efficient ones. Still others might use a process similar to our own and make a more data-driven decision, based on benchmarks and very extensive cross-validation.

Conceptually speaking, the last approach could be considered statistically superior as an analyst cannot possibly know in advance what the best algorithm would be at any given time to predict such time varying and ultra sensitive series. That is the reason why here the decision is made to establish not one but two such meta-models, one on the index itself and one based on forecasting and averaging over individual stocks.

The reasoning behind this is that the combination of those methodologies would provide us with a more sophisticated benchmark than a simple constant, and allow us a better glimpse into the inner workings of the data. From the results, at least one interesting insight is indeed found due to their simultaneous application:

On the more complex and longer-spanning training set, the model focusing on the Index itself proves superior. That may suggest that over longer, more tumultuous periods, an analyst should preferably focus on the higher-level history of the index in question, presumably due to its ability to average all the shocks and provide better evidence for future performance.

However on the shorter, and arguably easier to model test set, where a clear bullish trend was in place, the meta-model dissecting the Index to its individual parts proved superior. That could be due to stocks individually being more sensitive to such movements and trend cycles, and even when averaging over each one's prediction, can still provide a more accurate picture of performance in shorter-term horizons, particularly if there is a clear underlying trend.

These observations however do not imply these are the best models indeed, or even that the particular methodology employed to come up with them is the optimal one; There are still competitors clearly superior in the arguably most important metric, the Sharpe ratio, such as the CNN, and even in all other metrics they still fare rather modestly, pretty much in par with the rest. This however does make for good benchmarks, and conceptually, it does not invalidate at all the intuition with which they were conceived in the first place: That human intuition alone about the fitness of any model should not be trusted above analytical methodologies when choosing what to apply in practice, as it unfortunately is just not possible to know better beforehand, doubly so when intuitions about statistics are very often proven wrong and misleading.

4. Conclusion

The purpose of this exercise was to examine whether several machine learning algorithms were capable of producing any meaningful financial strategies, on the daily level and relatively constrained on the richness of features available. The empirical analysis indicated that while complex algorithms like CNNs or Random forests can result in good outcomes, as measured by the Sharpe ratio, it also indicates that complexity is not necessarily adding enough value to be justifiable in out-of-sample predictions.

The exercise also indicated that stock returns are not a particularly well-behaved time series, often demanding many involved preprocessing operations and a lot of data for even marginally useful results, and that many simplifying assumptions often underlying traditional econometric and financial models, like stationarity, often do not apply without handcrafted engineering or yet persist, eventually limiting their effectiveness. Then comes the need for more advanced models, particularly nonparametric, which allow the data itself to shape architecture instead of the other way, like it happens with most traditional, often oversimplifying models. This intense nonlinearity also demands more, better data and thus domain expertise to decide the most important factors.

This has been far from an exhaustive analysis, and there are many promising and fascinating avenues to extend such analysis. The most basic directions would be to obtain a richer feature space, via even more data, more varied indicators including fundamental and macroeconomic ones, and perhaps at different time intervals, as

daily data tend to be rather noisy. Also, more computational power would be necessary in order to train larger models much faster than what a conventional personal computer would allow, and for more extensive cross-validation techniques in reasonable time with smarter, 'informed-search' approaches like coarse-to-fine, Bayesian optimization and genetic algorithms.

Other very interesting ways to create more robust predictive models is to take advantage of computer vision and natural language processing: For example, the former could be used in predicting the traffic in ports and thus measure the direction in which economic activity could move, according to shipping activity. The latter has recently found interesting use in sentiment analysis of social media posts, which could be used to reflect the mood regarding a corporation, as sentiment is a major driving factor of prices, regardless of the rationality assumptions held by classical economics.

Moreover, generative adversarial models (GANs) could be trained in order to synthesize more artificial training data in areas of finance where availability of data is not high, like for example derivatives or emerging markets, and autoencoders could be utilized as very effective dimensionality reduction techniques, alongside other popular unsupervised learning techniques, like PCA and t-SNE, being particularly helpful in large, feature-rich datasets.

Reinforcement learning has been found to be a very promising future avenue for more effective machine learning in a wide variety of tasks not well suited for traditional supervised learning, like autonomous cars, learning games and of course trading, and is already slowly being employed by major banks, as its merits become clearer and the algorithms improve.

Another very interesting idea, would be to more effectively merge insights from psychology and behavioral economics into predictive models, even going so far as to insert elements of human irrationality into the models deployed for forecasting. That could paradoxically prove to improve out-of-sample performance, as it is quite apparent that human irrationality often is the driver of stock market movements, and it is at least an oversight on the part of modelling to overlook it entirely while deployed in the real, filled with biases and cognitive shortfalls, world.

Finally, a supremely exciting development, which still is in its early steps, is the application of Quantum mechanics' principles into the machine learning pipeline. This could be done via the still-to-be commercially deployed quantum computers that would enable unparalleled processing abilities, and via the clever exploitation of various quantum world properties, like Superposition and Entanglement, to even further tremendously improve performance.

5. Appendix

Besides predicting returns, is machine learning used in other financial contexts? The answer is a resounding 'yes', with broad applications including but not limited to risk assessment in the consumer and banking level, credit scoring, bond valuation and even volatility forecasting. A short, very much not all-inclusive, overview of these exciting applications follows:

Khandani et al (2010) attempt to construct consumer credit risk models using machine learning algorithms, like CARTs, radial basis functions and support vector machines. The results look promising with very high, out-of-sample R-squared values up to 85% for 6-12 month delinquency forecast horizons. Those results, the authors report, could lead to a conservation of resources in case of losses ranging from 6% to 25% of total losses. A final very important implication is that by aggregating the individual forecasts to build a measure of the systemic risk in the consumer lending sector, it is possible to model it in its entirety better thus reducing the chances of another big financial meltdown occurring out of the blue.

Closely related to the work by Khandani et al (2010), Ong et al (2005) again use varying methods to build credit scoring methods, like ANNs (Artificial Neural Networks) and decision trees. The novelty of their work is on utilizing genetic algorithms, which seem to outperform all other models, and point to the potential of such methods over backpropagation for ANN training. Martens et al (2007) focus on improving the explainability of SVMs in the context of credit risk models, and they do so with minimal loss of classification accuracy, thus resolving an important criticism of ML models working as black boxes, which especially in areas like finance where interpretability is critical, can be a major drawback.

Machine learning has also been used in the context of (corporate) bankruptcy risk assessment.Min and Lee (2005) test various models, like MLPs, MDA, Logit and SVMs, and the later are proved to perform better, with the added bonus that they do not require much data to be trained effectively, due to the transformations of the data they perform, essentially reducing the complexity of the problems. Atiya (2001) used indicators inspired by the work of Merton in the context of credit risk models, in conjunction with MLPs, and showed that they can provide a significant boost in predictive performance, up to 85.5% for a three-year ahead forecast.

Sirignano et al (2018) utilize deep learning to examine a particularly broad range of variables to assess mortgage risk, and they too find strong nonlinear interactions between them. Based on that, they reason that the standard linear models can significantly be misspecified and fail to recognize the effects of various key factors, like interest rates, unemployment and housing prices on consumer behavior on consumer borrowing and weaken the economic validity of their conclusions. Thus, with their extensive out-of-sample testing, they show that their deep neural network architecture can effectively deal with the massive dataset and naturally work out the nonlinearities, which *"significantly improves the accuracy of loan- and pool-level risk forecasts, the investment performance of mortgage trading strategies, and the valuation and hedging of mortgage-backed securities"* (Sirignano 2018, page 32).

Their work has two significant implications: First, the effectiveness of their neural network can pave the way to absolve econometric analysts from the extensive feature engineering they may have to perform in order to extend the linear models to more capably adapt to nonlinearities, as they are not naturally accustomed to it, by more readily applying such architectures. Second, the model's predictive prowess hints at it being useful for a variety of other important applications, given it is appropriately refitted to its problem domain, including the valuation and hedging of mortgage-backed securities as in Curley & Guttentag (1977), Schwartz & Torous (1989) and Stanton & Wallace (2011).

Butaru et al (2016) apply decision trees, random forests and regularized logistic regression, to measure risk in the credit card industry, and propose better risk management measures. They find that decision trees and random forests deliver better in- and out- of sample forecasts, especially in the short term. However, they find that the delinquency rates vary wildly among different banks, so even those better models cannot be broadly applied accurately to all cases, which are defined by their heterogeneity, for example in the management styles or other macroeconomic risk factors.

Similarly, Huang et al (2004) use MLPs and SVMs for credit rating analysis. They too found that they performed well in their given task (close to 80% classification accuracy for both, with SVMs offering slightly better performance) ,which was to learn the dynamics of past evaluations of credit risk and to use that to accurately predict the current credit score of corporations. This is in par with the literature's prior findings, which shows that AI systems regularly outperform classic statistical learning methods due to their increased complexity, even though it may more easily lead them to overfitting problems. Interestingly, they found that even though Taiwanese and American markets favor similar lists of variables crucial to the bond rating process, the feature importance between them was different, which could mean that the models should be retrained for maximum performance according to the market they are being applied to, and there is no 'one size fits all' system.

To cite but a handful of examples of the aforementioned literature which indicates ML systems can be superior more often than not, the authors mentioned Singleton and Surkan (1990) who used an MLP to classify Moody's bond ratings of four categories, where they achieved 88% accuracy; Kim's (1993) comparison of linear regression, discriminant analysis, logistic analysis and a rule-based system for bond rating, and found that neural networks outperformed everything, on a 6-level rating scheme using Standard and Poor's financial data; Moody and Utans (1995) who also concluded that neural networks reigned supreme in all of the 3, 5 and 16 category classification tasks; and Maher and Sen (1997) who again found neural networks gave the better performance (up to 70% accuracy) when compared with logistic regression.

Machine learning models have also been used to measure and predict volatility. Monfared and Enke (2014) used GJR-GARCH models in conjunction with three mainstream neural network architectures, in order to improve the GARCH model's performance in periods of both calm waters and crisis.They conclude that even though the Hybrid approach (combining the GARCH with a Neural Network) could improve the predictions in times of crisis, due to the increased complexity of the data which gets ameliorated by the added intricacy of the hybrid model, in peaceful times the hybrid models tended to perform subpar compared to the GJR-GARCH model used standalone, due to the added complexity of the hybrid. Finally, they too have come to the conclusion that there is no 'one size fits all' approach, as different architectures fare better in different circumstances, as the underlying dynamics are fluid, and no single framework reigns supreme.

6.Bibliography

Pedregosa Fabianpedregosa, F., Michel, V., Grisel Oliviergrisel, O., Blondel, M., Prettenhofer, P., Weiss, R., … Duchesnay Edouardduchesnay, Fré. (2011). Scikit-learn: Machine Learning in Python Gaël Varoquaux Bertrand Thirion Vincent Dubourg Alexandre Passos Pedregosa, Varoquaux, GramforT et al. Matthieu Perrot. *Journal of Machine Learning Research*, *12*, 2825–2830. Retrieved from http://scikit-learn.sourceforge.net.

Hutchinson, J. M., Lo, A., & Poggio, T. (1994). Massachusetts institute of technology center for biological and computational learning A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks. *Convergence*, *1879*(1471).

Min, J. H., & Lee, Y. C. (2005). Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters. *Expert Systems with Applications*, *28*(4), 603–614. https://doi.org/10.1016/j.eswa.2004.12.008

Atiya, A. F. (2001). Bankruptcy prediction for credit risk using neural networks: A survey and new results. *IEEE Transactions on Neural Networks*, *12*(4), 929–935. https://doi.org/10.1109/72.935101

Martens, D., Baesens, B., Van Gestel, T., & Vanthienen, J. (2007). Comprehensible credit scoring models using rule extraction from support vector machines. *European Journal of Operational Research*, *183*(3), 1466–1476. https://doi.org/10.1016/j.ejor.2006.04.051

Khandani, A. E., Kim, A. J., & Lo, A. W. (2010). Consumer credit-risk models via machine-learning algorithms. *Journal of Banking and Finance*, *34*(11), 2767–2787. https://doi.org/10.1016/j.jbankfin.2010.06.001

Huang, Z., Chen, H., Hsu, C. J., Chen, W. H., & Wu, S. (2004). Credit rating analysis with support vector machines and neural networks: A market comparative study. *Decision Support Systems*, *37*(4), 543–558. https://doi.org/10.1016/S0167-9236(03)00086-1

Sirignano, J., Sadhwani, A., & Giesecke, K. (2018). Deep Learning for Mortgage Risk. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2799443

Heaton, J. B., Polson, N. G., & Witte, J. H. (2016). Deep Learning in Finance, (February), 1–20. Retrieved from http://arxiv.org/abs/1602.06561

Gu, S., Kelly, B. T., & Xiu, D. (2018). Empirical Asset Pricing via Machine Learning. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3281018

Giglio, S. W. (2016). Inference on Risk Premia in the Presence of Omitted Factors. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2865922

Moritz, B., & Zimmermann, T. (2016). Tree-Based Conditional Portfolio Sorts: The Relation between Past and Future Stock Returns. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2740751

Zheng, A., & Jin, J. (2017). Using AI to Make Predictions on Stock Market, 1–6.

Monfared, S. A., & Enke, D. (2014). Volatility forecasting using a hybrid GJR-GARCH neural network model. *Procedia Computer Science*, *36*(C), 246–253. https://doi.org/10.1016/j.procs.2014.09.087

Ong, C. S., Huang, J. J., & Tzeng, G. H. (2005). Building credit scoring models using genetic programming. *Expert Systems with Applications*. https://doi.org/10.1016/j.eswa.2005.01.003

Surkan, A. J., & Singleton, J. C. (1990). Neural networks for bond rating improved by multiple hidden layers. In *IJCNN. International Joint Conference on Neural Networks*.

Henrique, B. M., Sobreiro, V. A., & Kimura, H. (2018). Stock price prediction using support vector regression on daily and up to the minute prices. *The Journal of Finance and Data Science*. https://doi.org/10.1016/j.jfds.2018.04.003

Hiransha, M., Gopalakrishnan, E. A., Menon, V. K., & Soman, K. P. (2018). NSE Stock Market Prediction Using Deep-Learning Models. In *Procedia Computer Science*. https://doi.org/10.1016/j.procs.2018.05.050

Maher, J. J., & Sen, t. k. (1997). Predicting Bond Ratings Using Neural Networks: A Comparison with Logistic Regression. *International Journal of Intelligent Systems in Accounting, Finance & Management*. https://doi.org/10.1002/(sici)1099-1174(199703)6:1<59::aid-isaf116>3.3.co;2-8

Lake, B., & Baroni, M. (2018). Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *35th International Conference on Machine Learning, ICML 2018*.

Schwartz, E. S., & Torous, W. N. (1989). Prepayment and the Valuation of Mortgage-Backed Securities. *The Journal of Finance*. https://doi.org/10.1111/j.1540-6261.1989.tb05062.x

Brealey, R. (1977). Value and Yield Risk on Outstanding Insured Residential Mortgages: Discussion. *The Journal of Finance*. https://doi.org/10.2307/2326774

Stanton, R., & Wallace, N. (2011). The bear's lair: Index credit default swaps and the subprime mortgage crisis. *Review of Financial Studies*. https://doi.org/10.1093/rfs/hhr073

Moody, J., & Utans, J. (1994). Architecture selection strategies for neural networks: Application to corporate bond rating prediction. *Neural Networks in the Capital Markets*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. https://doi.org/10.1109/CVPR.2015.7298594

Nayak, S. C., Misra, B. B., & Behera, H. S. (2018). Artificial chemical reaction optimization based neural net for virtual data position exploration for efficient financial time series forecasting. *Ain Shams Engineering Journal*. https://doi.org/10.1016/j.asej.2016.10.009

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*. https://doi.org/10.1109/72.279181

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*. https://doi.org/10.1109/MCSE.2011.37

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science and Engineering*. https://doi.org/10.1109/MCSE.2007.55

McKinney, W. (2010). Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*.

Oliphant, T., & Millma, J. k. (2006). A guide to NumPy. *Trelgol Publishing*. https://doi.org/DOI:10.1109/MCSE.2007.58

Kim, J. W., Weistroffer, H. R., & Redmond, R. T. (1993). Expert systems for bond rating: a comparative analysis of statistical, rule-based and neural network systems. *Expert Systems*. https://doi.org/10.1111/j.1468-0394.1993.tb00093.x

Hochreiter, S., Frasconi, P., & Schmidhuber, J. (2001). *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies - Abstract - UK PubMed Central*. *A Field Guide to Dynamical Recurrent Neural Networks*.

Donahue, J., & Darrell, T. (2017). Dversarial eature earning. *Iclr*.

Kingma, D. P., & Ba, J. L. (2015). Adam: A method for stochastic gradient descent. *ICLR: International Conference on Learning Representations*.

Yao, J., Li, Y., & Tan, C. L. (2000). Option price forecasting using neural networks. *Omega*. https://doi.org/10.1016/S0305-0483(99)00066-3

Breiman L. (2001). Machine Learning, 45(1), 5–32. *Statistics Department, University of California, Berkeley, CA 94720.* https://doi.org/10.1023/A:1010933404324

Moghaddam, A. H., Moghaddam, M. H., & Esfandyari, M. (2016). Predicción del índice del mercado bursátil utilizando una red neuronal artificial. *Journal of Economics, Finance and Administrative Science*, *21*(41), 89–93. https://doi.org/10.1016/j.jefas.2016.07.002

Butaru, F., Chen, Q., Clark, B., Das, S., Lo, A. W., & Siddique, A. (2016). Risk and risk management in the credit card industry. *Journal of Banking and Finance*, *72*, 218–239. https://doi.org/10.1016/j.jbankfin.2016.07.015

Chollet, F. (2015) keras, GitHub. https://github.com/fchollet/keras

Talib: https://github.com/mrjbq7/ta-lib

Seaborn: https://zenodo.org/record/12710#.XUCN1ntS-Uk

Statsmodels: www.statsmodels.org/stable/index.html

J. Dean, G. Corrado, R. Monga, et al (2012). Large scale distributed deep networks, *Advances in Neural Information Processing Systems*, pp. 1223-1231, 2012.