

# 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>

## 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

Explain below plot code

In the above code chunk, 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.

In the above 2 code chunks, we are providing 2 functions for normalizing and denormalizing 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. **Normalization** 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, **Denormalization** involves reversing normalisation by rescaling the data to its original range.

```
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 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`).

## 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.

```

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(1 = regularization)) %>%
    layer_dropout(rate=0.5) %>%
    layer_lstm(units = 10, kernel_regularizer = regularizer_l2(1 = 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.

## Insert Diagram and Explain

```

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)
}

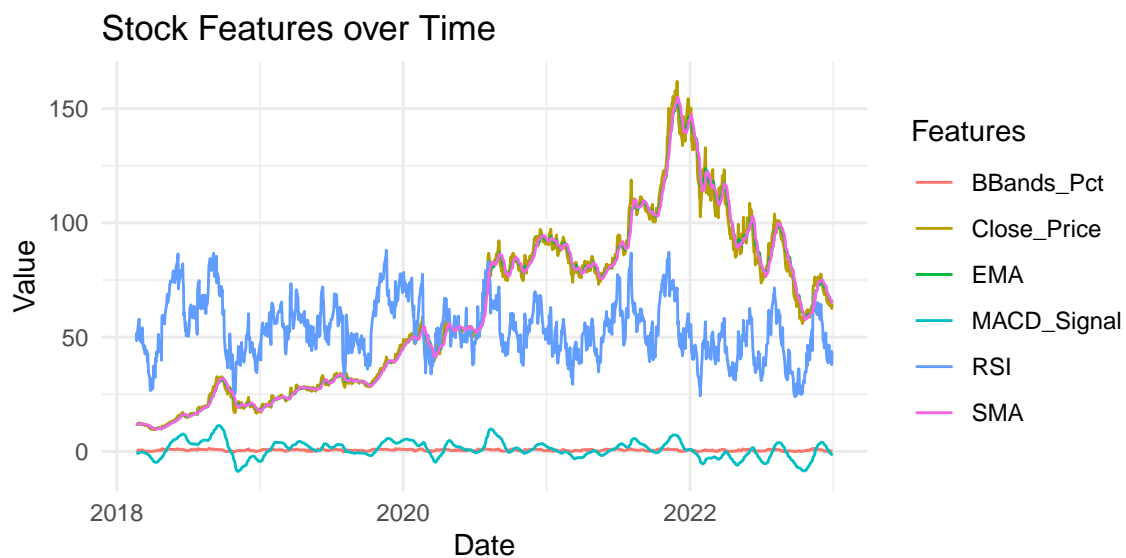
# Split the data into training, validation, and test sets

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
```

Explain the above plot

In the above code chunk, we are defining 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.

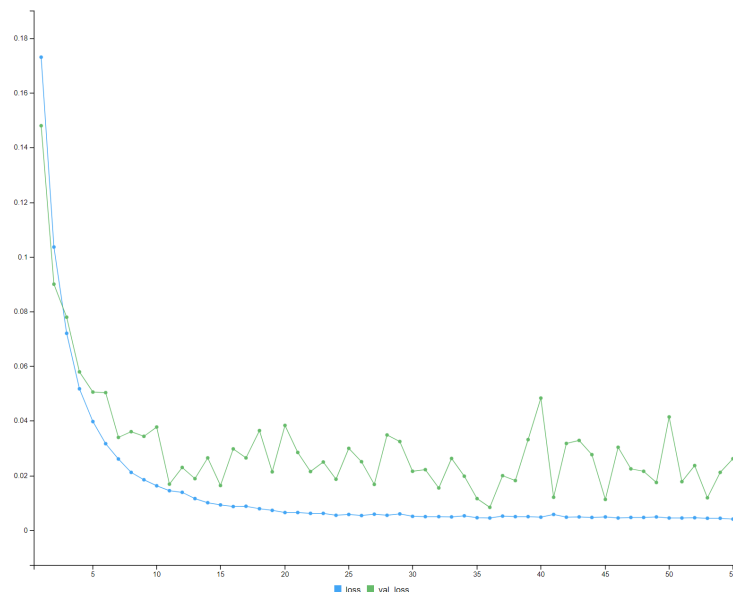
After running the PSO algorithm for 20 iterations, it found the below optimal parameters. We are putting them in here manually to avoid unnecessary time for re-computing the parameters by re-running the PSO process.

- We are getting the below hyperparameters from tensorboard.

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

-> Using the optimal hyperparameters to train the model for evaluation

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



-> evaluating the optimal model and producing the predicted values for the test set.

```
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)
}

```

## 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>

```

# 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]
}

```

Explain the above code chunk in brief.

Explain the plot generated from the code chunk above.

- Taking the trained model & predicting values and then using backtesting with a starting portfolio value

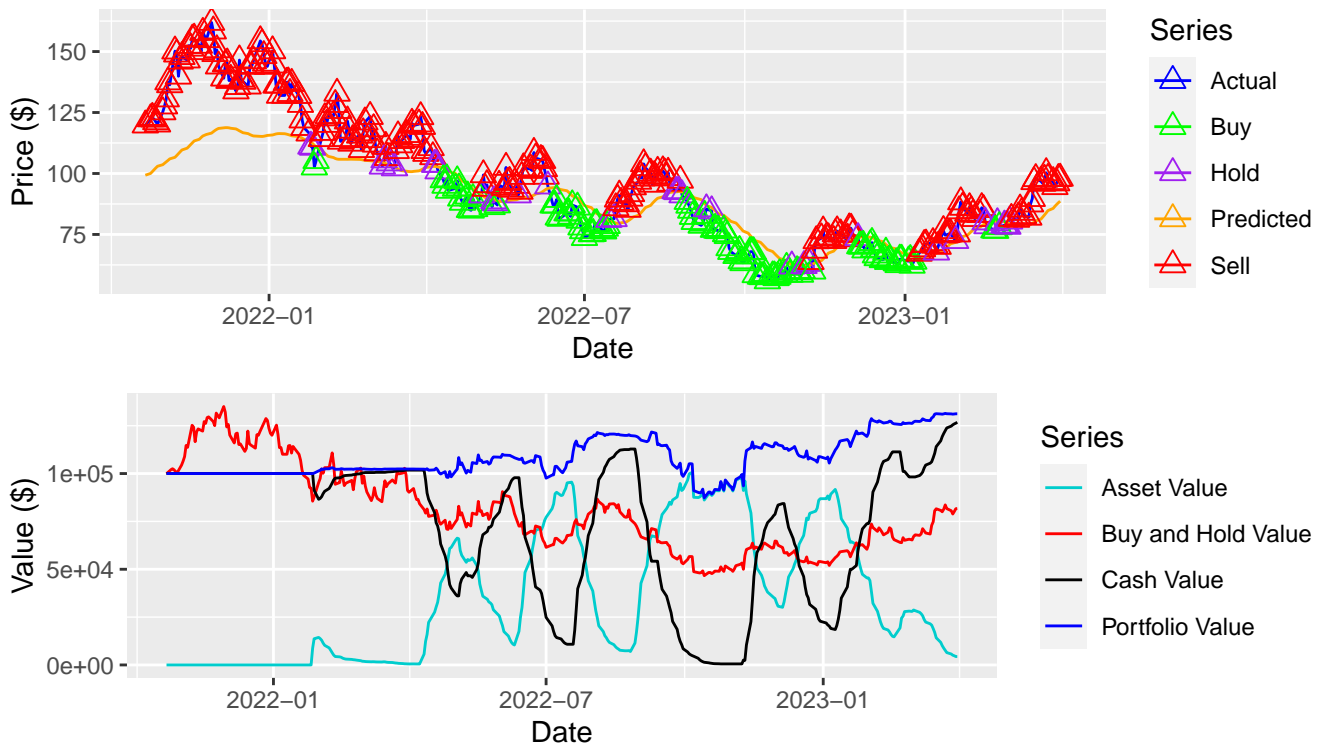
Compare both the outputs.

```

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)

```



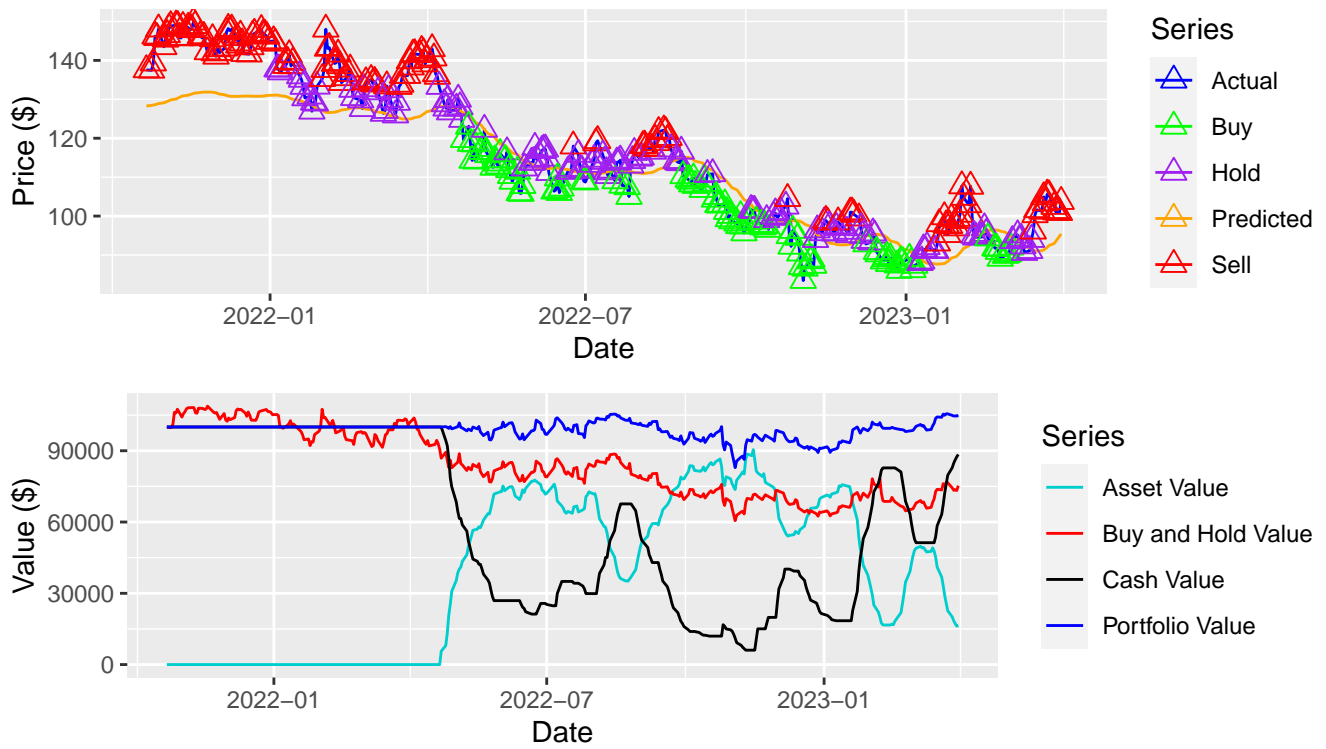
```

symbol <- "GOOGL"
start_date <- as.Date("2021-08-01")
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	185.0747	-0.03511417	Hold
## 2023-03-28	189.19	184.8211	-0.02309267	Hold
## 2023-03-29	193.88	184.6716	-0.04749536	Sell
## 2023-03-30	195.28	184.6837	-0.05426197	Sell
## 2023-03-31	207.46	184.8738	-0.10886997	Sell