Performances of the Moving Average Model and the Artificial Neural Network on the Forecast of Stock Market Indices

Peilin Jing, Jiachen Liu, Yaoyuan Zhang, Rashi Lodhi

Table of contents I

Section 1

Introduction

Abstract

This project delves into the theoretical framework of two types of models. It aims to compare and contrast the performances of the moving average model and the artificial neural network on the prediction of stock market indices. We used the data collected from Yahoo Finance with daily frequency for the period from 1 January 2000 to 31 December 2021. By using a rolling window approach, we evaluated ARIMA-SGARCH to reflect the specific time-series characteristics and have better predictive power than the simple ARIMA model and Recurrent Neural Network models. In order to assess the precision and quality of these models in forecasting, we compared their equity lines, their forecasting error metrics and their performance metrics. The main contribution of this research is to show that the hybrid ARIMA SGARCH model outperforms the other models over the long term.

Importance of this topic

The importance of this topic can be condensed to 4 points:

- **Interest in the area**: attracted attention of researchers, investors, speculators, and governments
- **ARIMA hybrid over ARIMA**: financial time series often do not follow ARIMA assumptions
- **newest ML techniques to improve models**: We use Recurrent Neural Network(RNN) model to determine if it can reflect the specific time series characteristics and predict better.

Section 2

Moving Average Model Methodology and Data

Data Analysis

The first step in the process was cleaning the data. Then, we transformed the adjusted price into a daily logarithmic return, which was calculated according to the following formula:

$$r_t = ln\frac{P_t}{P_{t-1}}$$

Reasons to choose log returns: - can be added across time periods to create cumulative returns - easy to convert between log return and simple return - log return follows normal distribution

Advantages to log return having normal distribution: - Distribution only dependent on mean and sd of sample - forecast with higher accuracy (log return) - Stock prices cannot be normal distribution

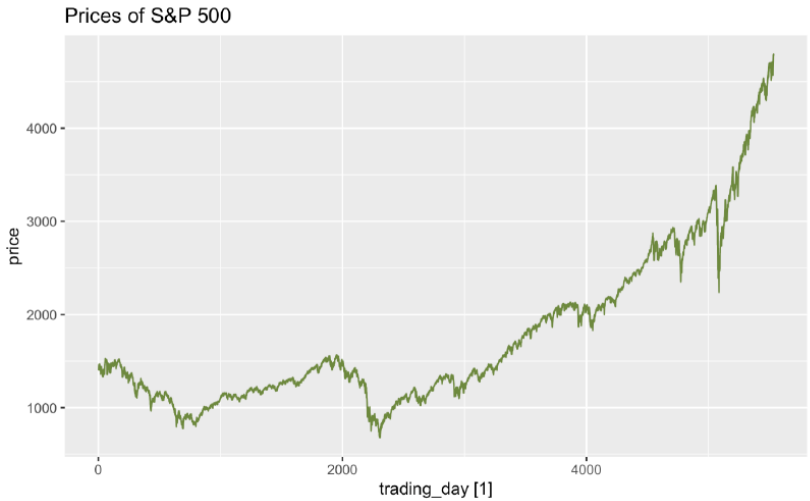
Descriptive Statistics - Stock prices

This presents the descriptive statistics of the adjusted closing prices

Statistics <chr>	Value <dbl>
Min Value	676.530029
1st Quantile	1173.805054
Median	1409.119995
3rd Quantile	2125.030029
Max Value	4793.060059
Mean	1772.378729
Skewness	1.400488
Kurtosis	1.457625

Descriptive Statistics - Stock prices II

There are a few periods, such as 2008, 2011, 2015, and 2018, that show high volatility of returns. We can expect to build more accurate forecasting models if we are able to mitigate and "smooth" such periods.



Stock prices of SP 500 are not normally distributed!

Descriptive Statistics - Log Returns

This presents the descriptive statistics of the adjusted closing prices

Statistics <chr>	Value <dbl>
Min Value	-0.1276521976
1st Quantile	-0.0047060675
Median	0.0006391235
3rd Quantile	0.0058129464
Max Value	0.1095719677
Mean	0.0002148568
Skewness	-0.4004359781
Kurtosis	11.0560760822

We use log returns to build models!

Methodology - ARIMA (p,d,q)

The ARMA process is the combination of the autoregressive model and moving average designed for a stationary time series.

Autoregression (AR) describes a stochastic process, and AR(p) can be denoted as shown below:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t,$$

where ε_t is white noise. This is a multiple regression with lagged values of y_t as predictors.

The moving average process of order q is denoted as MA(q) and the created time series contains a mean of q lagged white noise variables shifting along the series.

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q},$$

where ε_t is white noise. This is a multiple regression with past errors as predictors.

Methodology - ARIMA (p,d,q)

d is the number of differencing done to the series to achieve stationarity with I (d) so the ARIMA model can be expressed as

Expand: $$y_t = c + y_{t-1} + \phi_1 y_{t-1} - \phi_1 y_{t-2} + \theta_1 \varepsilon_{t-1} + \varepsilon_t$$

p is the number of autoregressive terms (AR) d is the number of differencing (I) q is the number of moving average terms (MA)

Methodology - ARCH(p)

The `ARCH(p)` model is given:

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i u_{t-i}^2$$

- Most volatility models derive from this
- Returns have a conditional distribution (here assumed to be normal)
- ARCH is not a very good model and almost nobody uses it.
- The reason is that it needs to use information from many days before t to calculate volatility on day t. That is, it needs a lot of lags.
- The solution is to write it as an ARMA model.
- That is, add one component to the equation, $\beta\sigma_{t-1}$.

Methodology - GARCH(r,s) and Hybrid ARIMA-SGARCH

The `GARCH(p,q)` model is

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i u_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2$$

Where: α is news. β is memory. The size of $(\alpha + \beta)$ determines how quickly the predictability of the process dies out.

This leads us to lastly, ARIMA-SGarch

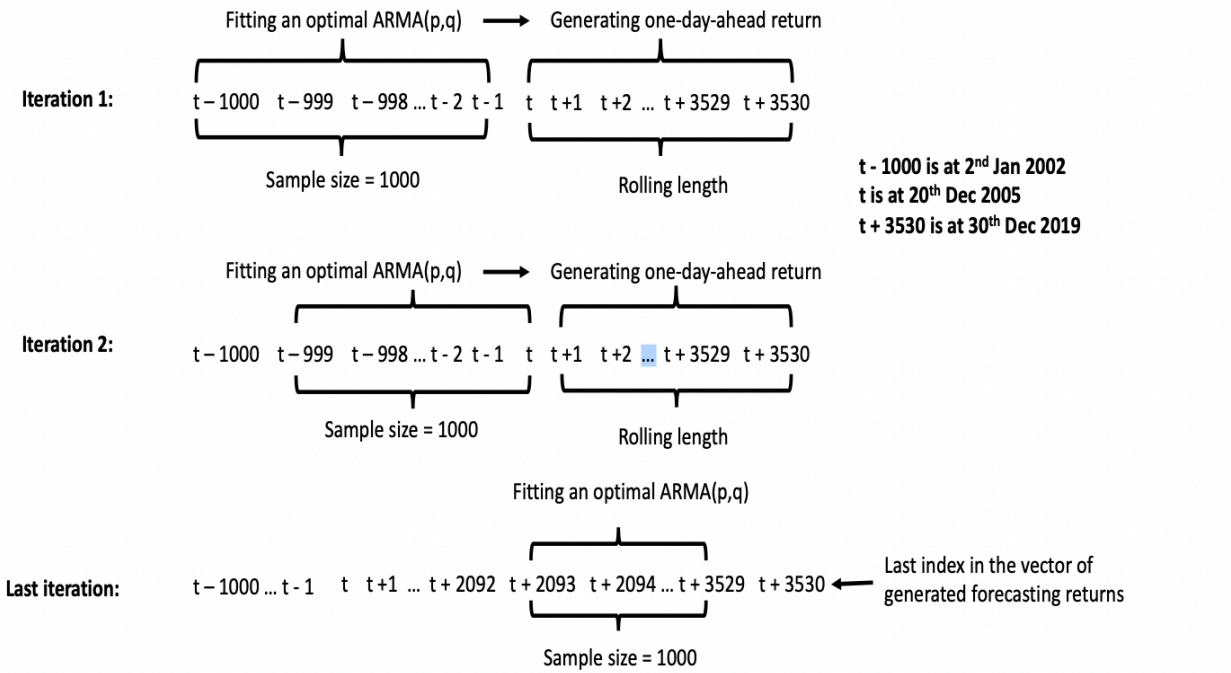
ARIMA SGARCH - Overview

Stock prices can be tremendously volatile during economic growth as well as recessions. When homoskedasticity presumption is violated, it affects the validity or power of statistical tests when using ARIMA models. We consider the SGARCH effect. The error term of the ARIMA model in this process follows SGARCH(1,1) instead of being assumed constant like the ARIMA model.

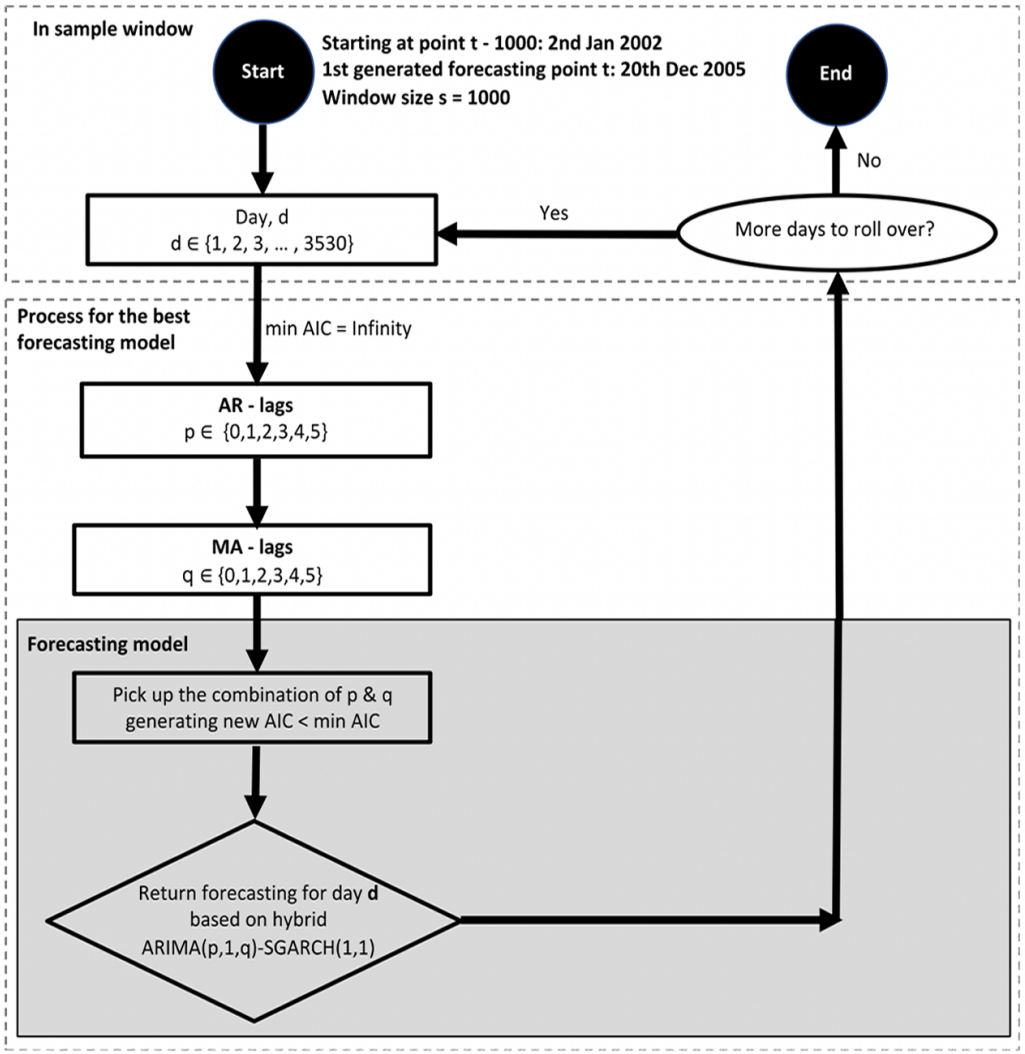
ARIMA SGARCH - Steps

1) We conduct a rolling forecast based on an ARIMA-SGARCH model with window size(s) equal to 1000.
2) The optimized combination of p and q which has the lowest AIC is used to predict return for the next point. At the end, the vector of forecasted values has the length of 3530 elements
3) We describe and review our implementation of dynamic ARIMA(p,1,q)-SGARCH(1,1) models with GED distribution and window size(s) equal to 1000.
4) we evaluate the results based on error metrics, performance metrics, and equity curves.

Iteration of the forecasting model ARIMA(p,1,q)-SGARCH(1,1)



Flowchart of the forecasting model ARIMA(p,1,q)-SGARCH(1,1).



Section 3

Moving Average Model - Coding

Data import

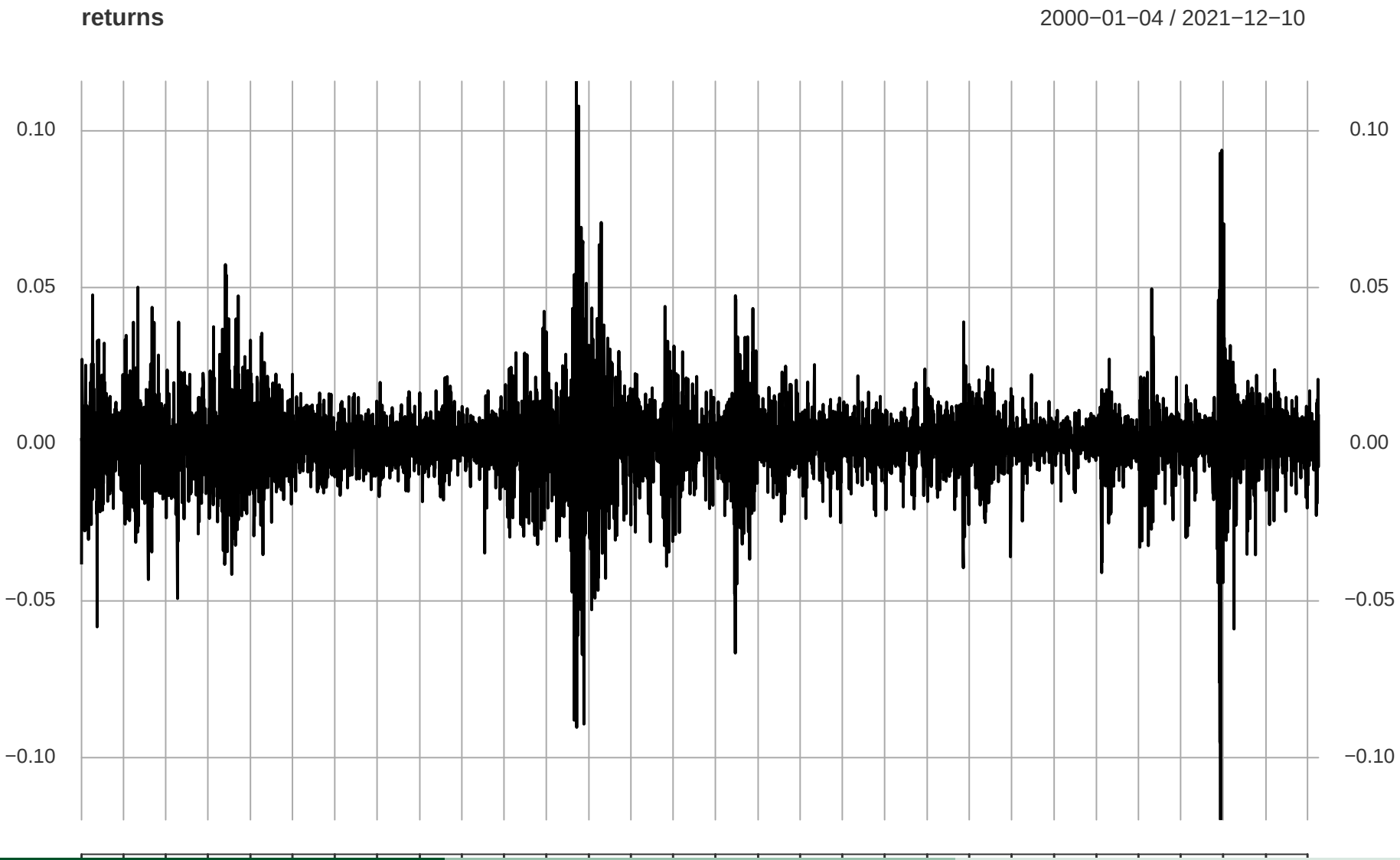
```
# Import Data
sp <- getSymbols(Symbols = "^GSPC", from = "2000-01-01",
                 to = "2021-12-13", src = "yahoo",
                 adjust=TRUE, auto.assign = FALSE)
sp_prices <- Ad(sp)
head(sp_prices)
```

```
           GSPC.Adjusted
2000-01-03       1455.22
2000-01-04       1399.42
2000-01-05       1402.11
2000-01-06       1403.45
2000-01-07       1441.47
2000-01-10       1457.60
```

Return calculation

```
#Compute the log returns
returns <- CalculateReturns(sp_prices) %>% na.omit()
data <- returns
```

Return calculation



'rugarch' package exploration

- **ugarchspec()**: Method for creating a univariate GARCH specification object prior to fitting.
- **ugarchfit()**: Method for fitting a variety of univariate GARCH models.
- **ugarchroll()**: Method for creating rolling density forecast from ARMA-GARCH models with option for refitting every n periods with parallel functionality.
- **ugarchboot()**: Method for forecasting the GARCH density based on a bootstrap procedures (see details and references).
- **ugarchforecast()**:Method for forecasting from a variety of univariate GARCH models.
- **ugarchfilter()**: Method for filtering a variety of univariate GARCH models.
- **ugarchpath()**: Method for simulating the path of a GARCH model from a variety of univariate GARCH models.

Specify sGarch model

```
# Specify sGARCH model
spec <- ugarchspec(
    variance.model =
      list(model = "sGARCH",
           garchOrder = c(1,1)),
      mean.model =
      list(armaOrder = c(0,0),
      include.mean = TRUE),
    distribution.model = "ged"
)
```

We choose the best model from the paper and reproduce it. The best model is hybrid model ARIMA(p,1,q)-SGARCH(1,1) with GED distribution (SGARCH.GED 1000), so we define the model = "sGARCH" and define the distribution model as ged.

Fit Arima-sGARCH Model

```
# Fit to the data
data -> y

sGARCH <- ugarchfit(spec = spec,
                    data = y,
                   solver = 'hybrid')
```

The solver parameter accepts a string stating which numerical optimizer to use to find the parameter estimates.The "hybrid" strategy solver first tries the "solnp" solver, in failing to converge then tries then "nlminb", the "gosolnp" and finally the "nloptr" solvers. The out.sample option is provided in order to carry out forecast performance testing against actual data.

Fit Arima-sGARCH Model

```
*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*

Conditional Variance Dynamics
-----------------------------------
GARCH Model  : sGARCH(1,1)
Mean Model   : ARFIMA(0,0,0)
Distribution    : ged

Optimal Parameters
------------------------------------
        Estimate  Std. Error  t value Pr(>|t|)
mu      0.000740    0.000089   8.3269 0.000000
omega   0.000002    0.000001   1.6854 0.091915
```

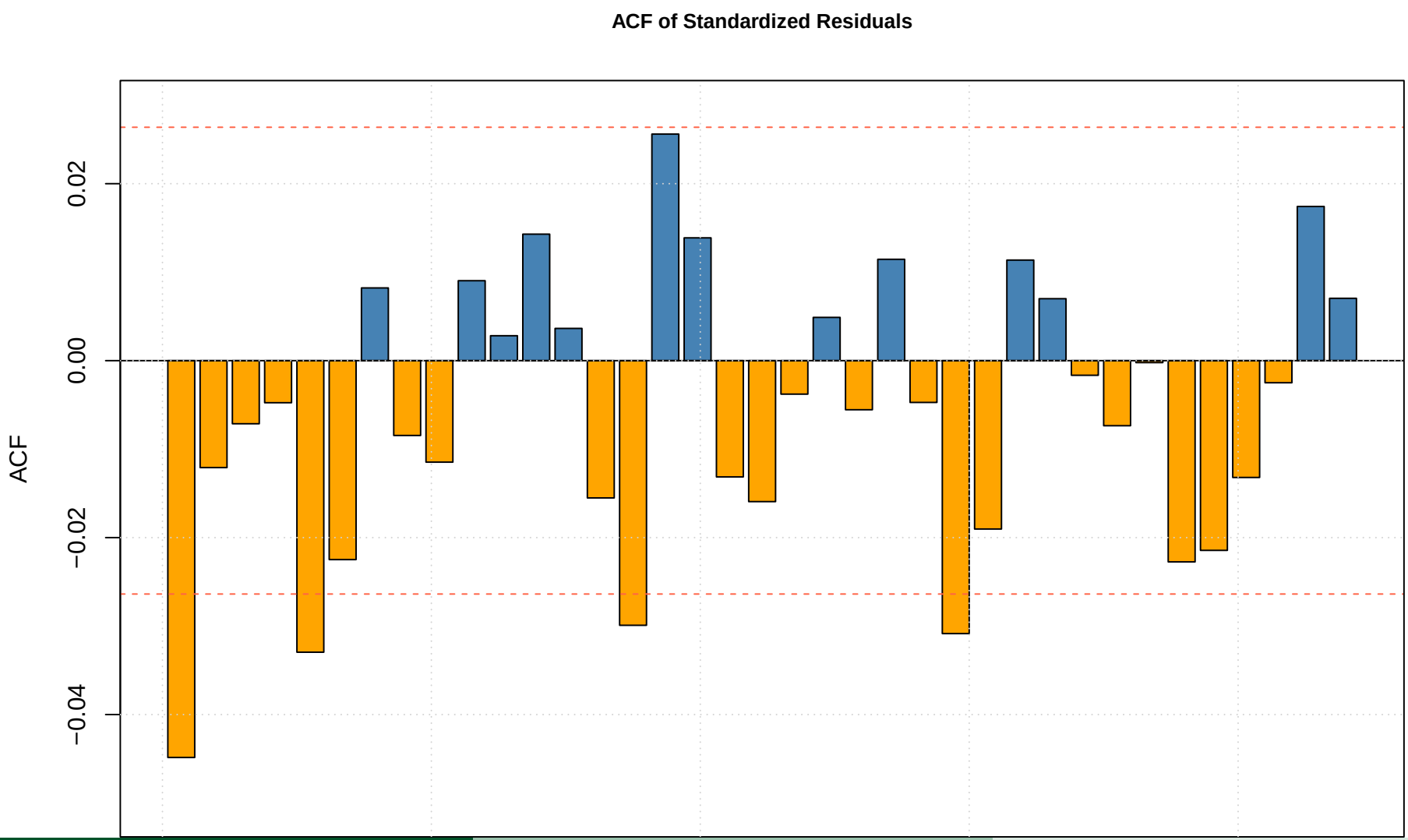
Fit Arima-sGARCH Model

```
# Results information criteria
infocriteria(sGARCH)
```

```
Akaike       -6.518885
Bayes        -6.512893
Shibata      -6.518886
Hannan-Quinn -6.516795
```

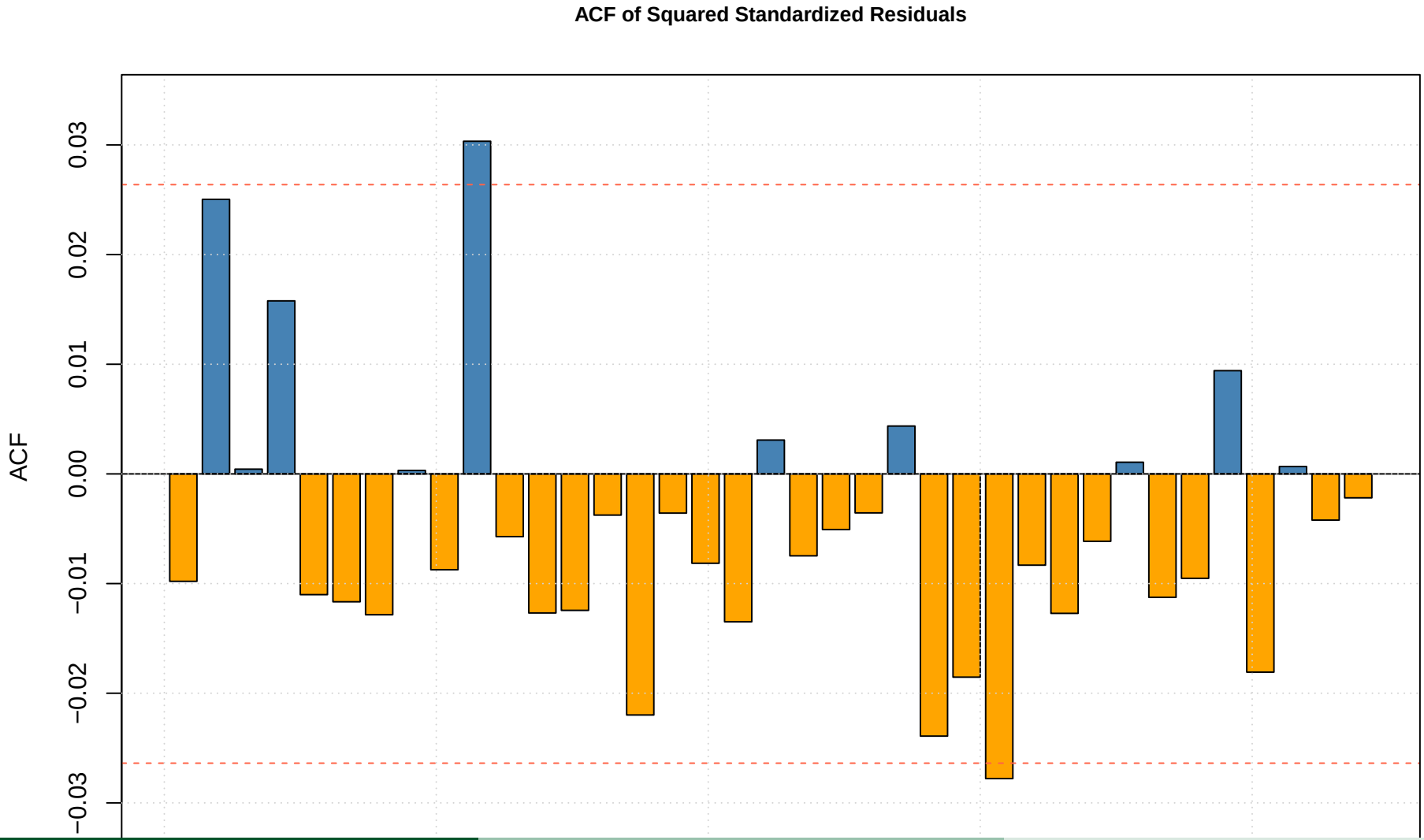
Residual Diagnostic

Standardized residual ACF



Residual Diagnostic

Standardaied Squared residuals ACF



Forecast for fitted model

```
*---------------------------------*
*       GARCH Model Forecast      *
*---------------------------------*
Model: sGARCH
Horizon: 5
Roll Steps: 0
Out of Sample: 0

0-roll forecast [T0=2021-12-10]:
       Series   Sigma
T+1 0.0007399 0.01151
T+2 0.0007399 0.01153
T+3 0.0007399 0.01155
T+4 0.0007399 0.01158
T+5 0.0007399 0.01160
```

Rolling Forecast for window size 1000

```
roll <- ugarchroll(spec = spec,
                   data = data,
                   n.ahead = 1,
                   n.start = 3000,
                   refit.every = 50,
                   refit.window = "moving",
                   solver = "hybrid",
                   window.size = 1000,
                   keep.coef = TRUE)
```

Refit in moving window where all previous data is used for the first estimation and then moved by a length equal to refit.every

Rolling Forecast for window size 1000

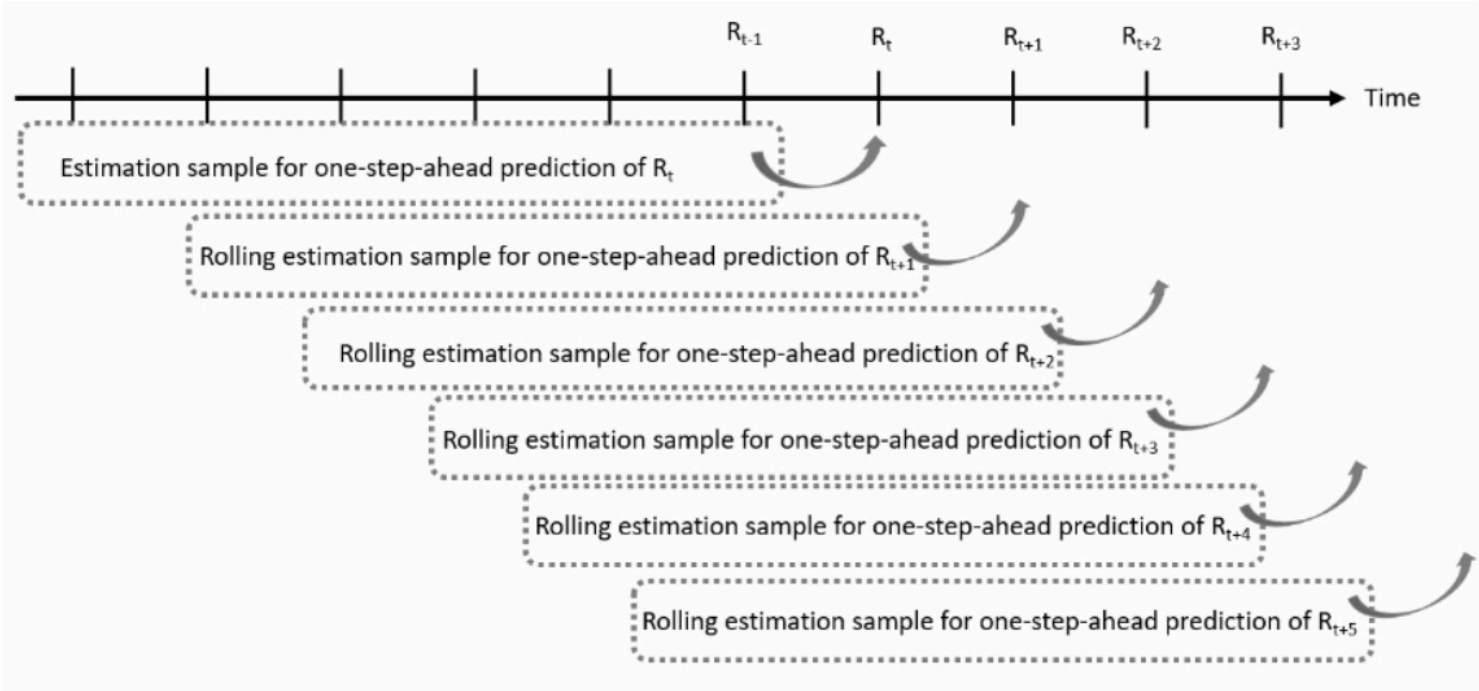
```
*-----------------------------------*
*              GARCH Roll           *
*-----------------------------------*
No.Refits        : 51
Refit Horizon    : 50
No.Forecasts     : 2521
GARCH Model      : sGARCH(1,1)
Distribution     : ged

Forecast Density:
                Mu  Sigma Skew  Shape Shape(GIG) Realized
2011-12-06 9e-04 0.0185    0 1.3032           0   0.0011
2011-12-07 9e-04 0.0175    0 1.3032           0   0.0020
2011-12-08 9e-04 0.0165    0 1.3032           0  -0.0211
2011-12-09 9e-04 0.0173    0 1.3032           0   0.0169
```

Rolling Forecast for window size 1000

refit.window

Moving window estimation

Only use a fixed number of the most return observations available at the time of prediction



Refit in moving window where all previous data is used for the first estimation and then moved by a length equal to refit.every. Another refit

Error Metircs

```
rugarch::report(roll, type = "fpm")
```

```
GARCH Roll Mean Forecast Performance Measures
------------------------------------------
Model        : sGARCH
No.Refits    : 51
No.Forecasts: 2521

       Stats
MSE 0.0001065
MAE 0.0065320
DAC 0.5518000
```

Section 4

RNN-LSTM Model

Recurrent Neural Network(RNN)

A **recurrent neural network** is a class of artificial neural network that uses sequential or time series data. Unlike Feedforward Neural Network, RNN allows the output from some nodes to affect subsequent input to the same nodes by using connections between nodes to create cycles. As a result, the hidden layers produce the outputs with the input information and prior "memory" received from previous learning.

Unroll RNN

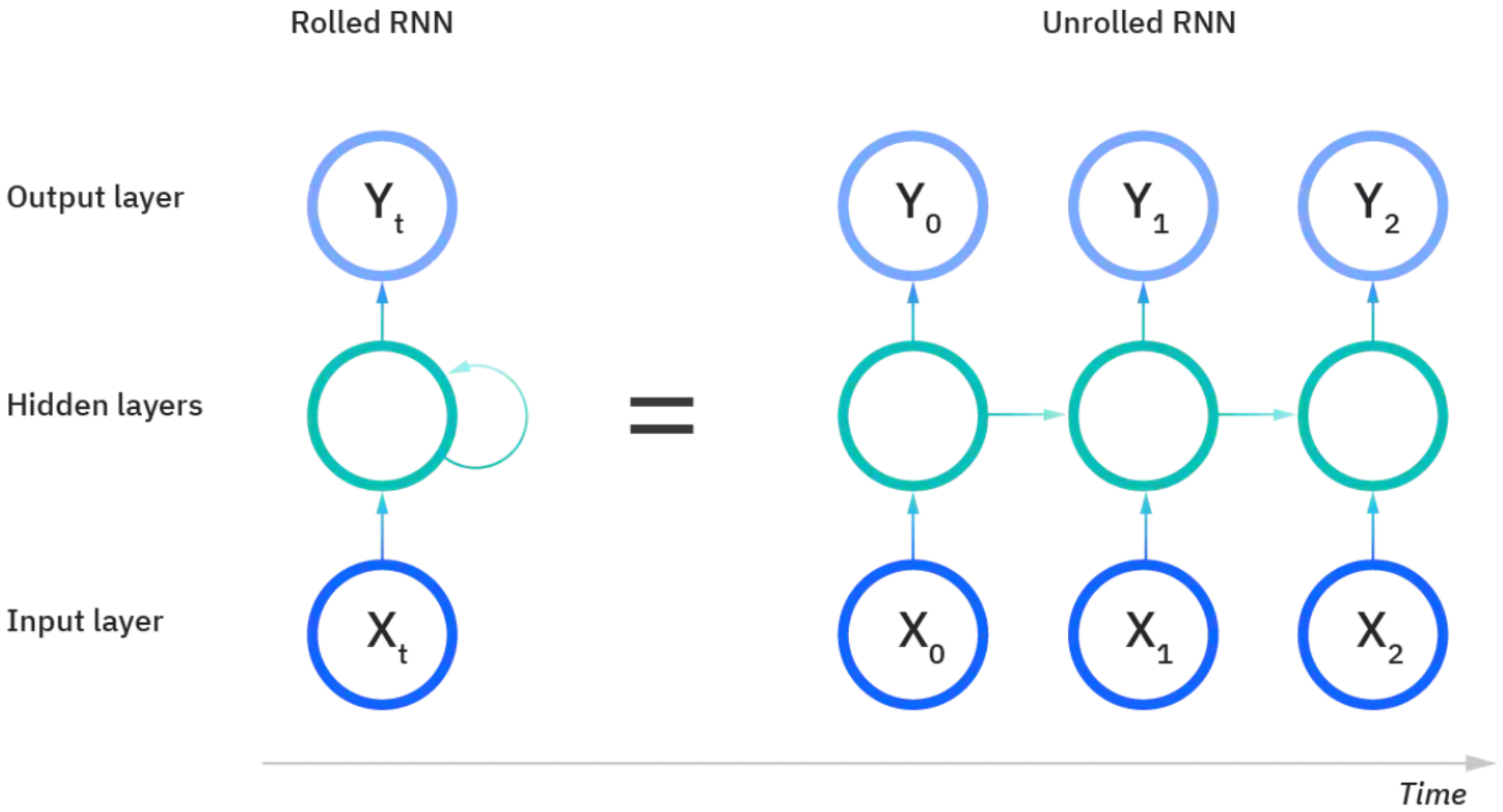


Figure 1: Rolled RNN and Unrolled RNN

Recurrent Neural Network(RNN)

Another distinguish characteristic of RNN is that they share parameters across each layer of the network. Unlike feedforward neural networks having individual weight and bias for each node in one layer, recurrent neural networks share the same weight parameter within each layer. However, these weights are still adjusted during the processes of backpropagation and gradient descent to facilitate reinforcement learning.

In feedforward neural network, backpropagation algorithm was used to calculate the gradient with respect to the weights. Recurrent neural network, on the other side, leverage backpropagation through time (BPTT) algorithm to determine the gradient as BPTT is specific to sequential data.

Activation Functions

In neural networks, an activation function determines whether a neuron should be activated and typically maps the input to $[0, 1]$ or $[-1, 1]$. The followings are two of the most commonly used activation functions and will be adopted later:

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Tanh (Hyperbolic tangent)

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

ReLU (Rectified Linear Unit) Activation Function

$$R(x) = max(0, x)$$

Long Short-term Memory (LSTM)

Long short-term memory network, usually known as LSTM, is a specific RNN architecture first introduced by Sepp Hochreiter and Juergen Schmidhuber as a solution to vanishing gradient problem. Recall with an RNN, similar with human reading a book and remembering what happened in the earlier chapter, it remembers the previous information and use it for processing the current input. The shortcoming of the NN is that it is not able to remember long term dependencies due to the vanishing gradient. The LSTM is designed to alleviate and avoid such issues.

The LSTM consists of three parts:

- **Forget Gate**: Choose whether the information coming from the previous time stamp should be remembered or can be forgotten
- **Input Gate**: Learn new information from the input to this cell
- **Output Gate**: Passes the updated information tot the next time stamp

Forget Gate

In an LSTM cell, the cell first need to decide if the information from previous time stamp should be kept or forgotten. The equation of the forget gate is:

$$f_t = \sigma(W_f \cdot [x_t, h_{t-1}] + b_f)$$

Where

- x_t = input to the current time stamp
- h_{t-1} = hidden state of the previous time stamp
- W_f = weight matrix associated with hidden state
- b_f = constant

After that, a `sigmoid` function is applied over f_t and make it a number between 0 and 1. Then f_t is multiplied with the previous cell state. If $f_t = 0$, the network will forget everything from the previous time stamp while $f_t = 1$ represents that the network will remember everything.

Input Gate and new information

Next we decide what new information we will store in the cell state. First, the input gate decides which values we'll update with `sigmoid` activation function:

$$i_t = \sigma(W_i \cdot [x_t, h_{t-1}] + b_i)$$

Where

- W_t = weight matrix of the input associated with hidden state

Next, the new information is sent through a `tanh` layer to create the new candidate values:

$$\tilde{C}_t = tanh(W_C \cdot [x_t, h_{t-1}] + b_C)$$

New Cell State C_t

With previous work, the LSTM cell now updates the new cell state for the current time stamp as:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The current cell state C_t combines how much we decide to remember from the previous cell state C_{t-1} scaled by the forget gate and how much we wish to take in from the new current input $\tilde{C}_t$ scaled by the input gate.

Output Gate

Finally, the cell needs to decide what it is going to output and by how much. The filter of the output is the output gate, with the following equation:

$$o_t = \sigma(W_o \cdot [x_t, h_{t-1}] + b_o)$$

The equation of the output gate is very similar with the forget gate and the input gate. Then, we push the cell state C_t through the `tanh` activation function to maintain the value staying in between -1 and 1, and multiply it by the output gate:

$$h_t = o_t * tanh(C_t)$$

Overall Module

The previous steps conclude the architecture of the LSTM. The whole process can be summarized and displayed as the following:

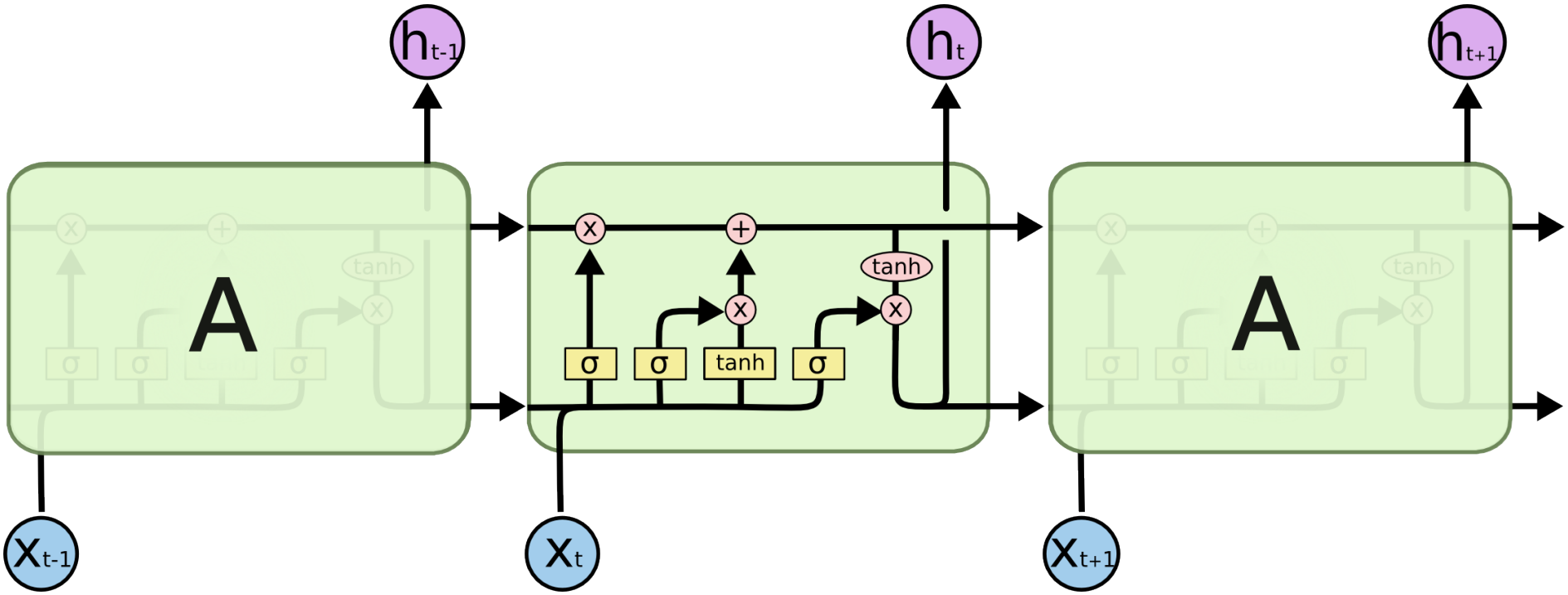


Figure 2: LSTM Chain

Section 5

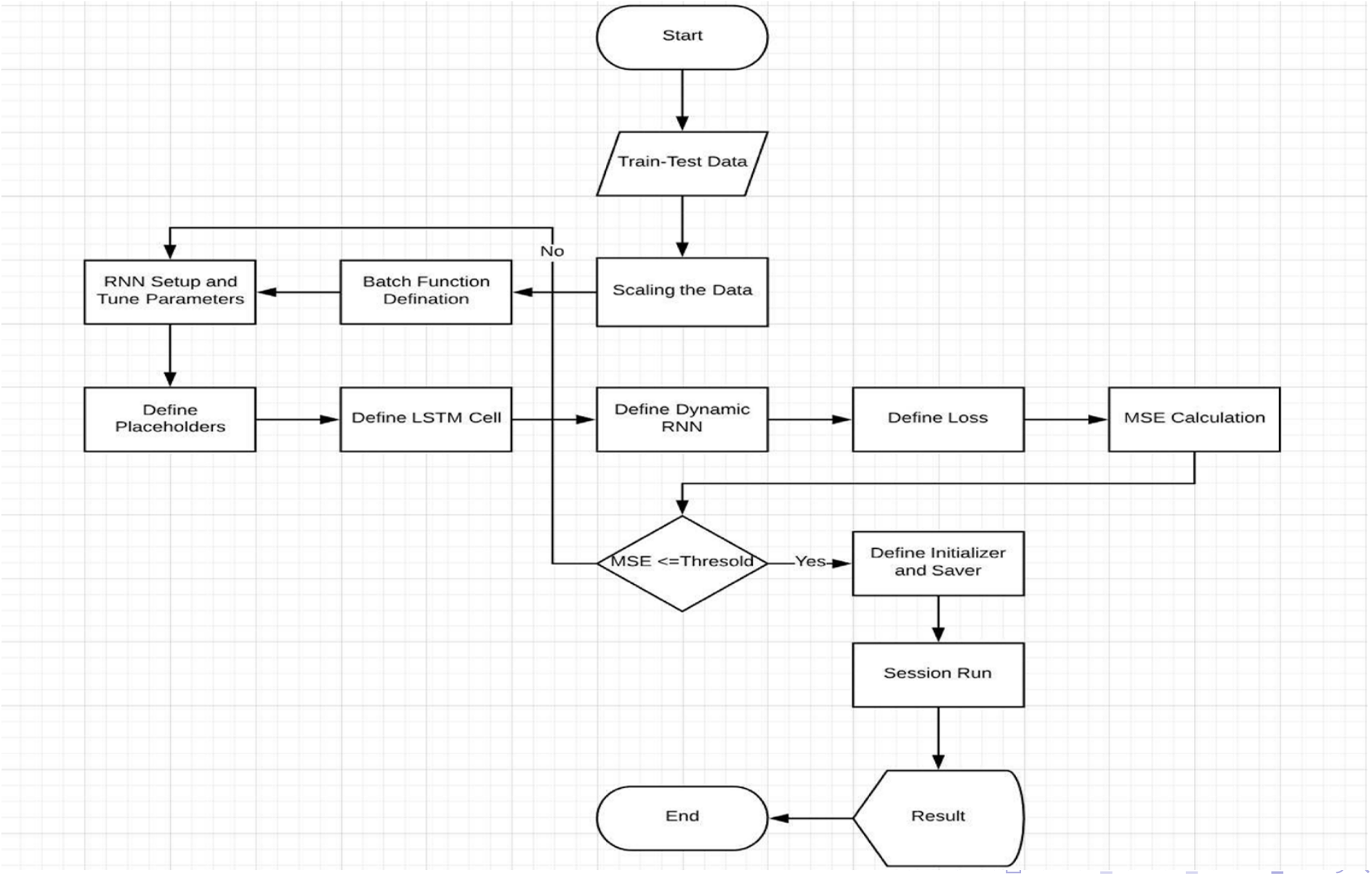
RNN-LSTM Model - Coding

Application in Tensorflow

We implement the LSTM by using the package `tensorflow`. The LSTM model in the `keras` package of `tensorflow` follows our previous description by using the `tanh` function as the activation function and the `ReLU` function as the recurrent activation function.

Let's start with Reproduce.

Reproduce - Understand Article



Reproduce - Understand Article

Data: Stock price of five companies from "2013-01-01" to "2017-12-31"

"In particular, a total of 1259 days of data was collected. The first 1008 data points are for the first 4 years (2013 to 2016) and the last 251 data are for each day in 2017."

Data Process: Scaling: Min-Max Scale

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

We will need to rescale data back after predicting.

Reproduce - Understand Article

Parameter

```
#Tuning parameters
num_inputs = 1
num_time_steps = 1
num_neurons = 500
num_outputs = 1
learning_rate = 0.002
num_train_iterations = 4000
```

LSTM Cell

```
#LSTM Cell
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicLSTMCell(num_units=num_neurons, activation=tf.nn.relu),
    output_size=num_outputs)
```

Loss

```
#Loss
loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train = optimizer.minimize(loss)
```

Reproduce - Understand Article

```
for i in range(1,251,+1):
    hold
    print("Pass ",i )

    train_set=stock[i:1008+i]

    #sklearn
    from sklearn.preprocessing import MinMaxScaler
    scaler = MinMaxScaler()
    train_scaled = scaler.fit_transform(train_set)
    #test_scaled = scaler.transform(test_set)
    #batch Defination
    #MSE CAlculation
    with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:
        sess.run(init)

        for iteration in range(num_train_iterations):

            X_batch, y_batch = next_batch(train_scaled,batch_size,num_time_steps)
            sess.run(train, feed_dict={X: X_batch, y: y_batch})

            if iteration % 100 == 0:

                mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
                print(iteration , "\tMSE:", mse)


        saver.save(sess, "./rnn_stock_time_series_model")
    #Session Run
    with tf.Session() as sess:

        saver.restore(sess, "./rnn_stock_time_series_model")

        train_seed = list(train_scaled[-num_time_steps:])
        for iteration in range(num_time_steps):
            X_batch = np.array(train_seed[-num_time_steps:]).reshape(1, num_time_steps, 1)
            y_pred = sess.run(outputs, feed_dict={X: X_batch})
            train_seed.append(y_pred[0, -1, 0])
    #Print Train_seed
    train_seed
    #results
    results = scaler.inverse_transform(np.array(train_seed[num_time_steps:]).reshape(num_time_steps,1))
    hold[i,1]=results
```

Figure 4: The Neural Network Loop

Reproduce - Data

```
start_date = "2013-01-01"; end_date = "2017-12-31"

nflx_prices <- tq_get("NFLX", get = "stock.prices",
                      from = start_date, to = end_date)

stock <- nflx_prices |> arrange(date) |>
  select(date, adjusted) |> column_to_rownames('date')

head(stock, 5)
```

```
            adjusted
2013-01-02 13.14429
2013-01-03 13.79857
2013-01-04 13.71143
2013-01-07 14.17143
2013-01-08 13.88000
```

Reproduce - Train-Test Split and Scale

```
# Max Min Scale
max_min_scale <- function(x, name = 'value') {
  df <- data.frame((x- min(x)) /(max(x)-min(x)))
  colnames(df) <- name; df}
max_min_scale_reverse <- function(y, x) {
  min(x) + y * (max(x)-min(x))}

train_set <- stock[1:1008,]
test_set <- stock[1009:nrow(stock),]
train_scaled <- train_set %>% max_min_scale
head(train_scaled, 3)
```

```
        value
1 0.000000000
2 0.005554876
3 0.004815041
```

Reproduce - Data Preperation for Fitting

```
data_prep <- function(scaled_data, prediction = 1, lag = 22){
  x_data <- t(sapply(1:(dim(scaled_data)[1] - lag - predictio
                     function(x) scaled_data[x: (x + lag - 1), 1
  x_arr <- array(data = as.numeric(unlist(x_data)),
                 dim = c(nrow(x_data), lag, 1))
  y_data <- t(sapply((1 + lag):(dim(scaled_data)[1] - predicti
                       function(x) scaled_data[x: (x + predicti
  y_arr <- array(data = as.numeric(unlist(y_data)),
                 dim = c(length(y_data), prediction, 1))
  return(list(x = x_arr, y = y_arr))}

x_train = data_prep(train_scaled)$x
y_train = data_prep(train_scaled)$y
cat('x_dim: (', dim(x_train), ') || y_dim: (', dim(y_train),
```

```
x_dim: ( 986 22 1 ) || y_dim: ( 986 1 1 )
```

Reproduce - LSTM Setup

```
num_neurons = 50
learning_rate = 0.002
# Define LSTM
get_model <- function(){
  model <- keras_model_sequential() |>
    layer_lstm(units = num_neurons,
               batch_input_shape = c(1, 22, 1),
               activation = 'relu',
               stateful = TRUE) |>
    layer_dense(units = 1)

  model %>% compile(loss = 'mse', metrics = 'mae',
    optimizer=optimizer_adam(learning_rate=learning_rate))
}
```

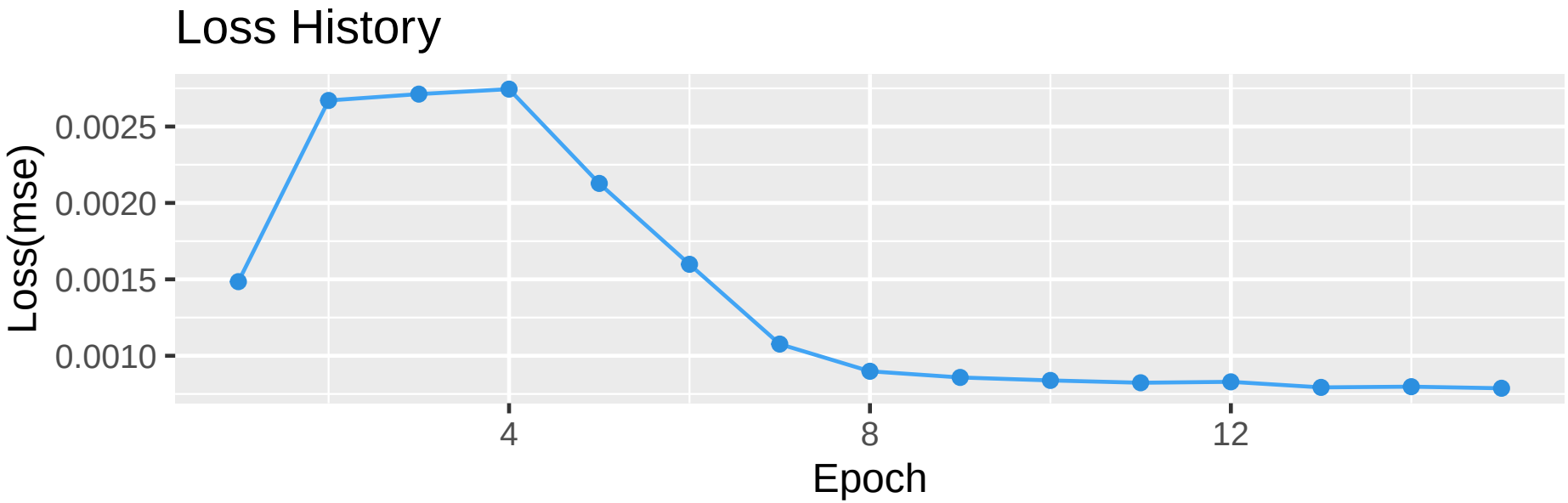
Reproduce - LSTM Structure

```
lstm_model <- get_model()
summary(lstm_model)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                                Output Shape
=================================================================
 lstm (LSTM)                                 (1, 50)
 dense (Dense)                               (1, 1)
=================================================================
Total params: 10,451
Trainable params: 10,451
Non-trainable params: 0
_________________________________________________________________
```

Reproduce - LSTM Fitting

```
set_random_seed(1209)
lstm_model <- get_model()
lstm_model %>% fit(x = x_train, y = y_train, batch_size = 1,
    epochs = 15, verbose = 2, shuffle = FALSE) -> history
```



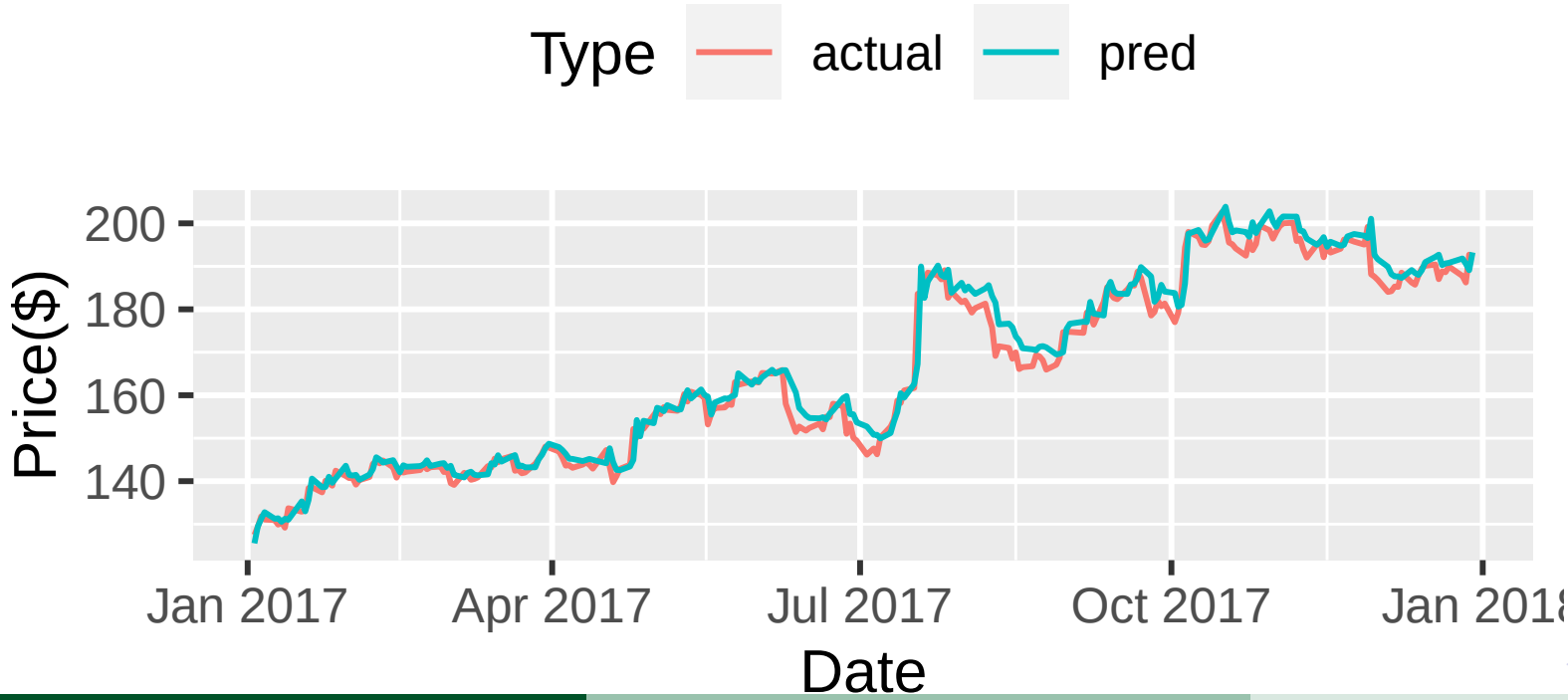
Reproduce - LSTM Result

```
model_prediction_lstm <- function(test_len, whole_data, model)
  t <- test_len # test size
  Tt <- nrow(whole_data) # whole data size
  t_start <- Tt - t + 1 # test start date
  predictions <- vector(length = t)
  for (i in 1:t){
    n <- t_start - 1 + i
    test_set = pull(whole_data)[(n-22):n]
    test_scaled <- test_set %>% max_min_scale
    x_test = data_prep(test_scaled)$x
    y_pred_scaled <- stats::predict(model, x_test, verbose = 0
    y_pred <- max_min_scale_reverse(y_pred_scaled, test_set)
    predictions[i] <- y_pred
  }
  pred_df <- data.frame('date' = rownames(whole_data)[t_start:
    'pred' = predictions, 'actual' = pull(whole_data)[t_start:
```

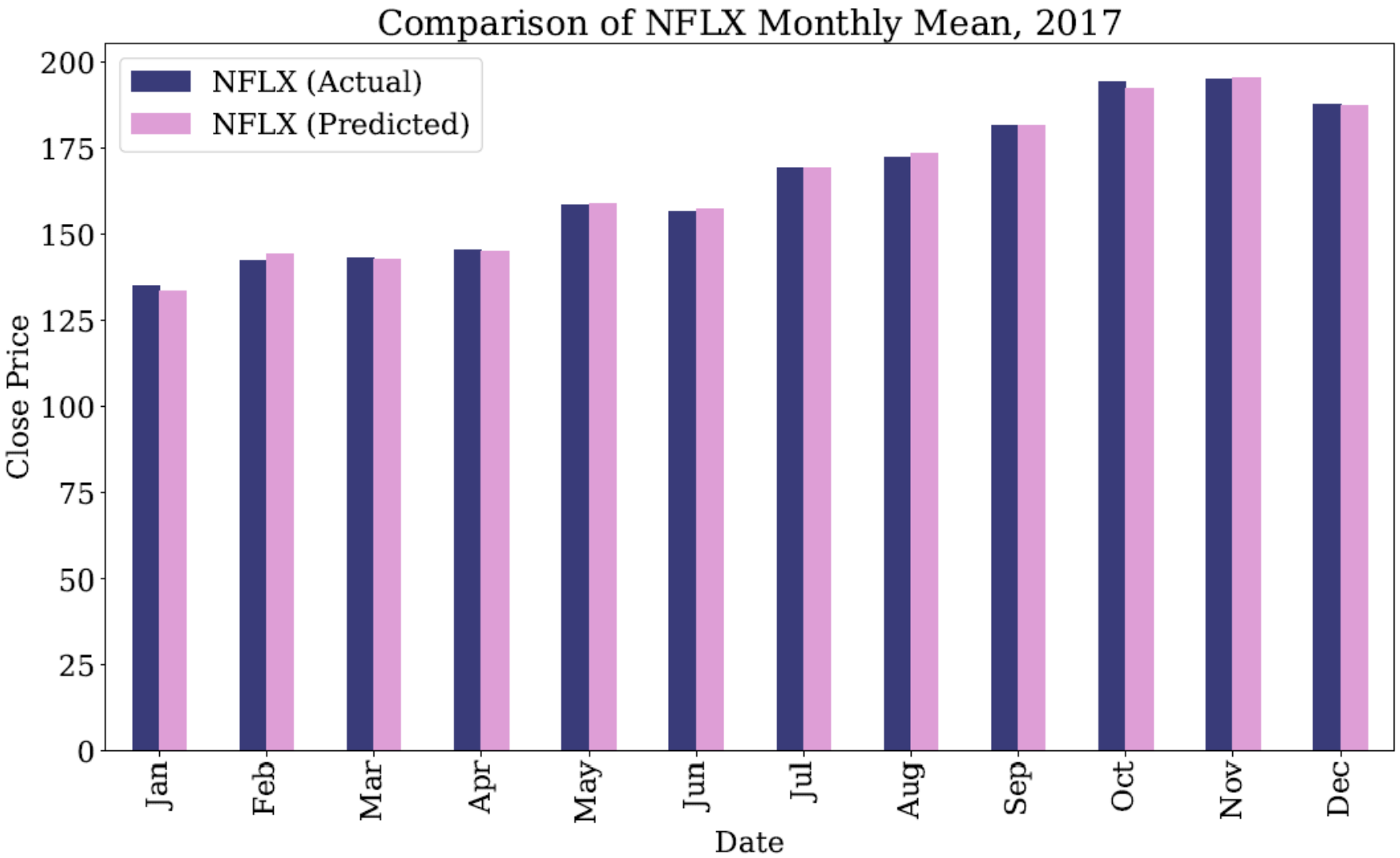
Reproduce - LSTM Result

```
model_prediction_lstm(length(test_set), stock, lstm_model) ->
pred_df %>% head(2)
```

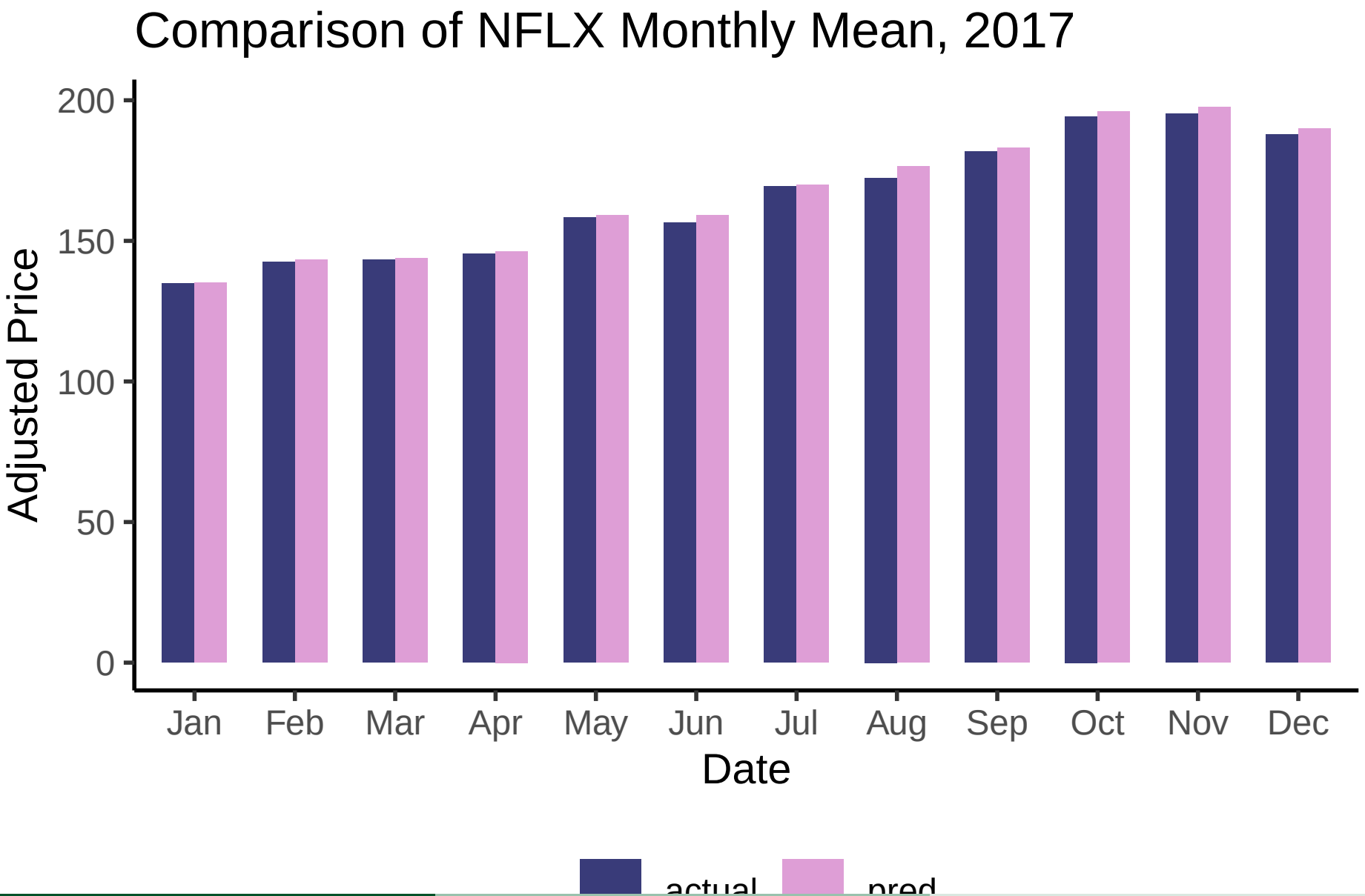
```
        date     pred actual
1 2017-01-03 125.5396 127.49
2 2017-01-04 129.4479 129.41
```



Result Comparison - Article Result



Result Comparison - Our Result



Result Comparison - |% of Error|

```
comparison %>% cbind('Our Result' = c(mean(per_err), StdDev(pe
                      StdDev(per_err)^2, quantile(per_err), rs
```

|% of Error|	Article	Our Result
1	mean	2.0000	1.4553167271
2	std	1.5400	1.3480666801
3	variabce	2.3800	1.8172837741
4	min	0.0000	0.0002329987
5	X25_percentile	0.8000	0.5238703257
6	X50_percentile	1.6500	1.1002221473
7	X75_percentile	2.9600	1.9550019530
8	max	9.2500	8.8855777088
9	R_squared	0.9589	1.0556739845

New Model - Get S&P 500, Set val

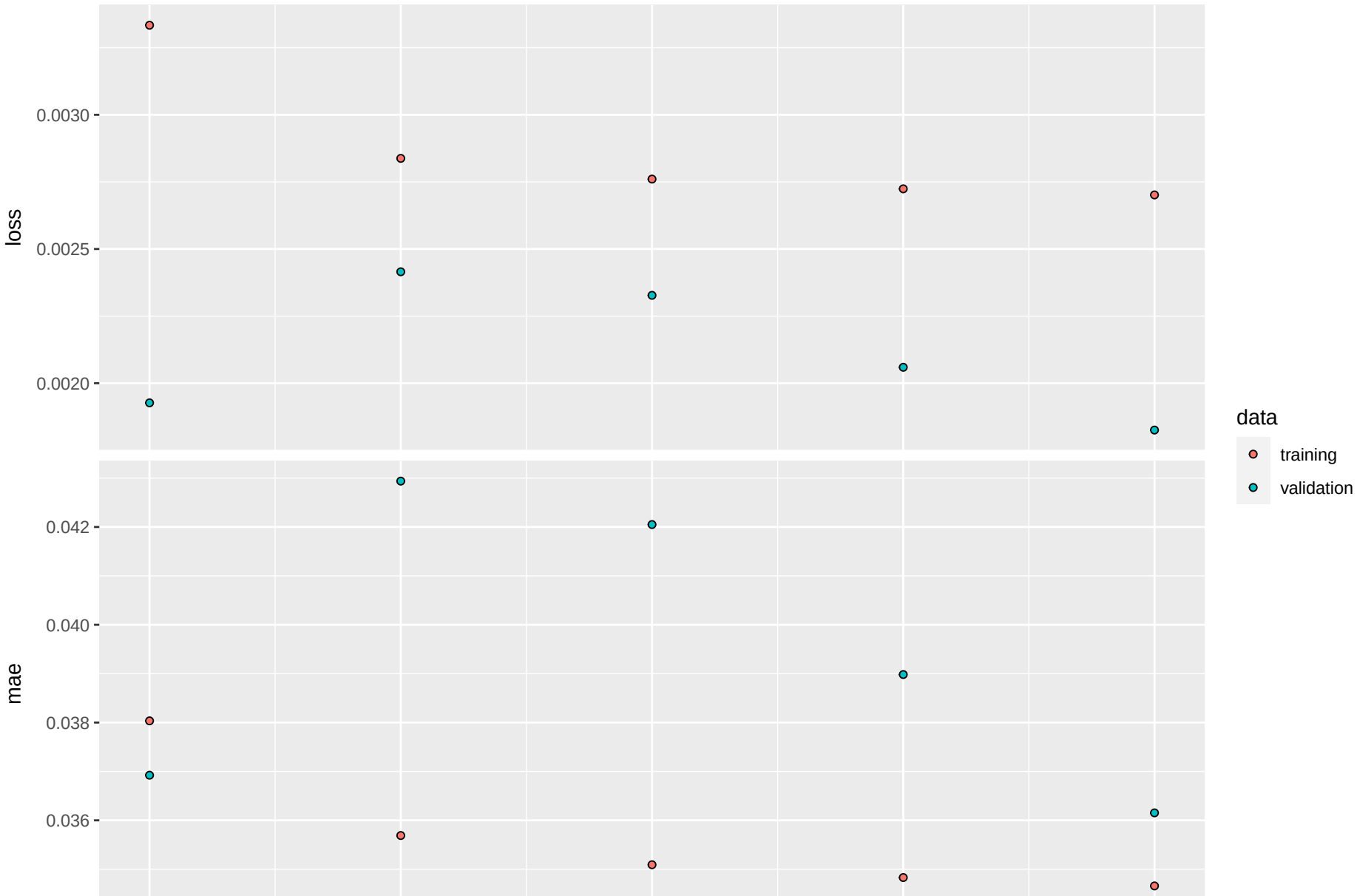
```
sp_prices <- tq_get("^GSPC", get = "stock.prices",
                    from = "2000-01-01", to = "2021-12-31")
return <- sp_prices |> arrange(date) |>
  mutate(return = log(adjusted) - log(lag(adjusted))) |>
  drop_na(return) %>% select(date, return)
# scale
return$return <- max_min_scale(return$return) %>% pull(value)
# train test split
val_date <- "2019-01-01"; test_date <- '2020-01-01'
train_data_n <- return |> filter(date <= val_date) %>% column
val_data_n <- return |> filter(date < test_date & date > val_
test_data_n <- return |> filter(date >= test_date) %>% column
# data preparation
x_train_new = data_prep(train_data_n)$x; y_train_new = data_pr
x_val_new = data_prep(val_data_n)$x; y_val_new = data_prep(va
```

New Model - Fit LSTM

```
validation_data <- list(x_val_new, y_val_new)
set_random_seed(1209)
lstm_model2 <- get_model()
lstm_model2 %>% fit(x = x_train_new, y = y_train_new, batch_si
    epochs = 5, verbose = 2, shuffle = FALSE,
    validation_data = validation_data) -> history2

pred_df2 <- model_prediction_lstm(nrow(test_data_n), return_ne
head(pred_df2, 3)
```

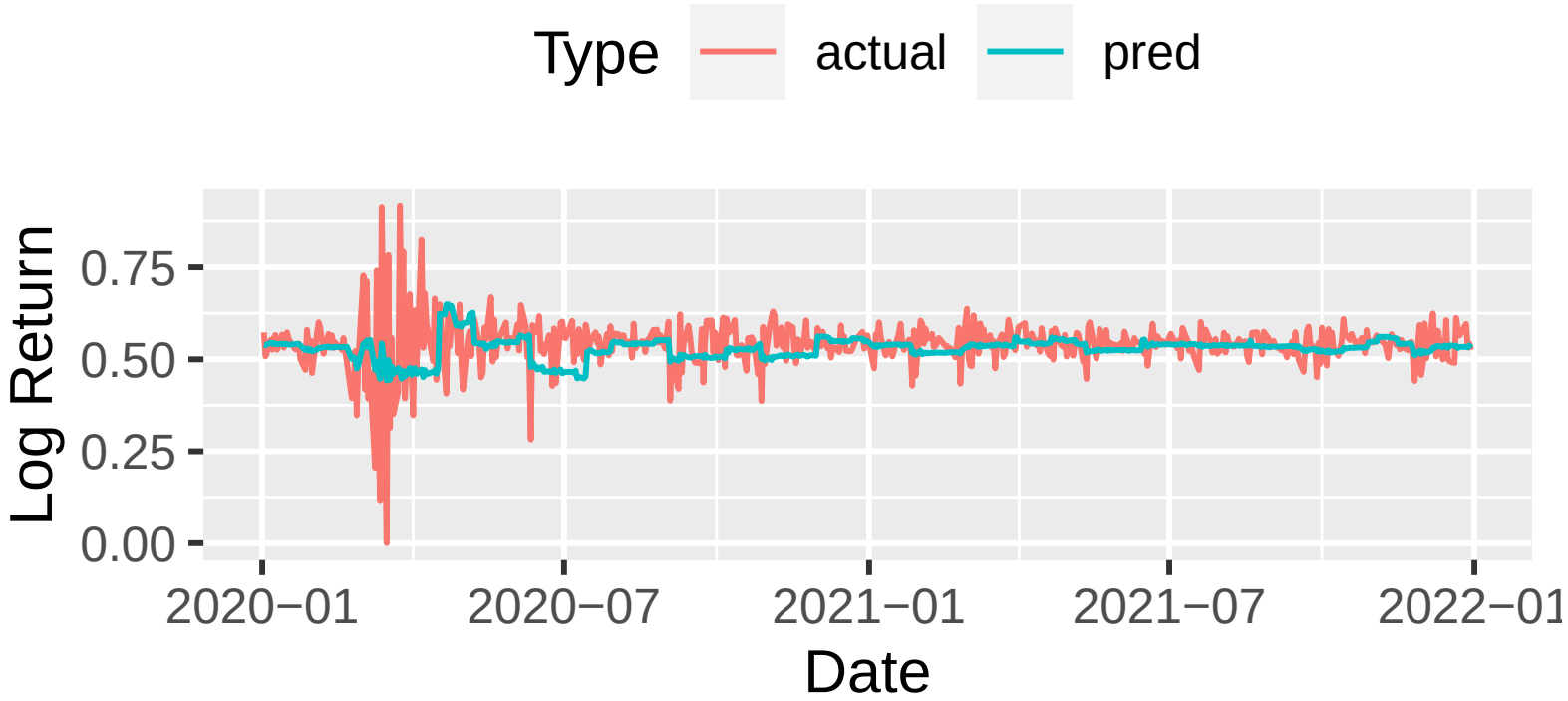
```
        date      pred    actual
1 2020-01-02 0.5391658 0.5732809
2 2020-01-03 0.5384816 0.5082420
3 2020-01-06 0.5453627 0.5529763
```

New Model - Loss History



New Model - Prediction

	MSE	MAE	RMSE	Rsq
1	0.00517588	0.04533519	0.01206681	0.2078431



Section 6

Model Comparison - ARIMA and LSTM

Error Metrics

	Error Metrics	sGARCH	RNN_LSTM
1	MSE	0.0001065	0.00517588
2	MAE	0.0065320	0.04533519

Thank Your

Any Questions?