Edward Churchill
ed.churchill00@gmail.com

Predicting the Sign of the 1-week Forward Return of S&P 500 Constituents

CONTENTS

Code and Datasets

The data required for this project was found on Kaggle. The links to the datasets are below. Note that some of these datasets are appended to daily and so the dates may slightly differ to those used in the project.

- S&P 500 Data
- Macroeconomic Data

The notebooks for the project were written on Kaggle and are stored in this GitHub folder. The 7 notebooks are as follows:

- EDA
- Feature Engineering
- Feature Selection
- Logistic Regression Model
- XGBoost Model
- LSTM Model
- Results

Objective

The objective of this analysis is to predict the sign (positive or negative) of the 1-week forward return of constituents of the S&P 500. The return of a constituent stock is simply its change in price over the period divided by its price at the start of the period. The prediction should be '1' for a positive return and '0' for a negative return (we choose '0' instead of '-1' for easier compatibility with certain binary classification algorithms). In the unlikely scenario that a stock has a return of exactly zero return over the period, we classify it as '1' to keep the problem binary. To calculate returns throughout, we use the adjusted closing price of each stock, which is the closing price adjusted for dividends and stock splits.

Data Description and Cleaning

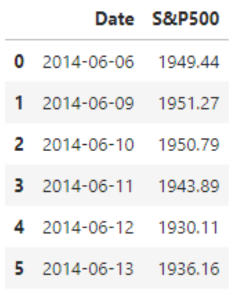
In order to come up with a dataset that can assist us in predicting these returns, we first take 4 tables found on Kaggle:

- S&P 500 Value
- S&P 500 Constituents Pricing/Volumes
- S&P 500 Constituents Static Data
- US Macroeconomic Indicators

These tables can be cleaned and joined together to create a wide table suitable for a supervised machine learning problem. We give a brief overview of each table below and outline how they are joined.

S&P 500 Value

The first table is simply a time series of the value of the S&P 500 index on each business day from 6th June 2014 to 6th June 2024, so there is roughly 10 years of data.



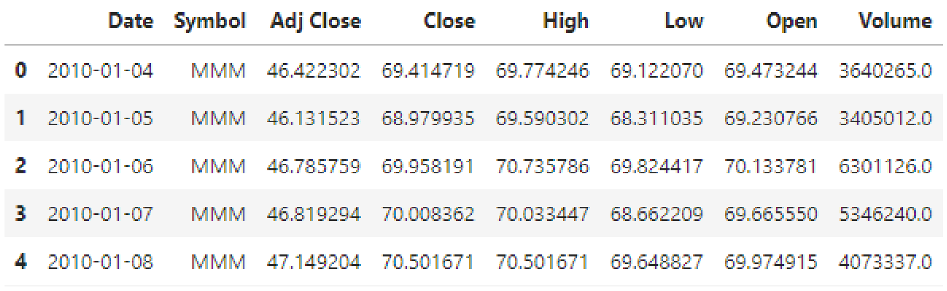
	Date	S&P500
0	2014-06-06	1949.44
1	2014-06-09	1951.27
2	2014-06-10	1950.79
3	2014-06-11	1943.89
4	2014-06-12	1930.11
5	2014-06-13	1936.16

Figure 1: First few rows of 'sp500_index.csv' containing the value of the S&P 500 index over time. The value is derived from the value of the underlying constituents, weighted by market capitalization.

S&P 500 Constituents Pricing/Volumes

The second table contains pricing and volume information for each constituent stock over time, with 1 row per date per stock. The data starts on 4th January 2010 and goes to 6th June 2024 so there is roughly 14 years of data.

- It is worth noting that this table only contains the *current* constituents over time, not the historical constituents. Some smaller companies may currently be in the index but may not have been previously. Similarly, some companies that were once in the index may have since dropped out. This is an unfortunate consequence of only having access to freely available data.
- The above remark means that the dataset introduces *survivorship bias*, since we are only considering stocks that have managed to 'survive' (remain in the index) until the latest date. This bias can cause inflated results when back-testing how a portfolio of stocks may perform over time, since we only consider stocks that remained in the index (so by definition have performed well) and ignoring those that have dropped out of the index.

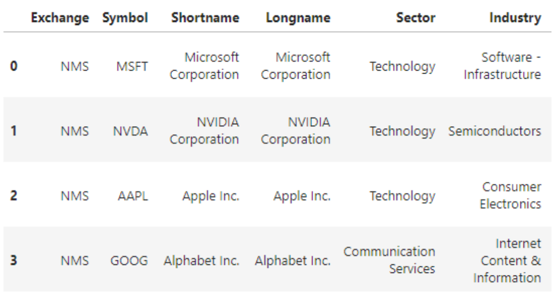


	Date	Symbol	Adj Close	Close	High	Low	Open	Volume
0	2010-01-04	MMM	46.422302	69.414719	69.774246	69.122070	69.473244	3640265.0
1	2010-01-05	MMM	46.131523	68.979935	69.590302	68.311035	69.230766	3405012.0
2	2010-01-06	MMM	46.785759	69.958191	70.735786	69.824417	70.133781	6301126.0
3	2010-01-07	MMM	46.819294	70.008362	70.033447	68.662209	69.665550	5346240.0
4	2010-01-08	MMM	47.149204	70.501671	70.501671	69.648827	69.974915	4073337.0

Figure 2: First few rows of 'sp500_stocks.csv' which contains pricing and volume information of each constituent stock over time.

S&P 500 Constituents Static Data

The third table contains static data for each constituent stock, such as its sector, its geography and which exchange it's traded on. This data doesn't change over time.



	Exchange	Symbol	Shortname	Longname	Sector	Industry
0	NMS	MSFT	Microsoft Corporation	Microsoft Corporation	Technology	Software - Infrastructure
1	NMS	NVDA	NVIDIA Corporation	NVIDIA Corporation	Technology	Semiconductors
2	NMS	AAPL	Apple Inc.	Apple Inc.	Technology	Consumer Electronics
3	NMS	GOOG	Alphabet Inc.	Alphabet Inc.	Communication Services	Internet Content & Information

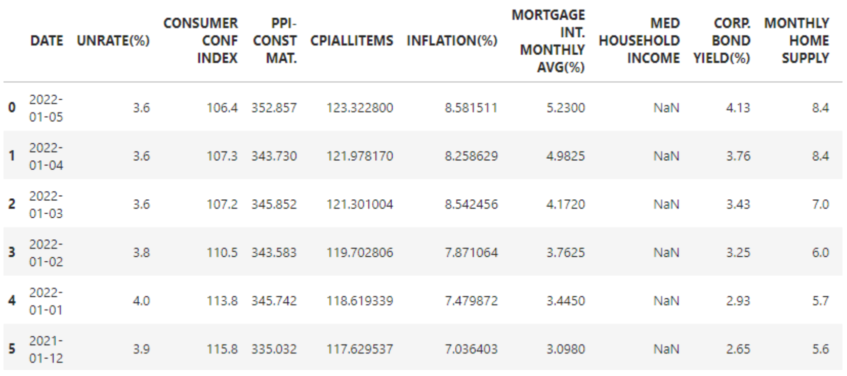
Figure 3: First few rows and columns of 'sp500_companies.csv' which contains static data for the current S&P 500 constituents.

US Macroeconomic Data

To predict the return of a stock, we'd ideally have additional information regarding the underlying company's finances (earnings, debt etc.). However, it is hard to obtain this information historically for free.

A less powerful but useful alternative is to bring in macroeconomic indicators. For example, inflation levels, house prices and corporate bond yields are all factors that may affect how various stocks perform. Some stocks may be more sensitive than others to changes in these macroeconomic indicators. For example, a company focused on real estate would be more heavily affected than others by changes in house prices and interest rates.

The fourth table therefore contains US macroeconomic data from 2002 to 2022.

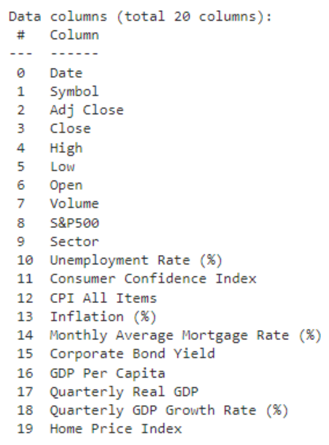


	DATE	UNRATE(%)	CONSUMER CONF INDEX	PPI-CONST MAT.	CPIALLITEMS	INFLATION(%)	MORTGAGE INT. MONTHLY AVG(%)	MED HOUSEHOLD INCOME	CORP. BOND YIELD(%)	MONTHLY HOME SUPPLY
0	2022-01-05	3.6	106.4	352.857	123.322800	8.581511	5.2300	NaN	4.13	8.4
1	2022-01-04	3.6	107.3	343.730	121.978170	8.258629	4.9825	NaN	3.76	8.4
2	2022-01-03	3.6	107.2	345.852	121.301004	8.542456	4.1720	NaN	3.43	7.0
3	2022-01-02	3.8	110.5	343.583	119.702806	7.871064	3.7625	NaN	3.25	6.0
4	2022-01-01	4.0	113.8	345.742	118.619339	7.479872	3.4450	NaN	2.93	5.7
5	2021-01-12	3.9	115.8	335.032	117.629537	7.036403	3.0980	NaN	2.65	5.6

Figure 4: First few rows and columns of 'DATA.csv' which contains US macroeconomic data such as unemployment rates and house prices over time.

Joined Dataset

To utilize all 4 tables, they can be joined on the 'Date' and 'Symbol' columns and some irrelevant columns can be dropped. Note that the various tables cover different time periods, but we have data for all 4 tables between 2014 and 2022. We therefore trim all tables to only contain data within these dates before joining. We also drop those stocks which have a lot of missing data. Finally, we ensure that the data is sorted by the 'Date' column. This leaves a final table representing 469 unique stocks with 20 columns.



```
Data columns (total 20 columns):
 #   Column
---  ------
 0   Date
 1   Symbol
 2   Adj Close
 3   Close
 4   High
 5   Low
 6   Open
 7   Volume
 8   S&P500
 9   Sector
 10  Unemployment Rate (%)
 11  Consumer Confidence Index
 12  CPI All Items
 13  Inflation (%)
 14  Monthly Average Mortgage Rate (%)
 15  Corporate Bond Yield
 16  GDP Per Capita
 17  Quarterly Real GDP
 18  Quarterly GDP Growth Rate (%)
 19  Home Price Index
```

Figure 5: The columns of the joined dataset. More columns will be calculated during the feature engineering process. The feature selection process will then select the ones with the most predictive power.

The data is now in a suitable format for a supervised learning problem. We will use these columns to generate many more calculated columns during our feature engineering process and then use feature selection techniques to select the columns with the most predictive power.

EDA

Below we show some exploratory plots that help us to better understand the data.

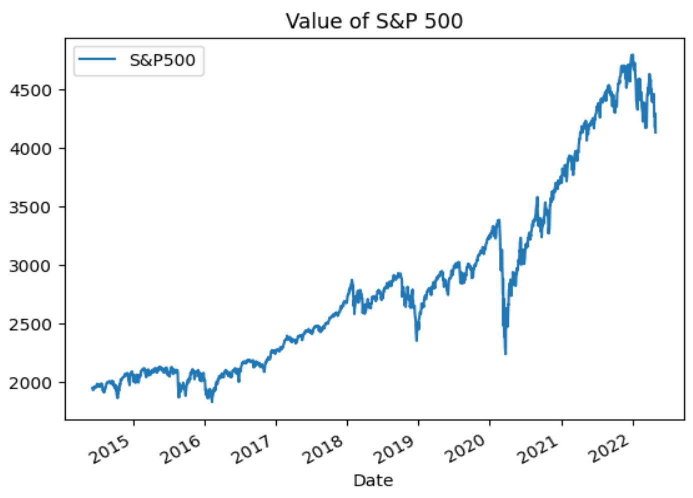


Figure 6: Value of the S&P 500 index over time. It has generally grown over time but notice the sharp drop in early 2020 corresponding the COVID-19 outbreak.

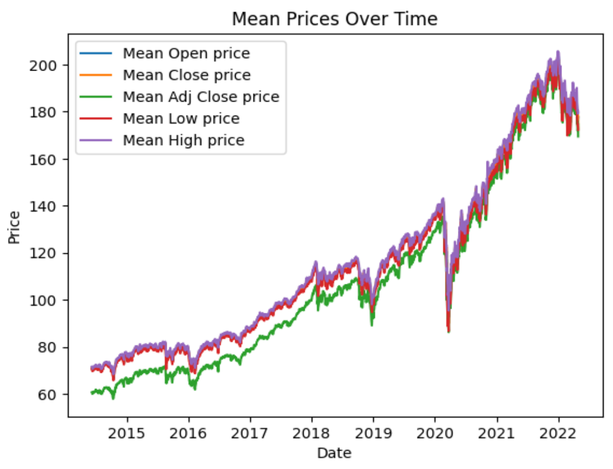


Figure 7: Mean prices (across constituent stocks) over time. Notice how the Adjusted Close price is consistently slightly lower than the other prices since it's been adjusted to reflect the stock price after any dividends have been paid to shareholders.

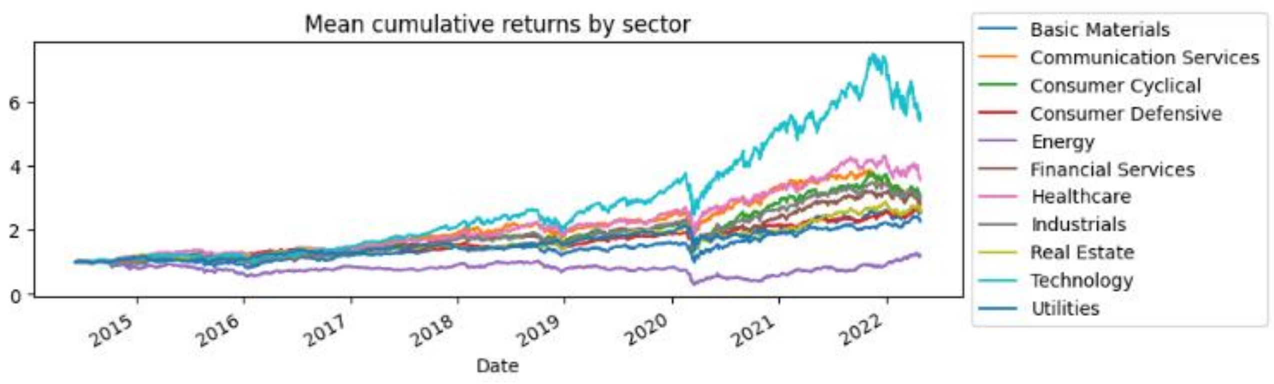


Figure 8: Mean cumulative returns by sector. Other than 'Energy' which remains fairly flat, all sectors trend upwards, with 'Technology' significantly outperforming other sectors.

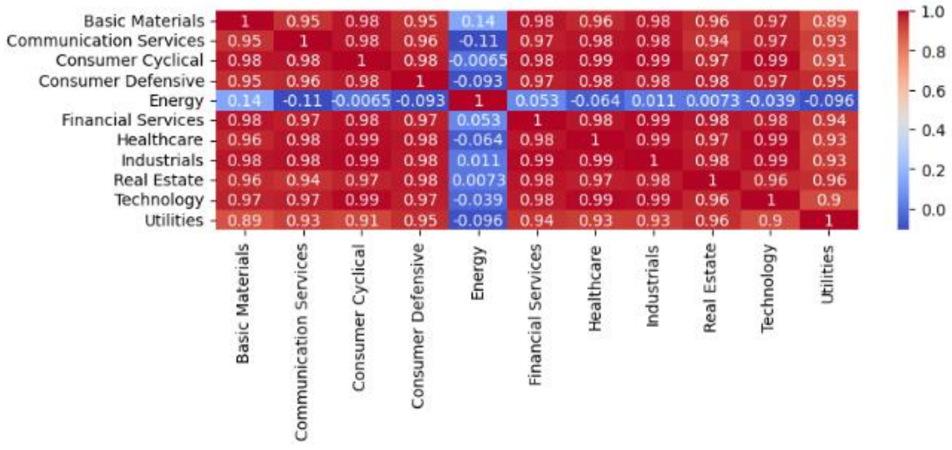


Figure 9: Correlation matrix of mean cumulative returns by sector. Note that most correlation coefficients are close to 1 (strong positive correlation) due to most sectors trending upwards with the exception of 'Energy'.

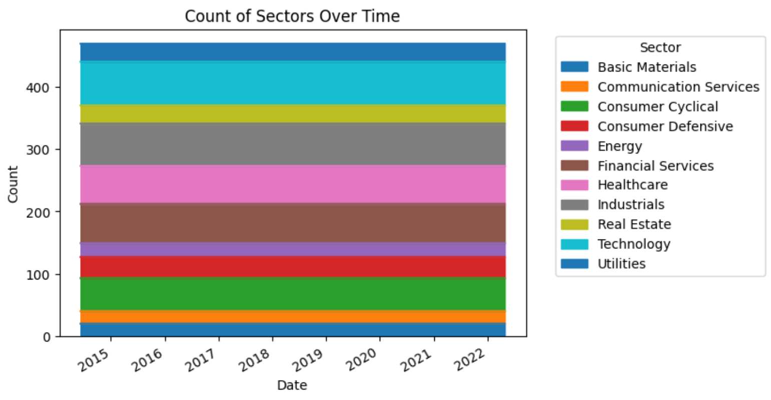


Figure 10: Sector distribution of the constituent stocks over time. The counts remain constant because we are only looking at current constituents historically, and their sectors are fixed.

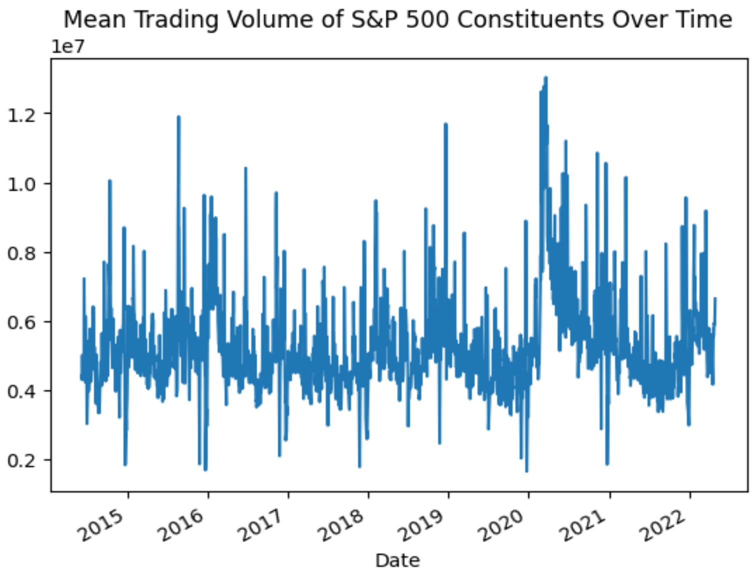


Figure 11: Mean trading volume (across constituent stocks) over time. Volume massively fluctuates over time but notice the large volumes throughout 2020 due to the COVID-19 outbreak.

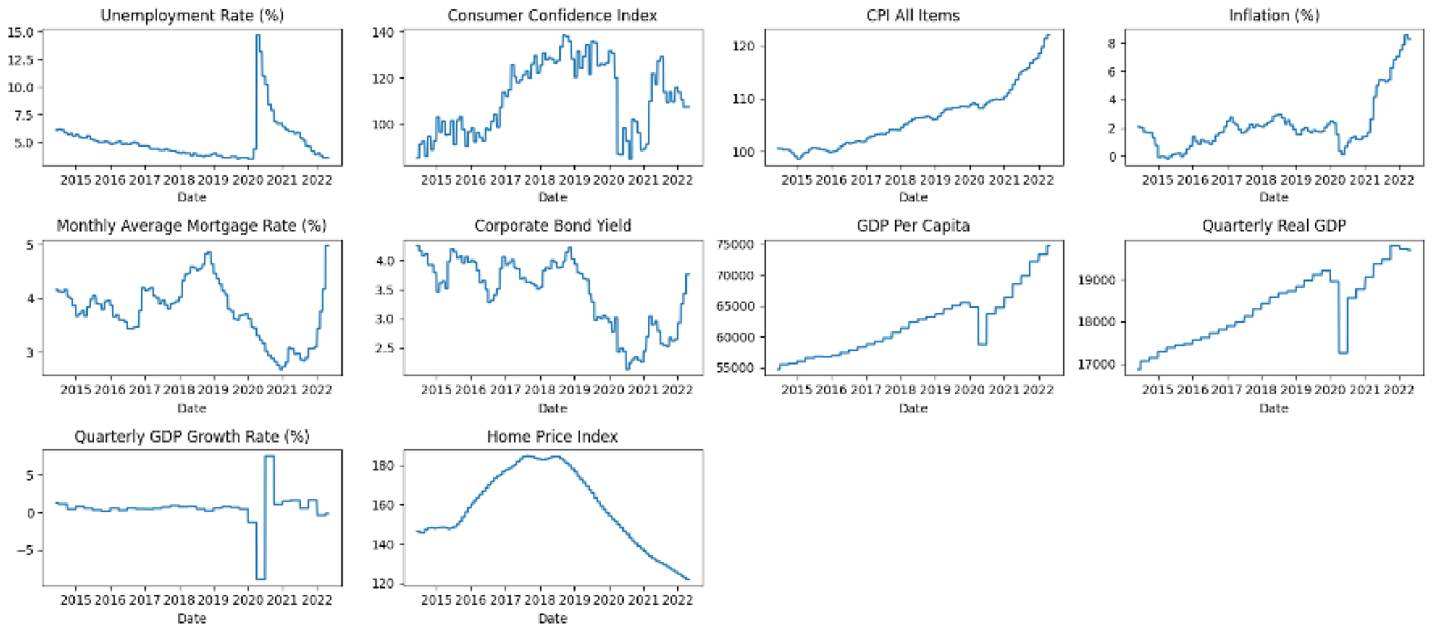
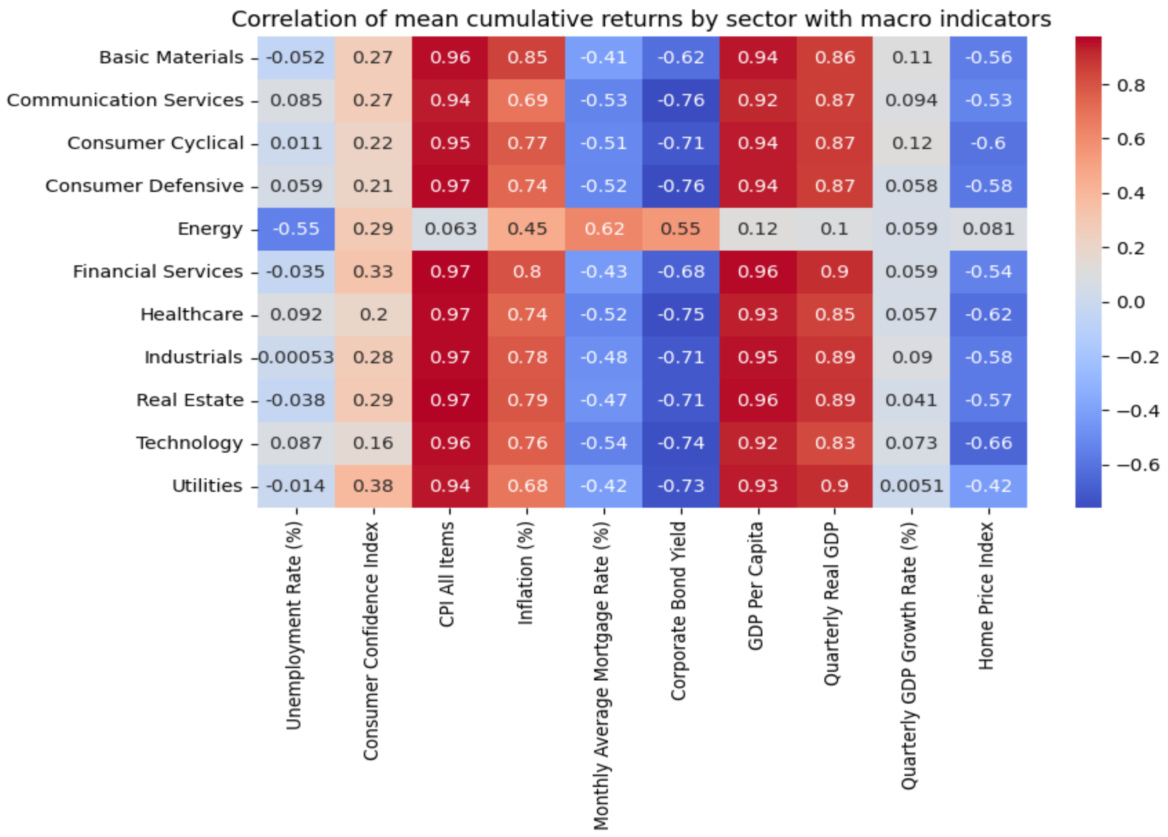


Figure 12: Various macroeconomic indicators over time. Notice once again how COVID-19 in 2020 has a significant impact on all of the indicators.



	Unemployment Rate (%)	Consumer Confidence Index	CPI All Items	Inflation (%)	Monthly Average Mortgage Rate (%)	Corporate Bond Yield	GDP Per Capita	Quarterly Real GDP	Quarterly GDP Growth Rate (%)	Home Price Index
Basic Materials	-0.052	0.27	0.96	0.85	-0.41	-0.62	0.94	0.86	0.11	-0.56
Communication Services	0.085	0.27	0.94	0.69	-0.53	-0.76	0.92	0.87	0.094	-0.53
Consumer Cyclical	0.011	0.22	0.95	0.77	-0.51	-0.71	0.94	0.87	0.12	-0.6
Consumer Defensive	0.059	0.21	0.97	0.74	-0.52	-0.76	0.94	0.87	0.058	-0.58
Energy	-0.55	0.29	0.063	0.45	0.62	0.55	0.12	0.1	0.059	0.081
Financial Services	-0.035	0.33	0.97	0.8	-0.43	-0.68	0.96	0.9	0.059	-0.54
Healthcare	0.092	0.2	0.97	0.74	-0.52	-0.75	0.93	0.85	0.057	-0.62
Industrials	-0.00053	0.28	0.97	0.78	-0.48	-0.71	0.95	0.89	0.09	-0.58
Real Estate	-0.038	0.29	0.97	0.79	-0.47	-0.71	0.96	0.89	0.041	-0.57
Technology	0.087	0.16	0.96	0.76	-0.54	-0.74	0.92	0.83	0.073	-0.66
Utilities	-0.014	0.38	0.94	0.68	-0.42	-0.73	0.93	0.9	0.0051	-0.42

Figure 13: Correlation matrix of sector-level mean returns vs some of the macro indicators. Returns are negatively correlated with things like corporate bond yields and house prices but positively correlated with GDP and inflation.

Feature Engineering

The next step is to generate some features that can help to predict returns. This is a difficult task so a common approach is to generate lots of features and then use feature selection techniques to choose a subset of features with the most predictive power. This section will give a brief overview of which features were generated. For more detail and the exact implementation, see the feature engineering notebook.

Categorical Features

Categorical features (e.g. sectors) were one-hot encoded to allow them to be used in the 3 models. The 'Symbol' column (representing the name of the stock) was also transformed to use an integer stock ID instead of a string.

Date-Related Features

The year, month and day were extracted from the 'Date' column to generate 3 new features.

Pricing Pressure Features

A few features were generated using a combination of the various prices for a given stock in a given day. These include:

- *Daily Variation*: (High - Low) / Open
- *Downward Pressure*: (High - Close) / Open
- *Upward Pressure*: (Low - Open) / Open

Lagged Features

Lagged features refer to those features which use a previous value of a particular column as a new column (e.g. a column which is the price of the stock 1 day ago). In our case, we generate several lagged features using 1, 3, 5, 10, 15, 30 and 60-day lookback periods on various columns such as the pricing/volume data as well as the macroeconomic data.

Returns/Changes Features

We also generate some features based on how much a column has changed (in percentage terms) over some lookback period. When the column is a price, this is simply the backward-looking return of the stock. We use the same lookback periods as the lagged features to perform percentage change calculations on the 'Adjusted Close', the 'S&P 500' and the macroeconomic columns.

Rolling Features

Rolling features are features that perform an aggregation on the last few time periods' worth of data. For example, you may look at the price of a stock over each of the last 10 days and calculate a mean. Then tomorrow, the 'window' rolls forward and we once again calculate the mean price over the last 10 days.

In our case, we define lookback periods of 7, 12, 14 and 26 days. Call this lookback period n. Then on the 'Adjusted Close' and 'Volume' columns, we perform the following aggregations for each stock:

- *Simple Moving Average (SMA):* Mean of the column over the last n days.
- *Exponential Moving Average (EMA):* Exponentially weighted mean of the column over the last n days. The exponential weighting places more weight on more recent data (in theory more recent data should be more relevant than older data).
- *Standard Deviation:* Standard deviation of the column over the last n days. This is a measure of the recent volatility of a stock's price/volume.

Technical Indicators

We also implement various well-known technical indicators and features derived from them. Exact formulas and implementation can be found in the notebook but we list some indicators below.

- *Average True Range (ATR)*: Measure of the recent volatility of the stock.
- *Bollinger Bands*: Lines plotted 2 standard deviations above and below a simple moving average of the adjusted close price.

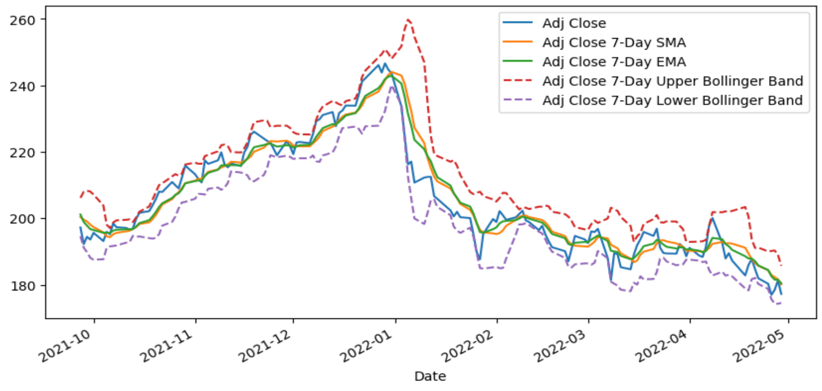


Figure 14: 7-Day Bollinger Bands for a random stock, along with 7-Day SMA and EMA. Notice how the Adjusted Close price generally remains between the bands. When the price hits the upper band, it tends to decrease and vice versa for the lower band.

- *Moving Average Convergence Divergence (MACD)*: Difference between a short-term and a long-term EMA of the stock's price. Along with a signal line, it can be used to predict the future direction of the stock price and therefore should be useful in predicting returns.

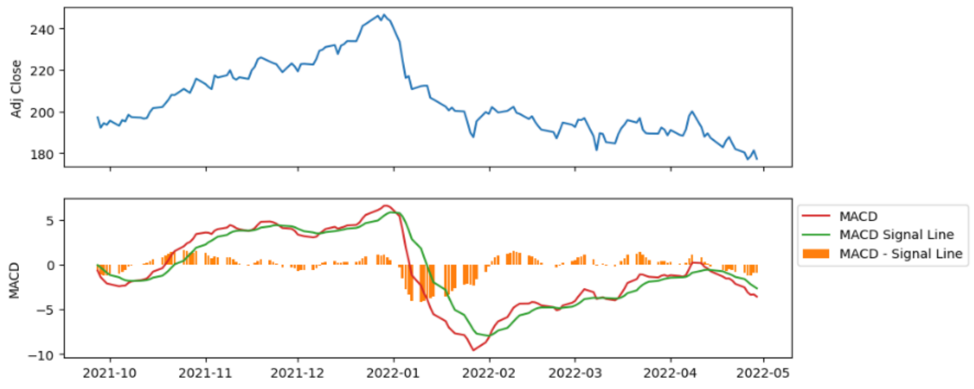


Figure 15: MACD indicator, its signal line and its histogram. The histogram is simply the difference between the MACD indicator and the signal line. When the MACD crosses the signal line from below, it indicates a possible price increase and vice versa from above

- *Stochastic Oscillator*: Indicator comparing a stock's closing price to a range of recent prices.

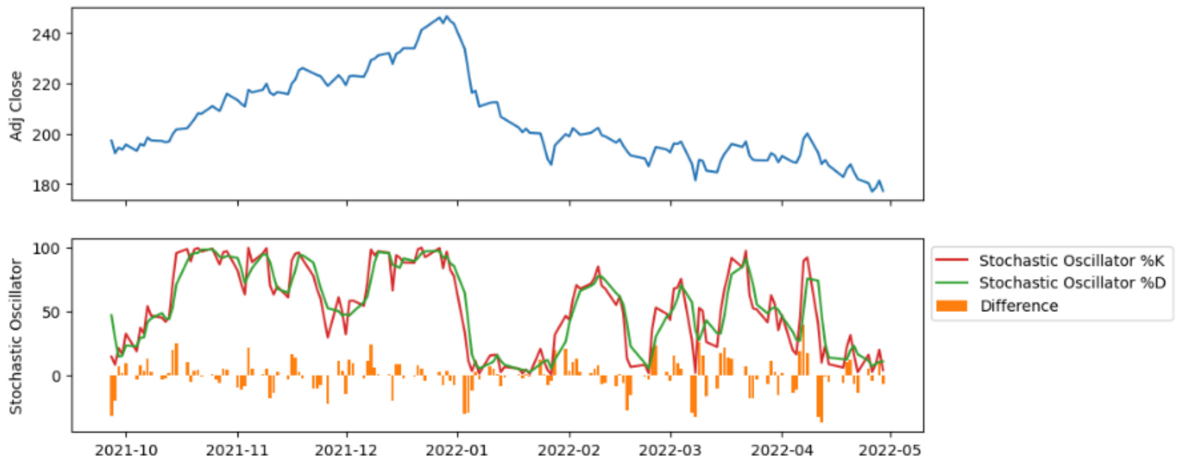


Figure 16: The two types of stochastic oscillators (%K and %D). Signals are generated when the lines cross.

Stationarity Analysis

The next step is to check the *stationarity* of the features that have been generated. A *stationary* time series is one whose statistical properties (mean and variance) do not significantly change over time (so there is no trend or seasonal effects). An example of a non-stationary feature and a stationary feature for the stock 'ETN' are shown below.

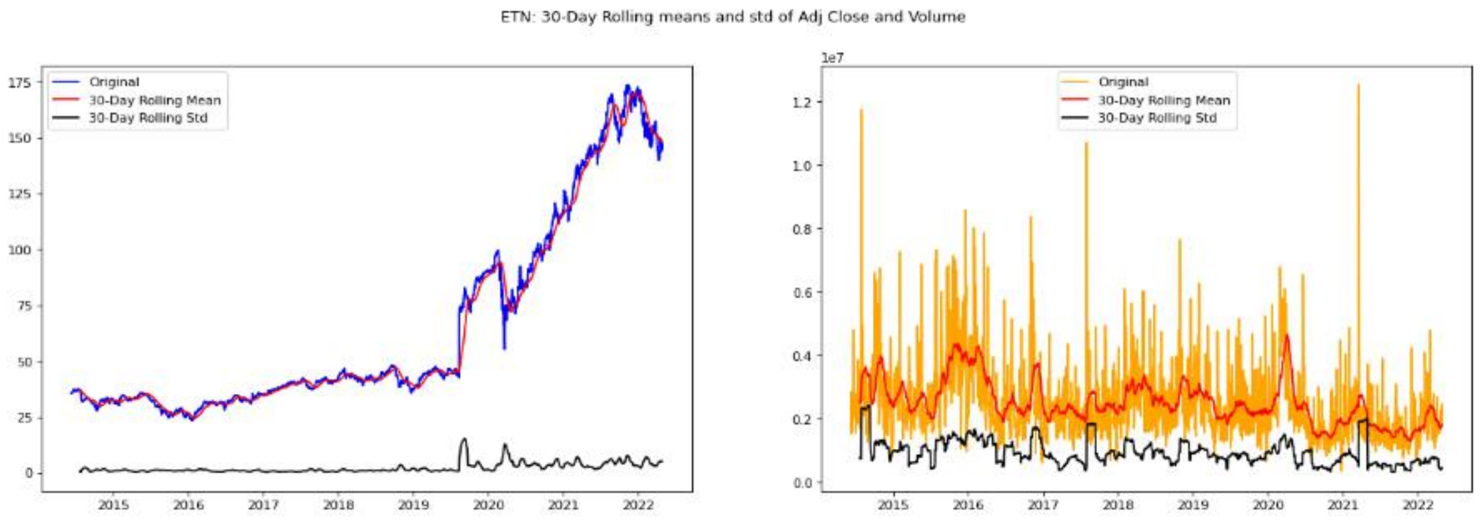


Figure 17: Left: A non-stationary feature whose mean increases over time. Right: A stationary feature whose mean and standard deviation stay fairly consistent over time. In general, prices trend upwards so won't be stationary but volumes fluctuate around a mean so should be stationary.

Why do we require our features to be stationary? Many ML models assume data is stationary and that samples observed at different times are independent. Furthermore, as an example, if columns have an upward trend (e.g. stock prices) our model may learn to just predict that the column will increase over time. This means that the model hasn't learnt anything deeper than what's already captured in the trend. This may lead to a live model performing no better than a naïve model that simply predicts that the target will increase.

To rigorously check for stationarity, we can perform a hypothesis test called the Augmented Dickey-Fuller (ADF) test which works as follows:

- The **null hypothesis** is that a *unit root* is present in the time series and it's therefore not stationary. This means that the value of the series is time-dependent.
- The **alternative hypothesis** is that there is no unit root present and it therefore is stationary. This means that the value of the series is not dependent on time.
- The **test statistic** is a number outputted from the test. If it is less than some critical value (which depends on the significance level), then the null hypothesis is rejected. Otherwise, there is insufficient evidence to reject the null hypothesis. Equivalently, if the p-value is less than the significance level, the null hypothesis is rejected.

An example of the test in action is shown below for the 'Adjusted Close' and 'Volume' columns for the stock 'RJF':

Stock	Column	Test Statistic	p-Value	1% Critical Value	5% Critical Value	1% Conclusion	5% Conclusion
RJF	Adj Close	0.518771	9.854326e-01	-3.433687	-2.863014	Fail to Reject Null: Non-Stationary	Fail to Reject Null: Non-Stationary
RJF	Volume	-8.876724	1.351718e-14	-3.433660	-2.863002	Reject Null: Stationary	Reject Null: Stationary

Figure 18: The ADF test in action. For the 'Adj Close' column, the test statistic isn't less than the critical value for both the 1% and 5% significance level, so we can't reject the null hypothesis and the feature is non-stationary. However, for the 'Volume' column, the test statistic is less than the critical value at both significance levels, so we can reject the null hypothesis and declare the feature stationary. Note that these tests align with our previous thoughts that prices don't tend to be stationary but volumes do.

The final step is to transform any non-stationary features to make them stationary. There are a few ways to do this but in this project we use a technique called *differencing*, which simply involves using the differences between consecutive observations rather than the observations themselves. These differences tend to be stationary and if they're not, we can perform differencing again until they are.

An example of a column before and after differencing is shown below:

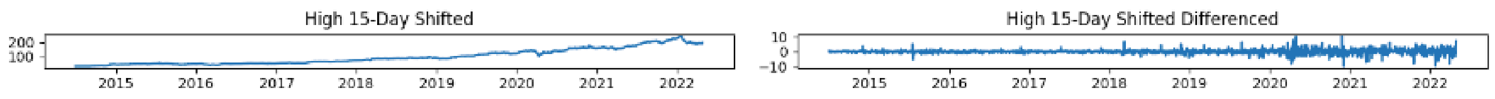


Figure 19: The column 'High 15-Day Shifted' for a particular stock before and after differencing. Note that the differencing removes the upward trend in the column. Also note the graph after differencing appears to get 'thicker' (has a bigger variance) as time goes on. This is because as the value of the original column increases, the absolute differences between observations get larger, despite the differences not changing much on a percentage basis. Fortunately, this increasing variance isn't enough to make the transformed feature non-stationary.

A final note on stationarity is that we should really only use data from the training set to test for stationarity, since a feature may be stationary over the whole dataset but not over the training set. However, this is unlikely since the training set will be the majority of the dataset and cover a large time period. Also note that the differencing technique is backward looking so we are not causing any *look-ahead bias*, which will be explained in the model training section. Using the whole dataset to test for stationarity is therefore acceptable.

Now that we have a very wide dataset with lots of generated features, we are ready to perform feature selection to pick the features with the most predictive power.

Feature Selection

The feature selection process contains many important steps:

- Split data into training, validation and test sets
- Standardize features
- Remove low-variance features
- Remove highly-correlated features
- Feature selection with SHapley Additive exPlanations (SHAP) values.

Split Data

Recall our data covers dates from 2014 to 2022. For the training set, we use data from 2014 to 2019 inclusive. Our validation set will be a subset of this training set but care needs to be taken due to the time series nature of the problem. We will talk more about this in the 'Model Training' section.

For the test set, it would be natural to use data from 2020 onwards given that the training set ends in 2019. However, it is good practice with time series prediction problems to leave a gap between the training and test sets. The reason for this is two-fold. Firstly, it makes any data leakage from the training set to the test set less likely. Secondly, if the test set comes immediately after the training set, a trained model may perform well on the test set because the test set is likely to be similar to the training set. However, the model may struggle to perform well on a different test set in the future when the nature of the data may have changed. This is especially the case in stock price prediction problems as market regimes change over time. We therefore leave out 2020 and have a test set that covers 2021 onwards.

Standardize Features

To ensure all features are on the same scale, we transform the features in both the training set and the test set using a standard scaler. This scaler standardizes each column so that it has a standard normal distribution with mean 0 and variance 1 (for every value in the column, subtract the column's mean and then divide by the column's standard deviation).

We once again have to be careful of information leakage here. We must scale both the training set *and* the test set using the mean and standard deviations of the *training set* columns. We can't scale the test set using the test set's means and standard deviations because this would mean that the test set no longer 'unseen' by the training process.

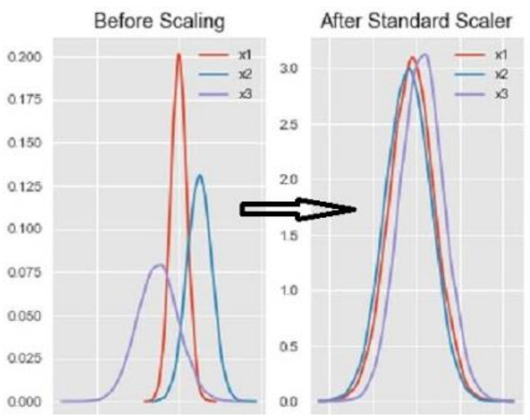


Figure 20: Effect of standard scaling on 3 example columns x1, x2 and x3.

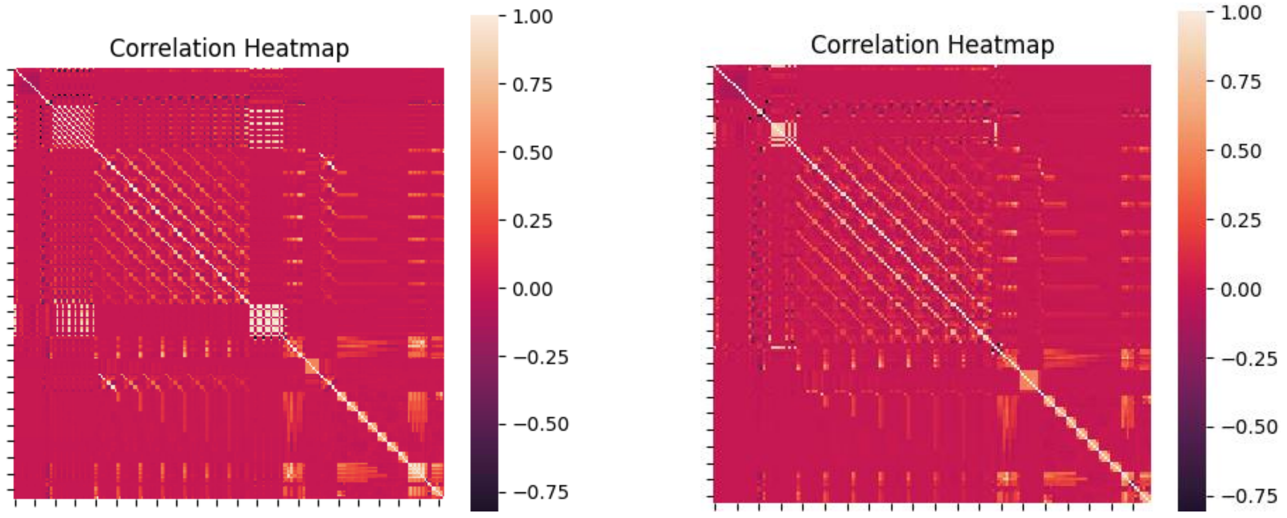
Remove Low-Variance Features

We check for features that are almost constant (have a very low variance) as these won't provide any meaningful information to the models. In our case, using a variance threshold of 0.01, there aren't any almost constant features anyway.

Remove Highly-Correlated Features

If any two features are very highly correlated, it's a good idea to remove one of them. This is because models such as Logistic Regression assume that there is no multi-collinearity between features. Furthermore, if one feature is highly correlated with another, it is not providing much additional information to the model.

We choose to remove any features that have a correlation coefficient greater than 0.9 with an already existing feature. This reduces the number of features from 213 to 159. This process is demonstrated below.



SHAP-Based

Figure 21: Correlation matrix of features before (left) and after (right) removing highly correlated features. Notice the white patches indicating high correlation in the left matrix are no longer present in the right matrix.

Feature Selection with SHAP values

Finally, we utilize a very useful function that is built in to the Python package *catboost*. This function trains various CatBoost models using subsets of features and recursively eliminates the least useful ones. It selects the best features using SHAP values which measure a feature's contribution to the final outcome.

We choose to remove around 50 features. The plot below shows the loss of each trained model decreasing as features are removed.

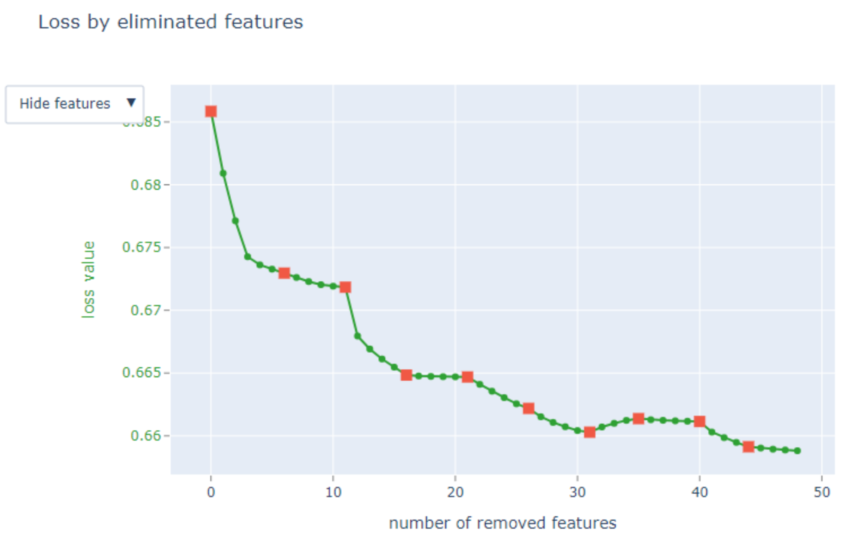


Figure 22: Plot showing the loss of each trained model decreasing as features are recursively removed. Each dot represents the removal of a single feature.

The final outcome of the feature selection process is that we are left with 113 features that have been properly standardized. This final dataset is the one used for model training and testing. We choose to drop 'Symbol' as a feature as we want the model to learn to pick up signals regardless of the stock.

Model Training

Now that we have selected the relevant features, we are finally ready to build some models. We build three models:

- Logistic Regression
- XGBoost (eXtreme Gradient Boosting)
- LSTM (Long Short-Term Memory network)

Logistic Regression is a simple regression model that can be used as a baseline. XGBoost is more complex and utilizes gradient-boosted decision trees. An LSTM is a type of Recurrent Neural Network (RNN) that is well-known for its strength in time series problems.

For all 3 models, we must consider the fact that the classes are imbalanced. There are more instances of a stock's return being positive than negative due to most stock prices going up on average over time. Fortunately, all three models have parameters that you can pass during training to account for the class imbalance so we make sure to supply these.

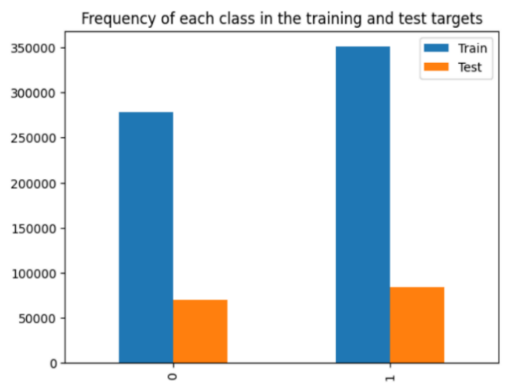


Figure 23: Class frequencies in the training and test set. Notice there are more 1s than 0s in both the training and test set.

We give an overview of the training of each model below. We won't go into detail on the underlying algorithms since this is well-covered elsewhere.

Classifier 1: Logistic Regression

To train a logistic regression model, we must tune its parameters to optimize performance on a validation set. We utilize the *sklearn* class *RandomizedSearchCV* to search for parameters (from a pre-defined grid) that maximize the accuracy of the model when making predictions on the validation set. How is this validation set chosen?

We must choose the validation set in such a way that the ordering of the time series data is respected so that we don't introduce *look-ahead bias*. This is a bias that occurs when data in the future (that would not be readily available in a live setting) is used to train a model.

This is most easily explained through an example. Suppose that we use k-means cross validation to train a model. This method randomly samples data points to add to the validation set on each fold. During a particular fold, a sample from 2020 is added to the training set and a sample from 2015 is added to the validation set. To test the trained model for accuracy, all targets in the validation set are predicted. Therefore, we are predicting the target for a sample from 2015 having used a sample from 2020 for training. This will cause the performance of the model on the validation set to look very strong but in reality, the model has been trained on data from the future which would not be available in a live model.

To get round this problem, we use a time-series cross validator that ensures that all points in any validation set come after those in the training set. An example of a 5-fold time-series split is shown below.

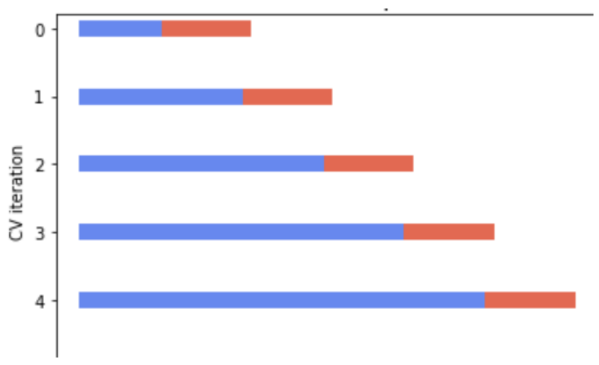


Figure 24: A 5-fold time series split. The blue bar represents training data and the orange bar represents validation data. Notice that on every fold, the validation always comes chronologically after the training data.

Using this time series cross-validation technique, we optimize the regularization parameter C as well as whether to use L1 or L2 regularization. Using *RandomizedSearchCV* leads to various models being trained with different parameter combinations. The best parameters are the ones that lead to the best average accuracy score across all folds of the time-series cross validation set.

Now that we have optimal parameters, we simply train a model with these parameters but using the whole training set. This lets us train on as much data as possible with parameters we know are close to optimal since they were optimal on the various folds during cross-validation.

Classifier 2: XGBoost

As with logistic regression, we tune the parameters of XGBoost using time-series cross validation. We use the same time-series technique to create folds as we did for logistic regression but with a slight tweak. Just like we left a gap between the training and test set while performing the initial data split, we leave a gap of about 10 days between the training and validation set in each fold. This helps to prevent information leakage and allows the trained models to learn longer term patterns.

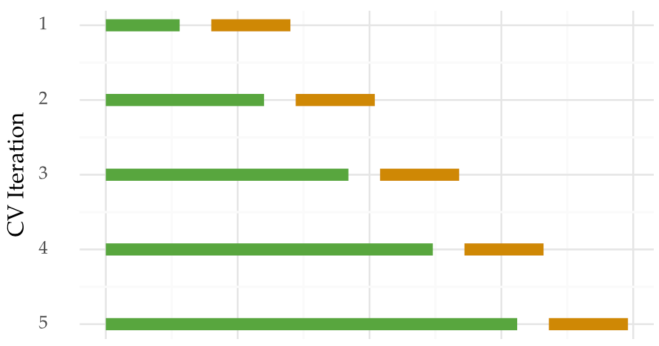


Figure 25: A 5-fold time series split with gaps. The green bar represents training data and the orange bar represents validation data. Notice the small gap between them.

For hyperparameter optimization, we use a popular library called *optuna*. For each parameter we would like to tune, we define a minimum and maximum allowed value, giving us a range of possible values. We also choose a probability distribution to associate with each parameter:

- A *uniform distribution* is used if you think all values in the range are equally likely to be optimal.
- A *log-uniform distribution* is used if you think smaller values in the range are more likely to be optimal than larger values.

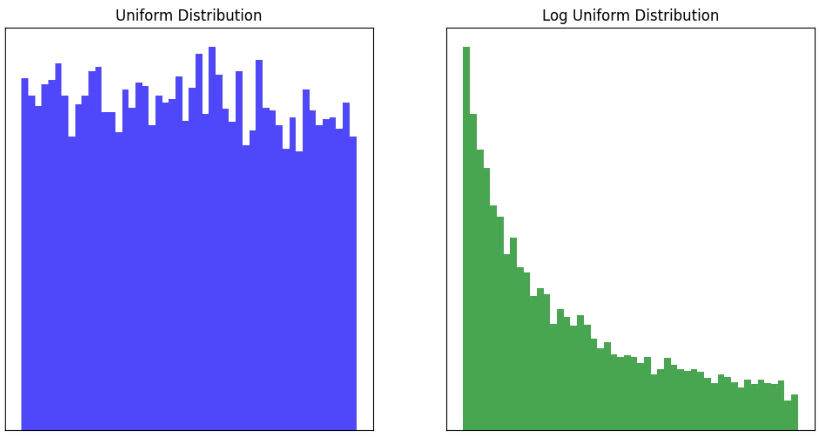


Figure 26: Uniform data vs log-uniform data. Notice that smaller values are more likely to be drawn in log uniform data.

Now that we have our parameter ranges and associated probability distributions, we run various ‘trials’. On each trial, we randomly draw a value from each parameter’s probability distribution to give us a set of parameters. We then perform time-series cross validation by training a model with these parameters for each fold. We then calculate the average AUC score across the validation folds.

This process is repeated for several trials. Once all the trials are finished, the optimal parameters are the parameters from the trial that produced the best average AUC score. The optimization history as well as the relative hyperparameter importances are shown below.

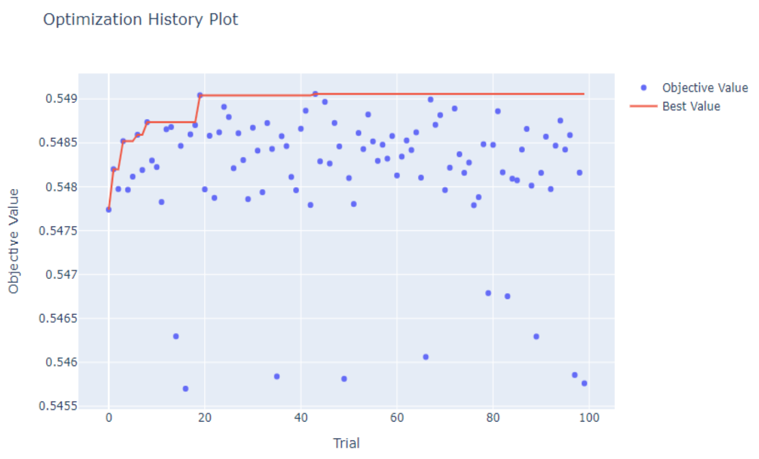


Figure 27: Parameter optimization history. Each blue dot represents a trial and shows us the average AUC score from that trial. The red line keeps track of the AUC score from the best trial so far. The 44th trial is the optimal trial.

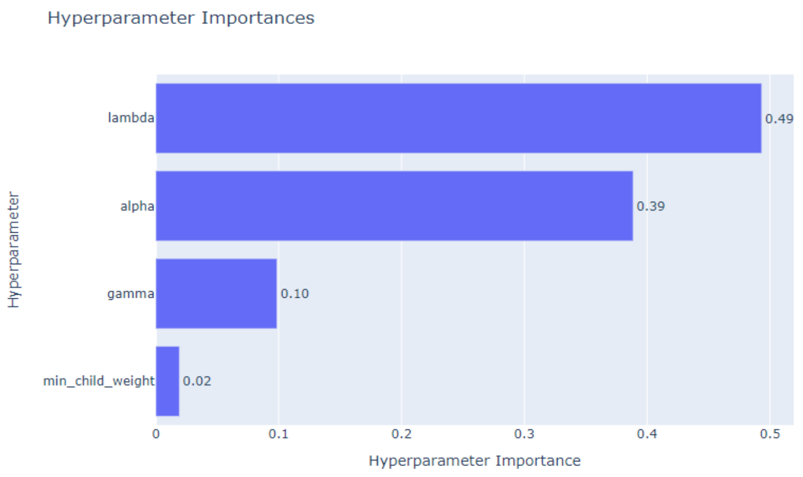


Figure 28: Relative hyperparameter importances. The regularization parameters 'lambda' and 'alpha' seem to be the most important.

In exactly the same way as for logistic regression, now that we have optimal parameters, we simply train a model with these parameters but using the whole training set. This lets us train on as much data as possible with parameters we know are close to optimal since they were optimal on the various folds during cross-validation.

Classifier 3: LSTM

The final model we train is an LSTM. Since this is a complex model and already prone to overfitting, we choose not to optimize its parameters for now and instead just build a simple model with reasonable parameters. In fact, we actually build 5 identically structured models by performing the same time-series split as we did for XGBoost and training one model per fold. The final ensemble model will make predictions by averaging the predictions of these 5 models.

However, some data prep is required first because LSTMs expect data in a slightly different format to models such as Logistic Regression and XGBoost. The idea is to pass *sequences* of data to the LSTM. We first define a lookback period, in our case we use 10 days. Then, on a stock-by-stock basis, we start 10 days into the dataset, gather all the features for that stock from the last 10 days, then append the current target. The window is then rolled forward 1 day and we once again look at all features from the last 10 days and append the current target. These sequences of data are what we pass to the LSTM for training and validation. The full implementation can be seen in the LSTM notebook.

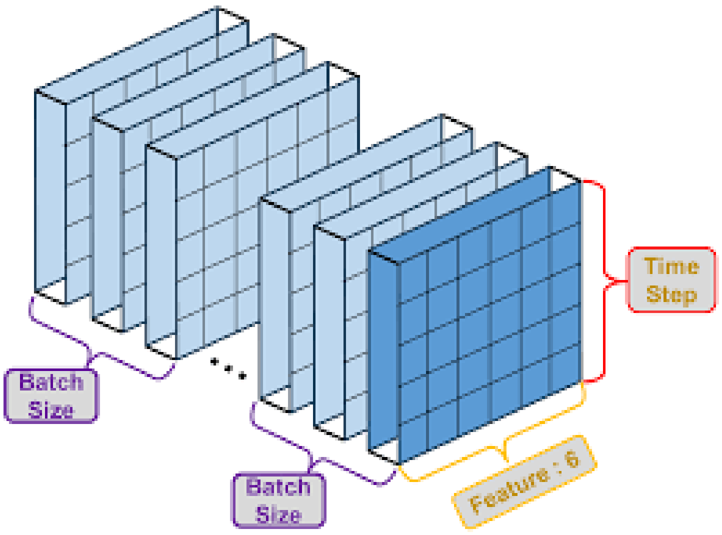


Figure 29: Visual representation of required data structure for training LSTMs. This example has 6 features, a window size of 5 timesteps, and 3 samples per batch.

Note that in theory, when looking back at the last 10 days of data, you could also include the target itself as a feature. However, we have to be very careful of look-ahead bias since the target is looking 1 week ahead. For example, if we included yesterday's target to help predict today's target, then yesterday's target is not actually available yet and won't be for another few days. (This mistake was initially made and it led to crazily good results. It's very easy to predict a target that looks 1 week ahead if you already have the target from yesterday's data!). To avoid any potential look-ahead bias, we choose to not include previous targets as a feature.

Now that the data is in the correct format, we are ready to construct the network. The network has 4 layers with brief overviews below:

- *Input Layer:* Layer to take in the sequences of data.
- *LSTM Layer:* Recurrent layer that can learn long-term dependencies in sequences.
- *Hidden Layer:* Takes the LSTM layer's output and transforms it to higher level feature space using a 'rectified linear unit' (ReLU) activation function to add some non-linearity without being computationally expensive.
- *Output Layer:* Layer that outputs a probability between 0 and 1 using a 'sigmoid' activation function. This can be interpreted as the probability that the sample belongs to the positive class so if it's greater than 0.5 we classify the sample as 1, otherwise we classify it as 0.

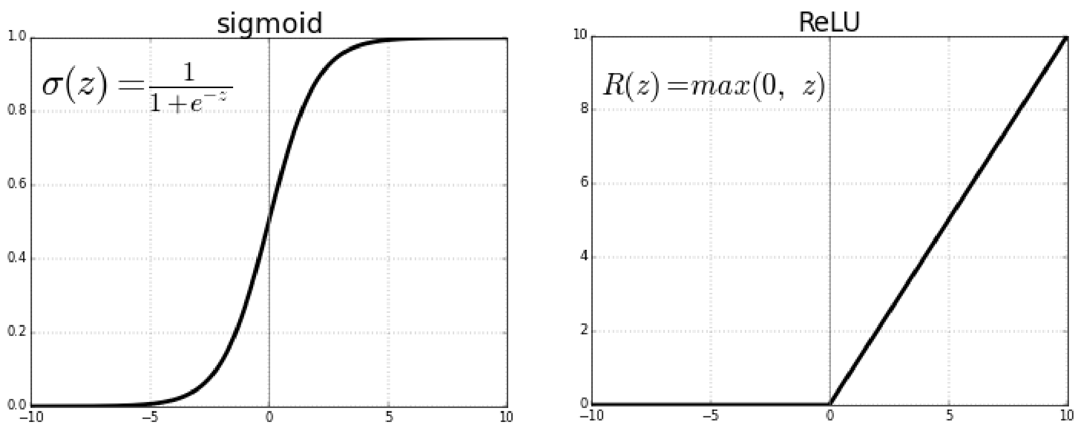


Figure 6: Left; Sigmoid activation function that outputs a number between 0 and 1 that can be interpreted as a probability. Right: ReLU activation function, allowing the model to capture some non-linear patterns without being too computationally expensive.

Now we address the problem of potential overfitting. We can help to prevent overfitting in two different ways. The first method is to supply a 'dropout' parameter to the LSTM layer. We supply a value of 0.3. This means that at each training time step, 30% of LSTM units will randomly be set to 0. This helps the trained model to not become over-reliant on any single unit. The second method is to add some L2 regularization to the hidden layer. This helps to reduce the magnitude of weights in the network which can help to reduce overfitting.

Now that the network is built, we simply fit it using the various training and validation folds. As previously mentioned, the final result is 5 trained models. Since each model outputs probabilities rather than classifications when making predictions, we can average the probabilities and then make a classification based on the average probability. Namely, if the average probability is greater than 0.5 then the sample belongs to class 1 otherwise it belongs to class 0.

Results

Now it's time to compare how the 3 models performed on the unseen test set. A good place to start is the confusion matrices. These show how many correct and incorrect classifications were made by each model.

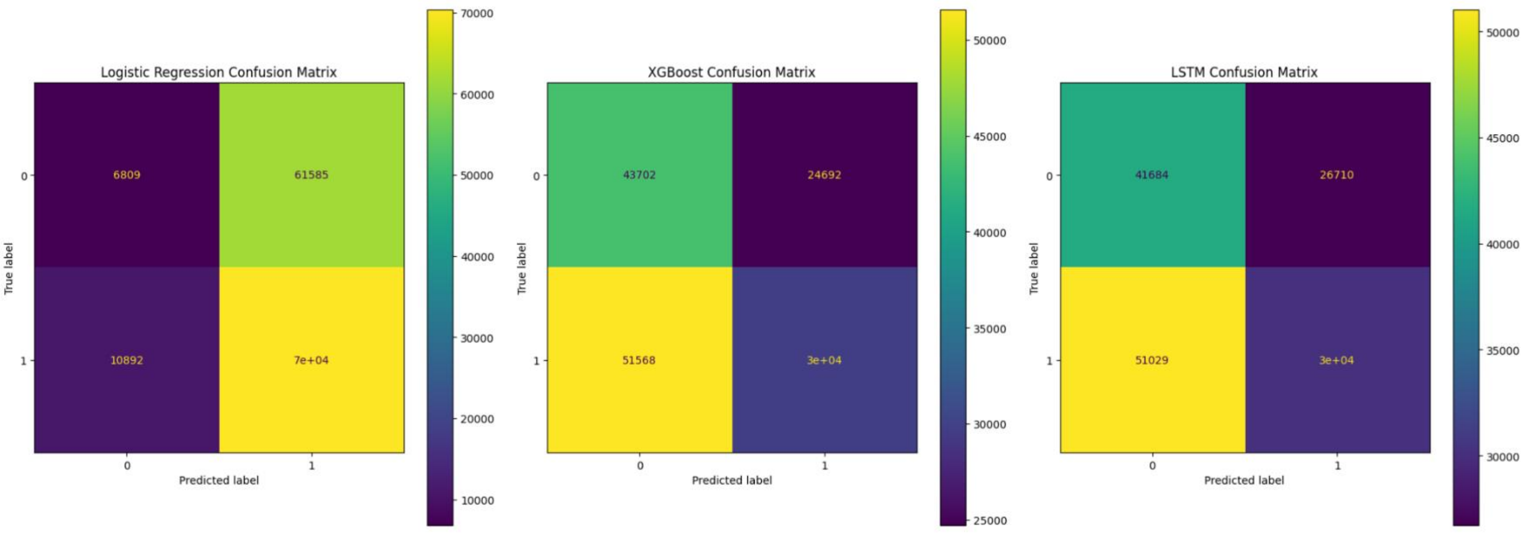


Figure 31: Confusion matrix for each model. XGBoost and LSTM have fairly similar confusion matrices but Logistic Regression massively overpredicts the positive class.

Both XGBoost and LSTM overpredict the number of samples in the negative class and underpredict the number of samples in the positive class. The opposite is true for Logistic Regression whose confusion matrix massively overpredicts the positive class.

The next step it to compare the accuracy of the various models, where accuracy is simply how many predictions were correct divided by the total number of predictions. The accuracies for each model are shown below. Note that we show accuracies for predictions made on both the training set and the test set. This is to check for overfitting.

	Logistic Regression	**XGBoost**	**LSTM**
Test Set	0.516	0.490	0.480
Training Set	0.581	0.648	0.582

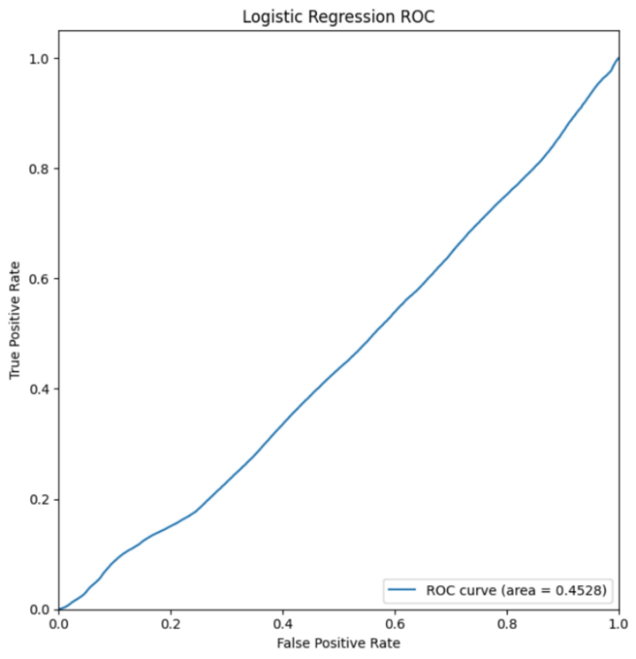
Figure 32: Accuracy of predictions on the test set and the training set for each model.

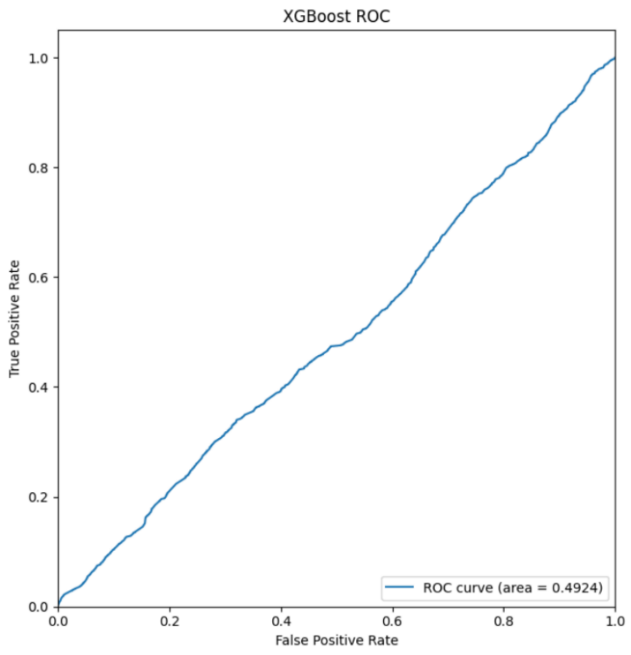
The first thing to notice is that the accuracy of all 3 models on the unseen test set is pretty poor, with accuracies around the 50% mark. There are a few potential reasons for this:

- The stock market is extremely unpredictable. While good companies tend to perform well in the long run, it is extremely difficult to predict what the price of a stock will do in the next week.
- We have very limited data for each stock. Pricing, volume and static data alone is not enough to accurately predict how a stock will perform in the short term. Other data sources are needed like company fundamentals, exchange/orderbook data and sentiment analysis.
- The training set spans a relatively small time period. Ideally we'd have 20+ years of data.
- More useful features could have been generated.

The accuracies on the training set are stronger as you would expect. However, XGBoost seems to be overfitted as it manages to achieve a 64.8% accuracy on the training set, which is very high. The LSTM may also be slightly overfitted.

In order to compare which model is best, we choose to look at AUC instead of accuracy, as this measure gives a better idea of the overall performance of a model when it comes to maximizing true positives and true negatives, and minimizing false positives and false negatives. The ROC curves are shown below with the areas (AUC) labelled.





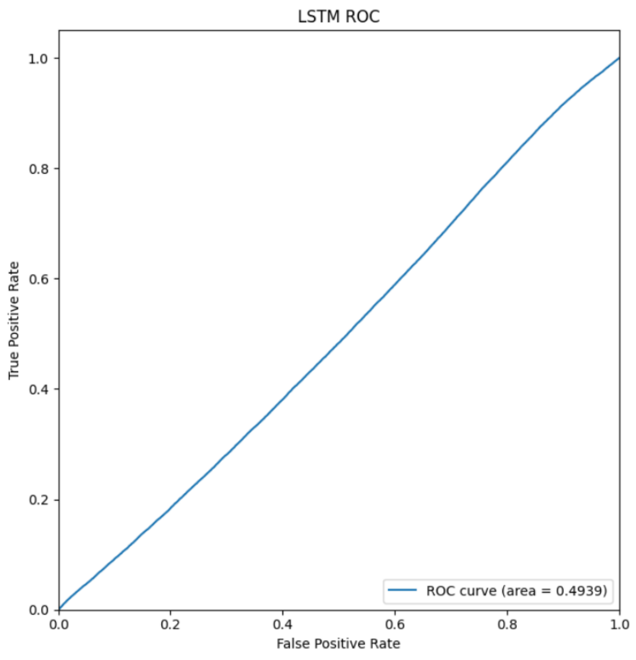


Figure 33: ROC curves for each model. The area under them gives us an idea of the overall performance of the models.

Both XGBoost and LSTM have AUC scores around 49-50% , while Logistic Regression has an AUC around 45%. While none of the values are particularly high (randomly making predictions by the toss of a coin would give an AUC of 50%), these graphs showcase the fact that XGBoost and the LSTM are better choices than Logistic Regression for this problem. This is expected due to Logistic Regression being unable to capture the non-linear patterns in stock market data.

Finally, we show how a trading strategy would perform using each model's predictions. The trading strategy is as follows (assuming no transaction costs):

- On day 1, make predictions for the direction each stock will move.
- Construct an equally-weighted portfolio. Go long the stocks whose prices are expected to go up. Go short the stocks whose prices are expected to go down.

- At the end of the week, rebalance by making predictions again and constructing a new equally weighted long-short portfolio based on the new predictions.

The results of running this strategy for each model are shown below. We also compare the strategies to the performance of the S&P 500 index itself (which is traditionally extremely hard to outperform).

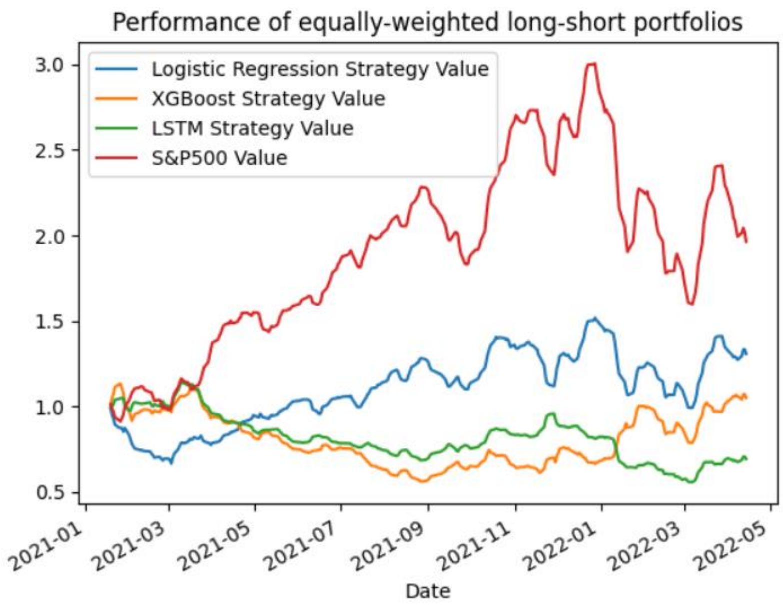


Figure 7: Relative performance of each model using the same strategy.

The S&P 500 massively outperforms these models as expected due to their poor accuracy. The XGBoost and LSTM strategies are heavily correlated and both perform poorly. The Logistic Regression strategy performs slightly better and isn't as correlated to the other two. However, this slightly better performance is simply due to the Logistic Regression model massively overpredicting the positive class, which is beneficial for this strategy's performance when stocks go up over time on average.

<u>Model Recommendation</u>

Overall, it would be foolish to recommend any of these models to use in the live stock market. However, given better data, it has been shown that boosted trees such as XGBoost can perform just as well as complex neural networks and can be less prone to overfitting if regularized properly. It would be an interesting project to see if an XGBoost with slightly better data could achieve an accuracy significantly higher than 50%.