Extreme Gradient Profit

MA 585 Final Project, Spring 2023

Yunhong Bao, Eli Gnesin, Su Lee, Thomas Misikoff, Will Tirone

MA 585 - Algorithmic Trading
Final Project

Duke University
April 27th, 2023

Abstract

This paper presents a novel approach to algorithmic trading that combines fundamental universe selection, regression using Gradient Boosting, linear programming to choose portfolio weights, and rigorous risk management controls to optimize portfolio performance. Our methodology involves selecting a subset of stocks from a large universe of U.S. equities based on fundamental criteria such as earnings, debt, and valuation metrics. We then use Gradient Boosting to predct returns the selected stocks as either buys or sells based on their expected performance. To construct an optimal portfolio, we use a linear programming model that maximizes the Sharpe Ratio while imposing constraints on risk and exposure. Our results show that the proposed approach outperforms the market during periods of economic crisis but underperforms during bull markets. Furthermore, we demonstrate the effectiveness of our risk controls in mitigating tail risk and preserving capital during market downturns. Overall, our findings suggest that incorporating fundamental universe selection, Gradient Boosting, linear programming, and risk controls can lead to superior investment outcomes with algorithmic trading during bear markets.

Contents

1 Introduction and Rationale

Our goal is to combine a fundamental-indicator value-based machine learning approach with algorithmic stock selection and portfolio optimization to maximize expected returns across a diverse stock portfolio while constraining our risk exposure. To accomplish the goal of generating profit while minimizing risks, our algorithm takes the following steps:

1. Filter to a universe of 10 securities with desirable value-investing indicators.
2. Train a gradient boosting regression model on each security using lagged indicators, lagged price, and news sentiment about the security to calculate expected returns.
3. Train gradient boosting quantile regression models on securities to predict the 5th (for stocks with positive expected returns) or 95th (for stocks with negative expected returns) quantile returns for each security.
4. With the expected returns and quantile returns for each security, use linear programming to find the optimal weights for a portfolio.
5. Use risk management controls such as Value-at-Risk (VaR) to ensure our portfolio meets desired criteria and targets and is not at risk of overexposure.
6. Allocate our portfolio to the selected securities with the risk-managed optimal weights.
7. After given holding periods, select a new universe, retrain the models, and re-balance the portfolio weights.

Our algorithm is constructed with some main focuses in mind. First, by filtering to a manageable, but not small universe of securities, we hope to be able to create diverse portfolios that are not overly focused on single industries or sectors. Then, by using gradient boosting, we hope to gain the benefits of iterative training on lightweight models without making any further assumptions about the underlying distribution of returns or the relationship between lagging indicators and sentiment analysis to the data. Gradient boosting further allows for quantile loss, providing an effective and consistent method of considering returns scenarios in the direction opposite to the predicted expected returns, again without requiring any normality assumptions on the distribution of stock returns or a linear relationship between the predictors and the target.

Then, we want to use linear programming to choose a portfolio, given the predictions made by the gradient boosting models, that maximize the expected returns, while simultaneously providing a bound for a "worst case" scenario based on the quantile returns. We also use this linear programming step to force diversity among portfolio holdings by restricting the weights of individual holdings.

Finally, before setting our holdings, we want to manage our risk with the optimal portfolio by capping the Value at Risk from our portfolio as a whole and from each individual security within our portfolio.

Initially, we had intended for this algorithm to be implemented with regular universe selection, model training, and portfolio re-balancing, at least weekly if not daily. However, due to computational constraints, addressed in Section 6, the time period for our model was resolved to be monthly universe selection and approximately weekly model training and portfolio weighting.

2 The Algorithm

2.1 Fundamental Selection Criteria

Though there is significant debate in the academic and investing communities about whether or not it is possible to generate abnormal returns, we believe a necessary tool to accomplish this is the use of fundamental indicators to remove stocks that have undesirable properties. We will use a slightly modified version of a Benjamin Graham [4] or Warren Buffett-style criteria:

1. Market Cap > \$300MM
2. Price > \$5 / Share
3. P/E < 100
4. Price / Earnings to Growth (PEG) < 3
5. Debt / Equity < 2
6. Current Ratio > 1

We are, of course, leaving out any consideration for dividends as we are more interested in growth of the stock rather than dividend income. If dividends are paid out, they are not being reinvested into the company and thus limits growth of share price. Piotroski gives evidence that identifying firms with high book-to-market value (BM) can provide a market-beating edge of up to 7.5%[7], though we found that this constraint limited our universe of stocks to only 3 or 4 and deemed it too restrictive. Thus, our goal is to provide a starting point that is not too restrictive that will later allow us to make the stocks more easily separable into winners and losers. Our reasoning for the criteria is as follows: a market cap > \$300MM will assure a large enough volume that there is enough liquidity to make trades on the stock, a price > \$5 will eliminate potential penny stocks, a P/E Ratio <100 will ensure that the company is not overvalued to an extreme degree, a PEG Ratio < 3 indicates the stock has a high chance for growth, a debt to equity ratio < 2 will avoid us choosing over-levered companies, and a current ratio > 1 ensures the company is able to meet its short-term liabilities.

Additionally, a dynamic universe selection avoids selection bias and survivorship bias of stocks. Since companies can be delisted over time, either voluntarily or involuntarily, it would introduce bias if we made a pre-determined list of stocks that were currently trading. For example, if we selected MSFT, GS, and JPM as stocks to trade, since they "survived" to the current day, it is biased against stocks that traded in the past but were delisted.

In our algorithm, this is accomplished in `SelectCoarse()` and `SelectFine()`, which are passed into `self.AddUniverse()` to dynamically create the active universe of securities that we will consider for trading. The initial universe considered in coarse filtering is the US Equity Security Master dataset provided by QuantConnect, which contains about 27,500 equities. The only parameter that is coarse filtered is the price of the stock. The rest of the properties, like the PE Ratio or Current Ratio are accessed in the Morningstar US Fundamental Data dataset. Since we have limited resources to train the Boosting model and we train the model on each security, we then select a maximum of 10 securities after fine filtering to consider trading. Universe selection is computationally intense, so to preserve RAM, we only perform selection on the first day of the month. With more resources, we believe more frequent selection would provide additional diversification benefits. We also utilize the QuantConnect event handler `OnSecuritiesChanged()` which triggers when securities move in and out of the universe. We found it particularly difficult to accurately retrieve the securities in the active universe, so we implemented code to manually keep track of them.

2.2 Gradient Boosted Decision Trees

Upon completion of the fine selection process, we use gradient boosted decision trees to predict the expected stock returns and the corresponding fifth percent or ninty-fifth percent quantiles. In this section, we introduce the gradient boosting algorithm and elaborate on the data collection process. Moreover, we discuss the hyperparameter tuning process and the computational constraints we are faced with on QuantConnect platform.

2.2.1 Decision Trees

Decision trees are a supervised learning algorithm used for classification and regression tasks. The goal is to predict the value of a target variable by learning decision rules inferred from training data features. The structure of a typical tree is as follows: starting with a root node, each internal node is a test or a decision on an attribute, each branch is an outcome of the test, and each leaf node is a class label or a numerical value.

Each internal node is a point where the feature space is split. If the feature space is m-dimensional, decision trees divide the feature space into non-overlapping m-dimensional rectangular regions and fit a simple model to each of the disjoint regions. Often this simple model is just a constant function. That is, for every observation that belongs to region R_j, we make the same prediction, which is the mean (for regression) or the mode (for classification) of the values of the training examples in R_j. The goal of the algorithm is to find regions $R_1, .., R_J$ such that the total loss

$$\sum_{j=1}^{J} \sum_{i \in R_j} \text{Loss}(y_i, \hat{y}_{R_j})$$

is minimized, where $\hat{y}_{R_j}$ is the predicted value for region R_j.

Decision trees can be grown in a top-down fashion, starting from the root node, where all observations belong to a single region. Then, successive splits are decided using a greedy approach so that the best split at that particular point is chosen, which minimizes the training error (mean squared error, Gini impurity measure, entropy, etc.). Since smaller trees with fewer splits are preferred for better interpretability and lower variance, large trees are grown first and then pruned afterwards to obtain a subtree, where the tuning parameter, which controls the trade-off between model complexity and fit to the training data, is chosen by cross-validation. [2]

One drawback of decision trees is instability, where a small perturbation in the data may result in a large change of the tree generated. In addition, decision trees are prone to overfitting the training data, especially as the depth of the tree increases. These problems can be resolved by using an ensemble model, which consists of a group of trees.

2.2.2 Gradient Boosting

Boosting is a method of accelerating the improvement in predictive accuracy to an optimum value. While decision trees are easy to interpret, a single tree alone may not yield the best results compared to other types of regression or classification algorithms. However, when a large number of trees are combined into an ensemble, dramatic improvements can be seen in prediction accuracy. Gradient boosted trees consist of a group of weak learners whose individual prediction accuracy is slightly higher than random guessing. The weak models are iteratively trained on a modified version of the data, where the weights of mispredicted samples are increased. Originally designed for classification problems, boosting can be largely beneficial for regression problems as well. Gradient-Boosted Decision Trees use decision trees as the weak models. In each iteration of the boosting process, a new decision tree is trained to predict the residual errors of the previous ensemble of trees. The output of the boosted tree model is the weighted sum of the predictions of each individual tree in the ensemble. Denote each weak learner as $\hat{f}^b$, boosting trees aims to minimize total loss defined as:

$$\sum_{i=1}^{n} \text{Loss} \left(y_i, \sum_{b=1}^{B} \lambda \hat{f}^b(x_i) \right)$$

where the loss function can be Gini impurity for classification problems or MSE for regression problems. The loss function defined above includes functions as parameters and cannot be optimized using traditional optimization methods in Euclidean space[3]. Thus, boosting models are trained in an additive manner. The procedure of training a boosted regression tree model is presented below[6]:

Step 1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.

Step 2. For $b = 1, 2, ..., B$, repeat:

(a) Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r).

(b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$$

(b) Update the residuals:

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$

Step 3. Output the boosted model:

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

As described in the algorithm above, several hyperparameter values need to be carefully considered for a boosting model. In real-life implementations, computational resources and memory load also need to be taken into consideration. Here we provide a brief discussion of three major hyperparameters that could be tuned.

1. The number of trees B. Since each iteration of boosting model trains on previous errors, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. The value of B can be tuned through cross-validation.

2. The shrinkage parameter λ, a small positive number. λ (η in XGBoost) is the learning rate of the boosting model. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Smaller values of λ generally require large values of B to acquire the desired accuracy. This shrinkage parameter is essential to prevent overfitting of the boosting model.

3. The max depth d allowed for each weak learner. Similar to decision trees, this max depth parameter controls the complexity of the boosted ensemble. Often d is chosen to be one so that each weak learner is a single stump. In this case, the boosted ensemble is a simple additive model which provides interpretability. For large d, the model is likely to overfit due to increased complexity. This max depth parameter is usually tuned through cross-validation.

2.2.3 Features (Data Aggregation)

Data aggregation is an essential part of our algorithm and model training. Inspired by momentum trading, we decided to train our model with historical pricing data and various indicators. Historical pricing data include open, close, high, low, and volume. Indicator data includes simple moving averages, Bollinger bands, and relative strength index. Data aggregation is performed in our custom function `get_all_data()`. The function takes a variety of steps to get historical data cleaned and ready for training or prediction. One of the most significant challenges we faced was accurately retrieving data for the stocks in our active universe. Ultimately, we managed to achieve this by passing in the list of tickers we manually created in `OnSecuritiesChanged()`.

2.3 Historical Price and Indicators

The methods of accessing data are somewhat different across different QuantConnect datasets which provided significant hurdles. Utilizing `self.History(ticker, lookback)` retrieves data with a lookback based on `self.Time` which, during backtesting, is the historical time from `Initialize()`. However, historical data of indicators, like RSI, etc., are not available with a self-based method and are only accessible by using a QuantBook object from the research environment in `main.py`. However, we discovered after implementing a QuantBook-based approach that it used a significant amount of memory that caused backtesting to crash. Thus assessing model performance over the given backtest period became difficult. One approach we came upon was to dice the backtest period into multiple time chunks and aggregate PnL statistics to assess the overall model performance. However, due to the complexity of compounding Sharpe ratios and managing unliquidated positions at the start of each backtest chunk, we aborted this approach. After investigation, we determined that QuantBook objects are only appropriate for research environments. To get the indicators, we manually calculated them in custom functions `bollinger_bands()` and `calculate_rsi()`.

We then have three scenarios for which we need to access data: training the model, backtesting the model, and live trading. To accomplish this, `get_all_data()` takes boolean arguments for historical, backtesting, and training. This divides the data retrieval into the appropriate time frame and lookback period. At a daily resolution, and for each stock, this gives us close, high, low, open, and volume. Then, we perform some calculations to pull in additional features like average gain, average loss, and bandwidth. For training, the function retrieves data for a predetermined lookback period. For backtesting, the function creates a data frame with one column containing the newest data on every trading day. For live trading, it gets all the real-time data from the market. Last, we use the retrieved data frame to create 5-day lags of the indicators with the Pandas `pd.shift()` function. Suppose the original indicator data frame has 50 rows and 14 columns. The lag function would create a data frame with 50 rows and 70 columns so that each data point contains information for today as well as the past 4 days.

2.4 Sentiment Analysis

QuantConnect provides access to datasets provided by Tiingo, a news and data aggregation service. To access this in our algorithm, we use the same historical approach as the indicators and add a subscription to the news source symbol. The resulting `TiingoNews` object includes data like `source, Url, PublishedDate, Description, and Title` for articles published during our lookback period. Using Python's `nltk` library, we performed sentiment analysis on the titles of all articles published in a day (since our resolution was daily) then combined them and appended to the main data frame. In Souza et. al., they note that "Twitter sentiment has a statistically-significant relationship with stock returns and volatility" and that "surprisingly, the Twitter's sentiment presented a relatively stronger relationship with the stock's returns compared to traditional newswires." [10] Since Twitter API access has been changed recently, and Tiingo data was freely accessible through QuantConnect, we hope that our news data analysis can perform similarly though note that Twitter sentiment data would likely provide superior performance. The analysis in the literature is also typically performed on a certain sector of the industry, like retail or automotive. Since we are industry-agnostic, sentiment analysis may be less effective.

2.5 Lookback Period Determination

One significant factor to consider is the length of the look-back period for model training. Large machine learning models usually prefer a longer lookback period with more data points. However, the length of look back period is restricted by the following factors:

1. QuantConnect live trading memory restriction: The live trading nodes we possess right now have 0.5GB of RAM, which largely restricts the allowable size of machine learning models and data to be used for model training. During live-trading deployment, we discovered that a look-back period of 50 days, as used in backtesting, would cause memory overload. More RAM is needed to improve the live performance of our algorithm.

2. QuantConnect backtesting runtime restriction: During backtesting, we discovered that QuantConnect would raise runtime errors if the code execution time for each trading day exceeds 10 minutes. This makes model training difficult since our algorithm requires the training of multiple models with large datasets. To combat this restriction, we utilized the `self.Schedule.On()` function of Quant Connect to schedule weekly training. However, the runtime cap for scheduled events is 1 hour in total so the same problem still exists.

3. Computational limits: One common way to resolve the runtime issue is through parallelization. Since our algorithm constructs separate models for each stock in the refined universe. Parallel model training would be ideal. However, QuantConnect has not yet developed the capacity for parallel backtesting. Thus, model training has to be done in a sequential manner.

Leveraging all the restrictions and the accuracy requirement of our predictive models, we eventually selected a lookback period of 50 days. However, to present the full capacity of our algorithm, we also try to divide backtesting periods into different chunks to allow larger model and training data sizes. More results will be discussed in the later sections.

2.6 Hyperparameter Selection and Model Training

With a functional data aggregation method, we are able to acquire data at any desirable time point and train the boosting models. In our algorithm implementation, model training is conducted on a weekly basis. Gradient-boosted trees are used as regressors to predict the 5-day expected return of chosen stocks. For stocks with positive expected returns, we use gradient boosting again to acquire a 5% quantile predictive model as a lower bound for our portfolio performance. For stocks with negative expected returns, a boosting model predicting the 95% quantile is trained. As discussed in previous sections, several hyperparameters for the boosting models should be selected. Ideally, such hyperparameters should be tuned through K-fold validation. However, due to the computational and memory constraints discussed above, embedding hyperparameter tuning in the trading algorithm is infeasible.

To select the best parameters without breaking any computational or memory constraints. We conducted extensive research in QuantConnect's research environment. A range of parameter values were tested. It is observed that a larger number of weak learners and a smaller learning rate would produce better results. However, such settings could cause memory explosions in live trading. It is ultimately decided that 150 weak learners with a no maximum depth and a learning rate of 0.05 provided a good balance of model size and accuracy. For consistency in hyperparameter selection and backtest results, we set an initial random seed across all of our Gradient Boosting models, and did not change that seed at any point during parameter selection or backtesting.

With the ideal hyperparameters in mind, we implemented our algorithm to train three gradient-boosting models (one for predicting expected return and one for predicting tail performance in each direction) for each selected security in the universe using Scikit-Learn's `GradientBoostingRegressor`. Models are trained on a weekly basis. How these models will aid trading decisions will be discussed in the next section.

2.7 Making Predictions

Once we have a set of trained Gradient Boosting models for each of our securities, the next step is to use those models to get an array of expected returns and expected quantile returns for our universe, which can be used to build the portfolio with the linear programming methods in Section 2.4. For our algorithm, we made the decision to make predictions on the first trading morning after the model was trained, which was either the first day of the month or any Monday morning, at 10:00 AM EDT. To make the predictions, we iterate through the securities in our active universe, and for each security, we gather the same data as was used in training the models, except we keep only the most recent "observation". We then use this data point to predict our expected return, μ_i, with the caveat that, if there is no model or prediction fails for some other reason, we predict $\mu_i = 0$. We then condition the expected quantile return based on μ_i. If we return $\mu_i > 0$, then we expect to have better success with long positions on the stock, so we want to bound our "worst case scenario" by predicting $q_{i,0.05}$ as the expected 5th quantile return for the security. Conversely, if we return $\mu_i < 0$, then we expect to have success with short positions on the stock, so our "worst case scenario" is when the stock goes up, and therefore our bound is $q_{i,0.95}$, the expected 95th quantile return for the security. Finally, if $\mu_i = 0$, either by happenstance or, more commonly, because we had a prediction or model failure, then we return $q_i = 0$, since we are unable to render judgement as to the direction of the stock's movement, and by the design of the weights optimization program below, we should never assign any weight to a stock which has 0 return because doing so would not help minimize our solution and we receive no benefit in any constraint by assigning weight to that stock.

2.8 Portfolio Weight Selection

Once we have the set of securities we want to consider for our portfolio, the final step is to determine the composition of those stocks in the final portfolio. The goal is three-fold: (1) to optimize the overall expected return from the portfolio, (2) without putting too much weight into any individual security, (3) bounded by a "worst case scenario" for portfolio performance. Given a set of returns for securities, as well as a "worst case scenario" and an upper bound for the weight of individual securities, this question becomes, in one sense, a convex optimization problem, one to which we can apply the methods of Linear Programming. Yen Sun, at Bina Nusantara University, explored a similar concept, specifically using Linear Programming to find the optimum portfolio combined with Mean-Variance theory (as is consistent with Markowitz Theory)

[11]. Sun's approach aimed to optimize the portfolio variance, by changing the weights of the stocks in the portfolio, subject to the constraints that the total weights of the stocks had to be equal to 1 and subject to a given expected return for the portfolio. Importantly, although Sun treats this as a linear programming problem (with the Microsoft Excel Solver tool), the function he aimed to optimize, which was the portfolio variance, is not a linear function, and therefore the problem is not an exact linear programming problem.

To resolve this, we propose the following linear programming model:

$$\begin{aligned} \min_{\mathbf{x}} & -\mu^T\mathbf{x} \\ \text{subject to: } & -q^T\mathbf{x} \leq 0.01 \\ & \sum_{i=1}^{n} x_i * \text{sgn}(r_i) \leq 1 \\ & \sum_{i=1}^{n} -1 * x_i * \text{sgn}(r_i) \leq 0 \\ & 0 \leq x_i \leq \min(0.6, \frac{3}{n}), \quad \mu_i > 0 \\ & \max(-0.6, \frac{-1.5}{n}) \leq x_i \leq 0, \quad \mu_i \leq 0 \end{aligned}$$

First, we seek to minimize the negative expected returns μ times the portfolio weights $\mathbf{x}$. This minimization is equivalent to the maximization function max $\mu^T\mathbf{x}$, but works with Python's Scipy minimizer, with which we built our algorithm. Initially, it would seem that we would never place weight on stocks with negative expected returns, because they will have positive coefficients in the minimization equation and therefore are anti-useful to minimization efforts. To resolve this, however, we restrict the bounds on each individual weight to be limited to the direction by which the product $\mu_i x_i \leq 0$, in the individual weight constraints addressed later.

The first constraint achieves the third goal outlaid originally; we want to bound our weights by a "worst case scenario" condition. Here, we again lean into Gradient Boosting. Specifically, we use the quantile loss function for Gradient Boosting Regressor models, as was trained above. More importantly, since we are able to train these models for any quantile, we trained both a 5th and 95th quantile boosting model for each stock. Since Gradient Boosting itself is a non-linear method and is distribution agnostic, this method does not depend on the assumption that a security's returns are normally distributed. Then, we choose to restrict the product of the quantile returns and the portfolio weights such that the portfolio losses would be capped at 1%. That is, if every stock (with the weight at the optimal value), has its quantile return in adverse direction (so 5th for stocks we have long positions of, 95th for stocks we have short positions of), we would expect to lose no more than 1% of our portfolio value. This constraint can be formulated as ($q^T\mathbf{x} \geq -0.01$, and then we multiply through by -1 to make the constraint an upper bound, rather than a lower bound, to fit into the Scipy framework.

For the weights themselves, two sets of constraints are necessary. First, we constrain the sum of the weights, multiplied by the array sgn(μ) to be less than or equal to 1. The sign function here does the work of converting all of our portfolio weights to positive values, such that we can confirm that the sum of weights in our portfolio does not exceed 1. This is equivalent to taking the absolute value of the weights, but the absolute value function is non-linear, so we instead manually flip the portfolio weights to be positive for this constraint. The adjacent constraint is that the sum of the portfolio weights, multiplied by sgn(μ), should be greater than 0, since it makes no sense to have a "negative" portfolio. We then multiply this constraint through by a -1 to align it with the structure of the Scipy optimization function. It is in the first of these constraints that our algorithm could allow for increased leverage by setting the sum of the weights of the portfolio to be greater than 1, but we did not do so in our implementation of our algorithm.

We then consider the weight of each potential stock in our portfolio. For each individual weight, it is necessary to cap the individual weight in order to achieve the goal of "not putting too much weight into any individual security". For this, we place the constrain that every weight x_i, for a stock with positive returns, should be greater than zero and less than the minimum of 0.6 and $\frac{3}{n}$ where n is the number of stocks considered for the portfolio. This minimum function is necessary because we do not want it to be possible for our entire portfolio, or even a large majority of it, to be comprised of a single stock. Thus, to ensure diversification, we allow no more than 60% of the portfolio to be comprised of a single stock, regardless of how many stocks we are considering. Then, if $n \geq 5$, $\frac{3}{n} \leq 0.6$, so we instead restrict each weight to be no greater than three times its weight if every stock were held as an equal proportion of the portfolio. In doing so, we require that, in any portfolio using all of our cash, we will have to have positions in multiple securities, and for $n = 10$, as in backtesting, we will have to hold positions in at least 4 securities to use our entire capital. For stocks with negative returns, the direction flips, and we instead constrain the stock weight x_i to be less than zero but greater than the maximum of -0.6 and $\frac{-1.5}{n}$, with n as the number of stocks considered. Through backtesting, we determined that allowing many large short positions was a risk we were unwilling to take, so we instead constrain the short position to be smaller, since $\frac{-1.5}{n} > -0.6$ for all $n \geq 3$, so the 60% cap on shorting stocks only appears in an instance where our portfolio is only considering a universe of two stocks, and otherwise we cap each individual short position lower so as to lower the risk of incurring a margin call on our positions. If we were to implement leverage using the previous constraint, the values of $\frac{3}{n}$ and $\frac{-1.5}{n}$ could be changed accordingly to allow for increased holdings of individual securities while still meeting the goal of diversification.

With this linear program setup, the algorithm then relies on Scipy's implementation of linear programming, provided in `scipy.optimize`. Scipy, notably, has deprecated use of the Simplex method in favor of the HiGHS solver, a revised implementation of the Simplex method proposed by Huangfu and Hall [5]. The algorithm then runs the optimization and, if the linear program converges, returns the weights as an array called `self.weights`. If the linear program does not converge, then the algorithm flags this for the user and returns back equal weights for every stock considered, in the direction given by the direction of predicted returns, $\text{sgn}(\mu)$. In a fully-operational environment, this flag would be passed to a trader who would be able to react to it before any trades were made, so the trader could determine why the linear program was unable to converge. Since our algorithm must be entirely self-contained, however, the next best scenario is to neutralize the risk by diversifying the portfolio as much as possible, or alternatively to make no trades at all. Given our holding period, however, we felt it was better to take on a diversified portfolio than to sit still for the length of the holding period.

2.9 Risk Management

2.9.1 Reality Modeling

To accurately backtest our strategy, we specify in `Initialize()` to use the Interactive Brokers (IB) brokerage model which sets default values for things such as slippage and fees. Fees on equities according to this model are \$0.005 / share with a \$1 minimum fee and 0.5% maximum fee [12]. However, the default slippage model for IB is zero slippage Since we want to model a more accurate scenario, we use QuantConnect's implementation of the Volume Share Model. If the volume of the current bar of data is zero:

$$\text{slippage }\% = (\text{volume Limit})^2 \cdot \text{price Impact}$$

And if the volume of the current bar is positive:

$$\text{slippage }\% = \min\left(\frac{|\text{order Quantity}|}{\text{bar Volume}}, (\text{volume Limit})\right)^2 \cdot \text{price Impact}$$

Thus, large buy or sell orders will impact the price in backtests which more accurately reflects reality. Additionally, to limit slippage, we only trade starting at 10:00 A.M., 30 minutes after the market opens, to opening avoid volatility. Because we are trading stocks with price > \$5 and market cap > \$300MM, this will prevent large bid-ask spreads and gives us order fills closer to the market price when the order was placed.

2.9.2 Value-at-Risk

Value-at-risk (VaR) is a measure of the maximum potential loss a portfolio could incur over a specified time period. We use the parametric method, also known as variance-covariance method, which calculates VaR as a function of mean and variance of the returns series. First, we compute the covariance matrix of the portfolio returns. We calculate the mean and the standard deviation of the returns. Then we calculate the inverse of the normal cumulative distribution function with a specified confidence level ($\alpha = 0.05$), standard deviation, and mean. The obtained value gives the VaR, which is the maximum loss the portfolio can incur in one day with 95% confidence. To obtain the VaR, over the period of n days, we multiply this value by $\sqrt{n}$. Mathematically, if X is the random variable for the portfolio loss, then VaR with confidence level α is related to X by the following equation:

$$P(X \geq \text{VaR}) = \alpha.$$

This method assumes the stock price returns are normally distributed. Below is a plot showing the distribution of returns for the 8 most traded stocks in our portfolio (PYPL, BABA, ATVI, ABT, V, CMG, LLY, PCLN, in order) during the 2017 - 2021 period, which shows that their historical returns are indeed approximately normally distributed.

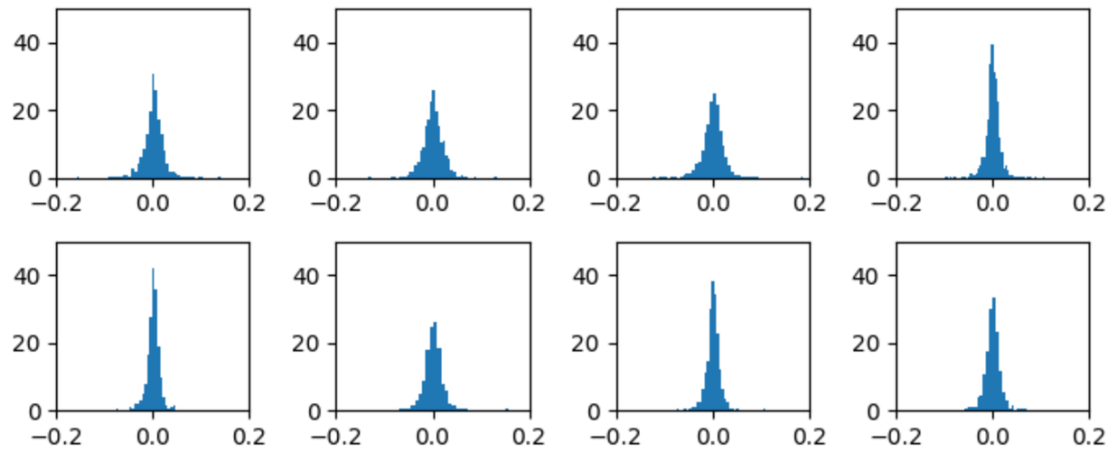

Figure 1: Distribution of returns in 2017 - 2021

We calculated the 95th percentile 1-day VaR each time before making a trade, with a VaR limit of -1.5% (maximum 1.5% loss). When our calculation showed that expected loss would exceed this limit, we reduced our portfolio position sizes accordingly so that our loss does not exceed the maximum risk threshold. Through backtesting, we found that reducing our positions to 60% of the weight that the optimizer returned was successful in mitigating these situations, so we chose to use this 40% reduction when the VaR limit was reached.

2.9.3 Stress Testing

To test our strategy against unpredictable scenarios and market shocks, and to see how well our risk management measures, such as VaR, perform, we conducted two stress tests. First, we conducted a stress test for the month of March 2020, to check for the resilience of our model to the market shock of the onset of the Covid-19 pandemic. Second, we conducted a stress test surrounding the 2008 Great Recession, to check the resilience of our model given both a market shock and an extended bear market of over a year. Detailed results and discussion of these backtests are in Sections 3.5 and 3.6.

3 Backtesting and Algorithm Performance

For the backtests below, each period started with \$10,000,000, and we have the following targets:

1. Drawdown duration $<$ 4 months
2. Sharpe Ratio $>$ 1
3. Drawdown $<$ 20%
4. Daily PnL Volatility $<$ 5% of Account Equity

We also note the win rate for each strategy, which is the proportion of trades that have a positive PnL. The only requirement during March 2020, was a drawdown $< 15\%$ and a net profit $> -15\%$. Additionally, in "Pseudo-Mathematics", the authors argue that, to avoid overfitting the model, with 4 years of in sample data, we could only try approximately 30 separate iterations of the model[1]. We kept this requirement in mind and utilized that many, but not more. We could have tuned further but would have risked overfitting which would harm performance in live trading.

3.1 In-Sample Backtest: 1/1/2017 - 1/1/2021

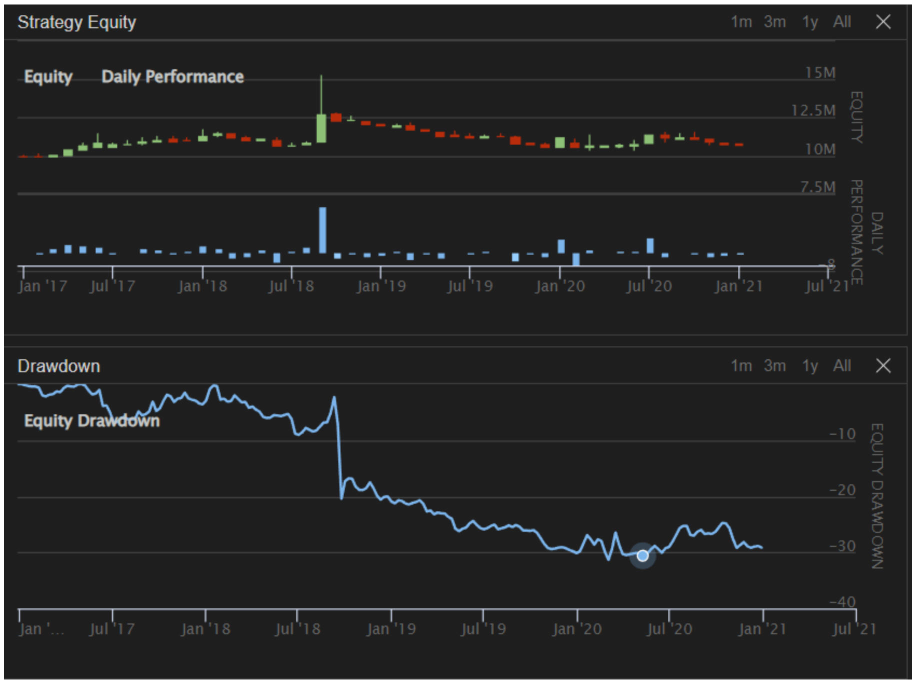

Figure 2: In-Sample Backtest: 1/1/2017 - 1/1/2021

For the in-sample backtest, we ran a single backtest over the 4-year period between January 1, 2017 and January 1, 2021, which is linked here. Overall, this backtest had a Sharpe Ratio of **0.157** and a maximum Drawdown of 32% with a Drawdown duration of 3 years between January 22, 2018 and the end of our backtest period. As seen in Figure 2, there is a spike during the week of September 17, 2018, during which the portfolio owned saw a profit of almost $3,000,000, and subsequently lost that entire profit over the course of the week. As a result of this spike and crash, we immediately saw a Drawdown of over 20%, and were unable to maintain the Drawdown target because of it. Likewise, though Drawdown duration exceeded 4 months in 2018, we would have reached a 0% Drawdown instance during that same week in September, and it is possible we would have been able to achieve shorter Drawdown duration across the rest of the backtest period as well. As discussed in Sections 3.1.1 and 6.3 below, in a real-life scenario, a human trader would likely take note of such a spike in profit and manually force a liquidation of the portfolio to capture and lock in those profits.

On the whole, the algorithm did achieve an overall Profit (PnL) of **7.21%**, which does mean that the algorithm turned a profit. More interestingly, it did so with a win rate of only 47%, which indicates that the average win was more successful than the average loss was detrimental. Looking at the securities which made up the portfolio, the largest volume traded securities in this backtest were PayPal, Alibaba Group Holding Ltd., and Adobe Inc., with other prominent securities including Visa, Chipotle Mexican Grill, and Abbott Laboratories. This suggests that our universe selection methods did successfully give our algorithm diverse stock universes, since we traded significant volumes of stocks from a variety of sectors and industries.

One other interesting note in this backtest was that, for the most part, the algorithm was stable, and even somewhat made a profit, during the market shock and instability caused by the Covid-19 Pandemic beginning in March 2020. From March 1, 2020 to the end of the backtest period, our algorithm had an approximately 1.6% return from its March 1 value, even as the country struggled through the early months of the pandemic and the instability of the 2020 presidential election.

3.1.1 Adjusted In-Sample Backtest with Profit Cap

To further explore the effect of the single spike week of September 17, 2018 on our algorithm, we ran an identical backtest, except for a single condition that liquidates the portfolio if the unrealized profit of the portfolio is greater than $2 million. This is a generally poor decision, artificially capping profits, while simultaneously being too high of an unrealized profit condition to handle most spike and crash scenarios. Furthermore, this conditional does not apply in any other backtest, nor in live-trading, so including it is deliberately fitting to maximize backtest profit, a poor motivation. In a real-world scenario, we would want our algorithm to, at a lower unrealized profit closer to 7.5% of portfolio equity, throw a flag so that a human trader could assess the situation and either reduce positions to capture profits or continue to maintain holdings with the expectation of even higher returns.

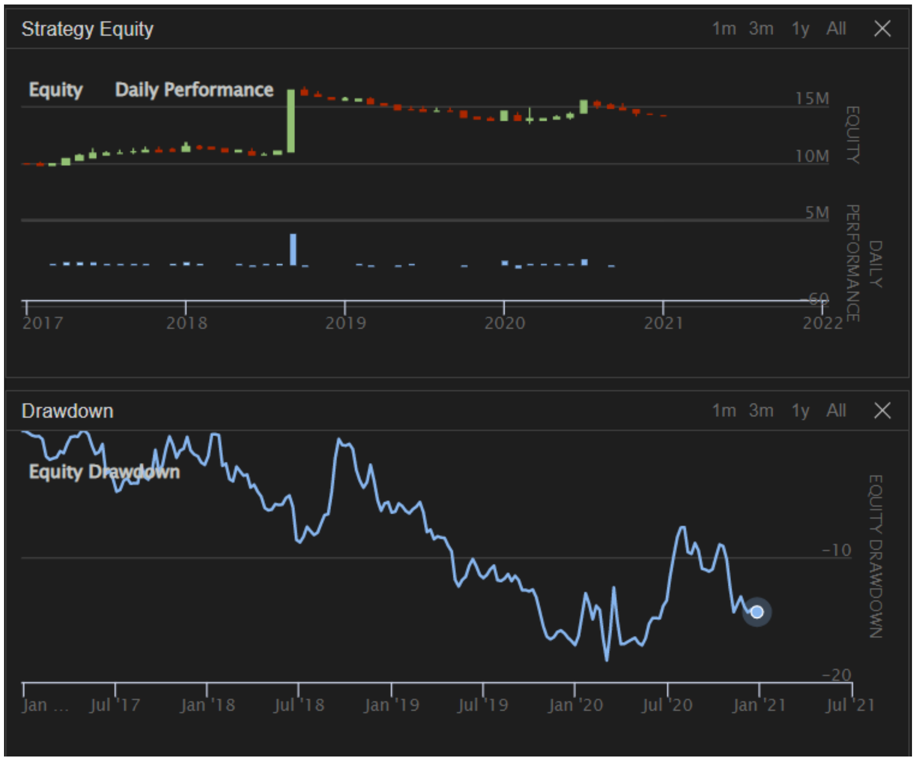

Figure 3: In-Sample Backtest with $2 Million Profit Cap

Looking at the results of this backtest, we can see that adding this conditional substantially improves the algorithm performance simply by liquidating with gains during the week of September 17, 2018. Overall, this backtest has a Sharpe Ratio of **0.572** and a net profit of **42.08%**, significantly higher than the initial backtest without the cap. Further, as noted above, the maximum Drawdown of this backtest is only 19.5%, within the 20% Drawdown target set for the backtest. Thus, this backtest is likely a much better representation of the actual efficacy of our algorithm, though it includes a conditional we choose not to include in our algorithm because it is a poor policy decision.

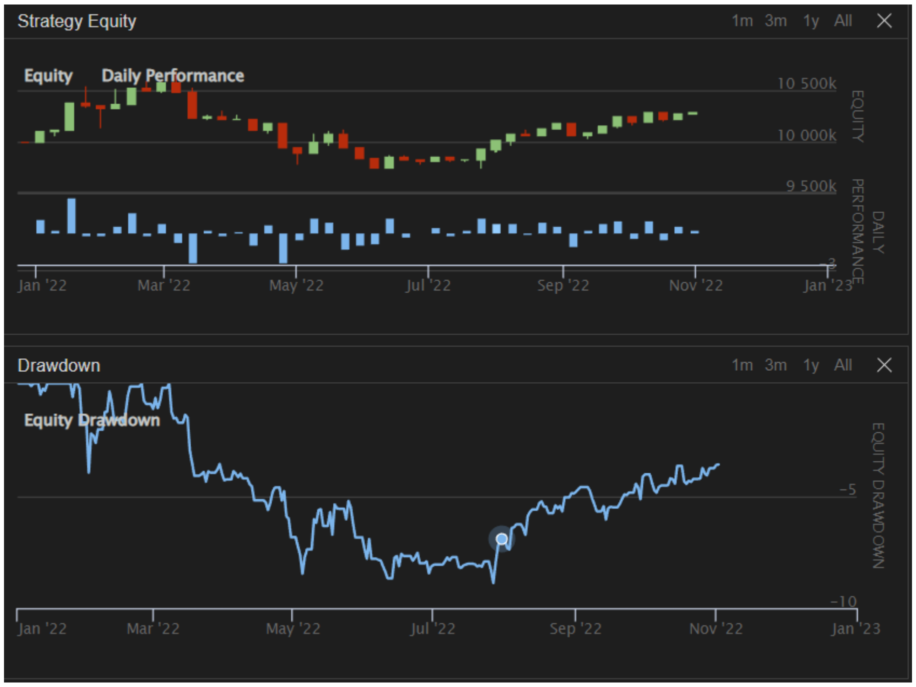

Figure 4: Out-of-Sample Backtest A: 1/1/2022 - 11/1/2022

3.2 Out-of-Sample Backtest A: 1/1/2022 - 11/1/2022

For the first out-of-sample backtest, we ran a backtest over the period between January 1, 2022 and November 1, 2022, which is linked here and displayed in Figure 4. Overall, this backtest had a Sharpe Ratio of **0.376** and an overall profit of **2.94%** on a maximum Drawdown of 8.900%, which means this backtest did satisfy the target of less than 20% Drawdown. We did, however, have a Drawdown duration of 8 months, even though the maximum Drawdown was significantly lower than the in-sample backtest.

Interestingly, our algorithm performed better in the Out-of-Sample test than it did in-sample. This is perhaps unsurprising, because although we used backtesting to decide on parameters such as the number of symbols in the universe and the lookback period, we were also significantly constrained by computational constraints, as discussed in Section 6.2. While active, however, the algorithm conducted its own universe selection, training, prediction, and portfolio building, so there was little opportunity for the model to be overfit to in-sample data. Unlike the in-sample backtest, the first out-of-sample backtest had a Win Rate of 55%, but the average win was slightly less successful than the average loss was detrimental.

When considering the most prominent securities in the portfolio over this backtest, the security with the highest volume was Intuit Inc., followed by Adobe Inc., Exxon Mobile Corporation, PayPal, and Lululemon, indicating that out-of-sample, the algorithm still succeeds in creating and buying from a diverse universe of stocks across several sectors.

3.3 Out-of-Sample Backtest B: 1/1/2016 - 1/1/2017

For the second out-of-sample backtest, we ran a backtest over the period between January 1, 2016 and January 1, 2017, which is linked here and displayed in Figure 5. This was a more successful backtest than the in-sample or first out-of-sample test periods, with a Sharpe Ratio of **0.727** and an overall profit of **7.66%** on a maximum Drawdown of only 5.4%. This backtest also achieved a longest Drawdown duration of only 3 months, from January 20 to April 29 of 2016, which was below our 4 month target. This backtest was most successful despite a Win Rate of only 51%, but the average win was 0.42%, whereas the average loss was only -0.32%, leading to over 7.5% profit over a single year.

Looking more closely at this backtest, we can see fairly consistent and steady gains over the course of the backtesting period, compared to either of the previous backtests. Ideally, this is how the algorithm should

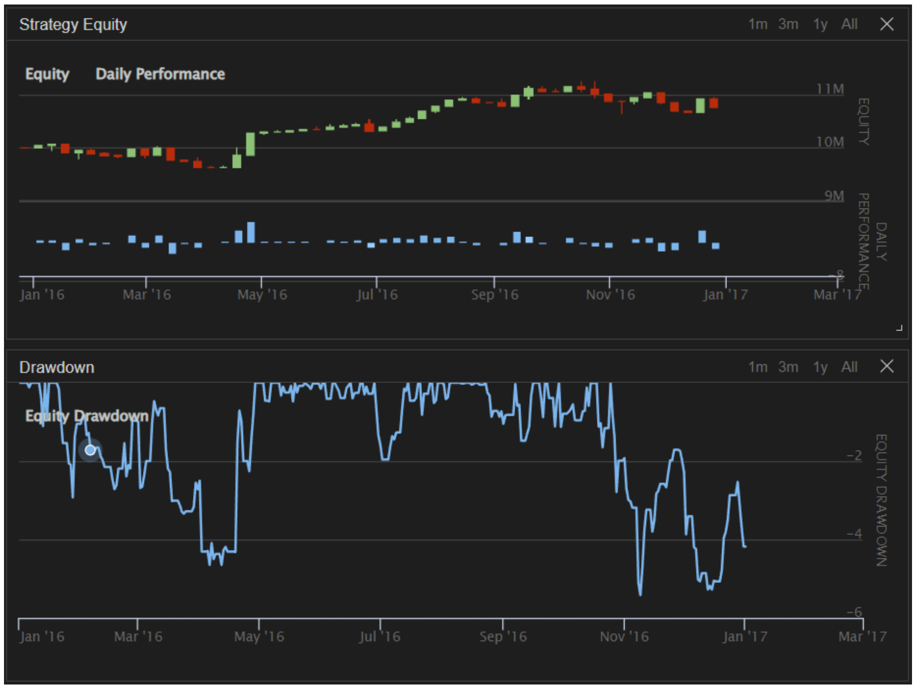

Figure 5: Out-of-Sample Backtest B: 1/1/2016 - 1/1/2017

perform, making incremental gains and quickly reversing any losses, but also rarely taking any significant loss because of an ability to anticipate and account for potential downturn among the securities in the algorithm universe. Looking at the top securities in the portfolios across this backtest, this backtest saw the least diversity among the most traded securities, with a heavy emphasis on securities from the sectors of Technology and Medicine and Pharmaceuticals. By name, the top securities traded by volume in this test were NVIDIA Corporation, Bristol-Myers Squibb Co., and Medtronic PLC, followed by Facebook, Microsoft, and Regeneron Pharmaceuticals Inc., although more diverse companies such as Starbucks, FedEx Corp. and Nike Inc. were traded at lower volume as well.

3.4 Out-of-Sample Backtest C: 12/23/2022 - 3/24/2023

For the third out-of-sample backtest, we ran a backtest from December 23, 2022 through March 24, 2023, linked here and displayed in Figure 6. This backtest was our most successful backtest overall, with a Sharpe Ratio of **2.585** and an overall profit of **5.22%** with a maximum drawdown of only 1.9%. The maximum drawdown period for this backtest was also only between February 16 and March 24, 2023. This backtest succeeded despite a win rate of only 48%, but the average win was 0.62% and the average loss was only -0.23%, meaning the algorithm's good choices were very good and its poor choices were not very poor.

Looking into this backtest, we see steady gains in profit for the first two months of the backtest period, followed by a leveling out and minor drop over the last weeks, during the longest drawdown duration. Further, as seen in the previous backtests, the algorithm successfully built diverse portfolios over the duration, with Spotify, Eli Lilly and Co, PayPal Inc., and International Flavors & Fragrances Inc. as the most prominent securities by volume, but no security consuming more than 15% of the overall volume.

3.5 Stress Test: March 2020

For our first stress test, we ran a backtest for the month of March, 2020, during the onset of the Covid-19 pandemic, which is linked here and shown in Figure 7. Unlike the previous backtests, while we still aspired for the same general targets, the focus was more to confirm that the algorithm was resilient in bear markets, and the target was to have a Drawdown of less than 15% and an overall profit (or loss) of greater than -15%.

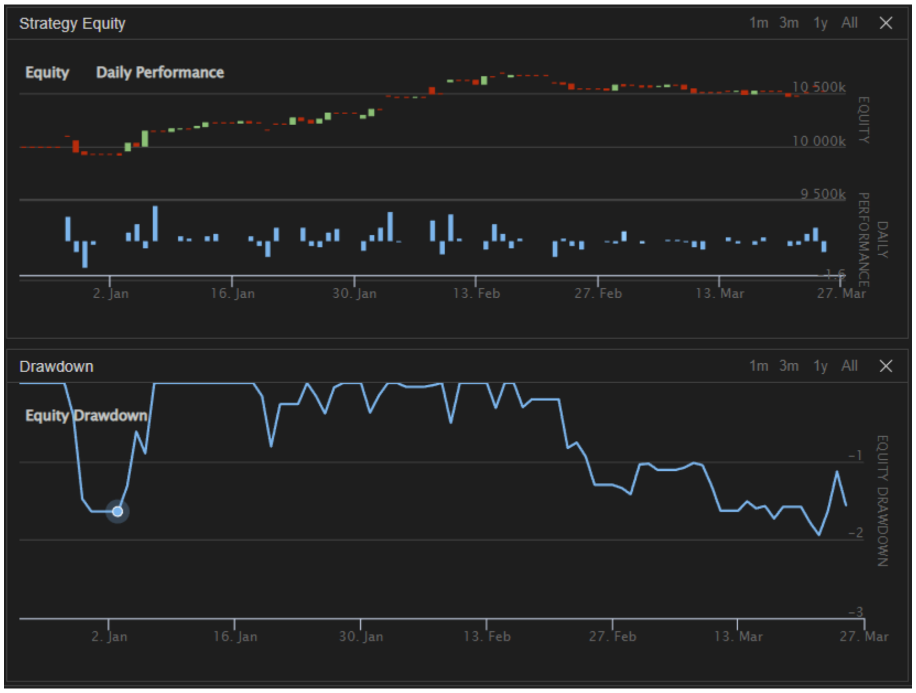

Figure 6: Out-of-Sample Backtest C: 12/23/2022 - 3/24/2023

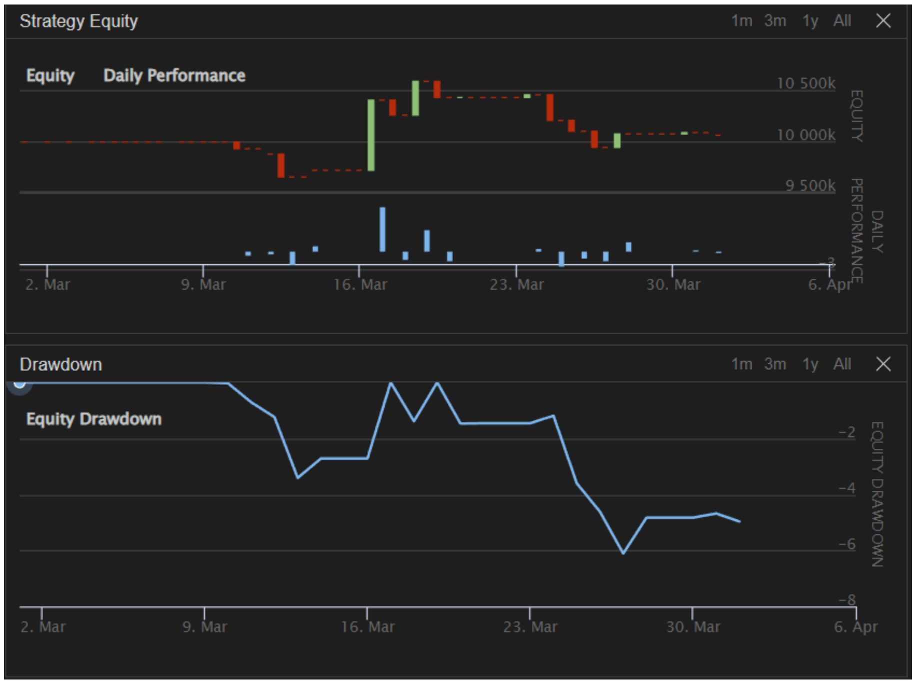

Figure 7: Stress Test: March 2020

Looking at the backtest, not only was the algorithm resilient to the market shock caused by the onset of the pandemic, it actually saw a **0.64%** profit and a Sharpe Ratio of **0.342** over the month of March 2020 with a Win Rate of 62% and a maximum drawdown of only 6.4%, which was well within the stress test target range.

Looking a little deeper into this backtest, the algorithm actually had its highest net profit during the week of March 16th. During that week, the largest holding in the portfolio was a large Short position in CME, which is the Chicago Mercantile Exchange. Although our algorithm is largely a black box method in backtests, it is possible that the sentiment analysis picked up on negative sentiment about the market as a whole during the week prior to March 15 and used that to predict negative returns for the Chicago Mercantile Exchange itself.

This stress test reveals a potential flaw in our algorithm, due to the computational constraints we had in live trading and in backtesting. Since the stock universe is selected only once per month, during this stress test, the only time universe filtering occurs is at the very beginning of the backtest. Because of this, although CME was in our universe during this month, it is entirely possible that other securities with similar properties were not included, so it may be beneficial to increase the number of available securities in the universe, as discussed in Section 6.3.

3.6 Optional Stress Test: Financial Crisis, 12/1/2007 - 6/15/2009

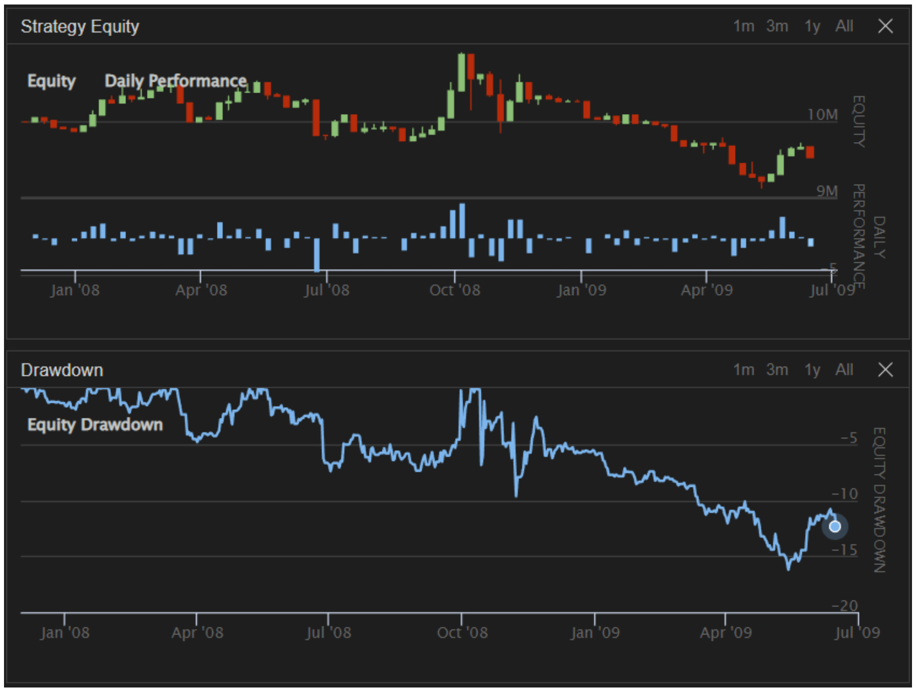

Figure 8: Optional Stress Test: Financial Crisis, 12/1/2007 - 6/15/2009

After running the stress test for March 2020 and seeing a profit, and noticing the algorithm's success during 2020 as a whole, we wanted to further explore the performance of the algorithm during bear markets with a stress test during the 2008 Financial Crisis. For this backtest, shown in Figure 8 and linked here, we considered the period from December 1, 2007 to June 15, 2009, which was the time period labeled as the "Great Recession" by Robert Rich at the Federal Reserve Bank of Cleveland [8].

Overall, our algorithm performed reasonably well over the duration of the backtest, with a Sharpe ratio of **-0.121** and an overall loss of **4.54%**. Although our algorithm sustains losses, those losses pale in comparison to the peak-to-trough losses of market benchmarks such as the S&P500, which fell over 56% between October 2007 and March 2009. Further, the maximum Drawdown of our algorithm during this backtest is only 16.2%, which is below the target drawdown, and although the drawdown duration is 8 months, between October 2008 and the end of the backtest period, this is still less than the Drawdown duration of the S&P500, as a market benchmark, which as noted fell from peak-to-trough over an 18-month period.

Looking further into this backtest, we again see the algorithm's universe selection and portfolio optimization lead to a diverse overall portfolio, with Amazon.com Inc., Blackberry, and the Chicago Mercantile Exchange among the top securities traded by volume, as well as Potash Corp. of Saskatchewan and Newmont

Corporation, two mining companies. Taken together with the stress test from March 2020 and the algorithm's performance during 2020 as a whole, the algorithm appears successful at making correct decisions during market shocks and bear markets, and in doing so limits losses compared to the overall market during these bear periods.

4 Live Paper Trading: 4/19/2023 - 4/26/2023

To further explore the efficacy of our trading algorithm, a live trading is deployed on April 19th, with detailed performance shown in Figure 9. At the beginning of the trading period, our boosting model successfully captures the upward trend of ENPH (Enphase Energy) and Eli Lily and entered long positions in both stocks. Throughout the period of 4/19/2023 to 4/25/2023, a 2-percent gain is secured by this trading decision.

On 4/26/23, abnormal volatility is noticed within our portfolio. After investigation, we found out that ENPH beat earnings estimates but delivered weak future guidance which caused the stock to fall about 24% during the day. Similarly, Eli Lilly was planning to announce earnings on Thursday (April 27th) thus its stock price also was experiencing turbulence. With a weekly model training schedule, our market sentiment data is not updated yet to track these earning changes. Despite the volatile market conditions, our algorithm managed to limit our loss with different risk management procedures including reasonable VaR threshold and optimized weights.

Throughout this short trading period, we have demonstrated our model's accuracy in capturing market trends but also discovered potential improvements. There are two strategies we can potentially employ to avoid the portfolio instability occurred during live trading. First, we would like to backtest a strategy that liquidates before earnings reports or chooses not to buy a stock if our holding period crosses over their earnings date. Though we have fairly short holding periods, our intention is not "playing earnings", so we think this would accurately reflect the purpose of our strategy even if we miss out on the potential upside from a positive earnings report. Furthermore, we could include a stop-loss function that computes sentiment scores regularly and liquidate all position if unfavorable events occur. This stop-loss function would further reduce the risk of our trading strategy.

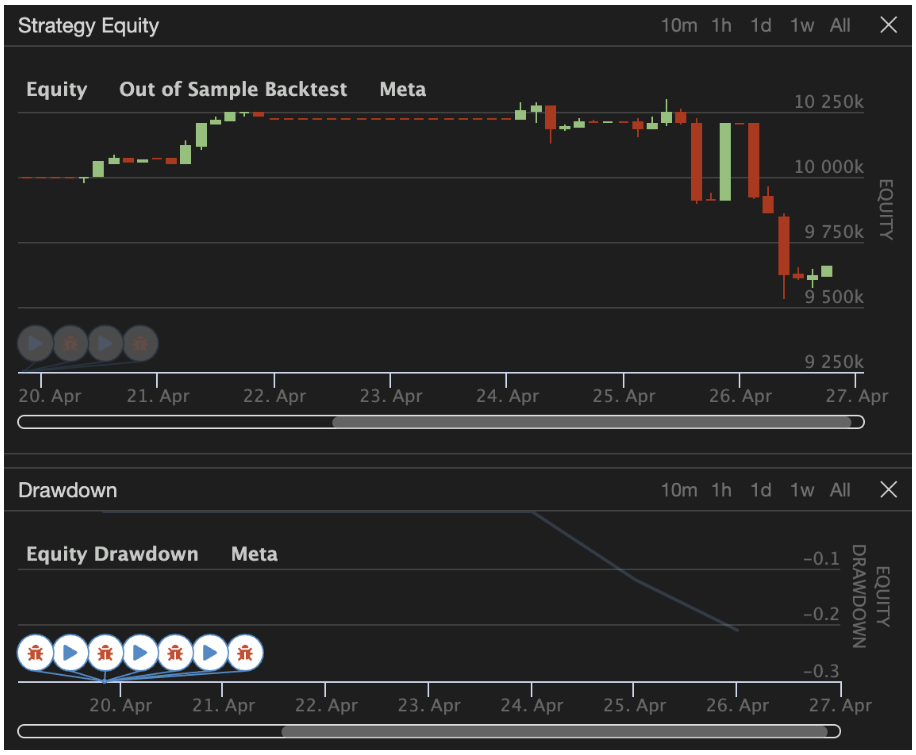

Figure 9: Live Trading, 4/18/2023 - 4/25/2023

5 Overall Algorithm Performance

In summary, our trading algorithm provides the following benefits:

1. Robust performance in bearish markets alongside moderate performance in bullish markets, as demonstrated in in-sample and out-of-sample backtests.

2. Diversification of portfolio, as a result of universal selection and portfolio optimization.

3. A lightweight, easily interpretable model and algorithm driven by publicly available financial data.

Interestingly, our algorithm performed better in some Out-of-Sample tests than it did in-sample. This is perhaps unsurprising, because although we used backtesting to decide on parameters such as the number of symbols in the universe and the lookback period, we were also significantly constrained by computational constraints, as discussed in Section 6.2. While active, however, the algorithm conducted its own universe selection, training, prediction, and portfolio building, so there was little opportunity for the model to be overfit to in-sample data. Though our algorithm did not perform as well as desired during bull markets, which was the case during the last 2 years post-COVID and several years prior, we performed very well during economic downturns. We believe our strategy would be a profitable complement to strategies that perform well during bull markets.

6 Limitations and Future Expansions

6.1 Limitation 1: The QuantConnect Platform

We faced difficulties learning and using the QuantConnect platform, resulting from the relatively short time frame of a month over which we built the algorithm along with the limitations of the platform itself. First, we note that all of us were completely new to algorithmic trading (though familiar with Python), so learning any new platform would have been difficult. During the construction of our algorithm, we needed to learn and understand every fine detail of universe selection, event handlers, order placing and filling, how training machine learning models functioned the best on the platform, live trading, and dozens of other concepts. This took a significant amount of time, and an individual more familiar with algorithmic trading probably would have had a much easier time learning the platform.

We found the documentation was often poorly written and not descriptive enough to use effectively. Searching in the documentation on `https://www.quantconnect.com/docs` and clicking the resulting link results in a message at the top of the page which states ”You were redirected from our old version of the docs, click here to read our archived version”. For example, searching for ”indicators” redirects to the ”Historical Data” section of ”Writing Algorithms” when it should redirect to the ”Indicators” section. This led to us frequently either googling something like ”Indicators QuantConnnect” to find the docs, or reading through community posts on the platform which were often outdated, though sometimes contained useful information. The QuantConnect community also has an active Discord, a message board platform that is free to join, though expecting fellow community members to answer documentation or strategy questions frequently results in questions going unanswered. In fact, we found several documentation bugs that we reported on the Discord platform that were then quickly addressed.

Our most significant issue occurred when we used a `QuantBook()` in `main.py` to pull in historical data for indicators like Bollinger Bands and RSI. While the scripting platform has access to current data indicators, it does not have the ability to pull historical indicators, which we needed to train our model. This is very straightforward in the research environment, which is simply a Jupyter Notebook that allows the programmer to access data on the platform for research purposes. Since historical indicators are not available in the script, and to avoid building historical indicators manually, we assumed we could use a `QuantBook()` to pull them. This actually worked and did not present us with any warnings, but during long backtesting periods (> 6 months), we noticed we were exceeding the memory of the backtest nodes (around 6 GB), which caused the backtest to crash. We debugged for hours and tried many potential solutions to no avail. Eventually, we opened a support ticket on the platform, where QuantConnect staff pointed out that we should not use a `QuantBook()` in the main file as it uses too much memory. Fortunately, they replied very quickly, and sure enough, making the change to manually construct indicators and using `self.History()` fixed our issue. We believe issuing a warning when accessing a `QuantBook()` outside of the research environment would be a useful addition.

Additionally, we had difficulties connecting our live trading to the Interactive Brokers platform and instead opted to live trade using the QuantConnect platform instead. Since we are accurately simulating the brokerage fees and slippage, though, we believe this is adequate for live trading. The CEO of QuantConnect, in fact, emailed us telling us there was an issue with our connection to Interactive Brokers unrelated to anything we were doing.

However, despite these difficulties, the platform, CEO, and community were quick to respond to issues and were very helpful. It seems unlikely that many trading platforms have a CEO that would personally email someone at 10 P.M. to ask them to deploy their algorithm again because of an internal bug. We also appreciate that the LEAN Engine, the code that integrates the algorithms into the markets, is open source and can be fully viewed or downloaded from GitHub.

6.2 Limitation 2: Resource Constraints

We discovered that even with code that ran well in backtesting, the same code could run out of memory during live trading. However, this was not a bug, and our machine-learning models need a fairly significant amount of computational power to run on large datasets. Since we train the model on a lookback period and for each symbol, we intend to trade, during tuning in the in-sample period, we found that a larger number of fine symbols and a longer lookback worked the best but this used up too much memory during live trading. We only have access to live trading nodes with 0.5 GB of RAM (note that live trading uses significantly less resources than backtesting), and the most expensive nodes have 4 GB of RAM. To get around this, we only live traded with 5 fine symbols and a 25-day lookback period, though would increase this to 10 fine symbols and a longer lookback if we had more compute power.

Moreover, Quant Connect has a runtime cap on model training which restricted the allowed model complexity and the size of data. Below, we provide a comparison of the same algorithm parameters with different lookback periods (500 and 50).

Figure 10 is the performance of our algorithm between 1/1/2022 and 6/1/2022 trained with a lookback period of 500 days. Note that a five-month period is the longest backtest we can run without reaching QuantConnect's runtime cap with a 500 lookback period. A profit of 9.6% is achieved with a Sharpe ratio of 1.418 and a 4.8% drawdown.

Figure 11 is the performance of the same model trained with a significantly shorter lookback (50 days). The return falls to -1.52 percent with a Sharpe ratio of -0.236 and a drawdown of 9.2%.

From the above comparison, it can be seen how model accuracy might be affected by the size of data. With a longer lookback period, the predictive model is more robust to different market conditions and produces more reliable results. From these results, it can be concluded that our algorithm performance would be boosted if we had more computational power.

6.3 Future Expansions

As noted, some of our progress with building and testing our algorithm was hampered by struggles with the QuantConnect platform and with resource constraints we contended with. Even still, there are several instances where we could continue to alter our algorithm going forward in seeking improved performance. In order to address the constraints mentioned in Section 6.2, we made the decision in our algorithm to only select our universe of stocks once per month. With selecting a universe once per month, we were unable to capitalize on short-term booms for individual securities, such as due to a successful earnings report, unless those securities were already in our universe. Therefore, we would want to consider selecting our universe more often, at a minimum bimonthly, if not weekly or even daily, in order to ensure our universe consistently met the guidelines we laid out in Section 2.1.

Likewise, due to the same resource constraints, we made the decision to only train our algorithms, make predictions, and trade stocks once per week, or on the first of the month with a new universe selected, and therefore to have a holding period of 1 week, except on weeks where a new month started between Tuesday and Friday. In a real scenario, where computing constraints are less of a concern, we would want to consider

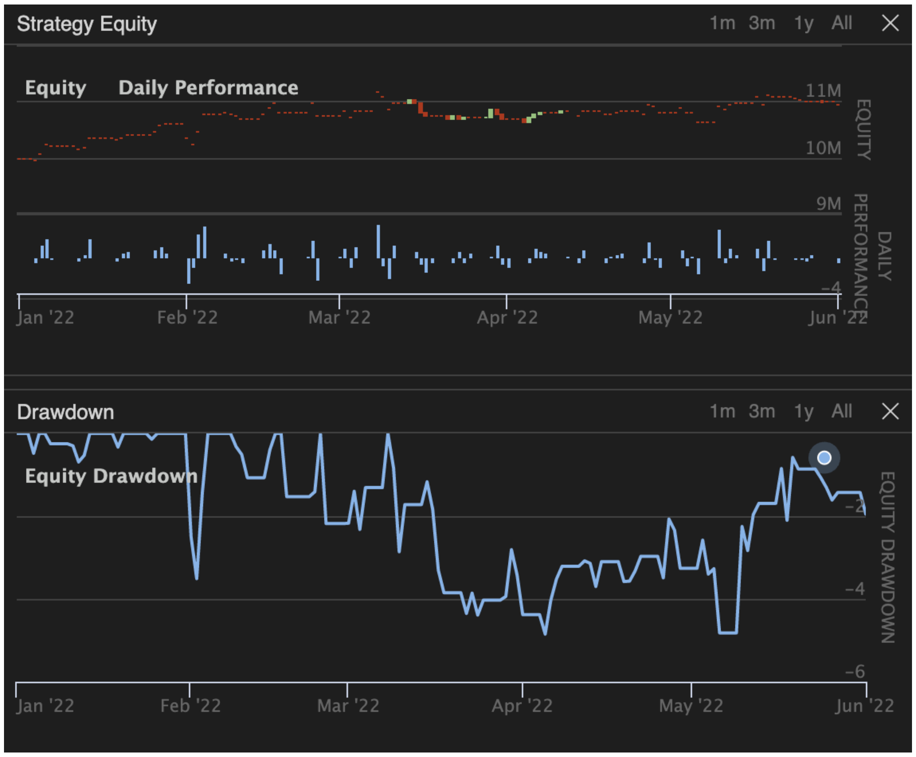

Figure 10: Out of Sample Back Test with 500 Lookback, 1/1/2022 - 6/1/2022

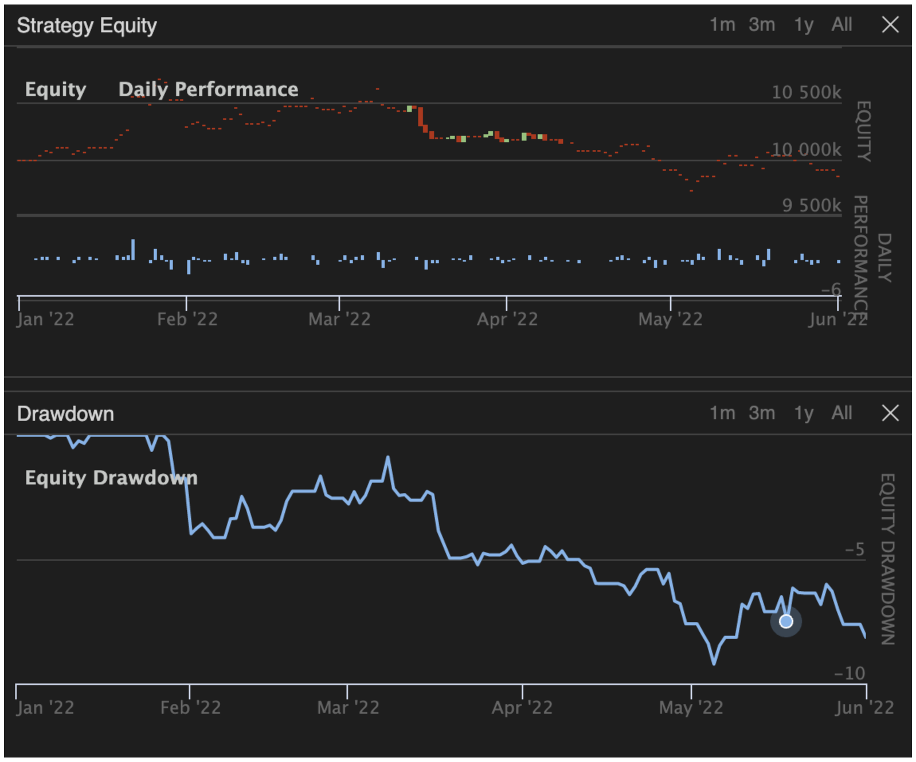

Figure 11: Out of Sample Back Test with 50 Lookback, 1/1/2022 - 6/1/2022

training our algorithm, making predictions, and building a new portfolio every day. This could open up other concerns, such as an overabundance of trading fees and losses via slippage which could impact the performance of the model, as well as increased volatility, but it is also possible that daily trading would allow our algorithm to capture profits more consistently, since we would be able to dynamically adjust our portfolio to have the best possible expected returns for a shorter time period.

As noted in the backtesting section, the issue with a week-long holding period is that, as currently constructed, there is no mechanism by which our algorithm can capture intermediate profits during the week. In an effort not to artificially cap profits or lock in losses, we elected not to include any constraint on our model to liquidate in the event of extreme mid-week profit or loss events. As evident in the In-Sample backtest in Section 3.1, during September 2018, there was a week during which the portfolio both saw 18% profit returns and lost those returns, which gives the backtest a drawdown over 30%. In a real-world implementation of the algorithm, a human trader likely would be observing the market and checking the returns of the model, and would be able to liquidate holdings during the spike during that week, locking in profits manually. For the sake of our algorithm being entirely self-contained, we did not include any mechanism to do so, but future expansions of this algorithm could consider profit or loss capping conditions.

Although VaR is a widely used measure of risk, it has shortcomings, such as providing no insight over scenarios in which loss significantly exceeds the VaR limit. In other words, VaR is indifferent to extreme tails. Therefore, we considered another measure of risk, conditional value-at-risk (CVaR), also known as expected shortfall. In contrast to VaR, CVaR accounts for extreme risks by quantifying the average loss over a specified time period of scenarios beyond the VaR cutoff point. If X is a random variable with a continuous distribution function, $\mathrm{CVaR}_\alpha(X)$ is the conditional expectation of X subject to $X \geq \mathrm{VaR}_\alpha(X)$. CVaR is calculated by taking the probability-weighted average of the tail [9]. Although we considered using CVaR as a more conservative risk-management measure, we found that our VaR limit was conservative enough. In future work, we would like to experiment with comparing strategies using CVaR compared to VaR.

Finally, due to the resource constraints, as noted in Section 6.2, we conducted our live-trading with only 5 securities in our universe and only a 25-day lookback period, which was half of what we conducted our backtesting with. The 10 fine symbols and 50-day lookback, which we found to be most successful during backtesting, was itself towards the upper edge of what the resources we were working with allowed. In a situation where computing resources were less of a concern, we would seek to experiment with considering a larger number of securities in our universe, upwards of 20, and a lookback with higher resolution, rather than the daily resolution we were restricted to. In this experiment, it could be possible that the constraints in the linear program, such as avoiding leverage and limiting the total stock size to a function of n, the number of universe securities, could be altered to allow for our algorithm to choose a small fraction of a larger universe while still maintaining a diverse portfolio.

References

[1] David H. Bailey et al. "Pseudo-mathematics and financial charlatanism: The effects of Backtest overfitting on out-of-sample performance". In: *Notices of the American Mathematical Society* 61.5 (2014), p. 458. DOI: 10.1090/noti1105.

[2] Suryoday Basak et al. "Predicting the direction of stock market prices using tree-based classifiers". In: *The North American Journal of Economics and Finance* 47 (2019), pp. 552–567. ISSN: 1062-9408. DOI: https://doi.org/10.1016/j.najef.2018.06.013. URL: https://www.sciencedirect.com/science/article/pii/S106294081730400X.

[3] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[4] Benjamin Graham. *The Intelligent Investor*. Harper, 2006.

[5] Q. Huangfu and J. A. Hall. "Parallelizing the dual revised Simplex method". In: *Mathematical Programming Computation* 10.1 (2017), pp. 119–142. DOI: 10.1007/s12532-017-0130-5.

[6] G. James et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2021. ISBN: 9781071614181. DOI: https://doi.org/10.1007/978-1-0716-1418-1_1.

[7] Joseph D. Piotroski. "Value Investing: The Use of Historical Financial Statement Information to Separate Winners from Losers". In: *Journal of Accounting Research* 38 (2000), pp. 1–41. ISSN: 00218456, 1475679X. URL: http://www.jstor.org/stable/2672906 (visited on 04/04/2023).

[8] Robert Rich. *The Great Recession*. Nov. 2013. URL: https://www.federalreservehistory.org/essays/great-recession-of-200709.

[9] R. Tyrrell Rockafellar and Stanislav Uryasev. "Optimization of conditional value-at risk". In: *Journal of Risk* 3 (2000), pp. 21–41.

[10] Tharsis T.P. Souza et al. "Twitter Sentiment Analysis Applied to Finance: A Case Study in the Retail Industry". In: *arXiv* ().

[11] Yen Sun. "Optimization Stock Portfolio With Mean-Variance and Linear Programming: Case In Indonesia Stock Market". In: *Binus Business Review* 1 (May 2010), p. 15. DOI: 10.21512/bbr.v1i1.1018.

[12] *Supported models - documentation quantconnect.com*. URL: https://www.quantconnect.com/docs/v2/writing-algorithms/reality-modeling/transaction-fees/supported-models#03-Interactive-Brokers-Model.