AI For Finance (Assignment-2)

2023-04-03

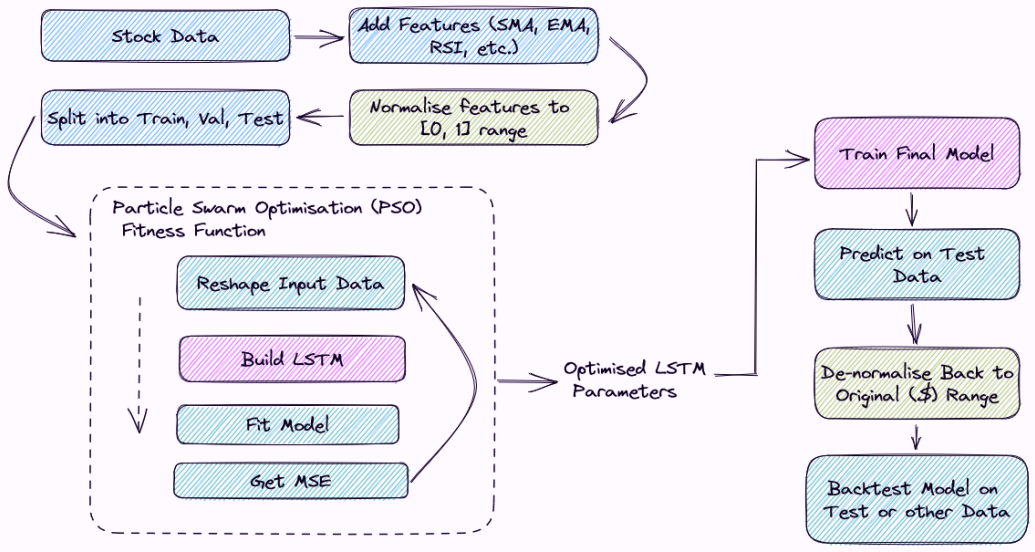
Price Prediction-Based Trading

Price Prediction-Based Trading is a strategy that utilizes machine learning and statistical analysis to forecast future asset prices, aiming to identify market trends and potential reversals. Key aspects include using historical financial data, selecting suitable prediction models, preprocessing data through feature engineering, evaluating model performance, implementing risk management strategies, and continuously updating models to maintain accuracy. The use of historical financial data, the selection of appropriate prediction models, the preprocessing of data through feature engineering, the evaluation of model performance, the implementation of risk management strategies, and the continuous updating of models to maintain accuracy are all crucial elements. Although this method has the potential to enhance decision-making and market understanding, it is important to remember that price predictions inherently involve some degree of uncertainty and that no model can guarantee a 100% success rate.

References: https://www.simplilearn.com/tutorials/machine-learning-tutorial/stock-price-prediction-using-machine-learning

https://www.investopedia.com/articles/active-trading/110714/introduction-price-action-trading-strategies.asp

Overview Diagram



The above diagram is an overview of our PSO training process, including the preprocessing steps (explained below), PSO meta-optimisation, and final model training and inference.

Technical Indicators as Features

Simple Moving Average (SMA)

In the world of finance, the **Simple Moving Average (SMA)** is a common tool for examining stock prices and trading volume over time. In the field of AI for Finance, SMA may be used as a component in ML models or as the foundation of an algorithmic trading system. In a nutshell, SMA "moves" along the time series by averaging a predetermined amount of data points inside a predetermined time interval. The goal is to eliminate noise caused by temporary variations so that the true trends become apparent. Simple moving averages (SMAs) are used by traders to track the long-term trend of a stock or other investments without being distracted by short-term fluctuations. This enables investors to evaluate longer-term and medium-term trends side by side. The price of an asset's bullish or bearish trend may be predicted with the use of a SMA. The market is considered to be in an uptrend if the SMA is moving higher, and a downtrend if the SMA is moving downward. Crossing the short-term SMA above the long-term SMA creates a buy signal, while crossing the short-term SMA below the long-term SMA results in a sell signal. **Reference:** https://www.investopedia.com/terms/s/sma.asp

Exponential Moving Average (EMA)

For analyzing financial time series data, such as stock prices, the **Exponential Moving Average (EMA)** is a popular technical indicator. EMA is a type of moving average that assigns more weight to recent data points, making it more responsive to price changes compared to the Simple Moving Average (SMA). EMA is more suited for spotting variations in trends early on because of the smoothing feature that helps eliminate latency. Technical analysis, trading strategies, and even machine learning models all make use of EMA, demonstrating its versatility in the financial sector. Commonly employed for identifying support and resistance levels or generating buy/sell signals, EMA crossovers are often used to gauge bullish or bearish trends in the market. **Reference:** https://www.investopedia.com/terms/e/ema.asp

Relative Strength Index (RSI)

When applied to financial time-series data like stock prices, the **Relative Strength Index (RSI)** acts as a momentum oscillator. It provides short-term buy and sell signals, identifies market trends, and detects potential trend reversals or price corrections. Values for RSI range from 0 to 100 and reflect the average gains and losses over a certain time frame. RSI calculates average gains and losses over a specific period, with values ranging from 0 to 100. Overbought (above 70) or oversold (below 30) conditions indicate impending price adjustments, while a value of 50 represents a neutral trading level. When the RSI indicates a divergence between price and the oscillator, it may be time to sell or buy. In addition, RSI's center line crossovers and the formation of chart patterns like double tops and bottoms may be used to spot bullish and bearish price fluctuations. Double tops ('M' shaped) imply a downward trend reversal, whereas double bottoms ('W' shaped) indicate an upward price movement. **Reference:** https://www.investopedia.com/terms/r/rsi.asp

Moving Average Convergence Divergence (MACD)

For financial time-series data like stock prices, the **Moving Average Convergence Divergence (MACD)** is a trend-following momentum indicator that can assist traders spot reversals in trend direction, momentum, and potential. It is calculated by subtracting the 26-period EMA from the 12-period EMA, resulting in the MACD line. To further aid in the identification of bullish or bearish trends, this line is drawn in conjunction with a 9-period EMA signal line. A bullish signal is given when the MACD line rises above the signal line. This might indicate a good time to purchase. When the MACD line drops below the signal line, however, it's interpreted as a bearish indication and may signify a good time to sell. MACD divergence from price action can signal an imminent trend reversal. For example, the 12-day EMA is $50, and the 26-day EMA is $48, resulting in a MACD line value of $2. The 9-day EMA signal line is $1.5. Since the MACD line ($2) is above the signal line ($1.5), it indicates that the shorter-term (12-day) momentum is stronger than the longer-term (26-day) momentum, suggesting a bullish trend. In this scenario, traders may consider it a potential buying opportunity.. If the MACD line and price action diverge, it may suggest a trend reversal, warranting caution in trading decisions. The MACD is most commonly used with daily periods, where the traditional settings of 12, 26, and 9 days are standard. **Reference:** https://www.investopedia.com/terms/m/macd.asp

Bollinger Bands

Bollinger Bands is a popular technical analysis tool used by traders and investors to measure the volatility of an asset and identify potential overbought or oversold conditions. Bollinger Bands consist of three lines: a moving average (typically a simple moving average) and two standard deviation lines plotted above and below the moving average. In AI for Finance, Bollinger Bands can serve as a signal for a variety of machine learning models and algorithmic trading methods.

Here's a brief explanation of how Bollinger Bands work:

1. Moving Average: Take the asset's closing prices over a predetermined time frame (say, 20 days) and derive a moving average. This line represents the central trend and serves as a basis for the upper and lower bands.
2. Standard Deviation: This statistic, which measures the asset's volatility, should be computed over the same time frame as the mean and standard deviation. Asset volatility increases as the standard deviation rises.
3. Upper and Lower Bands: Create the upper and lower bands by adding (for the upper band) and subtracting (for the lower band) a multiple of the standard deviation (usually 2) from the moving average. This creates an enclosing band around the moving average that expands and contracts in relation to the asset's volatility.

For example, A common Bollinger Bands trading technique is to purchase when the asset's price falls below the lower band (a possible oversold indication) and sell when it rises above the upper band (a possible overbought signal) (a potential overbought signal). In combination with other technical indicators or features, a machine learning model might utilise these signals to understand the patterns in the data and improve the quality of its forecasts or trade signals.

It's important to note that Bollinger Bands, like any other technical indicator, should not be used in isolation. Combining Bollinger Bands with other technical indicators, fundamental analysis, or machine learning models can provide a more robust and comprehensive approach to analyzing and trading financial markets.

Reference: https://www.investopedia.com/terms/b/bollingerbands.asp

As Features

The aforementioned technical indicators represent aggregate information about the asset's price movements. Because of this, we have integrated them in our dataset as features from which the neural network could potential learn predictive price patterns.

Train/Test Split

The train_test_split function in the provided R code takes a stock_data object and splits it into three subsets: training (70%), validation (15%), and test (15%). This custom function calculates the number of rows for each set, extracts the respective portions of data, and returns a list containing the three datasets for model training, hyperparameter tuning, and evaluation. We train the LSTM model on the training set, and evaluate using the validation data during training. Our final tests are performed using the validation and testing data.

Normalisation

We are using two functions for normalising and denormalising the data. The normalize_data function scales the input data to a range between 0 and 1, while the denormalize_data function reverses this process, restoring the data to its original range. **Normalisation** is a process where the numerical data are often normalised to a range between 0 and 1 as a preprocessing step. The performance of machine learning algorithms can be enhanced by using this method to reduce the effect of variables with varying scales. In contrast, **Denormalisation** involves reversing normalisation by rescaling the data to its original range. We are utilising normalisation as a preprocessing step because we found empirically that performance is drastically improved by normalisation. This is likely because neural networks perform best in the [-1, 1] range due to activations, as well as the fact that normalisation puts all the features in the same absolute range. When certain features are at a much different scale (e.g. RSI vs Closing Price) this can hurt the performance of the NN.

Preprocessing using Sliding Windows

```r
preprocess_data <- function(data, lookback_window, horizon) {
  # Normalize the data
  normalized_data <- data
  num_samples <- nrow(data) - lookback_window - horizon + 1
  x <- array(0, dim = c(num_samples, lookback_window, ncol(normalized_data)))
  y <- array(0, dim = c(num_samples, horizon))

  for (i in 1:num_samples) {
    x[i, , ] <- normalized_data[i:(i + lookback_window - 1), ]
    y[i, ] <- normalized_data[(i + lookback_window):(i + lookback_window + horizon - 1), "Close_Price"]
  }

  list(x = x, y = y)
}
```

In the above code chunk, the preprocess_data function in the provided R code preprocesses the input data for time series prediction by creating input-output pairs for a given lookback window and prediction horizon. In order to store the input data and the desired values, it makes two arrays, x and y. The function iterates through the data, extracting segments using a sliding window of the lookback window size for input and the corresponding target values, based on the prediction horizon. It returns a list containing the input-output pairs (x and y). This allows the NN model to see just the required input data with a ground-truth price for output comparison.

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem in RNNs, allowing them to effectively learn long-range dependencies in time series data or sequential information. LSTM networks consist of memory cells and gating mechanisms, which control the flow of information within the network, deciding what information to store, update, or discard. In the context of AI for Finance, LSTMs have been successfully applied to various financial time series prediction tasks, such as stock price forecasting, foreign exchange rate prediction, and market trend analysis. Their ability to model complex, non-linear patterns and capture dependencies over extended periods makes them a powerful tool for financial time series analysis.Investment strategies, risk management, and financial decision-making may all benefit from LSTMs because of the insights and forecasts they can provide from financial data.

Reference:

https://prvnk10.medium.com/how-lstms-solve-the-problem-of-vanishing-gradients-ea88f08c78ca

https://www.datacamp.com/tutorial/lstm-python-stock-market

Fjellström, C. (2022). Long short-term memory neural network for financial time series. arXiv preprint arXiv:2201.08218 https://arxiv.org/pdf/2201.08218.pdf

```
build_lstm_model <- function(input_shape, learning_rate, regularization) {
  model <- keras_model_sequential() %>%
    layer_lstm(units = 20, input_shape = input_shape, return_sequences=TRUE,
               kernel_regularizer = regularizer_l2(l = regularization)) %>%
    layer_dropout(rate=0.5) %>%
    layer_lstm(units = 10, kernel_regularizer = regularizer_l2(l = regularization)) %>%
    layer_dropout(rate=0.5) %>%
    layer_dense(units = 1)

  model %>% compile(
    loss = "mse",
    optimizer = optimizer_adam(learning_rate = learning_rate)
  )

  return(model)
}
```

In the above chunk of code, we have defined a function called build_lstm_model that creates and compiles an LSTM model using the Keras library. The function takes three arguments: input_shape, learning_rate, and regularization. The keras model sequential() function is used to build the model architecture, and it generates a linear stack of layers that may be appended to in sequence using the %>% pipe operator. The layers of the model are as follows:

layer_lstm: The first LSTM layer has 20 units and takes an input_shape as input. The return_sequences parameter is set to TRUE, which means that the layer will output the full sequence of hidden states for each time step, rather than just the final hidden state. The kernel_regularizer parameter is set to L2 regularization, with the strength controlled by the regularization argument.

layer_dropout: After the initial LSTM layer, a layer with a dropout rate of 0.5 is added. This layer randomly sets a proportion of 50% of the input units to 0 during training, which helps prevent overfitting.

layer_lstm: The second LSTM layer has 10 units and also uses L2 regularization, with the strength controlled by the regularization argument. Since the return_sequences parameter is not set, this layer will only output the final hidden state by default.

layer_dropout: Another dropout layer with a rate of 0.5 is added after the second LSTM layer, further helping to prevent overfitting.

layer_dense: A dense (fully connected) output layer with one unit is added at the end of the model. This layer is responsible for producing the final prediction.

The compile function is used to set up the model for training after the architecture has been defined. Mean Squared Error (MSE) is chosen as the loss function, which is typically employed for regression issues, and Adam is chosen as the optimizer with the learning rate given. Finally, the build_lstm_model function returns the compiled LSTM model, which can be used for training and predicting on financial time series data. **References:** https://medium.com/geekculture/10-hyperparameters-to-keep-an-eye-on-for-your-lstm-model-and-other-tips-f0ff5b63fcd4

Particle Swarm Optimization (PSO)

Particle swarm optimisation (PSO) is a robust optimisation technique used in artificial intelligence for finance to discover optimal solutions to complicated financial issues, and it was inspired by the social behaviour of flocking birds. This metaheuristic search technique is population-based; it maintains a swarm of particles (solutions) and repeatedly adjusts their positions depending on both local and global solutions found so far. This can be used with virtually any model type that has tunable parameters or weights. The choice of model depends on the specific problem you're trying to solve and the characteristics of the data. In the context of finance, PSO may be used for portfolio optimisation, with each particle representing a possible distribution of funds over a collection of assets.

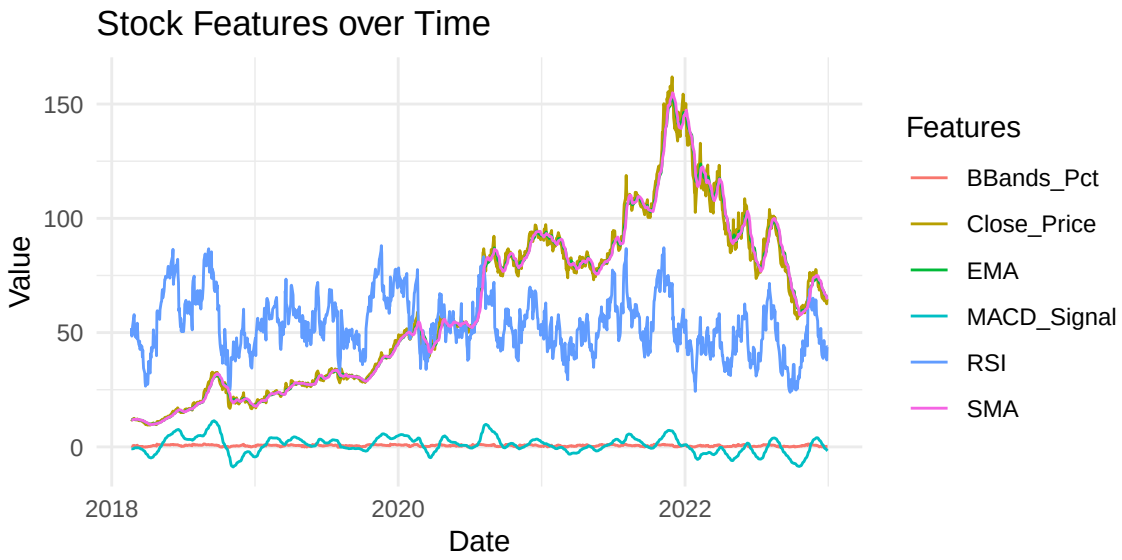
PSO Fitness Function

```
mse_fitness_function <- function(pso_params, train_data, val_data, horizon) {
  learning_rate <- pso_params[1]
  regularization <- pso_params[2]
  epochs = pso_params[3]
  lookback_window = pso_params[4]

  processed_train <- preprocess_data(train_data, lookback_window, horizon)
  processed_val <- preprocess_data(val_data, lookback_window, horizon)

  model <- build_lstm_model(
    input_shape = dim(processed_train$x)[-1],
    learning_rate = learning_rate,
    regularization = regularization
  )

  current_time <- format(Sys.time(), "%Y-%m-%d_%H-%M-%S")
  log_dir <- paste0("logs/fit/", current_time, "_lr_", learning_rate, "_reg_", regularization,
                    "_epochs_", epochs, "_lookback_", lookback_window)

  # Initialize tensorboard callback with the unique log directory
  tensorboard_callback <- callback_tensorboard(log_dir = log_dir)

  history <- model %>% fit(
    x = processed_train$x,
    y = processed_train$y,
    validation_data = list(processed_val$x, processed_val$y),
    epochs = epochs,
    batch_size = 32,
    verbose = 0,
    callbacks=list(tensorboard_callback)
  )

  val_mse <- tail(history$metrics$val_loss, 1)
  return(val_mse)
}
```

In the above code chunk we define our PSO fitness function as `mse_fitness_function`. We are using PSO to optimise the training hyperparameters of our LSTM network which will be used for future price prediction. In the fitness function, we preprocess the training and validation data with the appropriate lookback window and horizon value. We then build the LSTM model with the appropriate learning rate and regularisation values. Then, the model is trained with a batch size of 32 and the number of epochs decided by the PSO algorithm.

Training

```
# Split the data into training, validation, and test sets

symbol <- "AMD"
start_date <- as.Date("2018-01-01")
end_date <- as.Date("2023-01-01")
stock_data <- getSymbols(symbol, src = "yahoo", from = start_date, to = end_date, auto.assign = FALSE)

prepped_stock_data = prepare_stock_data(stock_data)

plot_stock(prepped_stock_data)
```




```
normal_data_training = normalize_data(prepped_stock_data)
split_data_training = train_test_split(normal_data_training)

train_data = split_data_training$train
validation_data = split_data_training$val
test_data = split_data_training$test
```

Data Selection

In the above plot, we have plotted a time series of all of our input features into the model. These are explained in detail in the previous sections. In the plot, it's easy to see how different features are at different scales, which is why the normalisation is necessary. In this example, we have decided to optimise and train our model using data from the closing prices of AMD (Advanced Micro Devices, Inc.) on the NYSE (New York Stock Exchange). The time period selected is from January, 2018 to January, 2023. This is then split into train, validation, and test sets and preprocessed with normalisation.

Hyperparameter Bounds Selection

We have defined the lower and upper bounds for each of the LSTM hyperparameters to be optimized by PSO. We are then generating an initial set of hyperparameters randomly within these bounds. The specific hyperparameters being optimized are:

Learning rate: Ranges from 0.0001 to 0.005. Regularization: Ranges from 0.001 to 0.01. Epochs: Ranges from 10 to 100. Lookback window: Ranges from 5 to 50.

With these initial hyperparameters, we can apply the PSO algorithm to find the optimal values for each parameter within their specified bounds. This will allow us to minimize the mean squared error (MSE) and improve the LSTM model's predictive accuracy for stock prices. The bounds for the algorithm were found empirically to be a range of values that maintain fairly stable performance. With that said, it's still necessary for the PSO algorithm to find the optimal set of parameters.

Above, we run the PSO algorithm for 20 iterations. More iterations would be preferable, but due to our lack of accss to appropriate GPU resources the training process was found to take too long for more iterations. After running the PSO algorithm for 20 iterations, it found the below optimal parameters. We are putting them in here manually to avoid unnecessary time for re-computing the parameters by re-running the PSO process.

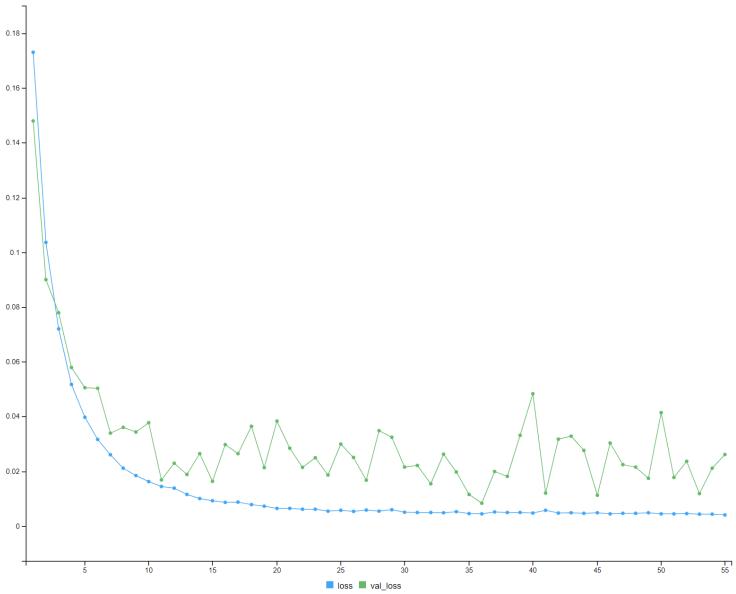
Our training sessions were logged to tensorboard, and by evaluating the performance of the different iterations we found the below optimal hyperparameters:

```
optimal_params = c(0.00221, 0.00415, 55, 24) # LR, L2 regularisation, epochs, lookback window
```

Evaluation

Using the optimal hyperparameters, we here train our final evaluation model using the training and validation data. The below plot shows the training and validations loss (MSE) per epoch of training.

```
train_model = train_final_model(optimal_params, train_data, validation_data, horizon=1)
```




```r
evaluate_final_model = function(optimal_params, model, test_data, horizon) {
  learning_rate <- optimal_params[1]
  regularization <- optimal_params[2]
  lookback_window = optimal_params[4]

  processed_test <- preprocess_data(test_data, lookback_window, horizon)

  # Evaluate the performance of the LSTM model on the test data
  test_loss <- model %>% evaluate(processed_test$x, processed_test$y)

  # Generate predictions for the test data
  predicted_test_prices <- model %>% predict(processed_test$x)

  # Convert the test data and predictions to data frames
  actual_test_prices <- as.data.frame(processed_test$y)
  colnames(actual_test_prices) <- c("Actual_Price")
  predicted_test_prices <- as.data.frame(predicted_test_prices)
  colnames(predicted_test_prices) <- c("Predicted_Price")

  # Combine the actual and predicted test prices
  price_comparison <- cbind(actual_test_prices, predicted_test_prices)

  # Calculate the error between the actual and predicted test prices
  error <- mean((price_comparison$Actual_Price - price_comparison$Predicted_Price)^2)

  list(actual = actual_test_prices, predicted = predicted_test_prices)
}
```

The code above is used to evaluate the model's final performance on validation data, and create set of price predictions from the test data set.

Backtesting

In finance, backtesting is an integral aspect of developing and assessing trading strategies or models. To determine how well a trading strategy or model would do in real-world trading conditions, it may be tested in a simulated environment using historical data. The main goal of backtesting is to determine the viability of a strategy or model and make necessary adjustments to improve its performance and reduce the risk of loss.

Machine learning models, such as LSTM or other neural networks, are tested on historical data to see how well they do at making predictions about asset prices or discovering trading opportunities in artificial intelligence for finance. Backtesting is most accurate if it uses historical data that represents the assets and includes periods of varying market circumstances.

Let's pretend you've built an LSTM model to forecast the stock price of a certain firm. Here are the measures you'd take to assess the effectiveness of the model:

1. Collecting firm stock price data (both training and testing data) from the past. The testing data would be utilised for backtesting after the LSTM model was trained with the training data.
2. Optimising the LSTM's hyperparameters to reduce the model's mean squared error (MSE) or some other relevant performance metric, then train the model using the training data.
3. Once the model is trained, use it to predict stock prices for the testing data period.
4. Using the testing data, evaluate the accuracy of the model's predictions by comparing them to the actual prices. This may be done by computing metrics like Mean Squared Error (MSE), Mean Absolute Error (MAE), or Return On Investment (ROI).

5.Examine the model's predictive accuracy by analysing the performance measures to see if it is suitable for use in real-world trading scenarios. If not, refine the model, its hyperparameters, or the input features and repeat the backtesting process.

To sum up, backtesting is crucial for evaluating the efficacy of artificial intelligence (AI)-driven trading techniques and models in the financial sector. It helps ensure that a model is reliable and accurate before applying it to real-world financial decisions, ultimately reducing risk and increasing the likelihood of successful trades.

References: https://www.investopedia.com/terms/b/backtesting.asp

For our backtesting process, we take the 'predicted' and 'actual' sets of prices from our evaluation function. Then, this data is 'denormalised' to return it back to its original range ($) rather than being [0, 1]. Then, the predicted values are shifted back by 1 day in order for the prediction for 'tomorrow' to be compared to the actual price for 'today'. The most important part of the backtesting code is the backtesting loop, repeated below:

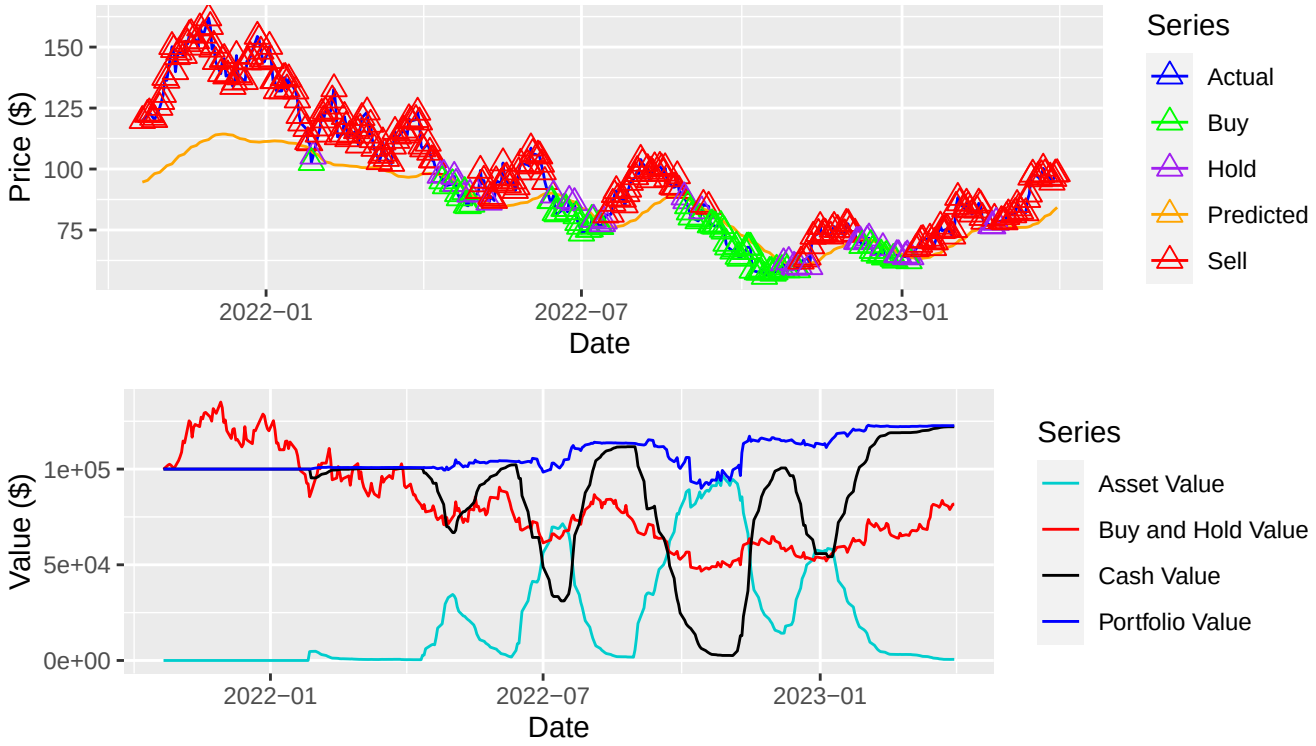
```
# Predict daily profit
predicted_profit_pct <- (shifted_predicted - actual_trimmed) / actual_trimmed
# Generate trading signals
trading_signals <- ifelse(predicted_profit_pct > buy_threshold, "Buy",
                          ifelse(predicted_profit_pct < sell_threshold, "Sell", "Hold"))

# Backtesting loop
for (i in 1:(length(signals) - 1)) {
  cash_value[i] <- cash
  buy_and_hold_value[i] = buy_and_hold_shares * actual_value[i]

  if (signals[i] == "Buy" && cash >= actual_value[i]) {
    investment_amount <- cash * abs(predicted_profit_pct[i])
    num_shares_bought <- floor(investment_amount / actual_value[i])
    num_shares <- num_shares + num_shares_bought
    cash <- cash - (num_shares_bought * actual_value[i])
  } else if (signals[i] == "Sell" && num_shares > 0) {
    sell_amount <- num_shares * actual_value[i] * abs(predicted_profit_pct[i])
    num_shares_sold <- floor(sell_amount / actual_value[i])
    num_shares <- num_shares - num_shares_sold
    cash <- cash + (num_shares_sold * actual_value[i])
  }

  asset_value[i] = num_shares * actual_value[i]
  total_portfolio_value[i] = cash + asset_value[i]
}
```

Basically, the actual/predicted values are used to predict a daily amount of profit (percent > 0) or loss (percent < 0). Then, this is used to generate a vector of trading signals; "Buy", "Hold", and "Sell". The signals are then iterated, and for each "Buy"/"Sell" signal a proportional investment is made based on the predicted profit or loss. For example, if the predicted profit for the following day is 3%, then 3% of our total "cash" is invested into the asset. If the predicted loss is 2%, then 2% of our total held asset is sold.

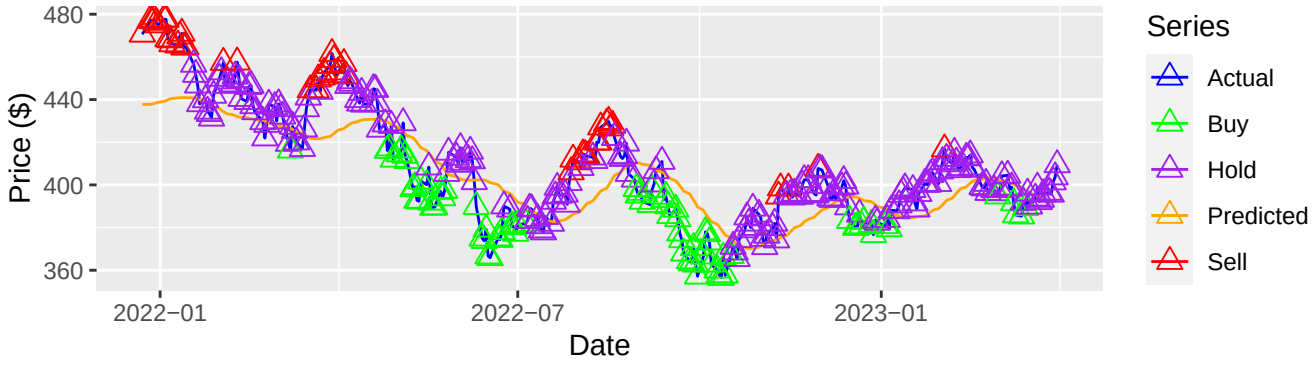
We also tried this with alternative strategies, such as only buy/selling a single share (or a constant number of shares) at each signal, or always buying/selling with a fixed proportion of our portfolio (e.g. always purchasing with 5% of our current "cash"). We found that the system that is proportional with the amount of predicted profit/loss performed the best. In the end, several time series are created to store this information so that the portfolio value over time can be monitored and plotted.

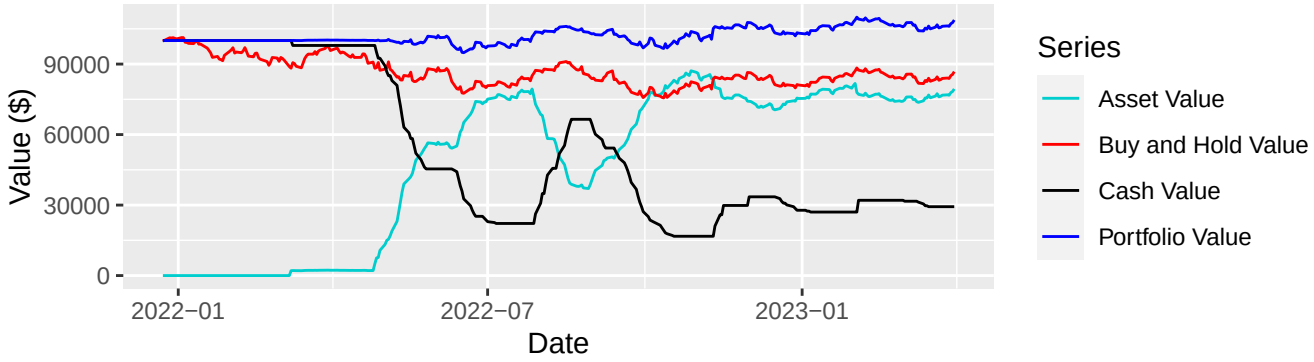
Compare both the outputs.

```
symbol <- "AMD"
start_date <- as.Date("2021-08-01")
end_date <- as.Date("2023-04-01")
stock_data_test <- getSymbols(symbol, src = "yahoo", from = start_date, to = end_date, auto.assign = FALSE)


normal_data2 = normalize_data2(prepare_stock_data(stock_data_test),
                               normal_data_training$col_mins, normal_data_training$col_ranges)
full_evaluation(optimal_params, train_model, normal_data2, normal_data_training, cash=100000)
```




```
symbol <- "SPY"
start_date <- as.Date("2021-10-01")
end_date <- as.Date("2023-04-01")
stock_data_aapl <- getSymbols(symbol, src = "yahoo", from = start_date, to = end_date, auto.assign = FALSE)

normal_data_aapl = normalize_data(prepare_stock_data(stock_data_aapl))
full_evaluation(optimal_params, train_model, normal_data_aapl$data,
                normal_data_aapl, cash=100000, buy_threshold=0.02, sell_threshold=-0.05)
```





Reasons for why the model is successful and what is it doing.

```
predicted_days = predict_real_days(optimal_params, train_model, normal_data_aapl$data,
                                   normal_data_aapl, buy_threshold=0.01, sell_threshold=-0.04)
```

```
print(predicted_days[(nrow(predicted_days) - 5):(nrow(predicted_days) - 1),])
```

```
##            Actual Predicted      Profit Signals
## 2023-03-27 191.81  181.5621 -0.05342743    Sell
## 2023-03-28 189.19  181.3123 -0.04163896    Sell
## 2023-03-29 193.88  181.0987 -0.06592356    Sell
## 2023-03-30 195.28  181.0036 -0.07310731    Sell
## 2023-03-31 207.46  181.1022 -0.12705000    Sell
```