The Highs and Lows of Stock Market Gambling: Pairs Trading with ARIMA and XGBoost

Julianne Cai, 1413991

November 8, 2024

1. Introduction

Pairs trading is a trading strategy characterised by choosing a pair of stocks which are highly correlated, and when their prices diverge, buy the lower one, and short the higher one. If the prices converge again, then we pocket the difference. This strategy was first implemented by Nunzio Tartaglia's quantitative group at Morgan Stanley in the 1980s [Richard, 2006], and generated profits of around 50 million dollars in 1987 [Uyumazturk and Portilheiro, 2017]. The group, however, later disbanded in 1989 due to unsatisfactory returns. Now it is our turn.

Let $\mathbf{X}_t$ and $\mathbf{Y}_t$ be stochastic processes denoting the price of a stock at time t. Then, the *spread* is the difference $\mathbf{S}_t = \mathbf{X}_t - \mathbf{Y}_t$. We say that two stocks are *cointegrated* if there exists some $\alpha, \beta \in \mathbb{R}$ such that the linear combination $\alpha\mathbf{X}_t + \beta\mathbf{Y}_t$ is a stationary process.

Pairs trading often rely on the assumption that the spread between two stocks is *mean-reverting* – that is, the long term behaviour of the stock has a constant mean. As such, when large divergences occur in the stock prices – and hence the spread – it is possible to enter into a short position in one stock, and a long position in another stock, and then make a profit by flattening the positions after the spread reverts to its mean.

However, determining the entry and exit thresholds for the long and short positions is a non-trivial problem, and is a problem that has been extensively studied (see [Zeng and Lee, 2014, Puspaningrum et al., 2009]). In [Uyumazturk and Portilheiro, 2017], the authors determined trade entry signals by determining the probability of the spread to the mean using Monte-Carlo simulation methods. Then, a probabilistic method was used to determine the local extrema of the spread $\mathbf{S}_t$, by using the assumption that

$$S_t | S_{t-1}, \cdots, S_{t-p} \sim N\left(\varphi_0 + \sum_{i=1}^{p} \varphi_i S_{t-i}, \sigma^2\right),$$

where φ_i are the fitted parameters of an autoregressive process AR(p) of order p (see Section 2). Using this, the probability $\mathbb{P}(|S_{t+1} - \mu_S| > |S_t - \mu_S|)$ was calculated, where μ_S is the historical mean of the spread.

[Figueria, 2022] forecasts the price of the spread $\mathbf{S}_t$ one time-step ahead – call this S_{t+1} – and then looks at the value $\Delta_{t+1} := S_{t+1} - S_t$. If Δ_{t+1} is greater than the transaction cost of the trade, then it enters a long posiiton, and if Δ_{t+1} is less than the negative of the transaction cost, then it enters a short position.

Another non-trivial aspect of the problem concerns the selection of such pairs of stocks. There exist many criteria for determining suitable pairs of stocks. A naive method uses pairs of stocks that

have the smallest average distance between the prices. However, this method negates many other subtle correlations between the two stocks. [Uyumazturk and Portilheiro, 2017] uses a maximum likelihood estimation of the parameters $(\varphi_1, \cdots, \varphi_p)$ of an AR(p) model, and chooses the pair of stocks that had the smallest norm $\|(\varphi_1, \cdots, \varphi_p)\|_2$.

In [Figueria, 2022], the author performed a test for cointegration on a list of stocks, and then selected the the pair that had the smallest p-value.

Our approach is primarily inspired by [Figueria, 2022], particularly in our use of ARIMA and XGBoost. The advantage of ARIMA over an AR model used in [Uyumazturk and Portilheiro, 2017] is that ARIMA models are able to accomodate non-stationary spreads. We also alter our strategy for generating long-short signals, instead opting to generate Bollinger bands using a moving average, and then taking the standard deviation of the moving average (see Section 6). While this methodology is more primitive than that used in sources like [Uyumazturk and Portilheiro, 2017], we find that this methodology generates more trading signals. For instance, the trading strategy implemented by [Uyumazturk and Portilheiro, 2017] made only about 8 trades over the course of 180 days of active trading.

We place our focus specifically on stocks in the technology sector, such as Microsoft (MSFT), NVIDIA (NVDA), Facebook (META), AMD, etc. Our criterion for selecting pairs is MacKinnon's asymptotic, approximate p-value test for cointegration [MacKinnon, 1996]. After performing the test, the pair of stocks with the lowest p-value is selected.

Our trading strategy involves using machine learning models – ARIMA and XGBoost in this case – to forecast the price of the spread one day ahead. Then, if the forecasted price is higher or lower than the Bollinger bands, a short or long position is entered, respectively. Then, once the spread reverts to being within the Bollinger bands, the position is flattened – that is, our short and long positions are sold off.

2. Preliminaries on ARIMA

Throughout, let $\{\mathbf{X}_t\}$ represent a discrete-time stochastic process – for our purposes, this will typically be a stock price or realised volatility at time t. We say that the time series is *(strictly) stationary* if $\mathbf{X}_t \overset{d}{=} \mathbf{X}_{t+h}$, where $\overset{d}{=}$ denotes equality in distribution, and h is any arbitrary shift in time. Equivalently, this can be formulated in terms of the distribution function as the equality $\mathbb{P}(X_t \leq x) = \mathbb{P}(X_{t+h} \leq x)$. This means that – assuming the distribution has finite second moments – the mean and variance must be constant. Then, we have *Wold's decomposition theorem*:

Theorem 1 (Wold, 1936)**.** *Let μ be the mean of the stochastic process. Then, every stationary, non-deteriministic stochastic process can be written as a linear combination of a sequence of uncorrelated random variables:*

$$X_t - \mu = \sum_{j=0}^{\infty} \psi_j a_{t-j}, \quad \psi_0 = 1,$$

where $\{a_t\}$ is a sequence of uncorrelated random variables, drawn from a fixed distribution with $\mathbb{E}a_t = 0$, and $\operatorname{Var} a_t = \sigma^2 < \infty$, with $\operatorname{Cov}(a_t, a_{t-k}) = 0$, for all $k \neq 0$.

2.1. Autoregressive Models

The statement of Theorem 1 is very general, but in practise particular choices of ψ are taken. As an example, let $\psi_j := \varphi^j$ allows us to write

$$X_t = a_t + \varphi a_{t-1} + \varphi^2 a_{t-2} + \cdots = a_t + \varphi X_t.$$

This is called a *first order autoregressive* (AR) model. This is often denoted by AR(1). Generally, an AR(p) model is defined as:

$$X_t = \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t,$$

where φ_i are the *parameters* of the model, and ε_t is an error term. Often, ε_t is assumed to be an i.i.d Gaussian distributed white-noise process with constant mean $\mu$and variance σ^2.

Lemma 1. *The* characteristic polynomial *of an autoregressive process* AR(p) *of order p is:*

$$1 - \varphi_1 z - \varphi_2 z^2 - \cdots - \varphi_p z^p = 0, \quad z \in \mathbb{C}.$$

Then, the time series $\{X_t\}$ is stationary if $|\varphi| < 1$ – or equivalently, if $|z| > 1$.

Proof. Let us restrict to the case for which $p = 1$. Then, the autoregressive process is given by:

$$X_t - \varphi X_{t-1} = \varepsilon_t,$$

and the characteristic polynomial is given by $1 - \varphi z = 0$. Thus,

$$|z| = \left|\frac{1}{\varphi}\right| > 1 \iff |\varphi| < 1.$$

Then, computing the mean, we have:

$$\mathbb{E}X_t = \sum_{i=0}^{\infty} \varphi^i \mathbb{E}\varepsilon_{t-i} = \sum_{i=0}^{\infty} \varphi^i \mu = \frac{\mu}{1-\varphi},$$

and thus it follows that X_t has finite first order moment if and only if $\varphi \neq 1$. Computing the variance, we have:

$$\begin{aligned} \operatorname{Var} X_t &= \varphi^2 \operatorname{Var} X_{t-1} + \sigma^2 \\ &= \varphi^2(\varphi^2 \operatorname{Var} X_{t-2} + \sigma^2) + \sigma^2 \\ &\vdots \\ &= \sum_{i=0}^{\infty} \varphi^{2i} \sigma^2 \\ &= \frac{\sigma^2}{1-\varphi^2}, \end{aligned}$$

and thus the autoregressive process has finite second order moment if and only if $|\varphi| < 1$, which is equiavlent to $|z| > 1$. Inducting on p gives the desired result. □

Definition 1. Consider a time-discrete stochastic process $\{X_t\}$ that can be written as an autoregressive process AR(p) of order p:

$$X_t = \varphi_1 X_{t-1} + \cdots \varphi_p X_{t-p} + \varepsilon_t.$$

Then, $\{X_t\}$ has a *unit root* if $z = 1$ is a root of the characteristic equation

$$1 - \varphi_1 z - \cdots - \varphi_p z^p = 0.$$

Alternatively, we say that X_t is *integrated* of order 1. If $z = r$ is a root of the characteristic equation, then X_t is *integrated* of order p.

Corollary 1. *A discrete-time stochastic process is non-stationary if it has a unit root.*

Proof. Suppose that the error term ε_t has constant mean μ and variance σ^2. In the $p = 1$ case, the time series X_t is integrated of order 1 if $\varphi = 1$, and we have: $X_t = X_{t-1} + \varepsilon_t$. Repeated substitution then gives us:

$$X_t = X_0 + \sum_{i=1}^{t} \varepsilon_i,$$

which has variance

$$\operatorname{Var} X_t = \sum_{i=1}^{t} \sigma^2 = t\sigma^2,$$

and thus X_t is non-stationary since its second moment depends on t. The result follows by inducting on p. □

2.2. Augmented Dickey-Fuller Test

Stationary is a desirable property for time series to have. In this section, we introduce a test for unit roots in a stochastic process, called the *Augmented Dicker-Fuller* (ADF). In particular, the ADF test tests the null hypothesis that a unit root is present in a time series sample, with the alternative hypothesis being that the time series is stationary.

The ADF test fits the following model:

$$\Delta X_t = \alpha + \beta t + \gamma X_{t-1} + \delta_1 \Delta X_{t-1} + \cdots \delta_{p-1} X_{t-p+1} + \varepsilon_t, \tag{1}$$

where α and β are constants. The number p is the *lag order* of the autoregressive process, and must be chosen prior to fitting the ADF test. The choice of lag order p can be determined by minimising either the Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC).

The constant β is called the *time trend* constant, and determines whether X_t is trend-stationary. If $\beta = 0$, then (1) simply tests for stationary. For our purposes, we will be setting $\beta = 0$ when we begin forecasting Stock prices. Using (1), the null hypothesis and alternative hypothesis can be formulaed as:

$$H_0 : \gamma = 0$$
$$H_1 : \gamma < 0.$$

2.3. ARIMA and ARMA Models

Suppose that the stochastic process is X_t stationary. Then, a *moving average* model MA(q) of order q is given by:

$$X_t = \mu + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i},$$

where θ_i are the parameters of the model, q is a lag hyperparameter, and $\mu = \mathbb{E}X_t$. The terms ε_i are i.i.d white noise error terms that are typically taken to be Gaussian random variables. An autoregressive moving average model ARMA(p, q) of order (p, q) is then given by adding a AR(p) and MA(q) model together:

$$X_t = \varepsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}. \tag{2}$$

Before fitting, one must choose appropriate hyperparameters p and q. This can be done by plotting the (partial) autocorrelation of the process X_t, and then choosing the lag values which are most statistically

significant. Alternatively, a grid search method can also be employed to tune these hyperparameters, using either AIC or BIC.

However, it is often the case that X_t is non-stationary. In this case, one considers differences of the series: $\{X_t - X_{t-1}\}$. This is a first-order difference. n-order differences are defined analogously. In the case of non-stationary processes, the order of differencing is taken into account in the model using the autoregressive *integrated* moving average (ARIMA) model. Define a lag operator $\mathcal{L}^i X_t := X_{t-i}$. Then, (2) can be re-written to be:

$$\left(1 - \sum_{i=1}^{p'} \varphi_i \mathcal{L}^i\right) X_t = \left(1 + \sum_{j=1}^{q} \theta_j \mathcal{L}^j\right) \varepsilon_t,$$

If the polynomial $1 - \sum_i \varphi \mathcal{L}^i$ has a unit root of multiplicity d, then, it can be re-written as:

$$\left(1 - \sum_{i=1}^{p} \varphi_i \mathcal{L}^i\right) (1-L)^d X_t = \left(1 + \sum_{j=1}^{q} \theta_j \mathcal{L}^j\right) \varepsilon_t,$$

where $p = p' - d$. This is a ARIMA(p, q, d) model, where d is the order of differencing of X_t.

3. Preliminaries on XGBoost

eXtreme[1] Gradient Boosting (XGBoost) is a machine learning model first introduced by [Chen and Guestrin, 2016]. Since its inception, it has become an extremely popular machine learning model, due to its ability to deal with large datasets and capture complex patterns in data. XGBoost uses *boosted trees*, which is a machine learning method that trains many classification and regression trees (CARTs) on a dataset, and then combines their predictions. The model itself is mathematically complex, and we can only give a brief overview of its mechanisms.

We will briefly talk about ensembling methods, but for a comprehensive overview of this topic, we refer the interested reader to [Zhou, 2012], which covers ensembling techniques for classification algorithms. [Breiman et al., 1987] covers both regression and classification trees. In particular, we follow [Zhou, 2012, Chapter 2] for our exposition on tree boosting, and [Chen and Guestrin, 2016, §2] for our exposition on gradient tree boosting.

The goal of this section is to present the content the XGBoost algorithm by [Chen and Guestrin, 2016] in a way that is more accessible. The expert reader, however, can simply skip straight to reading [Chen and Guestrin, 2016, §2.2], and further onwards.

3.1. Boosting Algorithms

Let $\mathcal{D}$ denote a dataset containing N instances, each with m features. That is,

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) : \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R}\},$$

where each $\mathbf{x}_i$, for $1 \leq i \leq N$, is an m-dimensional vector representing an instance, with the entries of the vector representing a feature. Here, y_i denotes the target feature. For example, if we are trying to predict the price of a stock at time $t+1$, the vector $\mathbf{x}_i$ could be a vector of the open price, close prices, trading volume, volatility, and y_i could be the price of the stock.

[1] This is not a typo. This is how it is stylised.

The term *boosting* refers to a family of algorithms that are able to convert "weak learners" into "strong learners" [Zhou, 2012, pg. 23]. Intuitively, a weak learner is a model whose performance is only marginally better than randomly guessing, whilst a strong learner is very close to perfect in terms of performance. Tree boosting is an example of an *ensemble* algorithm, which is a machine learning method characterised by the use of multiple algorithms, and then combining the predictions of each algorithm in order to improve performance. The idea of a boosting algorithm can be summarised using the pseudocode below [Zhou, 2012, pg. 24]:

Input: Sample dataset $\mathcal{D}$, base learning algorithm $\mathcal{L}$, number of learning rounds T
$t \leftarrow 0$
$\mathcal{D}_0 \leftarrow \mathcal{D}$
while $t < T$ **do**
  $\widehat{y}_t = \mathcal{L}(\mathcal{D}_t)$ ▷ prediction using learning algorithm
  $\varepsilon_t \leftarrow \ell(y_t, \widehat{y}_t)$ ▷ l denotes the loss function
  $\mathcal{D}_{t+1} \leftarrow$ Adjust Distribution$(\mathcal{D}_t, \varepsilon_t)$
end while
return Combine Outputs$(\{\widehat{y}_0, \cdots, \widehat{y}_T\})$

The benefit of a boosting algorithm is that it is a computationally cheap method that minimises instance bias. Thus, if we have a weak learner (e.g. a decision tree) that is consistently underfitting when trained on a dataset, boosting offers a way to improve the predictive performance of that model.

We will now formalise this idea a bit more. For our purposes, we may think of a *regression tree* as a tree that contains a continuous score on each leaf. Let w_i is used to denote the score on the i-th leaf of the tree. Once fitted to a dataset, the regression tree will resemble the following diagram:

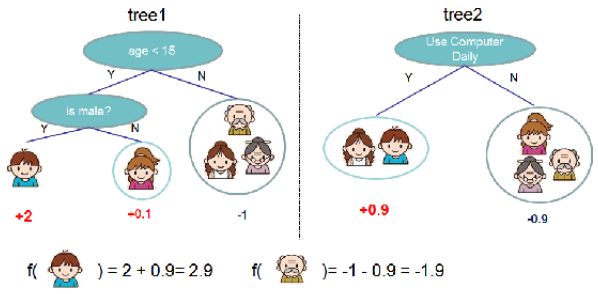


Figure 1: Example of a tree ensemble model. The final prediction is given by the sum of the predictions from each tree (retrieved from [Chen and Guestrin, 2016, Figure 1]).

Using the above example, suppose that we have trained a regression tree on a dataset of various people, and we want to determine whether or not they will be on their computers everyday. As features, we are given their age (a binary value: either < 15 or ≥ 15), and their gender (male or female). Then, when the model encounters an unseen instance given by a < 15 year old male, it will traverse through the decision rules in the tree until we arrive at a leaf for both trees, and then add the scores together. Same if it comes across an unseen instance given by an old man. In this case, a higher score represents a higher likelihood that the instance will be on their computer everyday.

Let us formalise this idea. The *space of regression trees* $\mathcal{F}$ is a space of functions of the from $f(\mathbf{x}) = w_{q(\mathbf{x})} \in \mathbb{R}^{|T|}$, where T is the set of leaf indexes, and $q : \mathbb{R}^m \to T$ is the *tree structure*, which maps an instance $\mathbf{x}$ to a leaf index. Intuitively, q here is the function that encodes the decision rules of the regression tree. $w \in \mathbb{R}^{|T|}$ is a vector representing the weights of each leaf. The function f then takes in an

instance $\mathbf{x}_i$, $1 \leq i \leq N$ as an argument, and outputs its corresponding weights.

A tree ensemble model generates predictions of the target feature y_i via the formula:

$$\widehat{y}_i = \phi(\mathbf{x}_i) := \sum_{i=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

for instance i. Each regression tree f_k corresponds to an independent tree with a given tree structure map q_k, and weight vector w_k. Intuitively, the regression trees f_k are fitted such that an appropriate loss function $\ell(y_i, \widehat{y}_i)$ is minimised. This can be formulated as an optimisation function using the loss function as the objective function. However, minimising the loss function directly can lead to overfitting, and thus a *regularisation* term $\Omega(f_k)$ depending on the tree is usually added to the objective function. Thus, we have the following optimisation problem:

$$\begin{aligned} \min \quad & \mathcal{L}(\theta) = \sum_i \ell(y_i, \widehat{y}_i) + \sum_k \Omega(f_k), \\ \text{s.t.} \quad & \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2, \end{aligned}$$

where l is a choice of differentiable, convex, loss function that measures the error between the prediction $\widehat{y}_i$ and the actual value y_i. The regularisation term $\Omega(f)$ is added in order to punish model complexity. The terms γ and λ are user-defined constants. γ punishes the model for forming too many leaves T, and λ reduces the prediction's sensitivity to individual observations [Figueria, 2022, pg. 45].

3.2. The XGBoost Algorithm

The idea behind the XGBoost algorithm is that it successive creates a collection of trees, each of which tries to improve on the predictions made by the previous one. After this, the contribution of all trees are then combined together in order to make a prediction.

In particular, each new tree is built such a way that the residuals from the previous trees are minimised. In this way, each new tree attempts to improve on the previous tree.

We let $\widehat{y}_i^{(k)}$ denote the prediction of the i-instance at the k-iteration. The loss function that we wish to minimise is then given by:

$$\mathcal{L}^{(k)} = \sum_{i=1}^{N} \ell(y_i, \widehat{y}_i^{(k-1)} + f_k(\mathbf{x}_i)) + \Omega(f_k). \tag{3}$$

Typically, such optimisation problems are solved using gradient descent techniques [Figueria, 2022, pg. 42-43]. However, [Chen and Guestrin, 2016] takes the second order Taylor expansion of (3), which gives us:

$$\mathcal{L}^{(k)} = \sum_i \left(\ell(y_i, \widehat{y}_i^{(k-1)}) + \frac{\partial}{\partial \widehat{y}_i^{(k-1)}} \ell(y_i, y_i^{(k)}) \cdot f_k(\mathbf{x}_i) + \frac{\partial^2}{\partial \left(\widehat{y}_i^{(k-1)}\right)^2} \ell(y_i, \widehat{y}_i^{(k-1)}) \cdot f_k^2(\mathbf{x}_i) \right) + \Omega(f_k) + O((\widehat{y}_i^{(k-1)})^3).$$

To simplify this, we write:

$$g_i := \frac{\partial}{\partial \widehat{y}_i^{(k-1)}} \ell(y_i, \widehat{y}_i^{(k-1)}, \quad h_i := \frac{\partial^2}{\partial \left(\widehat{y}_i^{(k-1)}\right)^2} \ell(y_i, \widehat{y}_i^{(k-1)}.$$

These terms are referred to as *gradient statistics* on the loss function. The authors in the original paper further simplify the expression by removing the constant terms to obtain the objective function:

$$\widetilde{\mathcal{L}}^{(k)} = \sum_{i=1} \left(g_i f_k(\mathbf{x}_i) + \frac{1}{2} h_i f_k^2(\mathbf{x}_i) \right) + \Omega(f_k). \tag{4}$$

Let

$$I_j := \{i : q(\mathbf{x}_i) = j\},$$

denote the set of leaves that are mapped to by the instance $\mathbf{x}_i$. Then, (4) can be written in terms of weights of the regression tree as follows:

$$\widetilde{L}^{(k)} = \sum_{j=1}^{T} \left(\left(\sum_{i \in I_j} g_i \right) \cdot w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) \cdot w_j^2 \right) + \gamma T.$$

This equation can actually be optimised exactly to obtain an optimal weight w_j^* (see [Chen and Guestrin, 2016, (4), (5)]. However, using this equation requires us to iterate over every tree structure q, which is generally impossible.

Thus, the authors use a *greedy algorithm*, which starts with a root node in the regression tree, and then iteratively splits by adding branches. Let $I = I_L \cup I_R$, where I_L and I_R are the instance sets of the left and right sub-trees of I are the split. Let $G = \sum_{i \in I} g_i$, $H = \sum_{i \in I} h_i$, $G_L = \sum_{i \in I_L} g_i$, $H_L = \sum_{i \in I_L} h_i$, and similarly for G_R and H_R. Then, the loss function after the split is given by

$$\mathcal{L} = \frac{1}{2} \left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G}{H + \lambda} \right) - \gamma.$$

This is then used to determine whether to further split the tree. Let us illustrate this with yet another example from the authors:

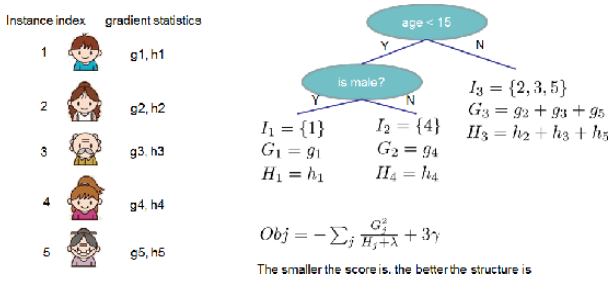


Figure 2: Summing up both gradient statistics on each leaf, and then applying the scoring function $\mathcal{L}$. The coefficient of γ comes from the fact that there are three leaves, and so $T = 3$.

In the above example, we are given some nodes. At each node, its G_L, G_R, H_L, and H_R values are calculated, and then substituted into $\mathcal{L}$. Then, the node with the largest $\mathcal{L}$ value is then chosen for the next branch split (c.f. [Chen and Guestrin, 2016, Algorithm 1]).

The paper goes on to discuss more technical aspects of split finding algorithms in [Chen and Guestrin, 2016, §3]. One common theme amongst the split finding algorithms is that the instance sets I have to be sorted (see [Chen and Guestrin, 2016, Algorithm 1, Algorithm 3]), which is "the most time-consuming part of tree learning" [Chen and Guestrin, 2016, pg. 5]. As such, [Chen and Guestrin, 2016, §4] is devoted to technical discussions of the system design that can optimise the time complexity for sorting I. [Chen and Guestrin, 2016, §5] gives a brief literature review of similar works, and [Chen and Guestrin, 2016,

§6] implements XGBoost on some publicly avaiable datasets, and reviews their performance.

4. Model Architecture

We take a modular approach to designing our model, with multiple classes, each performing a specific role. The `Trainer` class contains functions that handle feature engineering, time series tests, and model evaluation.

`ARIMATrainer` is a subclass of the `Trainer` class, and thus inherits these functions. But additionally, it is also able to train an ARIMA model, and make predictions.

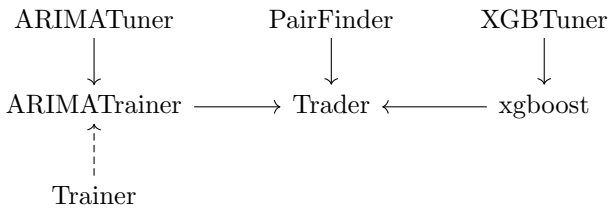
The `tuner.py` file contains two classes: `XGBTuner` and `ARIMATuner`. The `XGBTuner` class contains a function that optimises the hyperparameters of an `XGBRegressor` object from the `xgboost` package using Bayesian optimisation (see Section 5).

The `ARIMATuner` class uses a grid search method to optimise the hyperparameters (p, q, d) of an ARIMA model, using either the AIC or BIC as the scoring criterion.

The `ARIMATuner` class tunes the hyperparameters of the ARIMA(p, q, d) model by performing a grid search on a list of p, q, d values, and then selecting the best triple (p, q, d) based on either AIC or BIC. The `XGBTuner` class contains methods for tuning an `XGBRegressor` object from the XGBoost package, using Bayesian optimisation.

The `PairFinder` class finds a suitable pair of stocks by identifying pairs of stocks that are cointegrated. It also contains various methods that aid in data visualisation, such as plotting the spread, and the hedge ratio.

And finally, the `Trader` class uses the tuned ARIMA and XGBoost models to perform our pairs trading strategy on the pair of stocks selected by `PairsFinder`. It also contains functions that visualises the buy-sell signals generated by the autotraders. The dependency of each of the classes can be summarised by the following diagram:



where the dotted arrows denote the fact that a class inherits attributes from another class, and the solid arrows denote the fact that a class uses output from another class.

5. Model Tuning

The two machine learning models we use for this project – ARIMA and XGBoost – both requires the user to define certain parameters before the model can be trained. In the case of the ARIMA model, the lag parameters p and q, and the order of differencing d need to be predetermined before it can be fitted to the model.

In the case of the XGBoost model, there are also hyperparameters that have to be determined, such as the number of iterations T, the constants γ and λ. Using the documentation for the Python API, the number of iterations is assigned to the variable `n_estimators`, whilst γ is simply called `gamma`. The λ variable is called `reg_lambda`, in order to avoid conflict with the lambda functions in Python.

5.1. Tuning ARIMA Hyperparameters

For the ARIMA model, a grid search methodology was employed to determine the optimal parameters. First, we used the `adfuller` package in Python to perform an ADF test on the spread to determine if it is a stationary time series. We select 0.05 as the threshold for our p-value, and if $p < 0.05$, we simply set $d = 0$. The lag parameters are then initialised in the following way:

Parameter	Range
p	$[1, 4]$
q	$[1, 4]$

Then, we implement the following grid search algorithm:

Initialise P, Q, and D lists of integer ranges for (p, q, d)
best_score $\leftarrow \infty$
score $\leftarrow 0$
for p in P **do**
  for q in Q **do**
    for d in D **do**
      Fit ARIMA(p, q, d) model
      score $\leftarrow$ AIC/BIC of ARIMA(p, q, d)
      if score $<$ best_score **then**
        best_score $\leftarrow$ score
      end if
    end for
  end for
end for
return best_score

See `ARIMATuner.grid_search()` in `tuner.py` to see how this is implemented. In our implementation, P, Q, and D are stored in a dictionary whose keys are precisely labelled p, q, and d, and whose values are lists of integers, like the following:

```
param_space = {
    'p': [1,2,3,4],
    'q': [1,2,3,4],
    'd': [0, 1, 2]
}

arima_tuner = ARIMATuner()

best_params  = arima_tuner.grid_search(param_space, 'aic', spread_train)
```

Figure 3: Calling our custom method that tunes our ARIMA model using a parameter space dictionary, using AIC criterion. `spread_train` is the validation dataset of the spread time series S_t.

5.2. Feature Engineering for XGBoost

For the XGBoost model, several features were engineered from the dataset. For instance, the close prices of each stock was used to calculate the volaitility, which is a rolling standard deviation of the daily

returns.

The `yfinance` package gives the "high" and "low" for a given stock, which is the highest and lowest price that a stock has been traded at previously. The "open" and "close" prices are the prices of the stock when the exchange opens and closes, respectively. The volume is the number of stocks that are available for trading at the beginning of the trading day.

The features for each of the stocks that are being traded (labelled X and Y, respectively), are put into a `pandas.DataFrame` object, and then used to train the XGBoost model.

A couple of additional features were also engineered, such as the daily returns and historical volatility of each stock. Other features such as the day, month, and year from the training dataset was also considered in the training dataset. Additionally, lagged values of the stocks were also used as training data. For instance, a feature labelled `lag_n` denotes the value of the spread from n days ago. The following shows the code for generating the features, using some of our custom functions:

```
price_x = trainer.generate_features(price_x)
price_x = price_x.add_suffix('_X')

price_y = trainer.generate_features(price_y)
price_y = price_y.add_suffix('_Y')

df = pd.concat([price_x, price_y, spread], axis=1)

df = trainer.generate_out_of_sample_features(df, lags, target)
```

Figure 4: Code snippet using custom function `generate_features()` and `generate_out_of_sample_features()` method from `Trainer` class to generate features.

Then, `train_test_split()` from the `sklearn` package is used to separate the resulting dataframe into a training and validation set:

```
x = df.drop(target, axis=1)
y = df[target]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, train_size=0.8, shuffle=False)
```

Figure 5: Using `sklearn` to split out feature engineered dataset into a training and validation set.

5.3. Tuning XGBoost Hyperparameters

As aforementioned, there are a few hyperparameters that we also have to tune for XGBoost. These are `n_estimators`, `gamma`, and `reg_lambda`, which correspond to the variables T, γ, and λ from (3).

Another hyperparameter that is important is `max_depth`, which controls the depth of the regression trees that XGBoost produces. This is an important hyperparameter, since having a value of `max_depth` that is too high can easily lead to overfitting, and having it too low can lead to underfitting.

There is also another hyperparameter, called `learning_rate`, which controls how the weights at leaves of the regression trees are updated over each iteration. We select the following ranges for optimisation:

Hyperparameter	Range
`n_estimators`	Integer $[100, 1000]$
`max_depth`	Integer $[3, 13]$
`learning_rate`	Real $[0.01, 1.0]$
`gamma`	Real $[0.01, 5.0]$
`reg_lambda`	Real $[0.01, 1.0]$

Then, the `BayesSearchCV` package from `scikit-optimize` is then used to tune the hyperparameters, which uses Bayesian optimisation to find the optimal parameters for the XGBoost model. Unlike the ARIMA model, a grid search method would be too time-consuming, as some of the hyperparameters are continuous variables. Our custom function can be called as follows:

```
xgb_optimiser = XGBTuner().bayesian_optimisation(x_train, y_train)

print(xgb_optimiser.best_params_)
```

Figure 6: Calls the `bayesian_optimisation` method from class `XGBTuner`. The variables `x_train` and `y_train` are the independent and dependent variables

5.3.1. Bayesian Optimisation

Suppose that we have a learning algorithm given by f_γ, where $\gamma \in \Gamma$ are the hyperparameters of the algorithm, and Γ is a space of hyperparameters. Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)_{i=1}^N\}$ be a dataset of N instances. Suppose that the dataset follows some unknown probability distribution $(\mathbf{x}_i, y_i) \sim P_\mathcal{D}$. Then, the goal of Bayesian optimisation (BO) is to minimise the expected loss:

$$\mathbb{E}_{(\mathbf{x},y)\sim P_\mathcal{D}} \ell(y_i, f_\gamma(\mathbf{x}_i).$$

A holdout strategy is used, where the dataset is split into a training $\mathcal{D}_t$, and an evaluation dataset $\mathcal{D}_v$ (or a testing dataset). We then have the following optimisation problem:

$$\begin{aligned} \min \quad & \mathcal{L}(\gamma; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}_i, y_i)\in\mathcal{D}} \ell(y_i, f_\gamma(\mathbf{x}_i)), \\ \text{s. t.} \quad & \gamma \in \Gamma. \end{aligned}$$

Then, the BO algorithm can be summarised as follows:

Initialise γ_0 as first guess for hyperparameter
for $k < T$ **do**
  Select new γ_{k+1} by optimising "acquisition function" α
  $\gamma_{k+1} \leftarrow \text{argmax}_\mathbf{x}\, \alpha(\mathbf{x}; \mathcal{D}_k)$
  Plug into objective function to obtain y_{k+1}
  $\mathcal{D}_{n+1} \leftarrow \mathcal{D}_n \cup \{(\mathbf{x}_{k+1}(\gamma_{k+1}), y_{k+1})\}$
  Update "statistical model"
end for

There are generally many choices for a "statistical model" that can be used to fit $\mathcal{D}$, but a common choice is a "Gaussian process" (see [Shahriari et al., 2016], [Makarova et al., 2021]).

6. Pairs Trading using Tech Stocks

This section details our pairs selection procedure, as well as our procedure for determining entry and exit thresholds for long-short positions (or buy-sell signals).

When entering a *long position*, one simply buys the stock from the market, and we see a profit if the price of the stock moves up. The profit-and-loss that we make from this is then given by $X_{t+h} - X_t$ for some $h > 0$. When entering a *short position*, one borrows the stock from someone on the market, and then immediately sells it, with the expectation that the price of the stock will drop. If the price drops, then we can buy the stocks back at a reduced price, and then return it to the lender for a profit. Thus, the profit-and-loss is given by the $X_t - X_{t+h}$ for some $h > 0$.

This is a simplification of real-world trading scenarios. When buying a short contract, stockbrokers will typically require a premium, or require the stocks to be paid back with interest. Depending on the situation, short positions can also be very hard to source. For instance, the Goldman Sachs trading floor has a desk that is dedicated exclusively to sourcing short contracts.[2]

Similarly, when entering a long position there will typically be multiple sellers and buyers with different bid and ask prices, as well as transaction fees associated with buying a certain security. In very rare cases, there might even be liquidity issues with certain long positions. However, without access to stockbroker information, or order-book databases (which typically require considerable subscription fees), we were not able to implement this information into our model.

6.1. Pairs Selection

One of the big challenges of pairs trading is selecting a suitable pair of stocks to implement our strategy. In our approach, we choose a selection of stocks in the tech sector, those being: Apple (AAPL), NVIDIA (NVDA), HP (HPQ), AMD, IBM, MSFT, and Lenovo (LNVGY).

Pair selection was performed by testing each pair of stocks for cointegration using the `coint` function from `statsmodels`. This function implements a p-value test for cointegration from [MacKinnon, 1996]. We use $p < 0.05$ as our threshold for selecting a sufficiently cointegrated pair. A heatmap of the p-values for each of the stocks is given below:

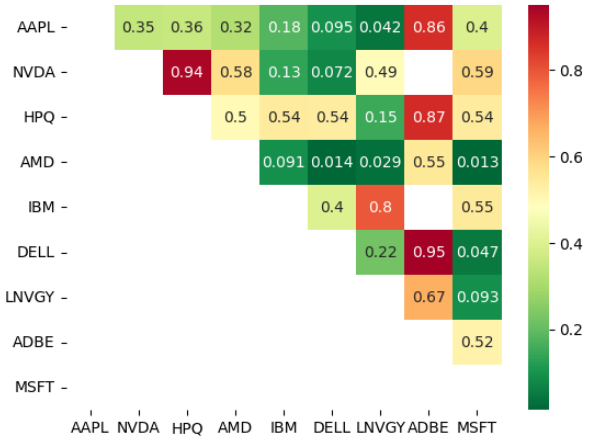


Figure 7: Heatmap of p-values from MacKinnon's test for cointegration.

[2]This information was obtained from a friend of the author.

From this, we can see that the p-value for AMD and MSFT are the lowest. Thus, we select this pair for our trading strategy. The code used to generate the heatmap is the following:

```
stocks = ['AAPL', 'NVDA', 'HPQ',
          'AMD', 'IBM', 'DELL',
          'LNVGY', 'ADBE', 'MSFT']

pairs_finder = PairsFinder()

pairs = pairs_finder.plot_coint_heatmap(stocks)

print(f'Selected pairs: {pairs}')
```

```
Selected pairs: [('AAPL', 'LNVGY'), ('NVDA', 'DELL'), ('AMD', 'DELL'), ('AMD', 'LNVGY'), ('AMD', 'MSFT'), ('DELL', 'MSFT')]
```

Figure 8: `PairFinder()` is a custom class that contains methods for identifying suitable pairs of stocks. The `plot_coint_heatmap()` method generates the above heatmap of p-values, and also returns all the pairs of stocks that have $p < 0.05$.

When implementing a pairs trading strategy, it is important to determine how much of each stock to buy or sell when significant price divergences are detected. The ratio of how much we buy to how much we sell is called the *hedge ratio*. Our method for determining the appropriate hedge ratio is relatively simple. We take the log prices of MSFT and AMD stocks, and plot them against each other. Then, the line of best fit was calculated using the `LinearRegression` package from `scikit-learn`, which fits a linear equation using ordinary least squares. From this, we obtain the following plot:

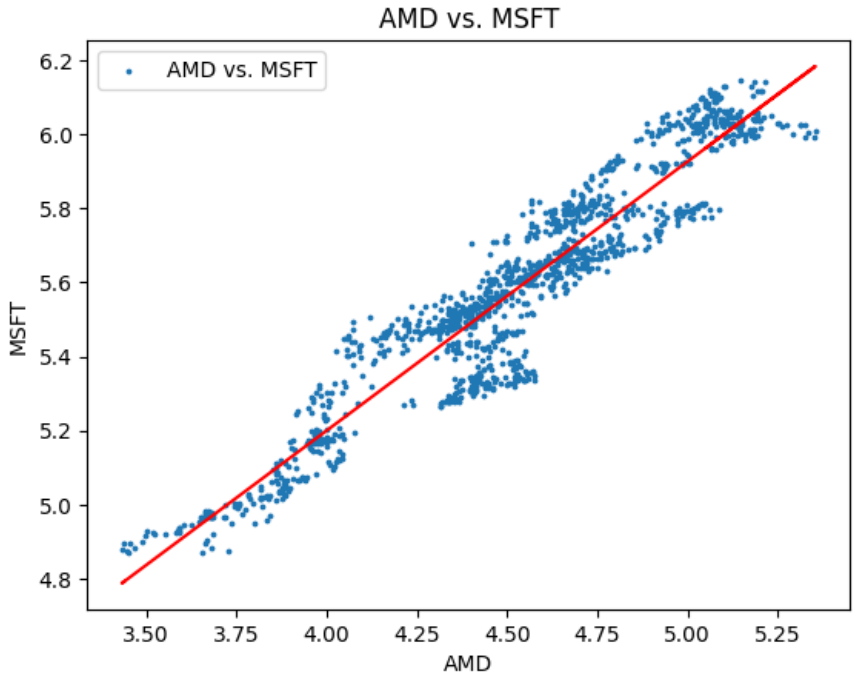


Figure 9: AMD and MSFT log prices plotted against each other, and with line of best fit (in red).

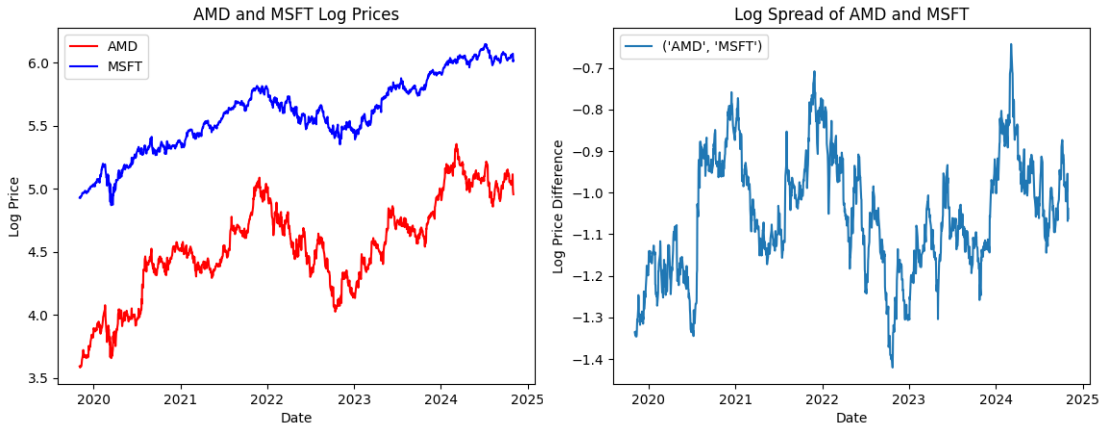
The code used to generate the scatter plots, and calculate the hedge ratio can be given by the code snippet below:

```
pair = ('AMD', 'MSFT')

pairs_finder.plot_hedge_ratio(pair)

_, hedge_ratio = pairs_finder.get_hedge_ratio(pair)

print(f'Hedge Ratio: {hedge_ratio}')

plt.title(pair[0] + ' vs. ' + pair[1])

plt.show()
```

```
Hedge Ratio: 1.0
```

Figure 10: `plot_hedge_ratio()` is another method from the `PairsFinder()` class that plots a scatter plot of both stocks, and draws the line of best fit through the plot. The slope of the line (when rounded up or down) is the hedge ratio. The function also calculates the hedge ratio for us, and returns it.

As we see, the hedge ratio is 1. So, when we short/long the spread, we simply buy one of the stocks, and then take out a short contract on the other. The log spread and log prices of both stocks can be seen below:



(a) Log prices of AMD and MSFT.

(b) Log price spread of AMD and MSFT.

Figure 11: Autotrader behaviour based on trading signals from ARIMA model.

The ADF test on the spread gave a p-value of 0.02, which indicates that the series is stationary:

```
pairs_finder.plot_spread(pair)

spread = pairs_finder.get_spread(pair)

spread_train, spread_test = train_test_split(
    spread,
    test_size=0.2,
    train_size=0.8,
    shuffle=False
)

trainer = Trainer()

_ = trainer.is_stationary(spread_train)
```

```
p-value of ADF test: 0.02209405949535845
The time series is stationary
```

Figure 12: Code snippet of `is_stationary` from our `Trainer` class being used to determine whether or not the spread is stationary. `train_test_split()` is used to split the spread time series into a training and validation set, and then `is_stationary()` is used on the training set.

6.2. Trading Strategy

Let X_t denote the log price of AMD, and Y_t denote the log price of MSFT, and $S_t := X_t - Y_t$ denote the spread of the time series. We adopt a trading strategy that compares the value of the spread forecasted by our models against the moving average and standard deviation of the spread. Denote by $\widehat{S}_{t+1}$ the value of the spread forecasted at time $t+1$ by the ARIMA/XGBoost model. Then, we construct a moving average, with upper and lower bands given by a moving standard deviation. Let μ_t and σ_t denote the moving average of the actual spread at time t. Then, our trading strategy is as follows:

Obtain $\widehat{S}_{t+1}$ using our choice of model
Compute lower band $\mu_t - 2\sigma_t$
Compute upper band $\mu_t + 2\sigma_t$
while $t < T$ **do**
  if $\widehat{S}_{t+1} > \mu_t + 2\sigma_t$ **then** ▷ When next-day forecast breaches upper band
    Enter short position on spread
  else if $\widehat{S}_{t+1} < \mu_t - 2\sigma_t$ **then** ▷ When next-day forecast breaches lower band
    Enter long position on spread
  else ▷ When spread returns to normal
    Exit positions
  end if
end while

We compare this strategy is compared against a *buy-and-hold* strategy, wherein one simply takes positions in the two stocks, and then holds it for the entire duration of the trading period. The buy-and-hold strategy will be used as a baseline. The *alpha* of our strategy is then calculated as the percentage performance improvement of our trading strategy over the baseline strategy.

To measure the performance of our models, we use the Sharpe ratio, which is a measure of a portfolio's performance relative to a risk-free asset. The risk-free asset is typically taken to be the yield of 10 year US treasury bonds, but in practise, it is often simply set to zero. The *annualised Sharpe ratio* is given by

$$\text{Sharpe} = \sqrt{252} \cdot \frac{\mu_{\text{PnL}}}{\sigma_{\text{PnL}}},$$

where μ_{PnL} is the average profit-and-loss, and σ_{PnL} is the standard deviation of the profit-and-loss. The factor of $\sqrt{252}$ is due to the fact that there are 252 trading days in a year. The Sharpe ratio gives us a measure of the performance of our portfolio after adjusting for risk.

A portfolio that generates consistent returns would have a much lower value of σ_{PnL}, and thus lead to a higher Sharpe ratio. In contrast, a risky investment would lead to a much higher σ_{PnL}, and thus a lower Sharpe ratio.

The return on our portfolio is calculated using the formula:

$$\frac{\text{Final position} - \text{Initial position}}{\text{Initial position}} \times 100\%.$$

We assume that we have an opening balance of $10,000$\$. The buy-and-hold strategy involves buying as many AMD/DELL shares using half of that money, and then holding onto them.

6.3. Results

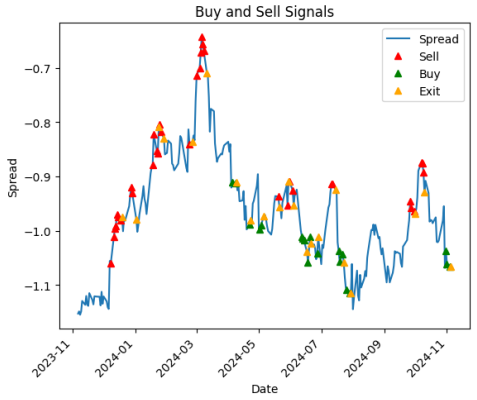
The *alpha* is the amount by which our trading strategy outperforms a baseline strategy – which, in this case is our buy-and-hold strategy. We record the performance of our ARIMA and XGBoost pairs trading strategy against the buy-and-hold baseline strategy:

	Baseline	ARIMA	XGBoost
Return	25.2%	8.41%	24.79%
Sharpe	0.87	3.45	5.43
Alpha	0%	−16.79%	−0.41%

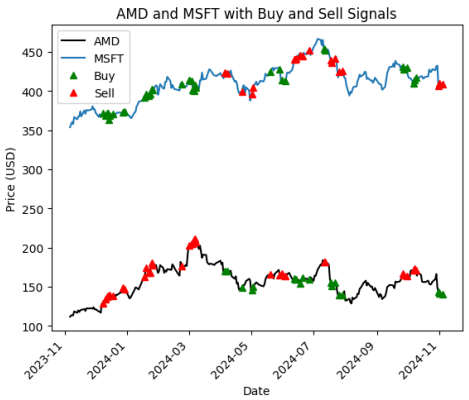
The buy-and-hold baseline strategy yielded more returns than the ARIMA model, but has a smaller Sharpe ratio, since the buy-and-hold strategy has more directional exposure. That is, simply buying a lot of stocks and then holding onto it, exposes the portfolio directly to market movements. As such, the σ_{PnL} of the buy-and-hold portfolio will be larger, which reduces the Sharpe ratio depsite the higher returns. In contrast, both the ARIMA and XGBoost strategies hedges its bets, and exits its position depending on market conditions, which led to more consistent returns.

So, while the baseline strategy can be profitable as well, it also exposes the portfolio to more directional risk. Our pairs trading strategy offers a way to mitigate such risks.

The ARIMA autotrader behaved in response to trading signals generated by the ARIMa model in the following way:



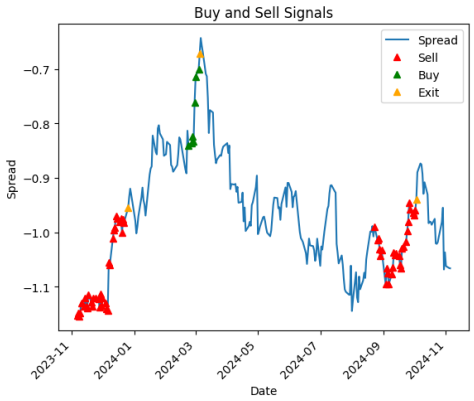
(a) Times at which ARIMA autotrader longed (green marker) or shorted (red marker) the spread. Orange markers denote times at which the autotrader exited its long and short positions.



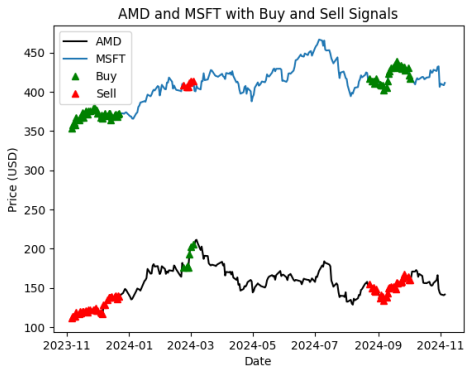
(b) Times at which the ARIMA autotrader bought (green marker), or took our short contracts (red marker) on each stock.

Figure 13: Autotrader behaviour based on trading signals from ARIMA model.

The autotrader behaved in response to trading signals generated by the XGBoost model in the following way:



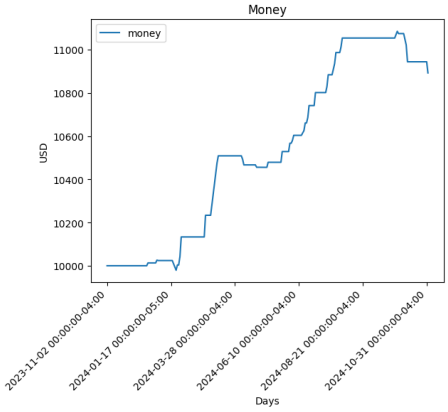
(a) Times at which XGBoost autotrader longed (green marker) or shorted (red marker) the spread. Orange marker denotes times at which the autotrader exited its position.



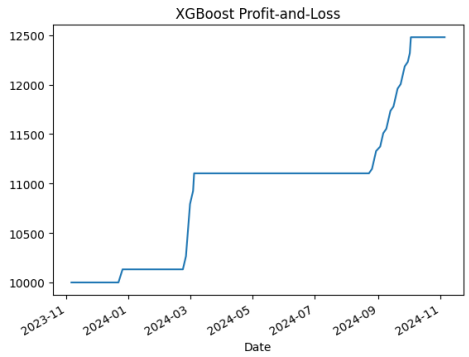
(b) Times at which the XGBoost autotrader bought (green marker), or took our short contracts (red marker) on each stock.

Figure 14: Autotrader behaviour based on trading signals from XGBoost model.

The resulting profit and loss from both of these trading strategies are given below:



(a) Profit and loss from ARIMA model.



(b) Profit and loss from XGBoost model.

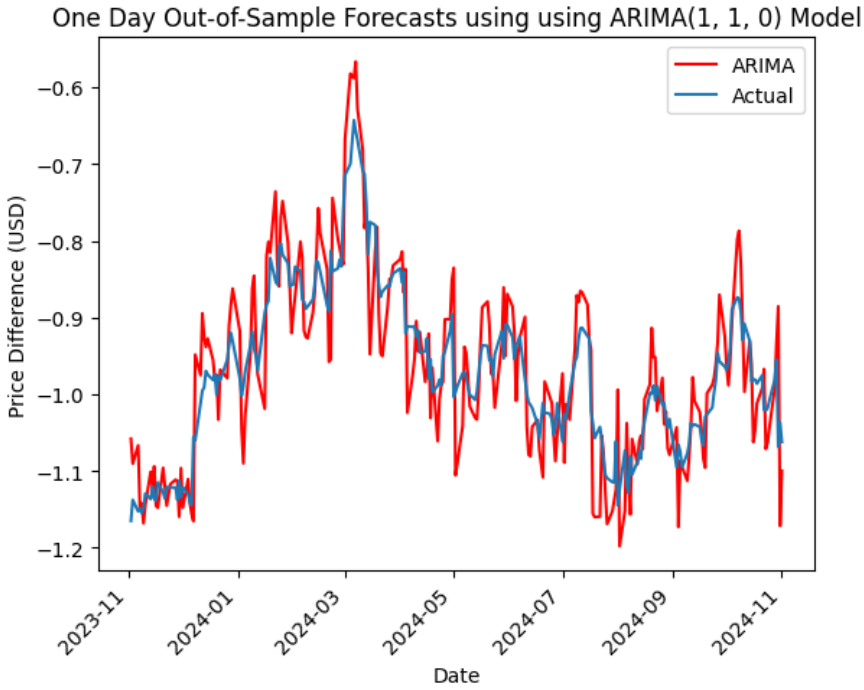
Figure 15: Profit and loss graphs, assuming initial balance of $10,000$\$.

6.4. Model Evaluation: ARIMA

An augmented Dickey-Fuller test was used on the training set using the `adfuller` package from Python. This returned a p-value of 0.013, which indicated that the time series was stationary. Thus, the series did not need to be differenced, and we could set $d = 0$. The ARIMA hyperparameters were tuned using a grid search algorithm that used AIC as the scoring function.

Hyperparameters that produced the lowest AIC was selected for the model. This process determined

that $(p, q, d) = (1, 1, 0)$ as the optimal hyperparameters. Forecasting was done out-of-sample by fitting an ARIMA model on the training dataset, and then using that to predict the value of the spread the next day. Then, the next day's observations are appended to the training set, and the ARIMA model is re-fitted. We obtain the following predictions from the ARIMA model:



An ARIMA$(1, 1, 0)$ is the same as an ARMA$(1, 1)$ model, which predicts the next-time step by fitting the equation:

$$X_t = \alpha X_{t-1} + \beta \varepsilon_{t-1} + \varepsilon_t.$$

The RMSE of the predictions is 0.056, which indicates that the model fits fairly accurately. However, the model experiences some fluctuations in its prediction when the spread experiences abrupt upwards or downwards tends.

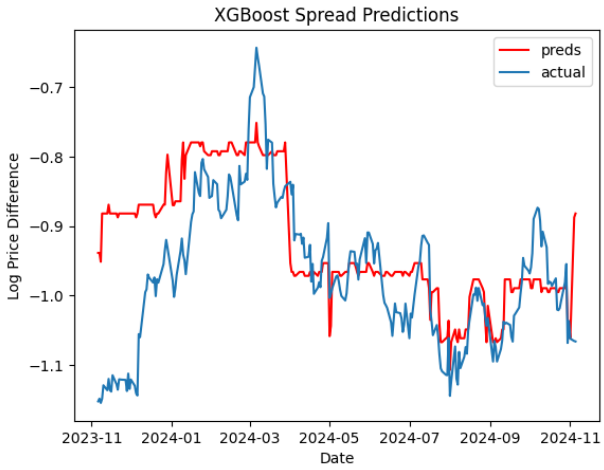
This is because the ARIMA$(1, 1, 0)$ only looks back one time-step to fit the coefficients of the model. So, if there is a sudden fluctuation in the spread, the model will need to record the observations from that day before fitting it, which results in the predictions lagging slightly behind the actual values of the model, or over-predicting the peak.

In relation to the trading strategy, the strategy overall performs quite well, though we experience some losses around January of 2024, and April of 2024. Around January, the autotrader shorted the spread, since it deteced sudden spikes in the spread. However, looking at the actual stock prices, we can see that the autotrader actually bought MSFT at a dip, and then shorted AMD at a peak, which thus resulted in a loss after selling off its positions.

Similarly, around April, there was a large peak in the spread, occurring at April, at which point the autotrader correctly shorted the spread. However, that resulted in the autotrader buying AMD at a peak, and then exiting the position after the price drops, which resulted in a loss.

6.5. Model Evaluation: XGBoost

Backtesting our XGBoost model on our training dataset yields the following predictions on the testing dataset:



The model exhibits poor performance around the beginning of the trading year, and overpredicts the spread, which led to the autotrader entering a significant amount of short positions. Meanwhile, the spread was actually increasing. This typically would have resulted in a loss, but the autotrader was still able to cover the losses with its substantial position in MSFT, which was beginning to trend upwards.

Let us evaluate what might have led to this poor performance. A *learning curve* is a plot of how our model learns with "experience" (estimated in sample sizes N). It works by training the model on increasing larger subsets of the training and validation dataset, and then recording its errors. The learning curve for our XGBoost model is given by:

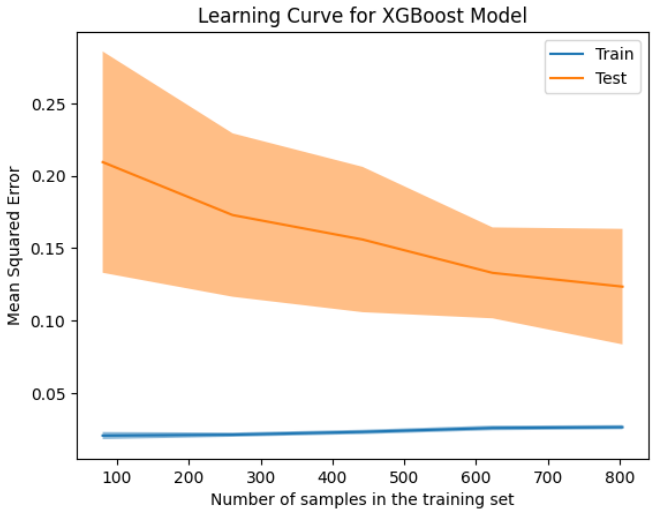


Figure 16: Learning curve of XGBoost model.

By inspection, we see that the error on the training dataset is very low over low sample sizes, but increases slightly as the number of samples increase. The error of the testing dataset is high at first, but then

slowly decreases over time, but does not converge to the testing error rate.

The learning curve shows that the model struggles to generalise the training dataset, which is indicative of overfitting. This occurs when the model fits the training data too closely, to the point where it learns the noise in the data. This results in low predictive accuracy when it attempts to generalise to unseen data.

To further diagnose the potential cause of this overfitting, we look at the feature importances of the model:

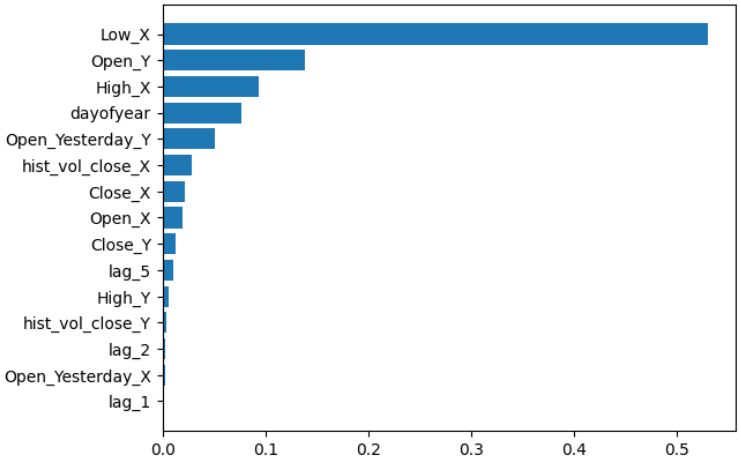


Figure 17: Feature importances determined using loss change for each split from trees.

Here, features with a subscript `_X` are features relating to the AMD time series X_t, and features with a subscript `_Y` relate to the MSFT time series Y_t. The `lag_n` values are lagged values of the spread $S_t = X_t - Y_t$ by n days.

From the plot, we can see that the `XGBoost` model is relying quite heavily on the `Low_X` feature in order to make its predictions, as that particular feature is ranked much higher than the other features. It is possible that the model is biased towards the `Low_X` feature, and is overly dependent on it during training. This could thus lead to poor performance on the testing data.

There is also literature that suggests that BO can cause overfitting, since the algorithm works by iteratively minimsing a loss function [Makarova et al., 2021]. A way to combat overfitting is to define an appropriate stopping criteria for the algorithm, but this is still an active area of research [Nguyen et al., 2017].

Regardless, the model performs quite well. It spends most of its time buying or selling the spread, and then holding on to the position for a while. The model only exits its position a total of 3 times throughout the trading year. Around March of 2024, the XGBoost model underpredicted the peak, which caused it to take long positions when the spread was actually increasing. However, it was able to exit the position just as MSFT was approaching its peak, which generated profits.

After the spike in March, the model seems to perform reasonably well, as there are no rapid fluctuations in the dataset. As such, the model was able to take some profitable short positions starting from around August, and then exiting it in October for a profit.

6.6. Strategy Evaluation

A clear drawback of our pairs trading strategy is that there are many pricing signals in the market that are not being capitalised upon. By only looking at the spread, we lose a lot of information about the underlying stocks in the portfolio, which – as we have seen – can lead to questionable trades. To counter this, one could run multiple trading strategies in tandem in addition to this one, though that can potentially lead to more issues, chief among those being how one should deal with conflicting trading signals from different strategies.

Moreover, it might be possible to use multiple models in conjunction in order to generate higher returns. As we have seen, the XGBoost model excels at predicting when the value of the spread becomes abnormally low, but fails to capture large upward-trending fluctuations in the spread. The ARIMA model gave more balanced trading signals, with a mixture of buy and sell signals for the trade. It is also better at knowing when to exit its positions. However, the XGBoost model outperformed the ARIMA model in terms of returns since the ARIMA occasionally made trades that lost money.

It follows that we can expect to make more money if the predictions from both models were instead combined. However, the high variance from the XGBoost model could also mean that we potentially put ourselves at risk of losing money.

In general, there are also many different methodologies for computing the entry-exit threshold for long short positions. The Bollinger band threshold that we employ here is comparatively quite primitive. It is possible that with a more sophisticated entry-exit strategy, we will be able to identify more buy-sell signals that we can act on, and thus increase our potential profit.

7. Conclusion

We explored the usefulness of XGBoost and ARIMA in pairs trading by using these models to generate trading signals. Our model choice was inspired by [Figueria, 2022], who also used ARIMA and XGBoost, as well as [Uyumazturk and Portilheiro, 2017], who used an AR model to perform pairs trading.

We found that our XGBoost model outperformed the ARIMA model, but the profits were mostly due to luck, as the XGBoost model was overfitted, and was performing poorly on the validation set. The ARIMA model, on the other hand, fit the validation set quite well, and was giving more accurate predictions of the spread.

The ARIMA strategy was also profitable, but the XGBoost overall yielded higher returns since the ARIMA strategy occasionally lost money. Neither strategy was able to outperform the baseline buy-and-hold strategy. However, both pairs trading strategies had much higher Sharpe ratios than the baseline strategy, which indicates that the pairs trading strategy is less risky than just buying and holding stocks.

The XGBoost model was overfitted due to a combination of its over-reliance on the `Low_X` feature, as well as the use of Bayesian optimisation, which is known to cause overfitting.

Future directions for research could involve selecting different models to see how performance is affected. For instance, random forest regressors could be used, which is another ensembling tree model that reduces bias. Different approaches to feature engineering could also be considered, with a view towards eliminating bias towards certain features during training. There is also an extensive wealth of literature concerning entry-exit thresholds for pairs trading strategies which could be explored. There are also many different stochastic times series modelling techniques that can be considered, such as Ornstein-Uhlenbeck

processes.

References

[Breiman et al., 1987] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1987). *Classification and Regression Trees*. Chapman and Hall/CRC.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge and Discovery and Data Mining*, pages pg. 785–794.

[Figueria, 2022] Figueria, M. (2022). Machine Learning-Based Pairs Trading Strategy with Multivariate Pairs Formed with Multi-objective Optimization. *Master's Thesis, Técnico Lisboa*.

[MacKinnon, 1996] MacKinnon, J. (1996). Numerical Distribution Functions for Unit Root and Cointegration Tests. *Journal of Applied Econometrics*, 11(6):pg. 601–618.

[Makarova et al., 2021] Makarova, A., Shen, H., Perrone, V., Klein, A., Faddoul, J., Krause, A., Seeger, M., and Archambeau, C. (2021). Overfitting in Bayesian Optimization: an empirical study and early-stopping solution. *Pre-print*.

[Nguyen et al., 2017] Nguyen, V., Gupta, S., Rana, S., Li, C., and Venkatesh, S. (2017). Regret for expected improvement over the best-observed value and stopping condition. *Asian Conference on Machine Learning*, pages pg. 279–294.

[Puspaningrum et al., 2009] Puspaningrum, H., Lin, Y., and Gulati, C. (2009). Finding the Optimal pre-set Boundaries for Pairs Trading Strategy Based on Cointegration Technique. *Centre for Statistical Survey Methodology Working Paper Series*.

[Richard, 2006] Richard, B. (2006). *A Demon Of Our Own Design: Markets, Hedge Funds, and the Perils of Financial Innovation*. Wiley.

[Shahriari et al., 2016] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., , and de Freitas., N. (2016). Taking the human out of the loop: A review of Bayesian optimization. 104(1):pg. 148–175.

[Uyumazturk and Portilheiro, 2017] Uyumazturk, B. and Portilheiro, V. (2017). Rise and Fall: An Autoregressive Approach to Pairs Trading.

[Zeng and Lee, 2014] Zeng, Z. and Lee, C. (2014). Pairs trading: optimal thresholds and profitability. *Quantitative Finance*, 14(11).

[Zhou, 2012] Zhou, Z. (2012). *Ensemble Methods: Foundations and Algorithms*. CRC Press.