STACKED ENSEMBLING AND ALGORITHMIC TRADING PROJECT

FIN40090

Group 7: Amy Qejvani, Jinal Khatri, Pranjal Kharbanda

Table of Contents

Introduction

This project is focused on forecasting returns of the FTSE 100 Index using a combination of historical data and a selection of technical indicators, using Julia as our coding language. We aim to employ different machine learning methods widely used to predict returns, and ultimately combine these methods in a stacked ensemble to evaluate the trading performance of suggested trading strategies.

The following five forecasting models were used in our analysis:

1. Linear Ridge Regression
2. Logistic Regression
3. XG Boost
4. Support Vector Machines
5. Random Forest

Our stacked ensemble will be performed using the Neural Network approach.
This report will detail our approach from data preparation through model evaluation, culminating in a trading performance analysis that includes out-of-sample testing for over 100 weeks. The codes used to conduct our analysis are provided in the appendix of the report.

Data Preparation

Index Selection and Data Acquisition

The data for the FTSE 100 index data was obtained from Yahoo Finance, spanning from April 18, 2009, onwards. Initial steps involved cleaning the data by removing any missing values and selecting relevant features, excluding open, high, low, and closing prices to focus solely on adjusted close prices. The plot of FTSE prices from the start of our dataset to date can be seen in Figure 1.

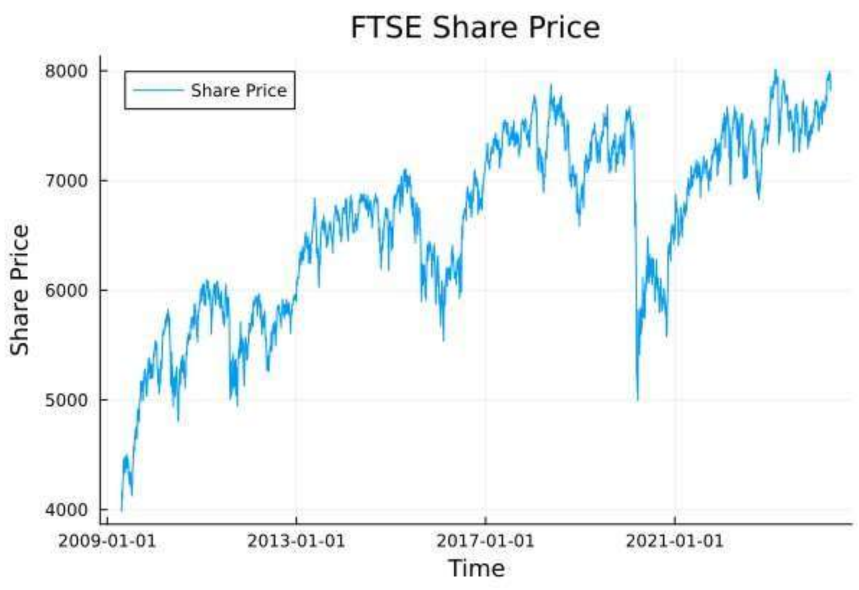


Figure 1

Technical Indicators

We incorporated several widely used technical indicators:

- **Exponential Moving Averages (EMA5 and EMA20)**: These indicators provide insights into the trend direction and momentum by averaging the prices over 5 and 20 days, respectively.
- **Moving Average Convergence Divergence (MACD)**: This indicator helps identify changes in the strength, direction, momentum, and duration of a trend in the stock's price.
- **Relative Strength Index (RSI)**: RSI is used to assess the speed and change of price movements, indicating overbought or oversold conditions.
- **Volume**: We consider the trading volume as it can validate the strength of a price move.

Models

Model 1-Linear Ridge Regression

Linear Ridge Regression is a method for fitting a linear model that aims to minimize the sum of squared residuals while also shrinking the coefficients toward zero. It achieves this by adding a penalty term to the traditional least squares objective function. This regularization helps to prevent overfitting, particularly when dealing with multicollinearity among the predictor variables. We start by determining the optimal lambda values, and then proceed to train ridge models to estimate the coefficients. These coefficients are then used collectively to evaluate the performance of out-of-sample data.

In-Sample Predictions

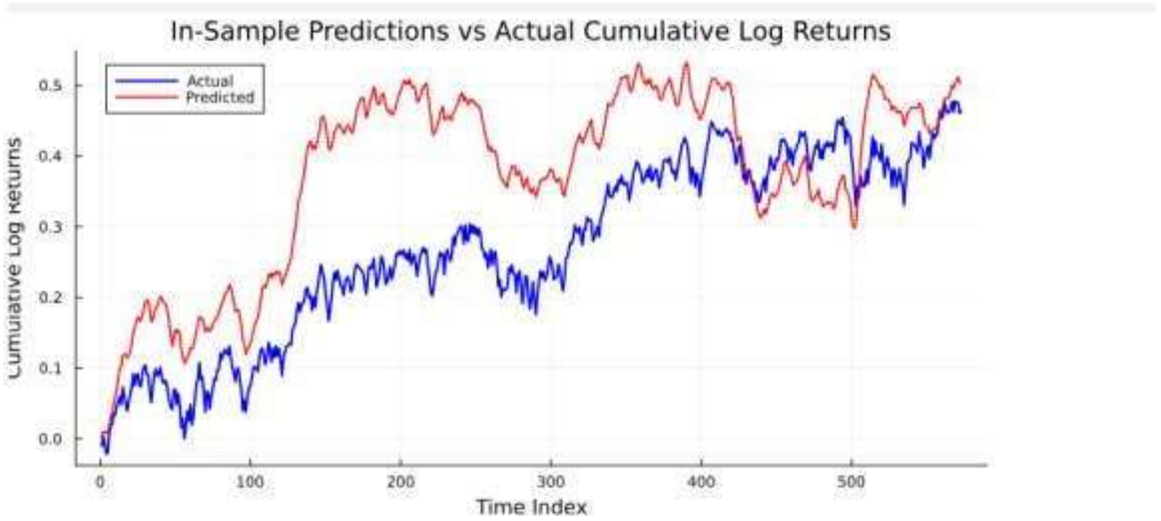


Figure 2: Performance of In-Sample predictions vs the actual data.

Sharpe Ratio	Treynor's Ratio	Directional Accuracy	Sample Accuracy
0.728	0.0084	84.97%	78.16%

Our in-sample predictions using the Linear Ridge Regression model showcase a strong alignment with the actual cumulative log returns over the observed period. The predictive trend closely mirrors the actual trend, though there are some deviations, especially in periods of high volatility.

The model's Sharpe Ratio is calculated to be 0.728, suggesting a robust risk-adjusted return above the risk-free rate. This high ratio underscores the model's effectiveness in generating excess returns for the level of volatility endured by the investor. Meanwhile, Treynor's Ratio, at a low 0.0084, indicates that the model's excess return does not scale as well with the market risk, potentially implying a less favorable performance in more volatile market conditions.

The Directional Accuracy is at 84.97%, indicating that the model has a strong capability to correctly predict the direction of market movement nearly 85% of the time. In combination with a Sample Accuracy of 78.16%, it suggests that while the model is generally reliable, it may require adjustments to refine its prediction accuracy for specific samples.

Out-Of-Sample Predictions

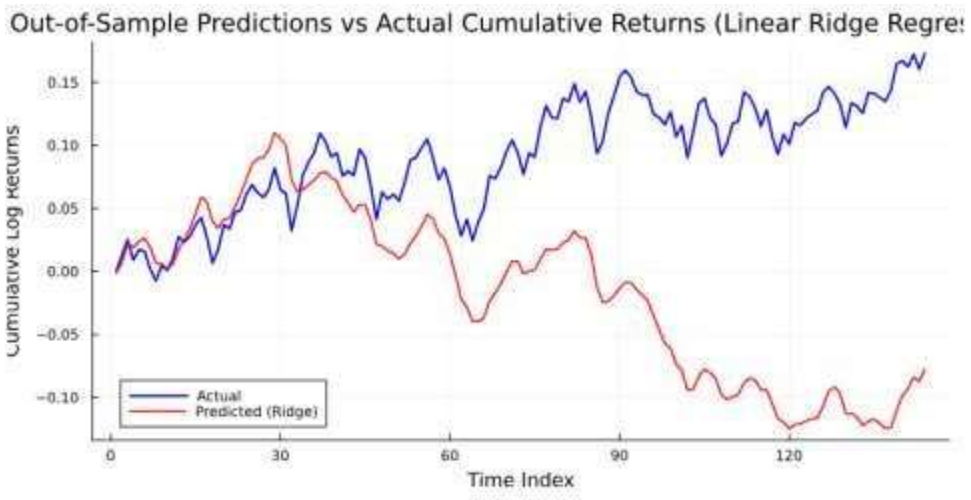


Figure 3: Out-of-sample predictions vs the Actual cumulative returns.

Sharpe Ratio	Treynor's Ratio	Directional Accuracy	Sample Accuracy
0.729	0.0077	89.33%	75.49%

In our examination of the out-of-sample predictions made by the Linear Ridge Regression model, we've observed a distinct performance profile. The graph illustrates that while the model's cumulative log returns tend to follow the general direction of the actual returns, there is a noticeable deviation in magnitude, particularly evident after the 60th time index.

The Sharpe Ratio stands at a robust 0.729, indicating that the model's predictive returns are substantially higher than the risk-free rate, suggesting a strong risk-adjusted performance. However, the Treynor's Ratio is markedly low at 0.0077, implying that the model’s performance is not as compelling when adjusted for market volatility. This disparity may point to the model's sensitivity to market risk being higher than ideal.

Notably, the model boasts a high Directional Accuracy of 89.33%, demonstrating that it predicts the correct direction of returns with high reliability. This is a significant metric for traders focusing on trend-following strategies. Sample Accuracy is slightly lower at 75.49%, which indicates that while the model is generally reliable, there are instances where it does not perfectly capture the market's movements. The analysis of these metrics reveals that our Linear Ridge Regression model is adept at forecasting the direction of market movements with commendable accuracy.

Model 2-Logistic Regression

Logistic regression is a statistical method used for binary classification tasks, where the outcome variable is categorical and has two possible outcomes. It models the probability that a given input belongs to a particular category using the logistic function. By fitting a logistic curve to the data, it estimates the likelihood of each outcome and makes predictions based on a threshold value.

<u>In Sample Predictions</u>

We calculated in-sample predictions by using the trained model to predict the probabilities of class membership for each observation in the training dataset. These probabilities were transformed into class predictions by applying a threshold of 0.65, where observations with predicted probabilities above the threshold are assigned to one class (e.g., positive or negative), and those below are assigned to the other class. This process allows us to evaluate how well the logistic regression model fits the training data by comparing the predicted class labels to the actual class labels.

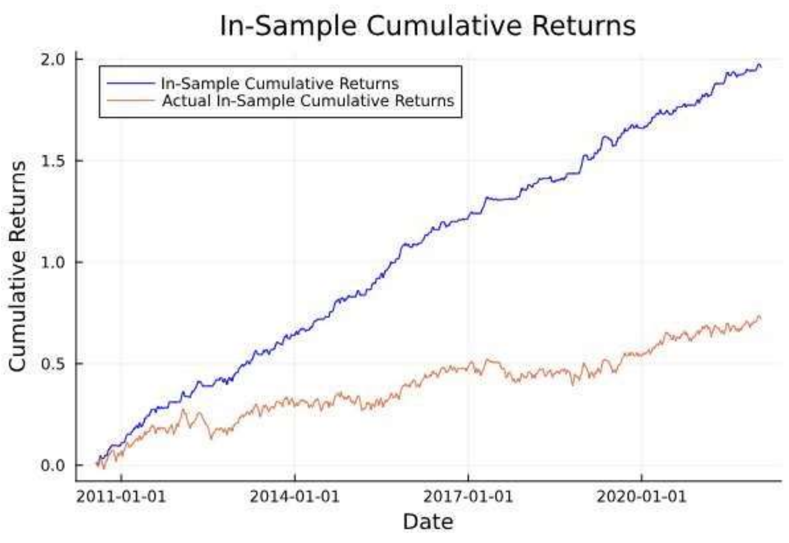


Figure 4: The In-Sample cumulative returns with the actual data.

Sharpe Ratio	Treynor's Ratio	Sample Accuracy
2.486	0.0235	67.22%

The logistic regression model demonstrates promising performance, with an in-sample Sharpe ratio of 2.734 and an accuracy of 64.56%. These results suggest that the model effectively captures underlying patterns in the data, providing valuable insights for classification tasks.

Out-of-Sample Predictions

Out-of-sample predictions in logistic regression involve using the trained model to forecast outcomes on data not previously seen during training. We achieved this by applying the learned coefficients to new input features, producing probability estimates for each class, thereby enabling the classification of unseen data points. In the below figure 4 we have demonstrated the performance of out-of-sample data vs the actual.

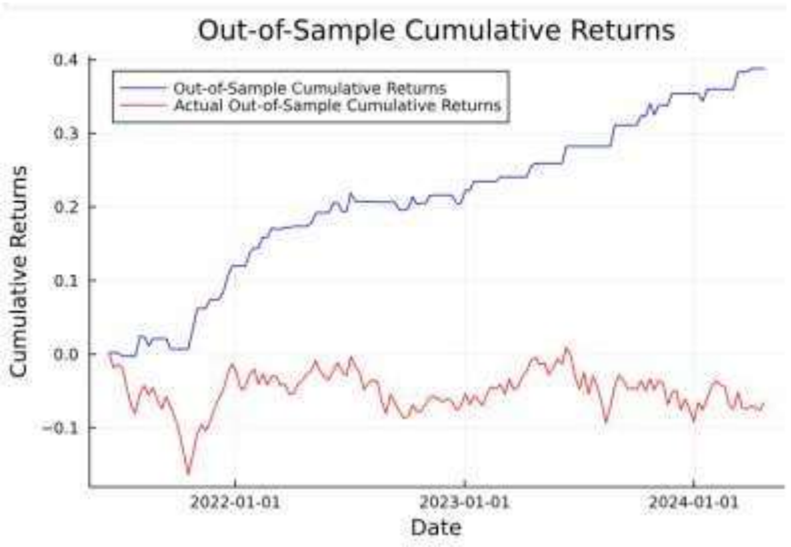


Figure 5: Out-of-Sample Cumulative Returns

Sharpe Ratio	Treynor's Ratio	Sample Accuracy
1.402	0.0.015	63.33%

Our Logistic Regression model's performance on out-of-sample data has presented a mixed outcome. The model's predicted cumulative returns exhibit a consistent uptrend over the observed period but fail to capture the volatility inherent in the actual out-of-sample cumulative returns. Notably, the model does not react to the downturns and upturns experienced in the actual market, as shown by the divergence between the predicted and actual return paths.

The Sharpe Ratio is exceptional at 1.402, which indicates that the model's returns are well above the risk-free rate, adjusted for volatility. This high ratio reflects strong risk-adjusted returns, albeit with the caveat that this may be overestimated due to the lack of volatility in predicted returns.

Conversely, Treynor's Ratio is exceedingly low at 0.0015, implying that when market risk is considered, the model's ability to generate excess return is minimal. This could be a critical concern for investors for whom market volatility plays a significant role in investment decisions.

The Sample Accuracy is at 63.33%, which suggests that just over half of the model's predictions match the actual market direction.

Model-3: XG Boost

XGBoost, short for Extreme Gradient Boosting, is a powerful machine learning algorithm known for its efficiency and accuracy in classification and regression tasks. It constructs a series of decision trees sequentially, optimizing a predefined objective function at each step to minimize loss.

In-Sample Predictions

The in-sample predictions were generated by aggregating the predictions from each tree in the ensemble. Each tree contributes a weighted prediction based on its output, with weights determined by the tree's performance on the training data. We then got the final prediction, which is the sum of these weighted tree predictions, providing a comprehensive estimate of the target variable for the training dataset.

Our XGBoost model's in-sample analysis reveals a generally positive alignment between the predicted and actual cumulative returns across an extended timeframe. The model tracks the rising trend of the actual returns with a notable degree of synchronicity, but it does exhibit some gaps, particularly around periods of market volatility.

Sharpe Ratio	Treynor's Ratio	Sample Accuracy
0.501	0.0072	54.07%

The Sharpe Ratio for the in-sample period is 0.501, signifying that the model delivers a modest excess return over the risk-free rate when considering the volatility of returns. This figure points to a reasonable level of efficiency in the model's performance, balancing return against inherent investment risk.

However, Treynor's Ratio is substantially low at 0.0072, suggesting that when adjusting for market risk, the model's performance may not be as strong. This could indicate that the model's predictions are not as effective in a broader market context, where systemic risk plays a significant role in asset price movements. The Sample Accuracy stands at 54.07%, indicating that the model correctly predicts the direction of returns slightly more than half the time.

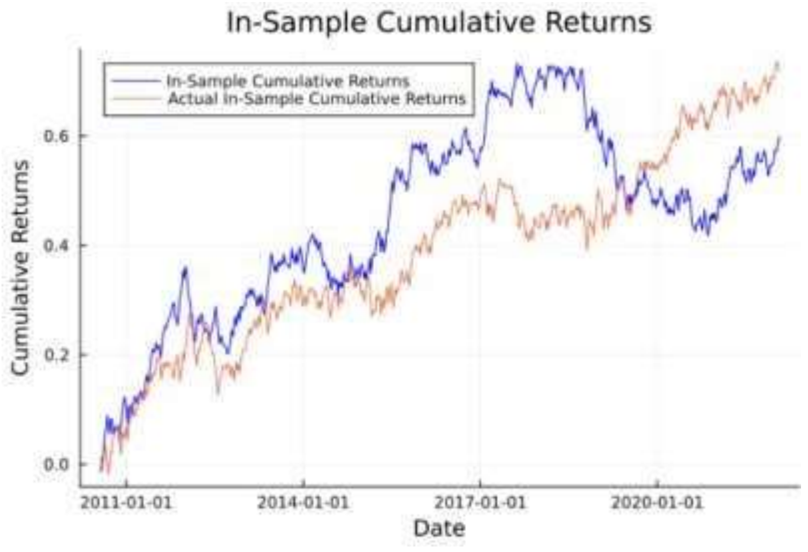


Figure 6: In-Sample Cumulative Returns

Out-of-Sample Predictions

Upon deploying our XGBoost model to generate out-of-sample predictions, we observed a marked divergence between the predicted and actual cumulative returns. The model's predictions tend to understate the peaks and overstate the troughs when compared to the actual returns, as evidenced by the pattern depicted in the provided graph.

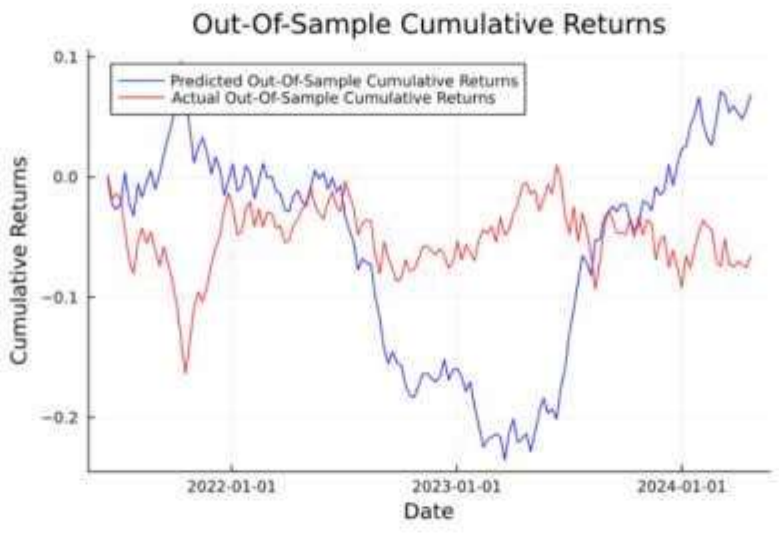


Figure 7: Out-of-Sample Cumulative Returns

Sharpe Ratio	Treynor's Ratio	Sample Accuracy
0.227	0.0033	51.33%

The Sharpe Ratio, a measure of risk-adjusted return, stands at 0.227. This indicates that the model's predictive power, although positive, offers only a slight improvement over a risk-free investment. The Treynor's Ratio is quite low at 0.0033, suggesting that the model's excess returns are not substantially higher when the market volatility is accounted for.

Sample Accuracy is noted at 51.33%, hovering just above the threshold for random chance. This suggests that while the model can predict the direction of market movements slightly better than a coin flip, its reliability is modest and may not instill high confidence for investment decisions based on these predictions alone.

Model 4: Random Forest

Random Forest is an ensemble learning method that constructs a multitude of decision trees during training and outputs the mode of the classes or the mean prediction of the individual trees. To model with Random Forest, we selected a subset of features at random for each tree and aggregated the predictions of multiple trees to improve accuracy and generalizability.

<u>In-Sample Predictions</u>

To calculate in-sample predictions for Random Forest, first we trained the model on the training dataset using the fit method. Then, we used the prediction method to generate predictions for the same dataset. Our in-sample analysis using the Random Forest algorithm reveals a favorable uptrend in the forecast cumulative log returns, closely tracking the ascent of the actual returns. The model demonstrates a robust capacity to follow the general market trend within the training dataset.

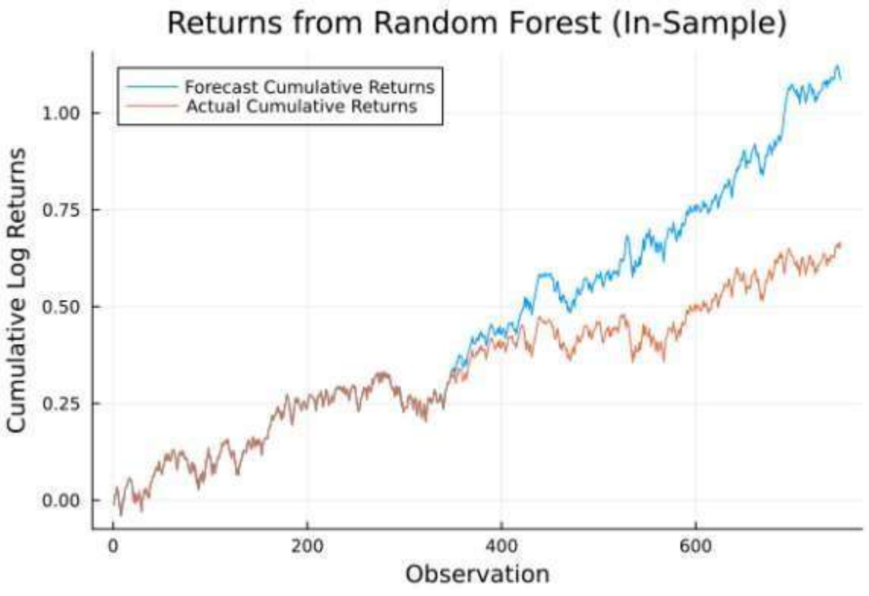


Figure 8: Returns from Random Forest (In-Sample)

Sharpe Ratio	Treynor's Ratio
0.730	0.0095

With a Sharpe Ratio of 0.730, the model yields a commendable risk-adjusted return, outpacing a risk-free investment and reflecting efficiency in its return generation process relative to the volatility experienced. However, Treynor's Ratio stands at a modest 0.0095, suggesting that when market risk is factored in, the model's performance in generating excess returns is less impressive.

It's crucial to note that while the model shows prowess in individual security selection and trend following, as evidenced by the Sharpe Ratio, its lower Treynor's Ratio underscores the need to consider market-wide movements in portfolio management decisions.

<u>Out-Of-Sample Predictions</u>

To calculate out-of-sample predictions using a Random Forest model, we first divided the dataset into a training set and a test set. Then we trained the Random Forest on the training set, and then used the trained model to predict the target variable for the unseen test set data.

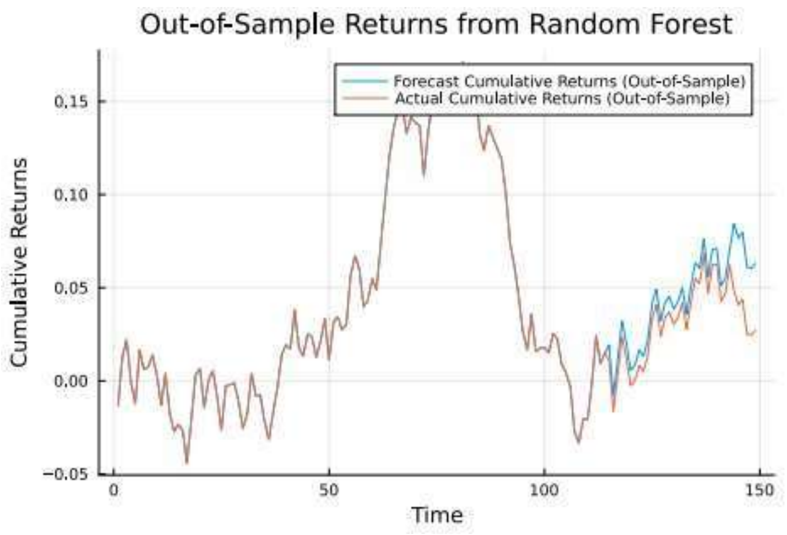


Figure 9: Out-of-Sample Returns from Random Forest

In our out-of-sample testing, the Random Forest model demonstrated an ability to generally follow the trend of the actual market returns. While the forecasted cumulative returns fluctuated alongside the actual returns, there were periods where the model's predictions diverged, indicating potential areas for improvement in the model's ability to handle market volatility and sudden shifts in trend.

Sharpe Ratio	Treynor's Ratio
0.216	0.437

The model's performance metrics reveal a moderate Sharpe Ratio of 0.216, which suggests that the returns, when adjusted for total risk, are above average but not exceptional. In contrast, the Treynor's Ratio of 0.437 is quite robust, indicating that the model performs significantly better when the returns are adjusted for systematic market risk.

Model 5: SVM

Support Vector Machine (SVM) is a powerful algorithm for classification and regression tasks, finding optimal hyperplanes to separate classes in feature space. To model with SVM, we chose a kernel function based on data characteristics. Then we trained the model on the labeled data, adjusting parameters like regularization (C) and kernel parameters.

In-Sample Predictions

Our Support Vector Machine (SVM) model, when assessed on in-sample data, has shown an exceptionally high level of predictive accuracy. The forecast cumulative log returns generated by the SVM markedly outperform the actual cumulative returns throughout the observation period. This is evidenced by the pronounced gap between the forecast and actual returns on the graph.

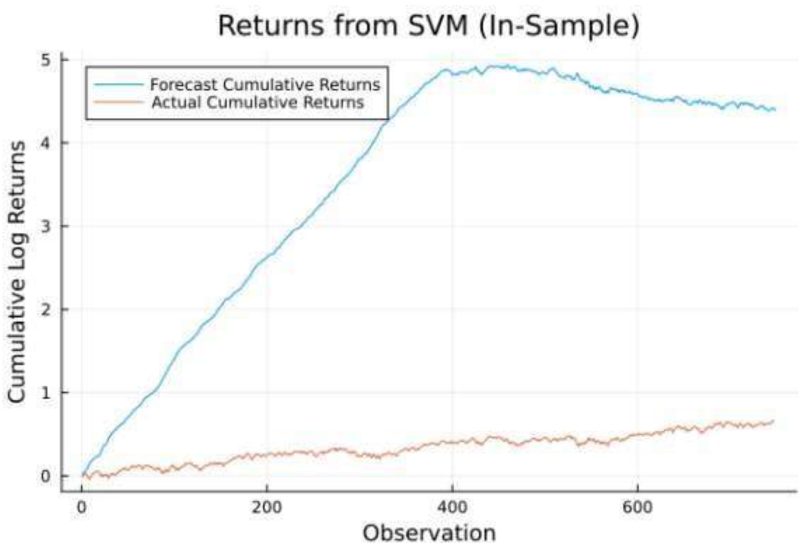


Figure 10: Returns from SVM (In-Sample)

Sharpe Ratio	Treynor's Ratio
3.222	0.304

The model's Sharpe Ratio is reported at an impressive 3.222, indicating that the returns are significantly higher than the risk-free rate even after adjusting for volatility. This suggests that the model is highly effective at generating returns on a risk-adjusted basis within the training dataset.

However, Treynor's Ratio stands at 0.304, which, while positive, suggests that the returns, when adjusted for systematic risk as measured by the market beta, are not as high. This might imply that while individual predictions are strong, the model may not be as effective in a diversified market scenario.

Out-Of-Sample Predictions

The SVM model's forecast of out-of-sample returns shows a conservative trend with less volatility compared to actual returns, suggesting a cautious strategy that might suit risk-averse investors. With a Sharpe ratio of 0.241, the risk-adjusted returns are low, indicating modest performance over a risk-free rate. The Treynor's ratio at 1.744 shows better returns adjusted for market risk, implying effectiveness in diversifying systemic risk. These metrics suggest the model is more attuned to minimizing risk than capturing high gains.

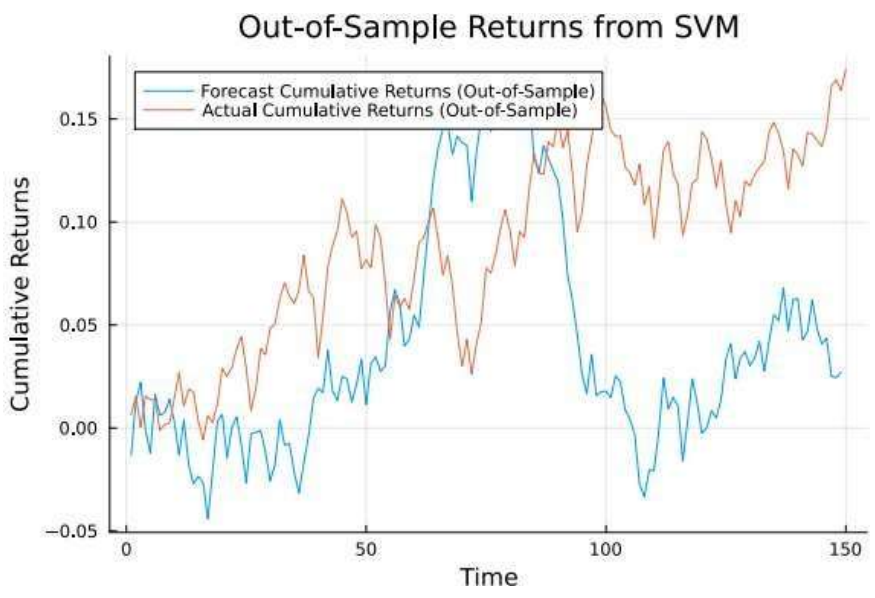


Figure 11: Out-of-Sample Returns from SVM

Sharpe Ratio	Treynor's Ratio
0.241	1.744

Ensembling

Neural Network (Prep work for ensemble)

In our analysis, we observed that the neural network model demonstrated a capability to mirror the general direction of market trends both in in-sample and out-of-sample data, with some discrepancies in capturing specific peaks and troughs of actual cumulative returns.

In Sample Predictions

When applied to in-sample data, the model showed a pattern closely aligned with actual returns but tended to underperform during periods of higher market gains and overperform during downturns. This suggests that while the model can be used to understand broader market movements, its predictions should be approached with caution for short-term investment strategies.

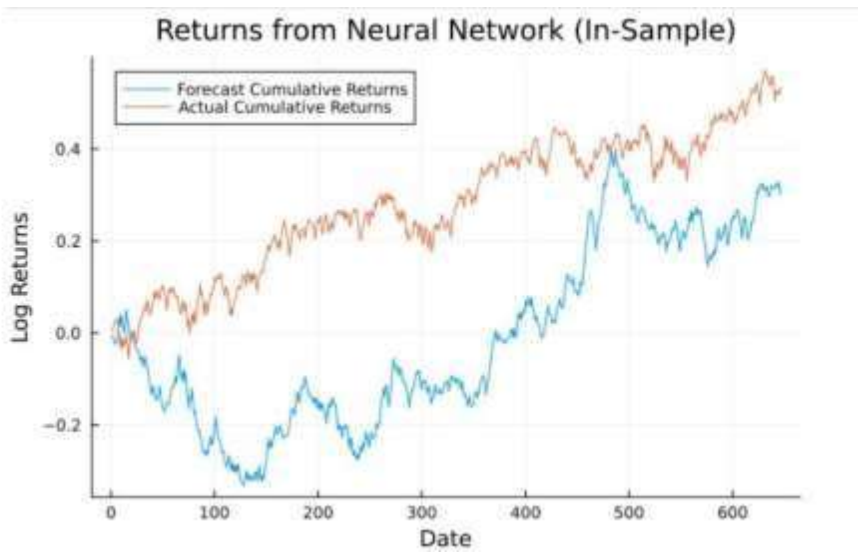


Figure 12: Returns from Neural Network (In-Sample)

Out-of-Sample Predictions

Moving to out-of-sample predictions, the model maintained its overall trend-following characteristic but exhibited notable deviations from actual returns. Interestingly, the out-of-sample Sharpe ratio is higher at 0.295 compared to the in-sample ratio of 0.235, indicating potentially better risk-adjusted returns when dealing with new data.

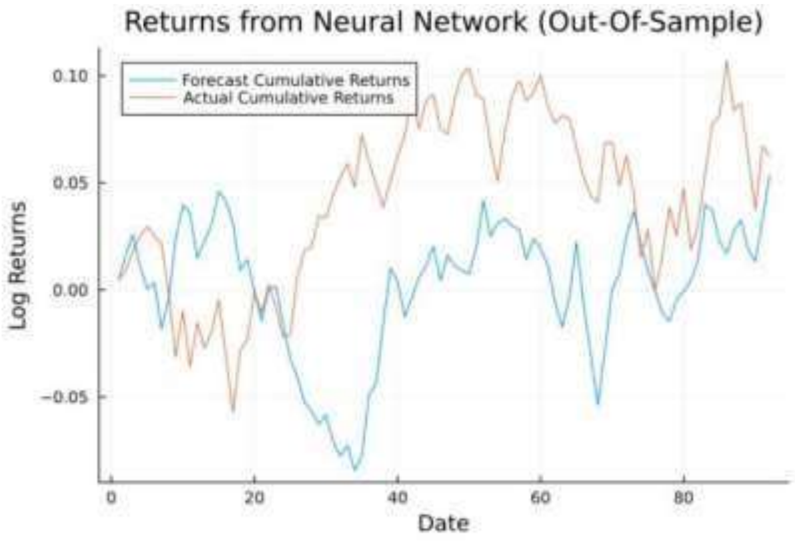


Figure 13: Returns from NN (Out-of-Sample)

Sharpe Ratio (in sample)	Sharpe ratio (out of sample)
0.235	0.295

Neural Network- Ensembling

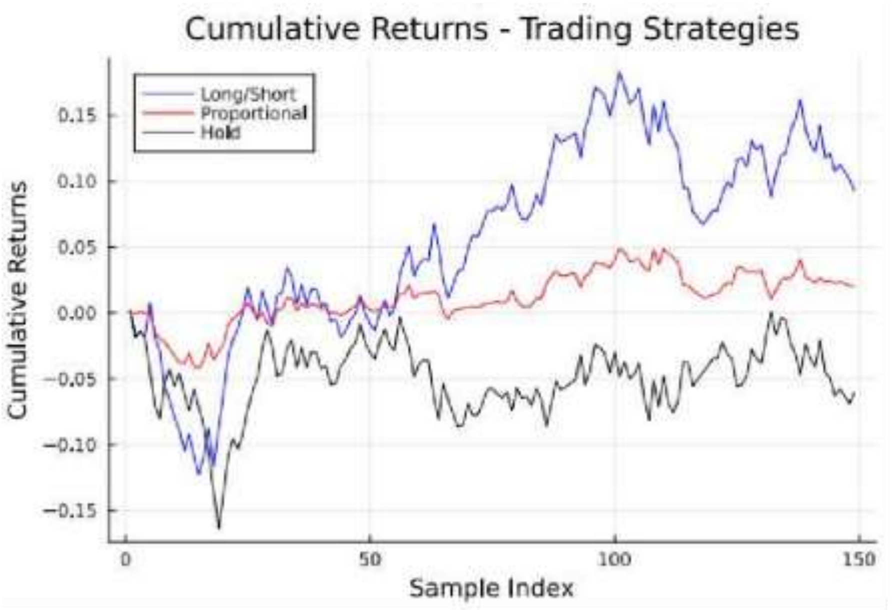


Figure 14: Cumulative Returns from Trading Strategies

Trading Strategies

In this section we present the findings from the final Stacked Ensemble neural network model, focusing on two distinct trading strategies: Long/Short investing and Proportional investing, using out-of-sample testing data

Long/Short Investing:

Our Long/Short strategy involves taking long positions in assets that are expected to increase in value and short positions in assets predicted to decline, as indicated by our model’s forecasts. This approach is designed to exploit the full range of the model's predictive abilities, regardless of market direction.

Sharpe Ratio	Treynor's Ratio
0.309	0.309

Our Long/Short strategy delivered promising results, as demonstrated by the cumulative returns plot. This approach capitalizes on both positive and negative forecasted trends, aiming to profit from the accuracy of the model's predictions on price movements in either direction. The strategy yielded a Sharpe Ratio and Treynor's Ratio of 0.309, indicating a moderate level of risk-adjusted return and market risk efficiency. This reflects a balanced performance, with the strategy successfully navigating through various market conditions and capturing profit opportunities on correctly forecasted trends.

<u>Proportional Investing</u>:

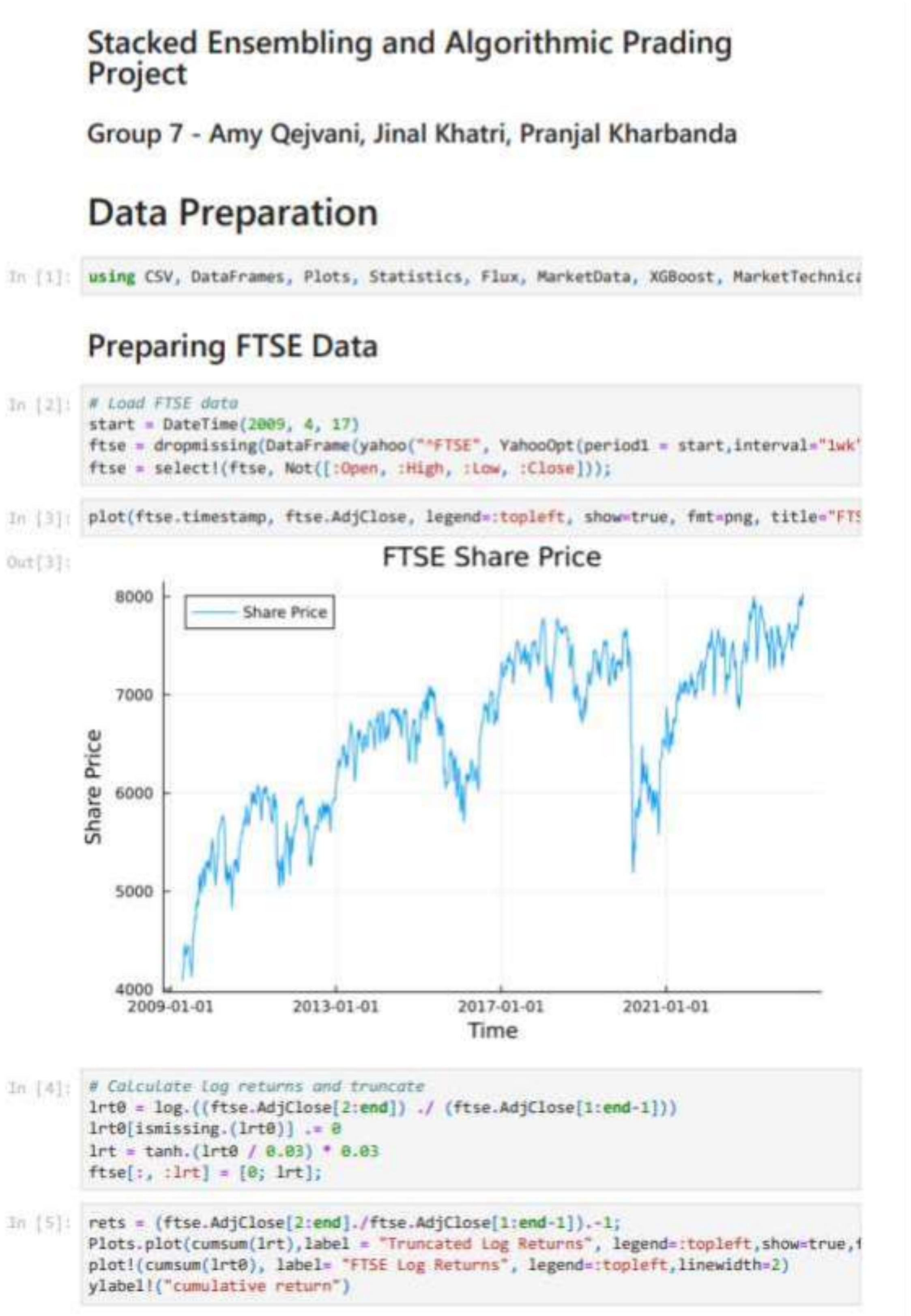
The Proportional strategy adjusts the size of the investment based on the confidence level of the model’s predictions. This more conservative approach means that greater capital is allocated to trades with higher forecasted confidence, and less to those with lower confidence.

Sharpe Ratio	Treynor's Ratio
0.168	0.168

The Proportional investing strategy, which scales positions based on the confidence level of the forecasts, showed a more conservative performance profile in comparison. The Sharpe and Treynor Ratios for this strategy were both 0.168, pointing to a lower return per unit of risk taken and less efficient adjustment for market risk than the Long/Short strategy.

Overall, the Long/Short strategy seems to be more robust in the given out-of-sample period, providing a stronger risk-adjusted return than the Proportional strategy within our Stacked Ensemble model's framework.

APPENDIX



Stacked Ensembling and Algorithmic Prading Project

Group 7 - Amy Qejvani, Jinal Khatri, Pranjal Kharbanda

Data Preparation

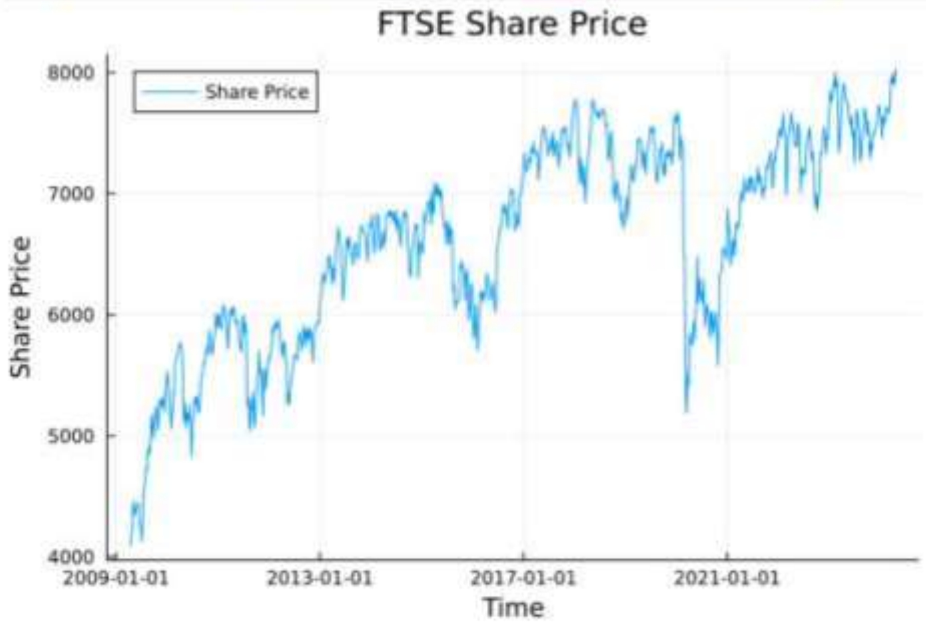
```
In [1]: using CSV, DataFrames, Plots, Statistics, Flux, MarketData, XGBoost, MarketTechnic
```

Preparing FTSE Data

```
In [2]: # Load FTSE data
        start = DateTime(2009, 4, 17)
        ftse = dropmissing(DataFrame(yahoo("^FTSE", YahooOpt(period1 = start,interval="1wk'
        ftse = select!(ftse, Not([:Open, :High, :Low, :Close]));
```

```
In [3]: plot(ftse.timestamp, ftse.AdjClose, legend=:topleft, show=true, fmt=png, title="FTS
```

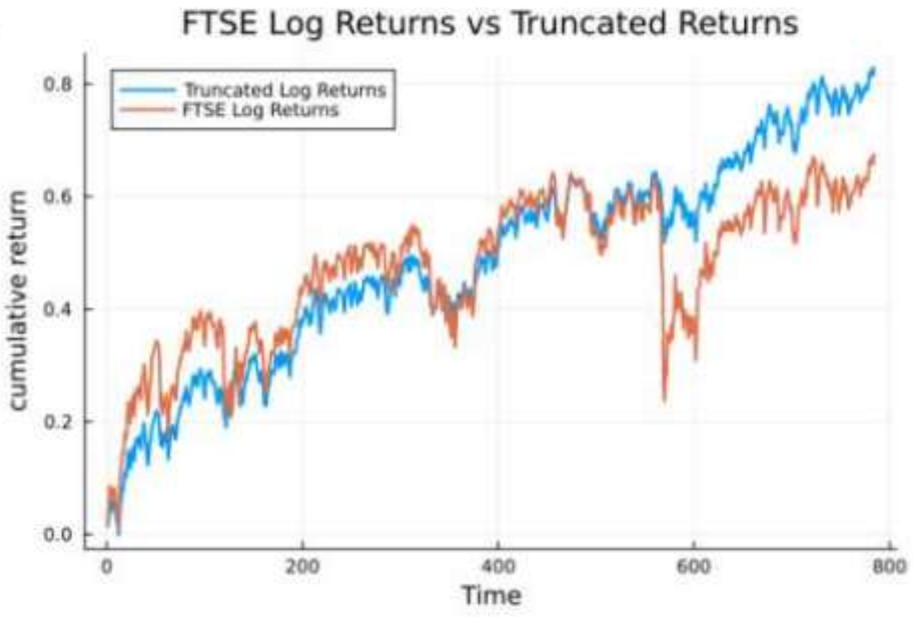
Out[3]:



```
In [4]: # Calculate log returns and truncate
        lrt0 = log.((ftse.AdjClose[2:end]) ./ (ftse.AdjClose[1:end-1]))
        lrt0[ismissing.(lrt0)] .= 0
        lrt = tanh.(lrt0 / 0.03) * 0.03
        ftse[:, :lrt] = [0; lrt];
```

```
In [5]: rets = (ftse.AdjClose[2:end]./ftse.AdjClose[1:end-1]).-1;
        Plots.plot(cumsum(lrt),label = "Truncated Log Returns", legend=:topleft,show=true,
        plot!(cumsum(lrt0), label= "FTSE Log Returns", legend=:topleft,linewidth=2)
        ylabel!("cumulative return")
```

Out[5]:



In [6]:
```julia
ta=TimeArray(ftse,timestamp=:timestamp);  # convert DataFrame to TimeArray requirec
```

Adding Technical Indicators

In [7]:
```julia
# Prepare TimeArray and indicators
ta = TimeArray(ftse, timestamp=:timestamp)
ema5 = ema(ta[:AdjClose], 5)
ema20 = ema(ta[:AdjClose], 20)
macd1 = macd(ta[:AdjClose], 12, 26, 9)
rsi1 = rsi(ta[:AdjClose]);
```

In [8]:
```julia
# Truncate indicators to align starts
#ema5 = ema5[30:end]
#ema20 = ema20[15:end]
#macd1 = macd1[34:end]
#rsi1 = rsi1[20:end];
```

In [9]:
```julia
ftse2 = ftse[1:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), DataF
```

In [10]:
```julia
ftse2 = ftse[34:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), DataF
DataFrames.rename!(ftse2, :AdjClose_ema_5 => :EMA5)
DataFrames.rename!(ftse2, :AdjClose_ema_20 => :EMA20)
DataFrames.rename!(ftse2, :AdjClose_rsi_14 => :RSI);
```

In [11]:
```julia
n_lags = 12

# Initialize an empty DataFrame for lagged return features
lags = DataFrame()

for lag in 1:n_lags
    lags[!, Symbol("Lag_", lag)] = [zeros(lag); lrt[1:end-lag]]
end
```

In [14]:
```
y = lags.lrt;
```

In [15]:
```
# Filter days with no trading
x = Matrix(lags)
x = x[y .!= 0, :]  # Remove rows where trading is zero as per 'y'
y = y[y .!= 0]     # Also filter 'y' to match 'x'

# Check the final sizes
println("Rows in x: ", size(x, 1))
println("Elements in y: ", length(y))
```

```
Rows in x: 751
Elements in y: 751
```

In [16]:
```
dates = ftse.timestamp
dates = dates[68:end];
```

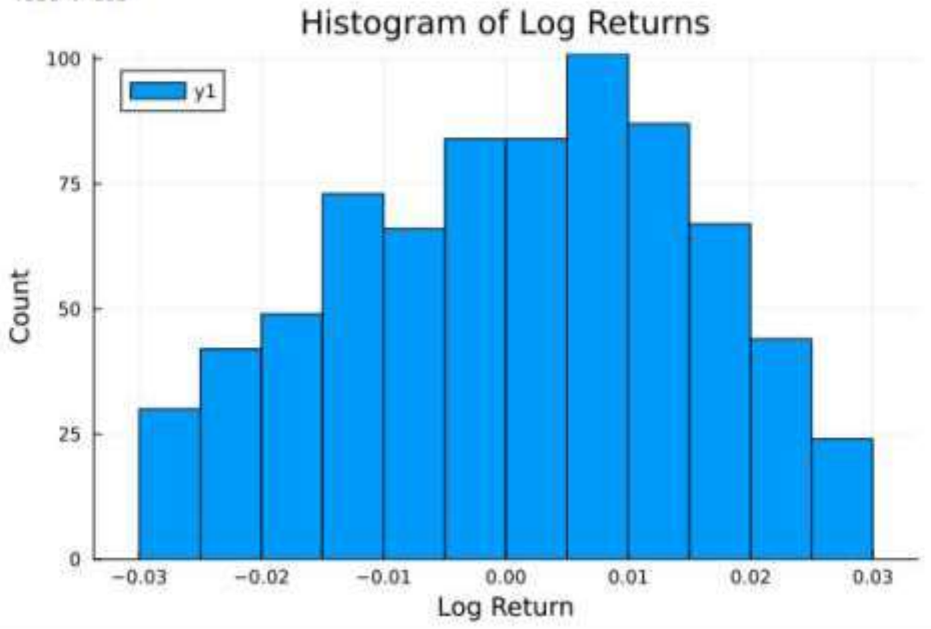
In [17]:
```
x=convert.(Float32,x);  #  This saves memory and makes computation much faster!!!
y=convert.(Float32,y);
ind_all=1:length(y)
ind_te=5:5:length(y)
ind_tr=ind_all[.!in.(ind_all,[ind_te])];
```

In [18]:
```
# Calculate the length of in-sample and out-of-sample periods
ilength = length(ind_tr)
olength = length(ind_te)

# Extract dates for in-sample and out-of-sample periods
idates = dates[1:ilength]
odates = dates[end-olength+1:end];
```

In [19]:
```
print("Full : ",length(ind_all),"\nTrain : ",length(ind_tr),"\nTest : ",length(ind_
histogram(y,legend=:topleft,fmt=png,title="Histogram of Log Returns",xlabel="Log Re
```

```
Full : 751
Train : 601
Test : 150
```

Out[19]:



5-Fold Training

In [20]:
```
tr_1 = 1:5:length(ind_tr)
tr_2 = 2:5:length(ind_tr)
tr_3 = 3:5:length(ind_tr)
tr_4 = 4:5:length(ind_tr)
tr_5 = 5:5:length(ind_tr)
```

Out[20]:
```
5:5:600
```

In [21]:
```
kfold_1 = vcat(tr_1,tr_2,tr_3,tr_4);
kfold_1t = vcat(tr_5);
kfold_2 = vcat(tr_1,tr_2,tr_4,tr_5);
kfold_2t = vcat(tr_3);
kfold_3 = vcat(tr_1,tr_2,tr_3,tr_5);
kfold_3t = vcat(tr_4);
kfold_4 = vcat(tr_1,tr_3,tr_4,tr_5);
kfold_4t = vcat(tr_2);
kfold_5 = vcat(tr_2,tr_3,tr_4,tr_5);
kfold_5t = vcat(tr_1);
```

Model 1 - Linear Ridge Regression

Classifying Predictors and Target

In [22]:
```
# First, ensure that 'lags' DataFrame includes the technical indicators
for col in names(tech_inds)
    lags[!, col] = tech_inds[!, col]
end

X = Matrix(lags)
y = ftse2[:, :lrt]

# To ensure that 'y' only includes days where trading occurred (non-zero log return
is_trading_day = y .!= 0
X = X[is_trading_day, :]
y = y[is_trading_day]

# Convert the data to Float32 if necessary (for efficiency and memory saving)
X = convert.(Float32, X)
y = convert.(Float32, y);
```

Training the model

In [23]:
```
# Split the data again and retrain the model.
train_size = floor(Int, size(X, 1) * 0.8)
X_train = X[1:train_size, :]
X_test = X[(train_size + 1):end, :]
y_train = y[1:train_size]
y_test = y[(train_size + 1):end];
```

In [24]:
```
using LinearAlgebra

# Add a column of ones to X_train for the intercept term
X_train_with_intercept = hcat(ones(size(X_train, 1)), X_train)
```

```
# Define the regularization strength (Lambda)
lambda = 1.0  # This is just an example value

# Compute the ridge regression coefficients manually using the normal equation with
ridge_coef = inv(X_train_with_intercept' * X_train_with_intercept + lambda * I) *
```

Running the Ridge Regression

In [25]:
```
wts = ones(size(ftse2[!, :lrt])) * lambda

ridge_model = glm(@formula(lrt ~ EMA5 + EMA20 + RSI), ftse2, Normal(), IdentityLink
```

Out[25]:
```
StatsModels.TableRegressionModel{GeneralizedLinearModel{GLM.GlmResp{Vector{Float6
4}, Normal{Float64}, IdentityLink}, GLM.DensePredChol{Float64, CholeskyPivoted{Flo
at64, Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

lrt ~ 1 + EMA5 + EMA20 + RSI

Coefficients:
──────────────────────────────────────────────────────────────────────────────
                   Coef.  Std. Error       z  Pr(>|z|)    Lower 95%    Upper 95%
──────────────────────────────────────────────────────────────────────────────
(Intercept)  -0.0564792   0.0047406   -11.91    <1e-31  -0.0657706   -0.0471878
EMA5         -6.35724e-5  4.10755e-6  -15.48    <1e-53  -7.1623e-5   -5.55217e-5
EMA20         6.38649e-5  4.18023e-6   15.28    <1e-51   5.56718e-5   7.2058e-5
RSI           0.00107124  4.89542e-5   21.88    <1e-99   0.000975287  0.00116718
──────────────────────────────────────────────────────────────────────────────
```

Splitting the Data

In [26]:
```
# Determine the size of the training set as 80% of the total data
train_size = floor(Int, size(X, 1) * 0.8)

# Split the predictors (X) into training and testing sets based on the calculated s
X_train = X[1:train_size, :]
X_test = X[(train_size + 1):end, :]

# Split the target (y) into training and testing sets in the same manner
y_train = y[1:train_size]
y_test = y[(train_size + 1):end]

# Confirm the dimensions
println("Size of X_train: ", size(X_train))
println("Size of X_test: ", size(X_test))
println("Size of y_train: ", length(y_train))
println("Size of y_test: ", length(y_test))

# Check if the number of columns in X_train matches X_test
if size(X_train, 2) == size(X_test, 2)
    println("Feature dimensions match.")
else
    println("Feature dimensions do not match.")
end
```

```
Size of X_train: (600, 20)
Size of X_test: (151, 20)
Size of y_train: 600
Size of y_test: 151
Feature dimensions match.
```

Checking the dimensions of the training and test sets

In [27]:
```
# Calculate means and standard deviations from the training set
means = mean(X_train, dims=1)
stds = std(X_train, dims=1)

# Normalize the training data
X_train_norm = (X_train .- means) ./ stds

# Normalize the test data using the same means and standard deviations
X_test_norm = (X_test .- means) ./ stds;
```

In [28]:
```
println("Size of X_train after preprocessing: ", size(X_train_norm))
println("Size of X_test after preprocessing: ", size(X_test_norm))
```

```
Size of X_train after preprocessing: (600, 20)
Size of X_test after preprocessing: (151, 20)
```

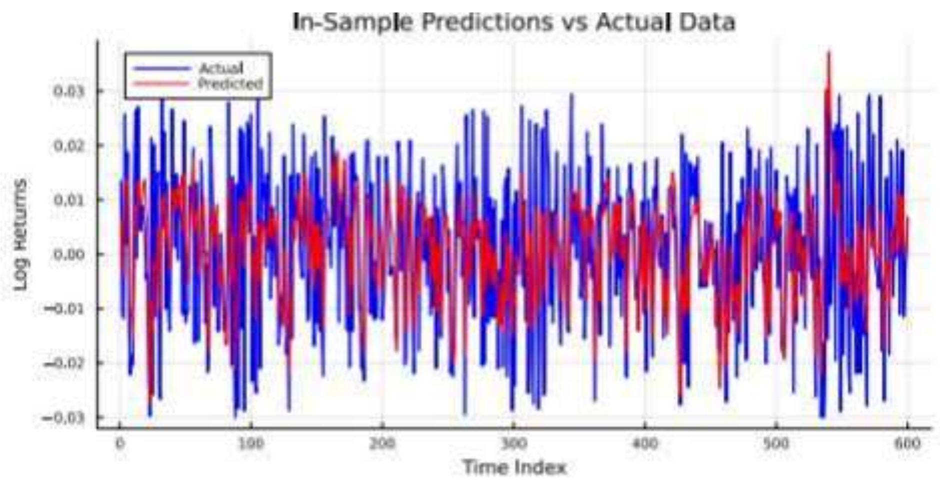
In [29]:
```
println(size(X_train))  # Should print (number_of_train_samples, number_of_features
println(size(X_test))   # Should print (number_of_test_samples, number_of_features)
```

```
(600, 20)
(151, 20)
```

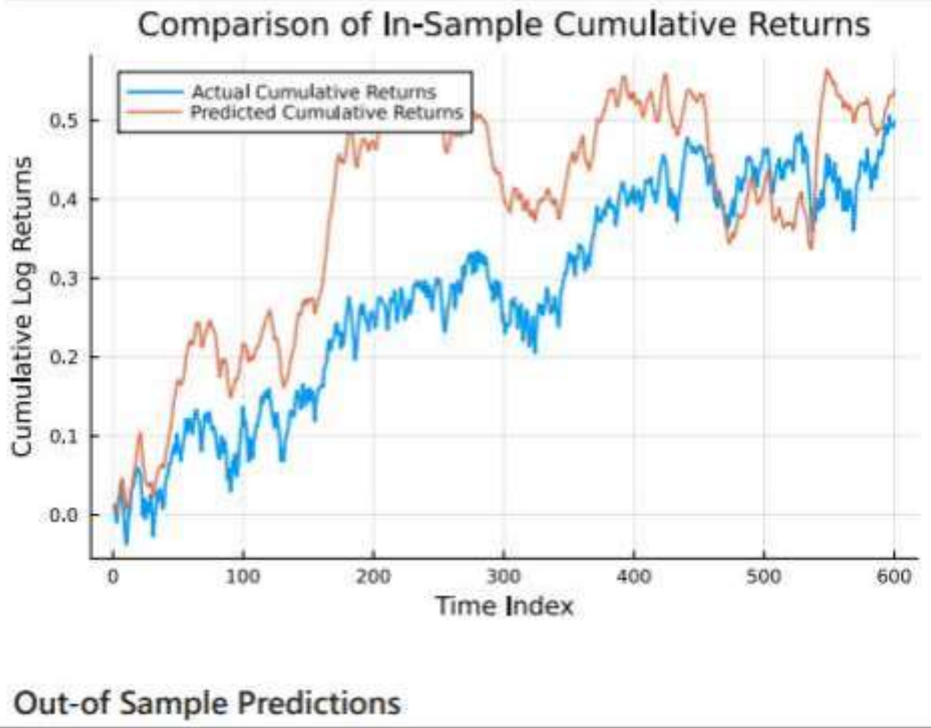
In-Sample Predictions

In [30]:
```
# Add intercept to X_train if not already included
if size(X_train, 2) == size(ridge_coef, 1) - 1
    X_train_with_intercept = hcat(ones(size(X_train, 1)), X_train)
else
    X_train_with_intercept = X_train
end

# Perform in-sample prediction
in_sample_predictions = X_train_with_intercept * ridge_coef;
```

In [31]:
```
# Plot the actual vs predicted values
plot_fig = plot(size=(800, 400))  # Set the size of the plot window

# Plotting the actual data
plot!(plot_fig, y_train, label="Actual", linewidth=2, color=:blue)

# Plotting the predicted data
plot!(plot_fig, in_sample_predictions, label="Predicted", linewidth=2, color=:red,

# Adding titles and labels
title!(plot_fig, "In-Sample Predictions vs Actual Data")
xlabel!(plot_fig, "Time Index")
ylabel!(plot_fig, "Log Returns")

# Display the plot
display(plot_fig)
```

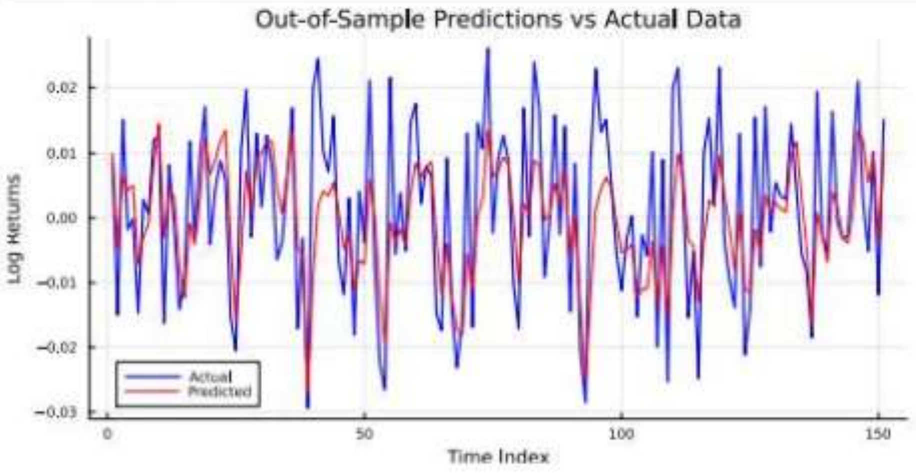


In [32]:
```
# Calculate cumulative sums
cumulative_actual_returns = cumsum(y_train)
cumulative_predicted_returns = cumsum(in_sample_predictions)

# Plotting the actual cumulative returns
isplot = plot(cumulative_actual_returns, label="Actual Cumulative Returns", show=tr

# Plotting the predicted cumulative returns
plot!(isplot, cumulative_predicted_returns, label="Predicted Cumulative Returns", l

# Adding titles and labels
title!(isplot, "Comparison of In-Sample Cumulative Returns")
xlabel!(isplot, "Time Index")
ylabel!(isplot, "Cumulative Log Returns")

# Display the plot
display(isplot)
```



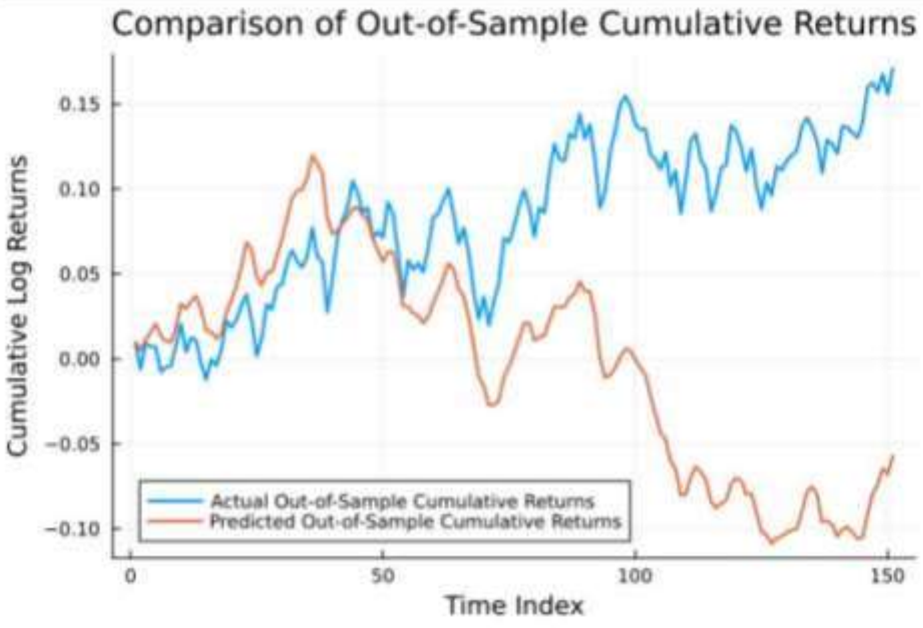
Out-of Sample Predictions

In [33]:
```
# Add intercept to X_test if not already included
if size(X_test, 2) == size(ridge_coef, 1) - 1
    X_test_with_intercept = hcat(ones(size(X_test, 1)), X_test)
else
    X_test_with_intercept = X_test
end

# Perform out-of-sample prediction
out_of_sample_predictions = X_test_with_intercept * ridge_coef

# If you want to plot these predictions as well, you can follow the same method use
plot_out_of_sample = plot(size=(800, 400))

# Plotting the actual data for out-of-sample
plot!(plot_out_of_sample, y_test, label="Actual", linewidth=2, color=:blue)

# Plotting the predicted data for out-of-sample
plot!(plot_out_of_sample, out_of_sample_predictions, label="Predicted", linewidth=:

# Adding titles and labels for out-of-sample plot
title!(plot_out_of_sample, "Out-of-Sample Predictions vs Actual Data")
xlabel!(plot_out_of_sample, "Time Index")
ylabel!(plot_out_of_sample, "Log Returns")

# Display the plot
display(plot_out_of_sample)
```



In [34]:
```
# Calculate cumulative sums for actual and predicted returns
cumulative_actual_returns_test = cumsum(y_test)  # Assuming y_test is your out-of-
cumulative_predicted_returns_test = cumsum(out_of_sample_predictions)

# Start a new plot with the actual out-of-sample cumulative returns
plot_out_of_sample = plot(cumulative_actual_returns_test, label="Actual Out-of-Samp

# Add the predicted out-of-sample cumulative returns
plot!(plot_out_of_sample, cumulative_predicted_returns_test, label="Predicted Out-c

# Adding titles and labels
title!(plot_out_of_sample, "Comparison of Out-of-Sample Cumulative Returns")
xlabel!(plot_out_of_sample, "Time Index")
ylabel!(plot_out_of_sample, "Cumulative Log Returns")
```

```
# Display the plot
display(plot_out_of_sample)
```

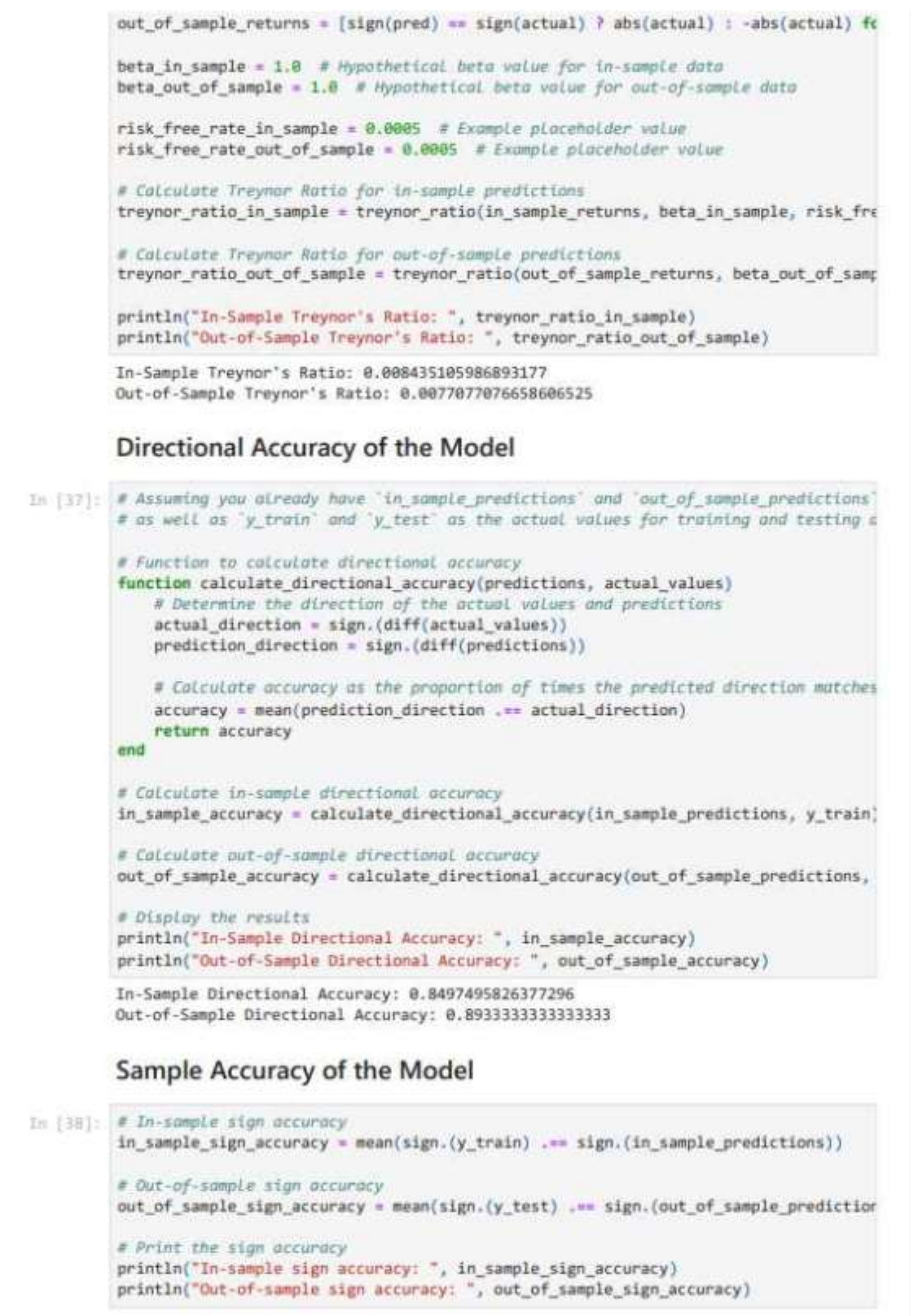


Sharpe Ratio

In [35]:
```
daily_returns_in_sample = [sign(pred) == sign(actual) ? abs(actual) : -abs(actual)
daily_returns_out_of_sample = [sign(pred) == sign(actual) ? abs(actual) : -abs(actu

mean_daily_return_in_sample = mean(daily_returns_in_sample)
mean_daily_return_out_of_sample = mean(daily_returns_out_of_sample)

std_daily_return_in_sample = std(daily_returns_in_sample)
std_daily_return_out_of_sample = std(daily_returns_out_of_sample)

risk_free_rate_in_sample = 0.0005  # Example placeholder value
risk_free_rate_out_of_sample = 0.0005  # Example placeholder value

# Calculate the Sharpe Ratio for in-sample predictions
sharpe_ratio_in_sample = (mean_daily_return_in_sample - risk_free_rate_in_sample) /

# Calculate the Sharpe Ratio for out-of-sample predictions
sharpe_ratio_out_of_sample = (mean_daily_return_out_of_sample - risk_free_rate_out_

println("In-Sample Sharpe Ratio: ", sharpe_ratio_in_sample)
println("Out-of-Sample Sharpe Ratio: ", sharpe_ratio_out_of_sample)
```

```
In-Sample Sharpe Ratio: 0.728291715971954
Out-of-Sample Sharpe Ratio: 0.7292112326939733
```

In [36]:
```
# Function to calculate Treynor Ratio
function treynor_ratio(portfolio_returns, portfolio_beta, risk_free_rate = 0.0)
    # Calculate the excess returns by subtracting the risk-free rate
    excess_returns = mean(portfolio_returns) - risk_free_rate
    # Treynor Ratio computation
    return excess_returns / portfolio_beta
end

in_sample_returns = [sign(pred) == sign(actual) ? abs(actual) : -abs(actual) for (p
```



```
out_of_sample_returns = [sign(pred) == sign(actual) ? abs(actual) : -abs(actual) fo

beta_in_sample = 1.0  # Hypothetical beta value for in-sample data
beta_out_of_sample = 1.0  # Hypothetical beta value for out-of-sample data

risk_free_rate_in_sample = 0.0005  # Example placeholder value
risk_free_rate_out_of_sample = 0.0005  # Example placeholder value

# Calculate Treynor Ratio for in-sample predictions
treynor_ratio_in_sample = treynor_ratio(in_sample_returns, beta_in_sample, risk_fre

# Calculate Treynor Ratio for out-of-sample predictions
treynor_ratio_out_of_sample = treynor_ratio(out_of_sample_returns, beta_out_of_samp

println("In-Sample Treynor's Ratio: ", treynor_ratio_in_sample)
println("Out-of-Sample Treynor's Ratio: ", treynor_ratio_out_of_sample)
```

```
In-Sample Treynor's Ratio: 0.008435105986893177
Out-of-Sample Treynor's Ratio: 0.0077077076658606525
```

Directional Accuracy of the Model

In [37]:
```
# Assuming you already have `in_sample_predictions` and `out_of_sample_predictions`
# as well as `y_train` and `y_test` as the actual values for training and testing d

# Function to calculate directional accuracy
function calculate_directional_accuracy(predictions, actual_values)
    # Determine the direction of the actual values and predictions
    actual_direction = sign.(diff(actual_values))
    prediction_direction = sign.(diff(predictions))

    # Calculate accuracy as the proportion of times the predicted direction matches
    accuracy = mean(prediction_direction .== actual_direction)
    return accuracy
end

# Calculate in-sample directional accuracy
in_sample_accuracy = calculate_directional_accuracy(in_sample_predictions, y_train)

# Calculate out-of-sample directional accuracy
out_of_sample_accuracy = calculate_directional_accuracy(out_of_sample_predictions,

# Display the results
println("In-Sample Directional Accuracy: ", in_sample_accuracy)
println("Out-of-Sample Directional Accuracy: ", out_of_sample_accuracy)
```

```
In-Sample Directional Accuracy: 0.8497495826377296
Out-of-Sample Directional Accuracy: 0.8933333333333333
```

Sample Accuracy of the Model

In [38]:
```
# In-sample sign accuracy
in_sample_sign_accuracy = mean(sign.(y_train) .== sign.(in_sample_predictions))

# Out-of-sample sign accuracy
out_of_sample_sign_accuracy = mean(sign.(y_test) .== sign.(out_of_sample_predictior

# Print the sign accuracy
println("In-sample sign accuracy: ", in_sample_sign_accuracy)
println("Out-of-sample sign accuracy: ", out_of_sample_sign_accuracy)
```

```
In-sample sign accuracy: 0.7816666666666666
Out-of-sample sign accuracy: 0.7549668874172185
```

Model 2 - Logistic Regression

In [39]:
```
using GLM # Note - removed tech indicators apart from RSI due to model overfitting
formula = @formula(BinaryOutcome ~ Lag_1 + Lag_2 + Lag_3 + Lag_4 + Lag_5 + Lag_6 +
```

Re-estimating folds

In [40]:
```
# Sort the training indices if not already sorted
sorted_ind_tr = sort(ind_tr)

# Split sorted_ind_tr into 5 folds
training_folds = []
fold_size = ceil(Int, length(sorted_ind_tr) / 5)

for i in 0:4
    start_idx = i * fold_size + 1
    end_idx = min((i + 1) * fold_size, length(sorted_ind_tr))
    push!(training_folds, sorted_ind_tr[start_idx:end_idx])
end

# Confirm the distribution of indices in each fold
for (i, fold) in enumerate(training_folds)
    println("Training fold ", i, ": ", length(fold), " indices")
end
```

```
Training fold 1: 121 indices
Training fold 2: 121 indices
Training fold 3: 121 indices
Training fold 4: 121 indices
Training fold 5: 117 indices
```

In [41]:
```
# Clear the existing test folds
test_folds = []

# Split ind_te into approximately equal parts for 5 folds
fold_size = ceil(Int, length(ind_te) / 5)
for i in 0:4
    start_idx = i * fold_size + 1
    end_idx = min((i + 1) * fold_size, length(ind_te))
    push!(test_folds, ind_te[start_idx:end_idx])
end

# Confirm the new distribution
for (i, fold) in enumerate(test_folds)
    println("Test fold ", i, ": ", length(fold))
end
```

```
Test fold 1: 30
Test fold 2: 30
Test fold 3: 30
Test fold 4: 30
Test fold 5: 30
```

In [42]:
```
# Prepare the DataFrame to include a binary outcome column for logistic regression
lags[!, :BinaryOutcome] = Int.(lags.lrt .> 0);  # Assuming lrt is already normalize
```

Model Training

In [43]:
```
models = []
full_predictions = zeros(Float64, size(lags, 1))

for (i, training_fold) in enumerate(training_folds)
    train_data = lags[training_fold, :]
    test_data = lags[test_folds[i], :]

    # Train the model
    model = glm(formula, train_data, Binomial(), LogitLink())
    push!(models, model)

    # Predict on training data immediately to verify output
    train_predictions = GLM.predict(model, train_data)
    #println("Train predictions for fold $i: ", unique(train_predictions))  # Check

    # Assign train predictions
    for (local_index, global_index) in enumerate(training_fold)
        full_predictions[global_index] = train_predictions[local_index]
    end

    # Predict on test data
    test_predictions = GLM.predict(model, test_data)
    for (local_index, global_index) in enumerate(test_folds[i])
        full_predictions[global_index] = test_predictions[local_index]
    end
end
```

In [44]:
```
# Assuming 'models' is an array of GLM models from your previous step
accuracies = []
for (i, test_fold) in enumerate(test_folds)
    test_data = lags[test_fold, :]
    predicted = GLM.predict(models[i], test_data)  # Explicitly using GLM.predict

    # Convert probabilities to binary outcomes
    predicted_classes = Int.(predicted .> 0.5)  # Assuming 0.5 as the cutoff

    # Calculate accuracy
    actual_classes = test_data[:, :BinaryOutcome]
    accuracy = mean(predicted_classes .== actual_classes)
    push!(accuracies, accuracy)
end

println("Accuracies for each fold: ", accuracies)
```

```
Accuracies for each fold: Any[0.6666666666666666, 0.6, 0.6, 0.6333333333333333, 0.
6666666666666666]
```

In [45]:
```
# Example: Mean accuracy
mean_accuracy = mean(accuracies)
println("Mean Accuracy across folds: ", mean_accuracy)
```

```
Mean Accuracy across folds: 0.6333333333333333
```

In and Out of Sample Results

In [46]:
```
inds=[ind_tr;ind_te];
```

In [47]:
```
# Binarize full predictions at a 0.5 threshold
binary_predictions = Int.(full_predictions .> 0.5)
```

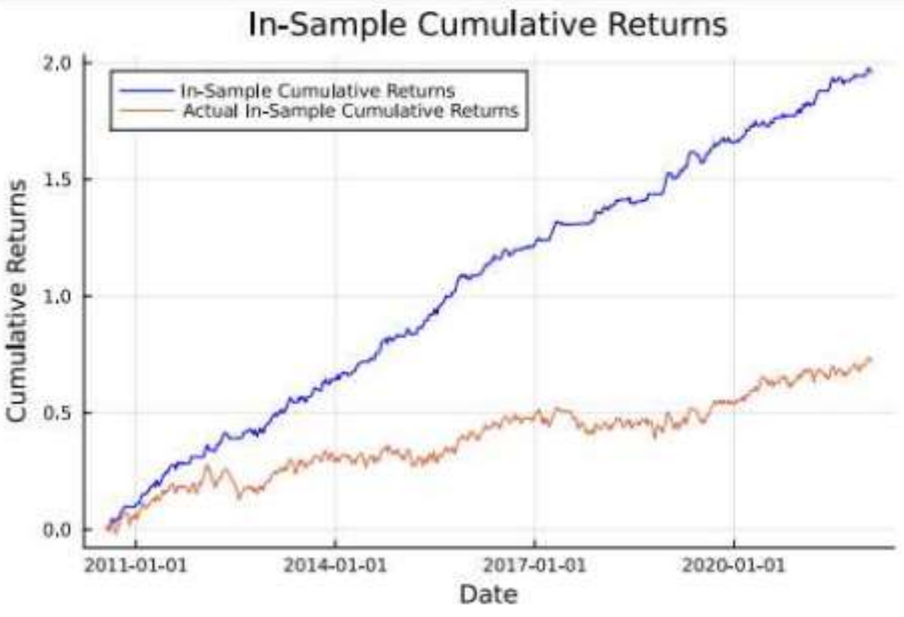
```
# Calculate binary outcomes based on log returns > 0
binary_outcomes = lags.BinaryOutcome;

full_predictions[ind_tr];
```

In [48]:
```
# Calculate log returns
log_returns = lags.lrt

# Compute returns: (predictions are binary, 1 if increase predicted, else 0)
# Since we're multiplying predictions directly with log returns
returns = binary_predictions .* log_returns

# Calculate cumulative returns
cumulative_returns = cumsum(returns);
```

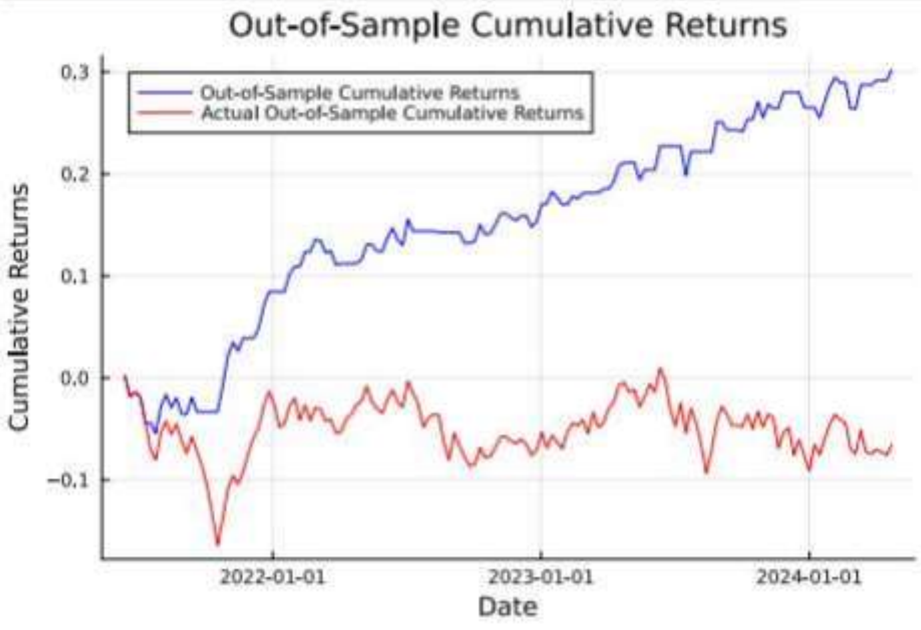
In [49]:
```
# Calculate cumulative returns specifically for in-sample and out-of-sample
in_sample_cumulative_returns = cumsum(binary_predictions[ind_tr] .* log_returns[inc
out_of_sample_cumulative_returns = cumsum(binary_predictions[ind_te] .* log_returns

# Calculate the length of in-sample and out-of-sample periods
ilength = length(ind_tr)
olength = length(ind_te)

# Extract dates for in-sample and out-of-sample periods
idates = dates[1:ilength]
odates = dates[end-olength+1:end];
```

Graphs

In [50]:
```
# Plotting in-sample cumulative returns
in_sample_plot = plot(idates, in_sample_cumulative_returns, label="In-Sample Cumula
plot!(in_sample_plot, idates, cumsum(log_returns[ind_tr]), label="Actual In-Sample
display(in_sample_plot)
```



In [51]:
```
# Plotting out-of-sample cumulative returns
out_of_sample_plot = plot(odates, out_of_sample_cumulative_returns, label="Out-of-S
plot!(out_of_sample_plot, odates, cumsum(log_returns[ind_te]), label="Actual Out-of
display(out_of_sample_plot)
```



Analysis

In [52]:
```
beta = 1 # Assumed for now

# In-sample and Out-of-sample indices
in_sample_returns = returns[ind_tr]
out_of_sample_returns = returns[ind_te]

# Sharpe Ratio calculation
in_sample_sharpe_ratio = sqrt(52) * mean(in_sample_returns) / std(in_sample_returns
out_of_sample_sharpe_ratio = sqrt(52) * mean(out_of_sample_returns) / std(out_of_sa

println("In-Sample Sharpe Ratio: ", in_sample_sharpe_ratio)
println("Out-of-Sample Sharpe Ratio: ", out_of_sample_sharpe_ratio)

# Treynor Ratio calculation, assuming beta = 1 for simplicity
in_sample_treynor_ratio = sqrt(52) * mean(in_sample_returns) / beta
out_of_sample_treynor_ratio = sqrt(52) * mean(out_of_sample_returns) / beta

println("In-Sample Treynor Ratio: ", in_sample_treynor_ratio)
println("Out-of-Sample Treynor Ratio: ", out_of_sample_treynor_ratio)
```

```
In-Sample Sharpe Ratio: 2.4857495827240856
Out-of-Sample Sharpe Ratio: 1.4002133114273678
In-Sample Treynor Ratio: 0.023504852359630735
Out-of-Sample Treynor Ratio: 0.01451128899293313
```

In [53]:
```
# In-sample accuracy
in_sample_accuracy = mean(binary_predictions[ind_tr] .== binary_outcomes[ind_tr])

# Out-of-sample accuracy
out_of_sample_accuracy = mean(binary_predictions[ind_te] .== binary_outcomes[ind_te

println("In-Sample Accuracy: ", in_sample_accuracy)
println("Out-of-Sample Accuracy: ", out_of_sample_accuracy)
```

```
logipredictions = binary_predictions[ind_te];
```

```
In-Sample Accuracy: 0.6722129783693843
Out-of-Sample Accuracy: 0.6333333333333333
```

Model 3 - XG Boost

In [54]:
```
# Sort the training indices if not already sorted
sorted_ind_tr = sort(ind_tr)

# Split sorted_ind_tr into 5 folds
training_folds = []
fold_size = ceil(Int, length(sorted_ind_tr) / 5)

for i in 0:4
    start_idx = i * fold_size + 1
    end_idx = min((i + 1) * fold_size, length(sorted_ind_tr))
    push!(training_folds, sorted_ind_tr[start_idx:end_idx])
end

# Confirm the distribution of indices in each fold
for (i, fold) in enumerate(training_folds)
    println("Training fold ", i, ": ", length(fold), " indices")
end
```

```
Training fold 1: 121 indices
Training fold 2: 121 indices
Training fold 3: 121 indices
Training fold 4: 121 indices
Training fold 5: 117 indices
```

In [55]:
```
# Clear the existing test folds
test_folds = []

# Split ind_te into approximately equal parts for 5 folds
fold_size = ceil(Int, length(ind_te) / 5)
for i in 0:4
    start_idx = i * fold_size + 1
    end_idx = min((i + 1) * fold_size, length(ind_te))
    push!(test_folds, ind_te[start_idx:end_idx])
end

# Confirm the new distribution
for (i, fold) in enumerate(test_folds)
    println("Test fold ", i, ": ", length(fold))
end
```

```
Test fold 1: 30
Test fold 2: 30
Test fold 3: 30
Test fold 4: 30
Test fold 5: 30
```

In [56]:
```
# Calculate the length of in-sample and out-of-sample periods
ilength = length(ind_tr)
olength = length(ind_te)

# Extract dates for in-sample and out-of-sample periods
```

```
idates = dates[1:ilength]
odates = dates[end-olength+1:end];
```

Training

```
In [57]: # Map these arrays to kfold_X and kfold_Xt variables properly
kfold_1 = training_folds[1]
kfold_2 = training_folds[2]
kfold_3 = training_folds[3]
kfold_4 = training_folds[4]
kfold_5 = training_folds[5]

kfold_1t = test_folds[1]
kfold_2t = test_folds[2]
kfold_3t = test_folds[3]
kfold_4t = test_folds[4]
kfold_5t = test_folds[5]
```

```
Out[57]: 605:5:750
```

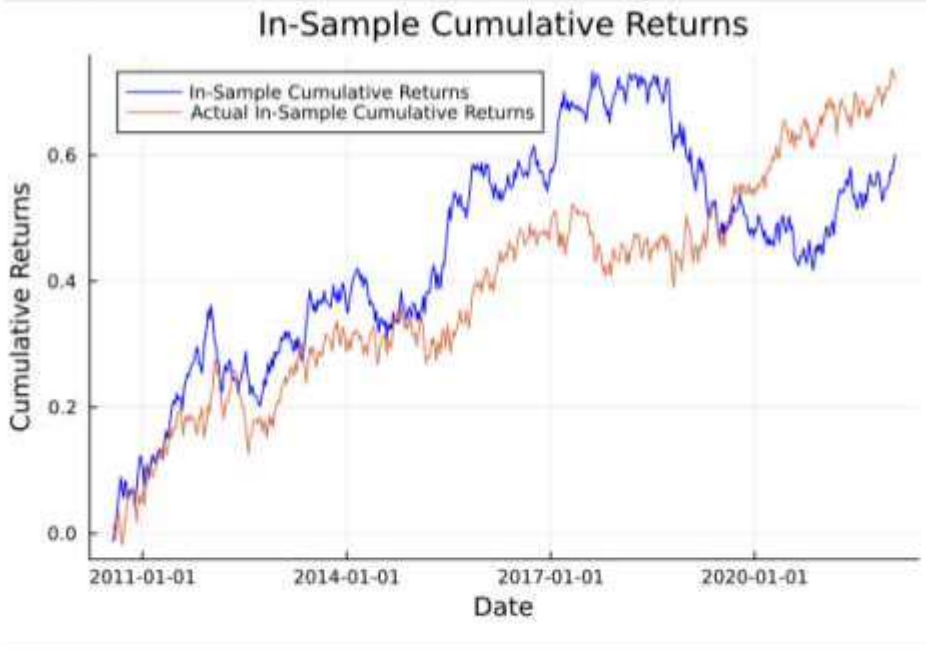
```
In [58]: # Prepare training data for prediction
dtrain_1 = DMatrix(x[kfold_1,:], label = y[kfold_1])
dtrain_2 = DMatrix(x[kfold_2,:], label = y[kfold_2])
dtrain_3 = DMatrix(x[kfold_3,:], label = y[kfold_3])
dtrain_4 = DMatrix(x[kfold_4,:], label = y[kfold_4])
dtrain_5 = DMatrix(x[kfold_5,:], label = y[kfold_5])

# Train the model for each fold
boost_1 = xgboost(dtrain_1, num_round=100, eta=0.1, max_depth=3, min_child_weight=1
boost_2 = xgboost(dtrain_2, num_round=100, eta=0.1, max_depth=3, min_child_weight=1
boost_3 = xgboost(dtrain_3, num_round=100, eta=0.1, max_depth=3, min_child_weight=1
boost_4 = xgboost(dtrain_4, num_round=100, eta=0.1, max_depth=3, min_child_weight=1
boost_5 = xgboost(dtrain_5, num_round=100, eta=0.1, max_depth=3, min_child_weight=1
In [59]: # Prepare test data for prediction using indices to map the indices
         dtrain_1 = DMatrix(x[kfold_1,:]);
         dtrain_2 = DMatrix(x[kfold_2,:]);
         dtrain_3 = DMatrix(x[kfold_3,:]);
         dtrain_4 = DMatrix(x[kfold_4,:]);
         dtrain_5 = DMatrix(x[kfold_5,:]);
```

```
In [60]: # Predict using the trained models
         vp1 = XGBoost.predict(boost_1, dtrain_1);
         vp2 = XGBoost.predict(boost_2, dtrain_2);
         vp3 = XGBoost.predict(boost_3, dtrain_3);
         vp4 = XGBoost.predict(boost_4, dtrain_4);
         vp5 = XGBoost.predict(boost_5, dtrain_5);
```

```
In [61]: # Prepare predictions
         ipreds = vcat(vp4, vp5, vp2, vp3, vp1)  # Concatenate predictions in the preferred
         ibets = sign.(ipreds);  # Calculate the sign of in-sample predictions
         length(ibets)

Out[61]: 601
```

Graphs

```
In [62]: # Plot in-sample cumulative returns
         in_sample_plotsxgb = plot(idates, cumsum(ibets .* y[ind_tr], dims=1), legend=:tople
         plot!(in_sample_plotsxgb, idates, cumsum(log_returns[ind_tr]), label="Actual In-Sam
         display(in_sample_plotsxgb)
```

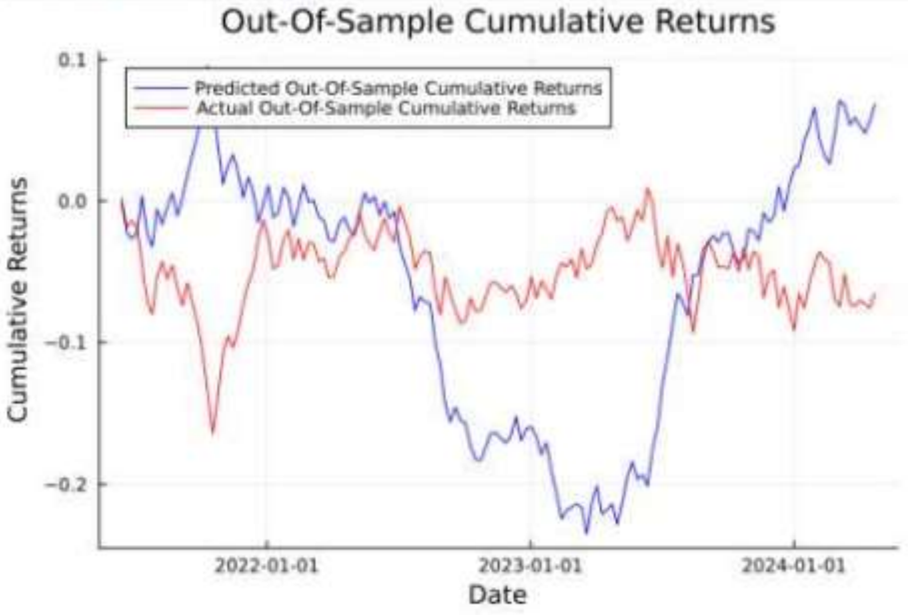


```
In [63]: # Prepare test data for prediction using indices to map the indices
         dtest_1 = DMatrix(x[kfold_1t,:]);
         dtest_2 = DMatrix(x[kfold_2t,:]);
         dtest_3 = DMatrix(x[kfold_3t,:]);
         dtest_4 = DMatrix(x[kfold_4t,:]);
         dtest_5 = DMatrix(x[kfold_5t,:]);
```

In [64]:
```
# Predict using the trained models
yp1 = XGBoost.predict(boost_1, dtest_1);
yp2 = XGBoost.predict(boost_2, dtest_2);
yp3 = XGBoost.predict(boost_3, dtest_3);
yp4 = XGBoost.predict(boost_4, dtest_4);
yp5 = XGBoost.predict(boost_5, dtest_5);
```

In [65]:
```
# Prepare predictions
opreds = vcat(yp5, yp4, yp2, yp3, yp1)  # Concatenate predictions in the preferred
obets = sign.(opreds);  # Calculate the sign of in-sample predictions
```

In [66]:
```
# Plot out-of-sample cumulative returns
out_of_samplexgb = plot(odates, cumsum(obets .* y[ind_te], dims=1), legend=:toplefl
plot!(out_of_samplexgb, odates, cumsum(log_returns[ind_te]), label="Actual Out-Of-S
display(out_of_samplexgb)
```



Analysis

In [67]:
```
beta = 1 # Assumed for now

# Calculate Sharpe and Treynor Ratios for in-sample and out-of-sample
in_sample_returns = sign.(ibets) .* y[ind_tr]
out_of_sample_returns = sign.(obets) .* y[ind_te]

in_sample_sharpe_ratio = sqrt(52) * mean(in_sample_returns) / std(in_sample_returns
out_of_sample_sharpe_ratio = sqrt(52) * mean(out_of_sample_returns) / std(out_of_sa

in_sample_treynor_ratio = sqrt(52) * mean(in_sample_returns) / beta
out_of_sample_treynor_ratio = sqrt(52) * mean(out_of_sample_returns) / beta

println("In-Sample Sharpe Ratio: ", in_sample_sharpe_ratio)
println("Out-of-Sample Sharpe Ratio: ", out_of_sample_sharpe_ratio)
println("In-Sample Treynor Ratio: ", in_sample_treynor_ratio)
println("Out-of-Sample Treynor Ratio: ", out_of_sample_treynor_ratio)
```

```
In-Sample Sharpe Ratio: 0.5031683943581213
Out-of-Sample Sharpe Ratio: 0.22686280742476123
In-Sample Treynor Ratio: 0.007208155469241227
Out-of-Sample Treynor Ratio: 0.0032874533557509197
```

In [68]:

```
# Calculate and print accuracies
in_sample_accuracy = mean(sign.(ibets) .== sign.(y[ind_tr]))
out_of_sample_accuracy = mean(sign.(obets) .== sign.(y[ind_te]))

println("In-Sample Accuracy: ", in_sample_accuracy)
println("Out-of-Sample Accuracy: ", out_of_sample_accuracy)

xgbpreds = obets;
```

```
In-Sample Accuracy: 0.540765391014975
Out-of-Sample Accuracy: 0.5133333333333333
```

.

Model 4 - Random Forest

Re-preparing Data

In [69]:

```
# Load FTSE data
start = DateTime(2009, 4, 18)
ftse = dropmissing(DataFrame(yahoo("^FTSE", YahooOpt(period1 = start,interval="1wk'
ftse = select!(ftse, Not([:Open, :High, :Low, :Close]));
#Compute log returns
lrt0=log.((ftse.AdjClose[2:end])./(ftse.AdjClose[1:end-1]))
lrt0[ismissing.(lrt0)].=0;
lrt=tanh.(lrt0/0.03)*0.03;  #  truncate at +/- 0.03
rets = (ftse.AdjClose[2:end]./ftse.AdjClose[1:end-1]).-1;
ta=TimeArray(ftse,timestamp=:timestamp);  # convert DataFrame to TimeArray requirec
# Prepare TimeArray and indicators
ta = TimeArray(ftse, timestamp=:timestamp)
ema5 = ema(ta[:AdjClose], 5)
ema20 = ema(ta[:AdjClose], 20)
macd1 = macd(ta[:AdjClose], 12, 26, 9)
rsi1 = rsi(ta[:AdjClose]);
ftse2 = ftse[1:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), DataF
ftse2 = ftse[34:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), DataF
DataFrames.rename!(ftse2, :AdjClose_ema_5 => :EMA5)
DataFrames.rename!(ftse2, :AdjClose_ema_20 => :EMA20)
DataFrames.rename!(ftse2, :AdjClose_rsi_14 => :RSI)
lrt0=log.((ftse2.AdjClose[2:end])./(ftse2.AdjClose[1:end-1]))
lrt0[ismissing.(lrt0)].=0;
lrt=tanh.(lrt0/0.03)*0.03;  #  truncate at +/- 0.03
#lrt and x_all are already defined as per your earlier code
number_of_lags = 12
# Initialize an empty DataFrame for lagged features
lagged_features = DataFrame()
# Create lagged return features
# Create lagged return features, avoiding initial rows with placeholder zeros
for lag in 1:number_of_lags
    lagged_features[!, Symbol("Lag_", lag)] = [zeros(lag); lrt[1:end-lag]]
end
tech_inds = ftse2[1:end-1,3:9]
# Assuming `lags` and `tech_inds` are already defined DataFrames
if nrow(lagged_features) > nrow(tech_inds)
```

```
    lags = lagged_features[1:nrow(tech_inds), :]
end

for col in names(tech_inds)
    lagged_features[!, col] = tech_inds[!, col]
end
x = Matrix(lagged_features)
y = lrt[1:end];
```

In [70]:
```
#Remove days when there is no trading
x=x[y.!=0,:];
y=y[y.!=0];
```

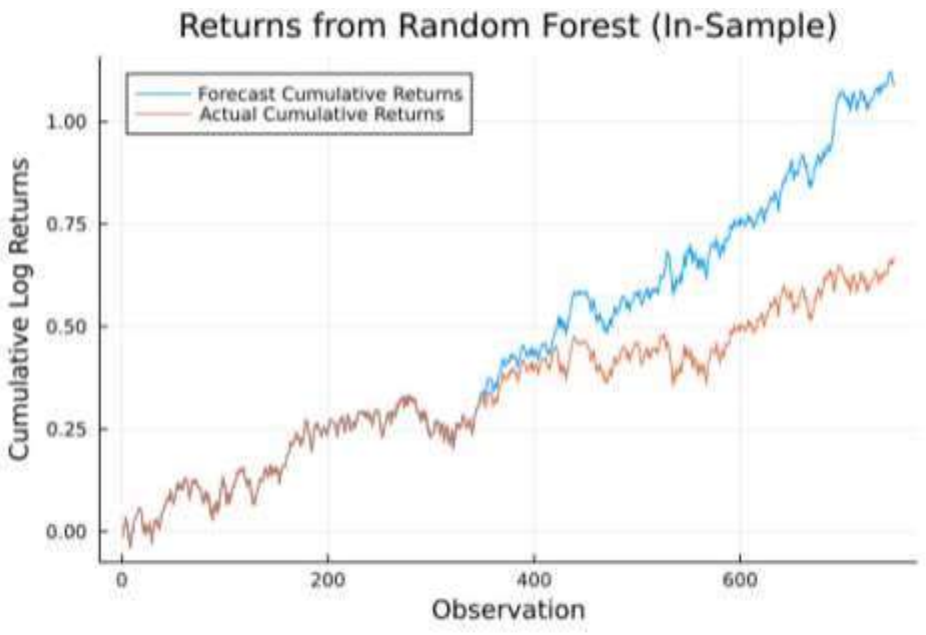
In [71]:
```
x=convert.(Float32,x);  #  This saves memory and makes computation much faster!!!
y=convert.(Float32,y);
ind_all=1:length(y)
ind_te=5:5:length(y)
ind_tr=ind_all[.!in.(ind_all,[ind_te])];
```

Model Estimation

In [72]:
```
using MLDataUtils
using DecisionTree
using Statistics

# Initialize the Random Forest Regressor
rf = RandomForestRegressor(n_trees=100, n_subfeatures=3, max_depth=1)

# Number of folds for cross-validation
k = 5

# Perform k-fold cross-validation
kf = kfolds(collect(1:length(y)), k) # Assume 'y' is your target vector

# Arrays to store the RMSE for each fold
fold_metrics = []
all_predictions = []
all_actuals = []

for (train_indices, val_indices) in kf
    # Split the data into training and validation sets
    X_train, y_train = x[train_indices, :], y[train_indices]
    X_val, y_val = x[val_indices, :], y[val_indices]

    # Train the model on the training set
    DecisionTree.fit!(rf, X_train, y_train)

    # Predict on the validation set
    predictions = DecisionTree.predict(rf, X_val)
    push!(all_predictions, predictions)  # Store predictions
    push!(all_actuals, y_val)            # Store actual values

    # Calculate the RMSE for this fold and store it
    rmse = sqrt(mean((y_val - predictions).^2))
    push!(fold_metrics, rmse)
end

# Calculate the average RMSE across all folds
average_rmse = mean(fold_metrics)
println("Average RMSE across all folds: $average_rmse")
```

```
Average RMSE across all folds: 0.014210279
```

Out[75]:



In [76]:
```julia
actual_cumulative_returns = cumsum(y);
cumulative_forecast_returns = cumsum(in_sample_bets .* y);
```

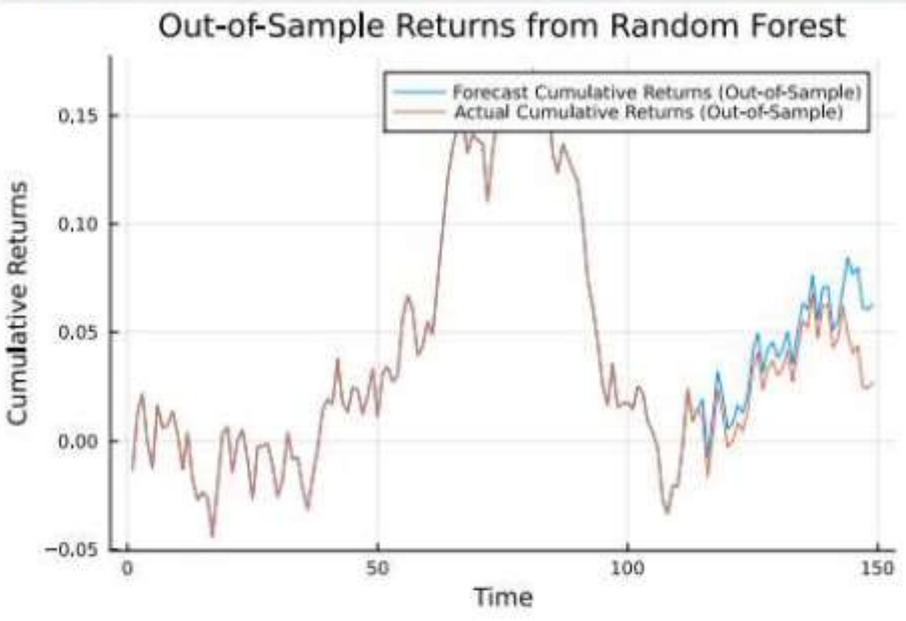
In [77]:
```julia
using Statistics

# Assuming `y` contains log returns and `in_sample_bets` represents the trading str

# Calculate annualized returns for actual and forecasted
annualized_actual_returns = mean(y) * 52
annualized_forecasted_returns = mean(in_sample_bets .* y) * 52

# Calculate annualized standard deviation for actual and forecasted
annualized_actual_std_dev = std(y) * sqrt(52)
annualized_forecasted_std_dev = std(in_sample_bets .* y) * sqrt(52)

# Assuming the risk-free rate is zero for this calculation
risk_free_rate = 0.0

# Sharpe Ratio calculation for actual returns
sharpe_ratio_actual = (annualized_actual_returns - risk_free_rate) / annualized_act

# Sharpe Ratio calculation for forecasted returns
sharpe_ratio_forecasted = (annualized_forecasted_returns - risk_free_rate) / annual

println("In-Sample Sharpe Ratio for Actual Returns: $sharpe_ratio_actual")
println("In-Sample Sharpe Ratio for Forecasted Returns: $sharpe_ratio_forecasted")
```

```
In-Sample Sharpe Ratio for Actual Returns: 0.446881336253227
In-Sample Sharpe Ratio for Forecasted Returns: 0.7304848163002802
```

In [78]:
```julia
using Plots

# Assume the full dataset `x` and `y` are available, and you have
# designated training indices `ind_tr` and testing indices `ind_te`

# Train the model on the full training dataset
DecisionTree.fit!(rf, x[ind_tr, :], y[ind_tr])

# Predict on the full testing dataset
```

```
test_predictions = DecisionTree.predict(rf, x[ind_te, :])

# Assuming `y` contains the log returns
# Calculate the sign of the predictions as betting decisions
bets_test = sign.(test_predictions)

# Calculate cumulative returns based on the model's test predictions
cumulative_returns_test = cumsum(bets_test .* y[ind_te])

# Calculate the actual cumulative returns for the testing data
actual_cumulative_returns_test = cumsum(y[ind_te])

# Plot the cumulative returns for the testing data
plot(cumulative_returns_test, label="Forecast Cumulative Returns (Out-of-Sample)",
     title="Out-of-Sample Returns from Random Forest", xlabel="Time", ylabel="Cumul

# Overlay with actual cumulative returns
plot!(actual_cumulative_returns_test, label="Actual Cumulative Returns (Out-of-Samp
```

Out[78]:



In [79]:
```
actual_cumulative_returns_test = cumsum(y[ind_te])
cumulative_returns_test = cumsum(bets_test .* y[ind_te]);
```

In [80]:
```
# Calculate the annualized returns and standard deviation for Sharpe Ratio
annualized_test_returns = mean(bets_test .* y[ind_te]) * 52
annualized_actual_returns = mean(y[ind_te]) * 52
annualized_test_std_dev = std(bets_test .* y[ind_te]) * sqrt(52)
annualized_actual_std_dev = std(y[ind_te]) * sqrt(52)

# Assuming the risk-free rate is zero for this calculation
risk_free_rate = 0.0

# Sharpe Ratio calculation for test predictions and actual returns
sharpe_ratio_test = (annualized_test_returns - risk_free_rate) / annualized_test_st
sharpe_ratio_actual = (annualized_actual_returns - risk_free_rate) / annualized_act


# Display Sharpe Ratios in the console
```

```
println("Out-of-Sample Sharpe Ratio (Actual): $sharpe_ratio_actual")
println("Out-of-Sample Sharpe Ratio (Forecasted): $sharpe_ratio_test")
```

```
Out-of-Sample Sharpe Ratio (Actual): 0.09294587461192481
Out-of-Sample Sharpe Ratio (Forecasted): 0.21649775188820414
```

In [81]:
```
#Calculationg Beta
market_returns = actual_cumulative_returns_test[1:end]
#x_market = market_returns[inds]
beta = cov(cumulative_returns_test, market_returns) / (var(market_returns))
```

Out[81]:
```
0.9924557f0
```

In [82]:
```
treynor_ratio1 = (mean(cumulative_returns_test*10) / beta)
println("Treynor Ratio: ", treynor_ratio1)
```

```
Treynor Ratio: 0.43676066
```

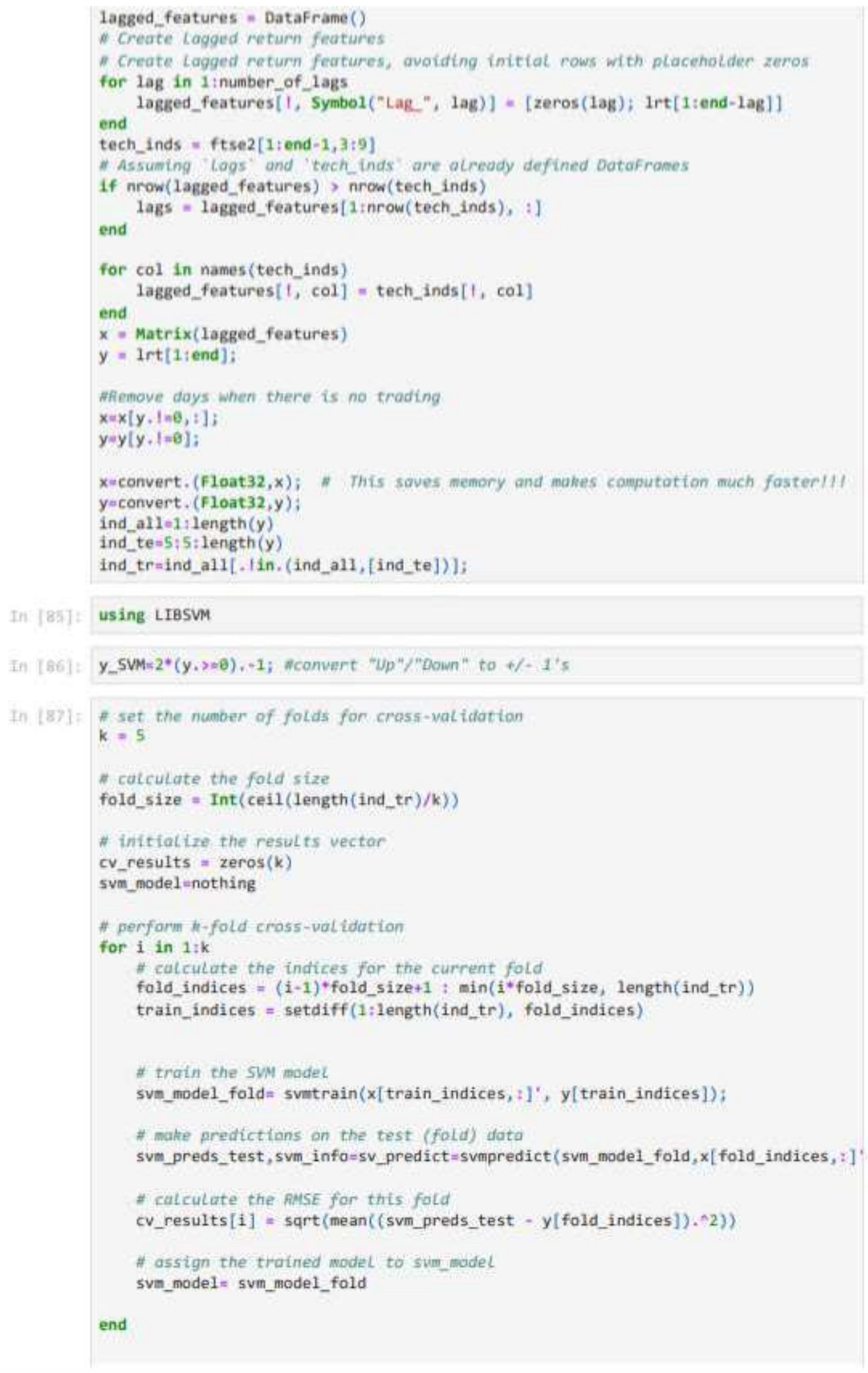
In [83]:
```
excess_returns = y[ind_te] .- (risk_free_rate / 52)  # Convert annual risk-free rat
annualized_excess_returns = mean(excess_returns) * 52  # Annualize weekly returns

treynor_ratio_oos = annualized_excess_returns / beta
```

Out[83]:
```
0.00954033033474536
```

Model 5 - SVM

Data Prep

In [84]:
```
# Load FTSE data
start = DateTime(2009, 4, 18)
ftse = dropmissing(DataFrame(yahoo("^FTSE", YahooOpt(period1 = start,interval="1wk'
ftse = select!(ftse, Not([:Open, :High, :Low, :Close]));
#Compute Log returns
lrt0=log.((ftse.AdjClose[2:end])./(ftse.AdjClose[1:end-1]))
lrt0[ismissing.(lrt0)].=0;
lrt=tanh.(lrt0/0.03)*0.03;  #  truncate at +/- 0.03
rets = (ftse.AdjClose[2:end]./ftse.AdjClose[1:end-1]).-1;
ta=TimeArray(ftse,timestamp=:timestamp);  # convert DataFrame to TimeArray requirec
# Prepare TimeArray and indicators
ta = TimeArray(ftse, timestamp=:timestamp)
ema5 = ema(ta[:AdjClose], 5)
ema20 = ema(ta[:AdjClose], 20)
macd1 = macd(ta[:AdjClose], 12, 26, 9)
rsi1 = rsi(ta[:AdjClose]);
ftse2 = ftse[1:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), Dataf
ftse2 = ftse[34:end, :]
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), Dataf
DataFrames.rename!(ftse2, :AdjClose_ema_5 => :EMA5)
DataFrames.rename!(ftse2, :AdjClose_ema_20 => :EMA20)
DataFrames.rename!(ftse2, :AdjClose_rsi_14 => :RSI)
lrt0=log.((ftse2.AdjClose[2:end])./(ftse2.AdjClose[1:end-1]))
lrt0[ismissing.(lrt0)].=0;
lrt=tanh.(lrt0/0.03)*0.03;  #  truncate at +/- 0.03
#Lrt and x_all are already defined as per your earlier code
number_of_lags = 12
# Initialize an empty DataFrame for lagged features
```



```
lagged_features = DataFrame()
# Create lagged return features
# Create lagged return features, avoiding initial rows with placeholder zeros
for lag in 1:number_of_lags
    lagged_features[!, Symbol("Lag_", lag)] = [zeros(lag); lrt[1:end-lag]]
end
tech_inds = ftse2[1:end-1,3:9]
# Assuming `lags` and `tech_inds` are already defined DataFrames
if nrow(lagged_features) > nrow(tech_inds)
    lags = lagged_features[1:nrow(tech_inds), :]
end

for col in names(tech_inds)
    lagged_features[!, col] = tech_inds[!, col]
end
x = Matrix(lagged_features)
y = lrt[1:end];

#Remove days when there is no trading
x=x[y.!=0,:];
y=y[y.!=0];

x=convert.(Float32,x);  #  This saves memory and makes computation much faster!!!
y=convert.(Float32,y);
ind_all=1:length(y)
ind_te=5:5:length(y)
ind_tr=ind_all[.!in.(ind_all,[ind_te])];
```

```
In [85]: using LIBSVM
```

```
In [86]: y_SVM=2*(y.>=0).-1; #convert "Up"/"Down" to +/- 1's
```

```
In [87]: # set the number of folds for cross-validation
         k = 5

         # calculate the fold size
         fold_size = Int(ceil(length(ind_tr)/k))

         # initialize the results vector
         cv_results = zeros(k)
         svm_model=nothing

         # perform k-fold cross-validation
         for i in 1:k
             # calculate the indices for the current fold
             fold_indices = (i-1)*fold_size+1 : min(i*fold_size, length(ind_tr))
             train_indices = setdiff(1:length(ind_tr), fold_indices)


             # train the SVM model
             svm_model_fold= svmtrain(x[train_indices,:]', y[train_indices]);

             # make predictions on the test (fold) data
             svm_preds_test,svm_info=sv_predict=svmpredict(svm_model_fold,x[fold_indices,:]'

             # calculate the RMSE for this fold
             cv_results[i] = sqrt(mean((svm_preds_test - y[fold_indices]).^2))

             # assign the trained model to svm_model
             svm_model= svm_model_fold

         end
```

```
# calculate the average RMSE across all folds
avg_rmse = mean(cv_results)
```

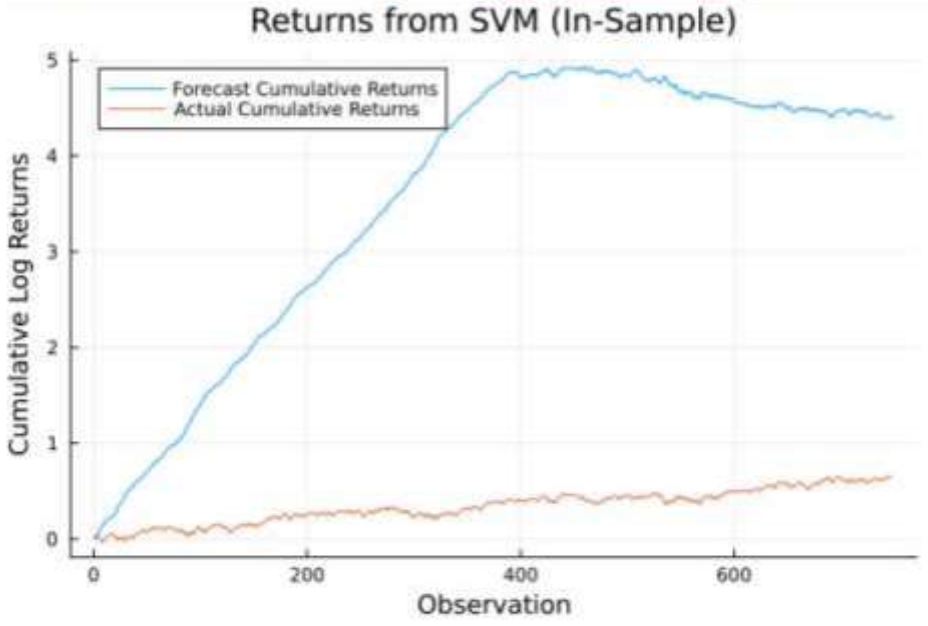
Out[87]:
```
0.020829033479094504
```

In [88]:
```
sv_preds,sv_info=sv_predict=svmpredict(svm_model,x');
sv_preds=vec(sv_preds); # Predictions
sv_bets=sign.(sv_preds);
```

In [89]:
```
# Train the SVM model
svm_model = svmtrain(x', y_SVM)

# Make predictions with the SVM model
predictions = svmpredict(svm_model, x')[1];
```

In [90]:
```
svm_model = svmtrain(x', y_SVM)
in_sample_predictions, _ = svmpredict(svm_model, x')
in_sample_bets = sign.(in_sample_predictions)
cumulative_forecast_returns = cumsum(in_sample_bets .* lrt[1:749])
actual_cumulative_returns = cumsum(lrt[1:747])

# Plot cumulative returns
plot(cumulative_forecast_returns, legend=:topleft, fmt=:png,
     label="Forecast Cumulative Returns", title="Returns from SVM (In-Sample)",
     xlabel="Observation", ylabel="Cumulative Log Returns")

# Overlay the plot with actual cumulative returns
plot!(actual_cumulative_returns, label="Actual Cumulative Returns")
```

Out[90]:



In [91]:
```
# Prepare the training and testing data
x_train = x[ind_tr, :]
y_train = y_SVM[ind_tr]
x_test = x[ind_te, :]
ind_te
```

Out[91]:
```
5:5:745
```

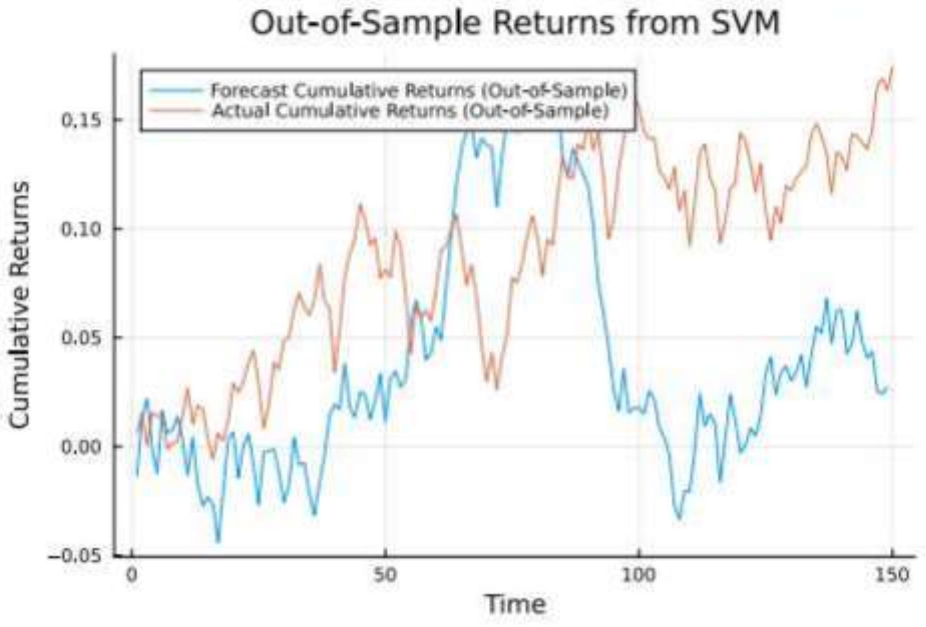
In [92]:
```
y_train = y_SVM[ind_tr];
```

In [93]:
```
# Train the SVM model on the full training dataset
svm_model = svmtrain(x_train', y_train);
```

In [94]:
```
# Predict on the full testing dataset
test_predictions, _ = svmpredict(svm_model, x_test');
```

In [95]:
```
# Convert raw outputs to binary betting decisions
# If the SVM output is already binary labels, this step might be unnecessary
bets_testsvm = sign.(test_predictions)

# Assuming 'y' contains the log returns and is properly aligned with 'x'
# Calculate the cumulative returns based on the model's test predictions
cumulative_returns_test = cumsum(bets_testsvm .* y[ind_te])

# Calculate the actual cumulative returns for the testing data
actual_cumulative_returns_test = cumsum(y[598:747])

# Plot the cumulative returns for the testing data
plot(cumulative_returns_test, label="Forecast Cumulative Returns (Out-of-Sample)",
     title="Out-of-Sample Returns from SVM", xlabel="Time", ylabel="Cumulative Retu

# Overlay with actual cumulative returns
plot!(actual_cumulative_returns_test, label="Actual Cumulative Returns (Out-of-Samp
```

Out[95]:



In [96]:
```
# Calculate in-sample returns based on the model's predictions
model_returns_in_sample = in_sample_bets .* lrt[1:749]

# Calculate the out-of-sample returns based on the model's predictions
model_returns_out_sample = bets_testsvm .* y_SVM[ind_te];
```

In [97]:
```
# Assuming risk-free rate is negligible and beta is 1.0 for both in-sample and out-
risk_free_rate = 0.0  # Risk-free rate is negligible
beta = 1.0  # Beta is assumed to be 1

# In-sample calculations
```

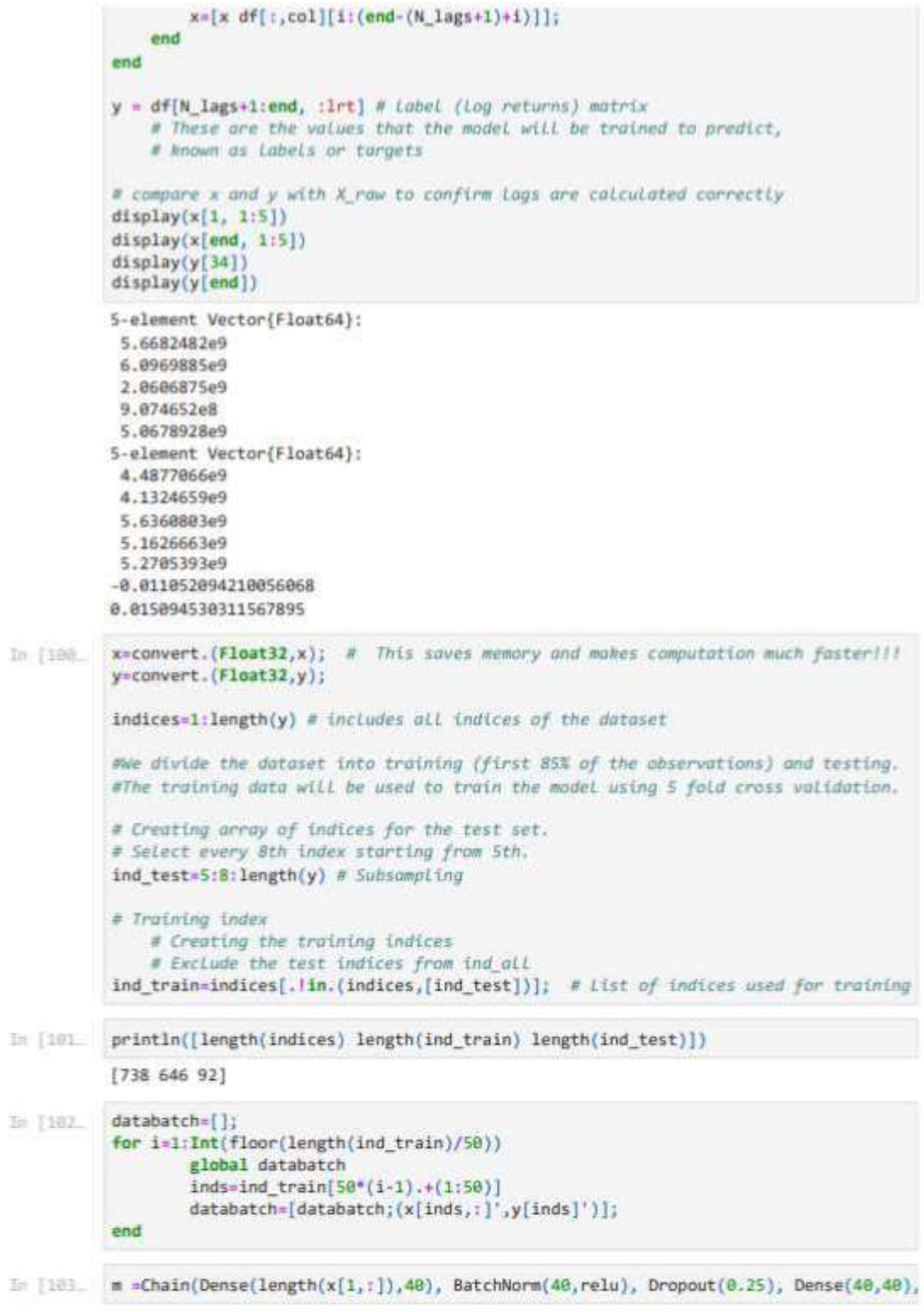
```
mean_daily_return_in_sample = mean(model_returns_in_sample)
std_daily_return_in_sample = std(model_returns_in_sample)
annual_factor = sqrt(52)  # Annualization factor for weekly data

annualized_mean_return_in_sample = mean_daily_return_in_sample * 52
annualized_std_return_in_sample = std_daily_return_in_sample * annual_factor

sharpe_ratio_in_sample = annualized_mean_return_in_sample / annualized_std_return_i
treynor_ratio_in_sample = annualized_mean_return_in_sample / beta  # since beta is

println("In-sample Sharpe Ratio: ", sharpe_ratio_in_sample)
println("In-sample Treynor Ratio: ", treynor_ratio_in_sample)

# Out-of-sample calculations
mean_daily_return_out_sample = mean(model_returns_out_sample)
std_daily_return_out_sample = std(model_returns_out_sample)

annualized_mean_return_out_sample = mean_daily_return_out_sample * 52
annualized_std_return_out_sample = std_daily_return_out_sample * annual_factor

sharpe_ratio_out_sample = annualized_mean_return_out_sample / annualized_std_returr
treynor_ratio_out_sample = annualized_mean_return_out_sample / beta  # since beta i

println("Out-of-sample Sharpe Ratio: ", sharpe_ratio_out_sample)
println("Out-of-sample Treynor Ratio: ", treynor_ratio_out_sample)
```

```
In-sample Sharpe Ratio: 3.2218554656994427
In-sample Treynor Ratio: 0.3048435719031614
Out-of-sample Sharpe Ratio: 0.24130581805214166
Out-of-sample Treynor Ratio: 1.74496644295302
```

Neural Network (prep work for ensembling)

In [98]:
```
ftse3 = ftse[34:end-1, :]
ftse3 = innerjoin(ftse3, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), Datal
DataFrames.rename!(ftse3, :AdjClose_ema_5 => :EMA5)
DataFrames.rename!(ftse3, :AdjClose_ema_20 => :EMA20)
DataFrames.rename!(ftse3, :AdjClose_rsi_14 => :RSI)

ftse3[!, :lrt] = lrt

using DataFrames

# Assuming ftse3 is your existing DataFrame
df = select(ftse3, Not(:AdjClose));
```

In [99]:
```
N_lags = 12  # or N_lags = 5
# Creating features based on 6 previous time steps
# no. lags: For weekly data, this means looking 7 weeks into the past.

# create features (lags of indicators & log returns) matrix
# No. of rows equal to number of rows in X_raw minus no. of lags
# This matrix will be filled with lagged features

global x=zeros(nrow(df)-N_lags,0)
for col in 2:ncol(df)
    for i=1:N_lags
    global x
```



```
        x=[x df[:,col][i:(end-(N_lags+1)+i)]];
    end
end

y = df[N_lags+1:end, :lrt] # Label (Log returns) matrix
    # These are the values that the model will be trained to predict,
    # known as Labels or targets

# compare x and y with X_row to confirm lags are calculated correctly
display(x[1, 1:5])
display(x[end, 1:5])
display(y[34])
display(y[end])
```

```
5-element Vector{Float64}:
 5.6682482e9
 6.0969885e9
 2.0606875e9
 9.074652e8
 5.0678928e9
5-element Vector{Float64}:
 4.4877066e9
 4.1324659e9
 5.6360803e9
 5.1626663e9
 5.2705393e9
-0.011052094210056068
0.015094530311567895
```

In [100…

```
x=convert.(Float32,x);  #  This saves memory and makes computation much faster!!!
y=convert.(Float32,y);

indices=1:length(y) # includes all indices of the dataset

#We divide the dataset into training (first 85% of the observations) and testing.
#The training data will be used to train the model using 5 fold cross validation.

# Creating array of indices for the test set.
# Select every 8th index starting from 5th.
ind_test=5:8:length(y) # Subsampling

# Training index
    # Creating the training indices
    # Exclude the test indices from ind_all
ind_train=indices[.!in.(indices,[ind_test])];  # List of indices used for training
```

In [101…

```
println([length(indices) length(ind_train) length(ind_test)])
```

```
[738 646 92]
```

In [102…

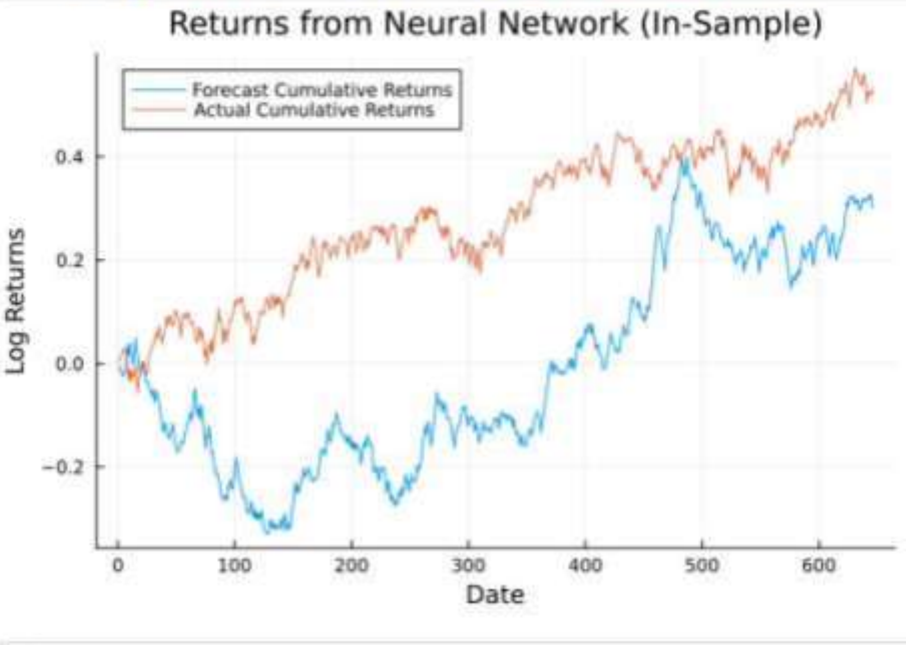
```
databatch=[];
for i=1:Int(floor(length(ind_train)/50))
        global databatch
        inds=ind_train[50*(i-1).+(1:50)]
        databatch=[databatch;(x[inds,:]',y[inds]')];
end
```

In [103…

```
m =Chain(Dense(length(x[1,:]),40), BatchNorm(40,relu), Dropout(0.25), Dense(40,40),
In [414…  using Random

          # Initialize models and train them
          models = [Chain(Dense(length(x[1,:]),50,relu),Dropout(0.50), Dense(50,50,relu), Der
          opt = ADAM(0.001)

          # Train each model using the respective fold data
          for i in 1:length(models)
              train_model(models[i], x[eval(Symbol("kfold_$i")), :]', y[eval(Symbol("kfold_$i
          end

          # Predictions
          predictions = [model(x[eval(Symbol("kfold_$(i)t")), :]')' for (i, model) in enumera
          predictions_vector = predictions[:];
          predictions_vector = vcat(map(vec, predictions)...)
          predictions_float32_vector = Float32.(predictions_vector);

          Early stopping triggered at epoch 50
          Early stopping triggered at epoch 45
```

```
In [415…  NN_preds = vcat(predictions_vector)
          rtn = y[ind_train]
          bets_is =sign.(NN_preds)
          plot(cumsum(bets_is.*y[ind_train],dims=1),legend=:topleft,fmt=png,label="Forecast C
          plot!(cumsum(y[1:length(ind_train)]),label="Actual Cumulative Returns")
```

Out[415]:

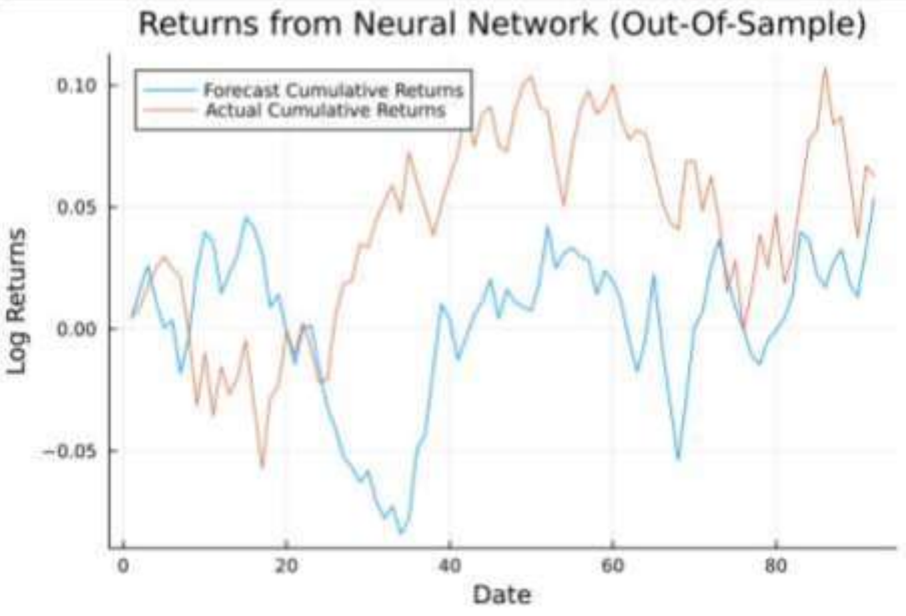


```
In [416…  # Sharpe in-Sample
          mean(sign.(NN_preds).*y[ind_train])./std(sign.(NN_preds).*y[ind_train])*sqrt(52)

Out[416]: 0.23473685888719303
```

```
In [417…  # Predict using each of the models trained
          predictions_oos = [models[i](x[ind_test, :]')' for (i, model) in enumerate(models)]
          predictions_oos = mean(hcat(predictions_oos));

          NN_preds_OOS = vcat(predictions_oos)
          rtn_os = y[ind_test]
          bets_os =sign.(NN_preds_OOS)
```

```
plot(cumsum(bets_os.*y[ind_test],dims=1),legend=:topleft,fmt=png,label="Forecast Cu
plot!(cumsum(y[1:length(ind_test)]),label="Actual Cumulative Returns")
```

Out[417]:



In [418…

```
# Sharpe Out-Of-Sample
mean(sign.(NN_preds_OOS).*y[ind_test])./std(sign.(NN_preds_OOS).*y[ind_test])*sqrt(
```

Out[418]:

```
0.29476646236235643
```

Neural Network Ensembling

Re-preparing data

In [458…

```
# Preparing FTSE Data
# Load FTSE data
start = DateTime(2009, 4, 17)
ftse = dropmissing(DataFrame(yahoo("^FTSE", YahooOpt(period1 = start, interval="1wk
ftse = select!(ftse, Not([:Open, :High, :Low, :Close]));
plot(ftse.timestamp, ftse.AdjClose, legend=:topleft, show=true, fmt=:png, title="FT

# Calculate log returns and truncate
lrt0 = log.((ftse.AdjClose[2:end]) ./ (ftse.AdjClose[1:end-1]))
lrt0[ismissing.(lrt0)] .= 0
lrt = tanh.(lrt0 / 0.03) * 0.03
ftse[:, :lrt] = [0; lrt];
rets = (ftse.AdjClose[2:end] ./ ftse.AdjClose[1:end-1]) .- 1;
Plots.plot(cumsum(lrt), label="Truncated Log Returns", legend=:topleft, show=true,
plot!(cumsum(lrt0), label="FTSE Log Returns", legend=:topleft, linewidth=2)
ylabel!("Cumulative Return")

ta = TimeArray(ftse, timestamp=:timestamp);  # Convert DataFrame to TimeArray requi

# Adding Technical Indicators
# Prepare TimeArray and indicators
```

```
ta = TimeArray(ftse, timestamp=:timestamp)
ema5 = ema(ta[:AdjClose], 5)
ema20 = ema(ta[:AdjClose], 20)
macd1 = macd(ta[:AdjClose], 12, 26, 9)
rsi1 = rsi(ta[:AdjClose]);
ftse2 = ftse[34:end, :]  # Align starts by truncating initial rows where indicators
ftse2 = innerjoin(ftse2, DataFrame(ema5), DataFrame(ema20), DataFrame(macd1), DataF
DataFrames.rename!(ftse2, :AdjClose_ema_5 => :EMA5)
DataFrames.rename!(ftse2, :AdjClose_ema_20 => :EMA20)
DataFrames.rename!(ftse2, :AdjClose_rsi_14 => :RSI)

n_lags = 12

# Initialize an empty DataFrame for lagged return features
lags = DataFrame()

for lag in 1:n_lags
    lags[!, Symbol("Lag_", lag)] = [zeros(lag); lrt[1:end-lag]]
end

tech_inds = ftse2[1:end, 3:10]
if nrow(lags) != nrow(tech_inds)
    resize!(lags, nrow(tech_inds));  # This adjusts lags to have the same number of
end

# Append each technical indicator from ftse2 to the lags DataFrame
for col in names(tech_inds)
    lags[!, col] = tech_inds[!, col]
end

y = lags.lrt;
# Filter days with no trading
x = Matrix(lags)
x = x[y .!= 0, :]  # Remove rows where trading is zero as per 'y'
y = y[y .!= 0]  ;   # Also filter 'y' to match 'x'
```

Ensembling

[135

```
# Preprocess predictions for each model
function preprocess_predictions(predictions, threshold=0.5)
    return [x >= threshold ? 1 : -1 for x in predictions]
end

# Apply preprocessing to each model's output
out_of_sample_predictions = preprocess_predictions(out_of_sample_predictions, 0)  #
logipredictions = preprocess_predictions(logipredictions, 0.5)  # for Logistic Regr
xgbpreds = preprocess_predictions(xgbpreds, 0)  # for XGB
bets_test = preprocess_predictions(bets_test, 0.5)  # for Random Forest
bets_testsvm = preprocess_predictions(bets_testsvm, 0.5);  # for SVM

# Assuming bets_testsvm has the correct length (150)
reference_length = length(bets_test)  # Use the length of bets_testsvm as the refer

# Adjust out_of_sample_predictions to have the same length as bets_testsvm
if length(out_of_sample_predictions) > reference_length
    out_of_sample_predictions = out_of_sample_predictions[1:reference_length]
end

if length(logipredictions) > reference_length
    logipredictions = logipredictions[1:reference_length]
end

if length(xgbpreds) > reference_length
```

```
    xgbpreds = xgbpreds[1:reference_length]
end

if length(bets_testsvm) > reference_length
    bets_testsvm = bets_testsvm[1:reference_length]
end

# Print Lengths to verify
println("Length of out_of_sample_predictions: ", length(out_of_sample_predictions))
println("Length of logipredictions: ", length(logipredictions))
println("Length of xgbpreds: ", length(xgbpreds))
println("Length of bets_test: ", length(bets_test))
println("Length of bets_testsvm: ", length(bets_testsvm))
```

```
Length of out_of_sample_predictions: 149
Length of logipredictions: 149
Length of xgbpreds: 149
Length of bets_test: 149
Length of bets_testsvm: 149
```

In [460…

```
# Create an ensemble matrix
ensemble_matrix = hcat(out_of_sample_predictions, logipredictions, xgbpreds, bets_t
```

In [461…

```
# Combine ensemble with labels and original inputs (assuming y and x[ind_te,:] are
neural_data_test = hcat(y[ind_te], ensemble_matrix)
neural_matrix_test = hcat(neural_data_test, x[ind_te,:])
neural_matrix_test = convert.(Float32, neural_matrix_test);
```

In [462…

```
# Extract features and labels
Xt = Array(neural_matrix_test[:, 2:end])
yt = 2 * (Array(neural_matrix_test[:, 1]) .>= 0) .- 1
yt = convert.(Float32, yt);
```

In [463…

```
# Neural Network Model Definition
input_size = size(Xt, 2)  # Dynamically assign input size
m = Chain(
    Dense(input_size, 40, relu),
    BatchNorm(40),
    Dropout(0.25),
    Dense(40, 40, relu),
    BatchNorm(40),
    Dropout(0.25),
    Dense(40, 20, relu),
    BatchNorm(20),
    Dense(20, 1)
)
```

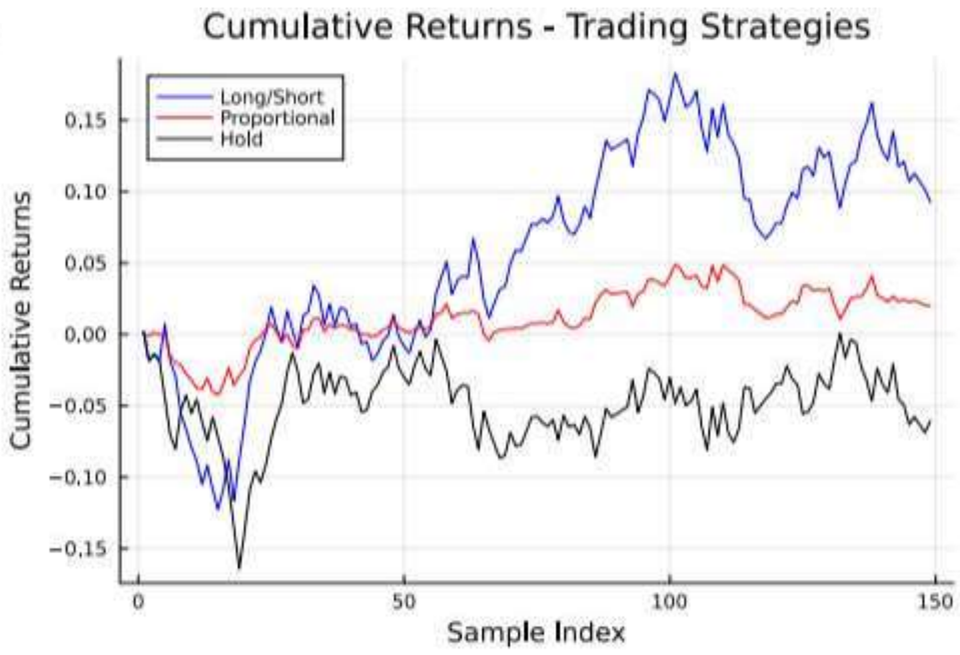
Out[463]:

```
Chain(
  Dense(25 => 40, relu),                # 1_040 parameters
  BatchNorm(40),                        # 80 parameters, plus 80
  Dropout(0.25),
  Dense(40 => 40, relu),                # 1_640 parameters
  BatchNorm(40),                        # 80 parameters, plus 80
  Dropout(0.25),
  Dense(40 => 20, relu),                # 820 parameters
  BatchNorm(20),                        # 40 parameters, plus 40
  Dense(20 => 1),                       # 21 parameters
)         # Total: 14 trainable arrays, 3_721 parameters,
          # plus 6 non-trainable, 200 parameters, summarysize 16.559 KiB.
```

In [464…

```
# Custom loss function
loss(x, y) = mean((m(x) .* y .- 1).^2)
```

Out[145]:



In [146…

```julia
# Assuming a beta value for the portfolio (typically calculated via regression agai
portfolio_beta = 1.0  # Placeholder value

# Update Treynor Ratio calculations
function treynor_ratio(returns, beta, risk_free_rate)
    average_returns = mean(returns .- risk_free_rate)
    return average_returns / (std(returns) * beta) * sqrt(52)
end

# Comprehensive metrics display
println("Accuracy: ", mean(nn_preds .== yt))
println("Sharpe Ratio (Long-Short): ", long_short_sharpe)
println("Sharpe Ratio (Proportional): ", proportional_sharpe)

# Calculate Treynor Ratio for both strategies
treynor_long_short = treynor_ratio(nn_preds .* rtn2, portfolio_beta)
treynor_proportional = treynor_ratio((nn_scores * 3) .* rtn2, portfolio_beta)
println("Treynor Ratio (Long-Short): ", treynor_long_short)
println("Treynor Ratio (Proportional): ", treynor_proportional)
```

```
Accuracy: 0.5234899328859061
Sharpe Ratio (Long-Short): 0.3097401448232795
Sharpe Ratio (Proportional): 0.1683434826105973
Treynor Ratio (Long-Short): 0.3097401448232795
Treynor Ratio (Proportional): 0.1683434826105973
```