PSTAT 231 Final Project: Predicting S&P 500 Stock Prices with RNN models

Ashwath Ekambaram

2024-06-09

Contents

Introduction

This project focuses on predicting the S&P 500 stock index's price using time series data from the previous five days. The number of days may be adjusted to achieve the highest accuracy without overfitting. The prediction will indicate the percentile change in the stock price, either an increase or decrease. This approach is suited for regression, as it deals with quantitative variables and does not involve classification. Predicting the behavior pattern of the stock market is a common algorithm many data scientists are trying to replicate as an accurate model can lead to great profitability.

The project will demonstrate the use of a recurrent neural network (RNN) for stock price prediction. The goals are threefold: to build an RNN model utilizing S&P 500 time series data, to explain the RNN's architecture and training process compared to standard feed-forward networks, and to compare the RNN's performance with other predictive models, including a feed-forward network and a SARIMA time-series model. The

ultimate aim is to create a predictive tool that can potentially generate profits for its users by accurately forecasting stock behavior. # Exploratory Data Analysis

Data Description

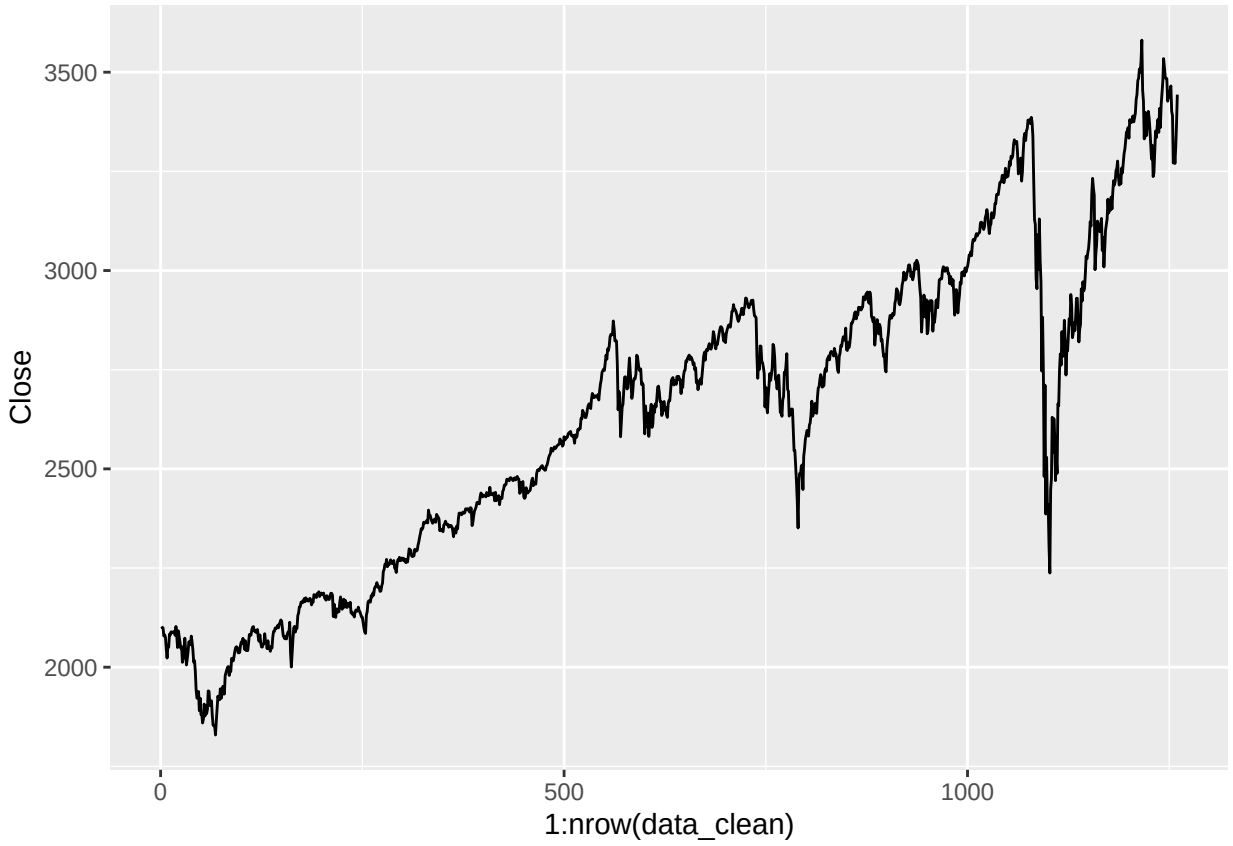
The data I used will be from [Kaggle, Henry Han] (https://www.kaggle.com/datasets/henryhan117/sp-500-historical-data/). This dataset includes daily observations of data regarding the S&P 500 index.

```
data_raw <- read.csv("data/spx_raw.csv")

data_raw$Date <- as.Date(data_raw$Date)
data_clean <- data_raw %>% filter(Date >= '2015-11-04')
head(data_clean)
```

```
##         Date    Open    High     Low   Close Adj.Close     Volume
## 1 2015-11-04 2110.60 2114.59 2096.98 2102.31   2102.31 4078870000
## 2 2015-11-05 2101.68 2108.78 2090.41 2099.93   2099.93 4051890000
## 3 2015-11-06 2098.60 2101.91 2083.74 2099.20   2099.20 4369020000
## 4 2015-11-09 2096.56 2096.56 2068.24 2078.58   2078.58 3882350000
## 5 2015-11-10 2077.19 2083.67 2069.91 2081.72   2081.72 3821440000
## 6 2015-11-11 2083.41 2086.94 2074.85 2075.00   2075.00 3692410000
```

Taking a look at the data that has been filtered.

```
# Looking at the data
ggplot(data_clean, aes(x = 1:nrow(data_clean), y = Close)) + geom_line()
```

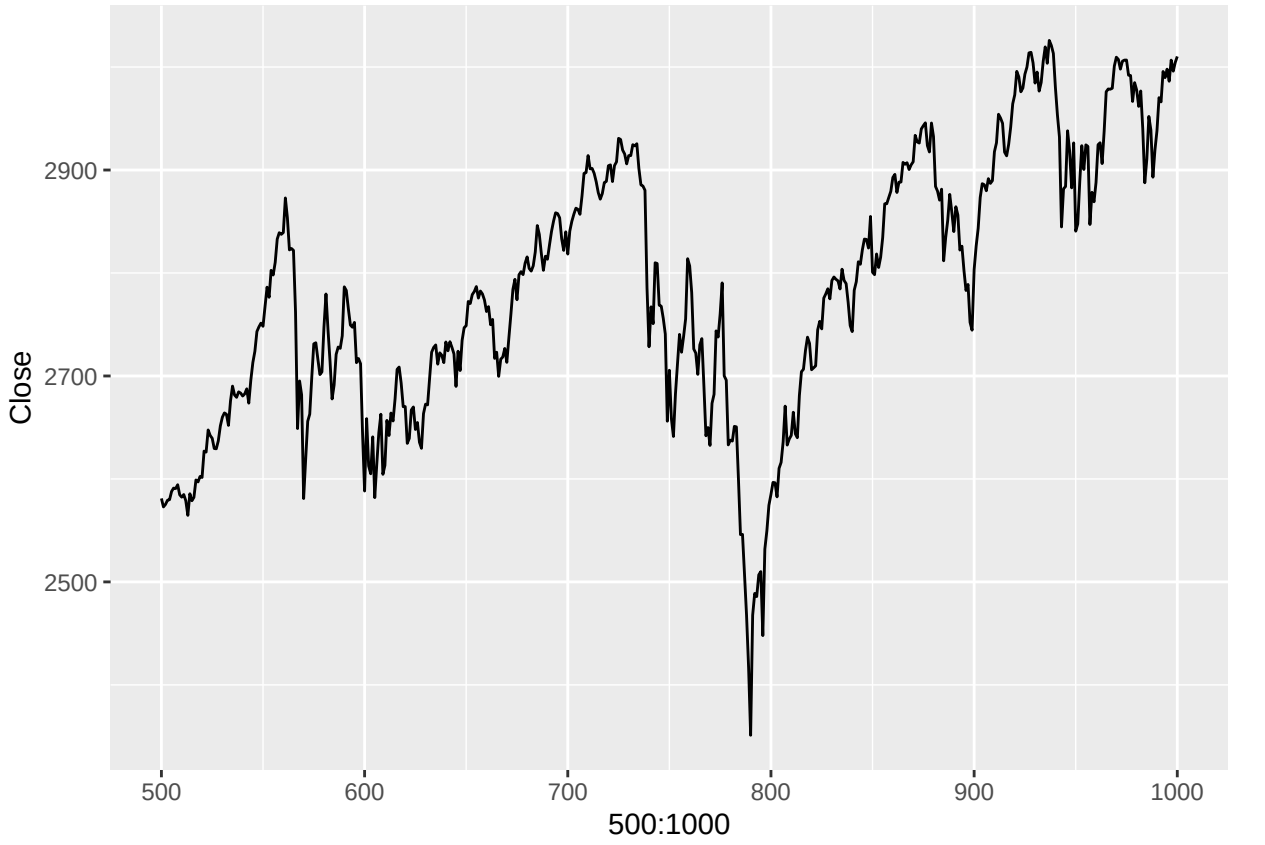



```
#data_clean
```

Taking a look at the portion I am interested in for training. Using 500 to 1000(29%) for training and 1000 to

1200 for testing(71%).

```
ggplot(data_clean[(500:1000),], aes(x = 500:1000, y = Close)) + geom_line()
```



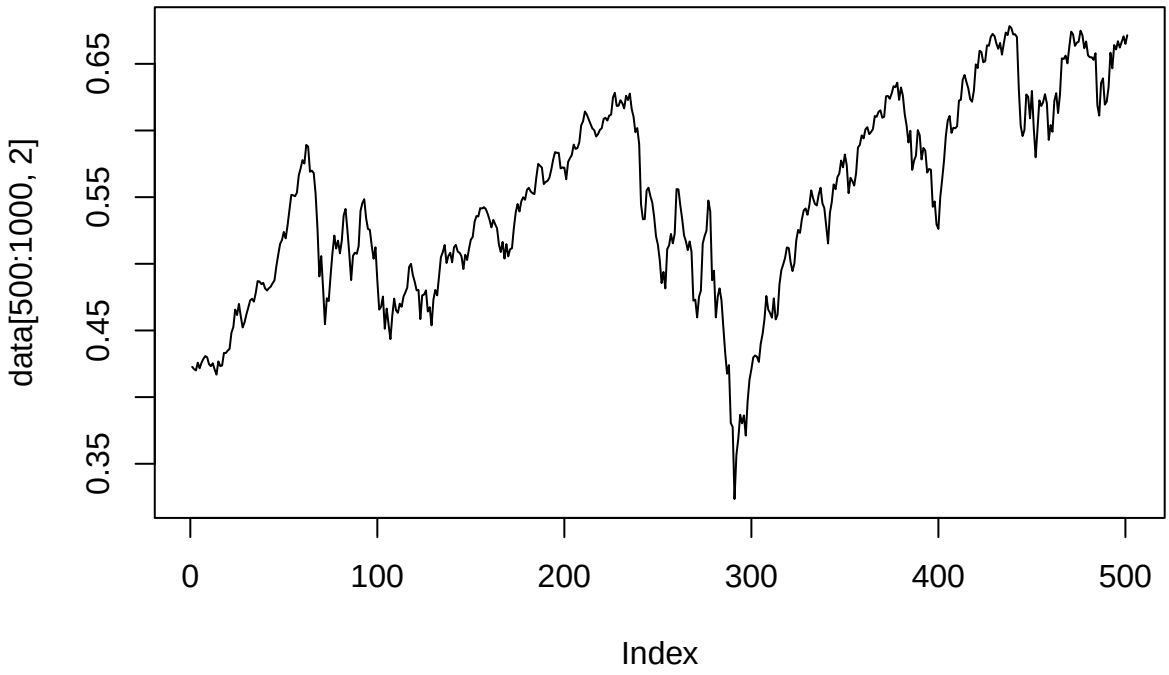
Stratifyied Sampling

Firstly, I cleaned the data so that each column is standardized, and normalized.

Normalizing and standardizing data before feeding it into LSTM models serves to create consistency in scale and distribution among the input features. Normalization rescales data to a common range, between 0 and 1, ideal for handling varying feature ranges of dynamic data- in this case time series stock prices. Meanwhile, standardization centers data around mean zero and unit variance, which is beneficial for dealing with diverse feature units- such as volume vs price. These preprocessing techniques help LSTM model convergence by ensuring all inputs fall within manageable ranges, acting as a safeguard from dominant features from skewing learning. They also enhance model robustness by minimizing the impact of outliers and avoiding numerical instabilities during optimization, ultimately allowing the model to capture complex temporal patterns within sequential data.

```
### Standardize and Normalize data
data <- data.matrix(data_clean[,-1])

# Standardize data --> center around mean for each column
mean <- apply(data, 2, mean)
std <- apply(data, 2, sd)
data <- scale(data, center = mean, scale = std)

# Normalize, create func. --> make between 0 and 1 for activation function
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
```

```
}

max <- apply(data, 2, max)
min <- apply(data, 2, min)

# Normalize data & get rid of adjusted close
data <- apply(data, 2, normalize)

# Shape of standardized, normalized data is the same as before
plot(data[500:1000, 2], type = 'l')
```



The Generator Function

Next is the generator function used to generate training sequences out of sequential stock price data.

This function creates a specified amount of training observations (batch_size), each the length of the lookback value defined by the user. The step argument specifies when to resample the sequence, after every step timepoints. This allows for sequences of training data observations within the min/max indices to be specified. It also allows the model to learn on multiple sequential observations of data rather than the one sequence of time series in the raw data. The output of the generator function is a list of two arrays. The first array contains the set of sequences (# rows = batch_size) of training data (# columns = lookback) for each feature in the dataset. The second array contains the target or the response for each set of sequences (each row) with a shape of (batch_size x 1).

In all, the generator function is crucial to ensure that our model is able to learn on multiple sequences of sequential stock price data, rather than the single sequence of raw data. In a standard feed forward neural network, that does not utilize time-series data, training data would look like a flat representation of data. It could be in the form of individual pixel values of an image, or raw data values that are independent of one another.

LSTM RNN Model using all Features

Recurrent Neural Networks (RNNs) are a class of neural networks designed to process sequential data by retaining information in memory across time steps. They do well in handling time-series data or sequences due to their ability to maintain context and dependencies within the data. Long Short-Term Memory networks (LSTMs), which I will be using in this project, represent a specialized type of RNN, engineered to address the vanishing gradient problem and better capture long-term dependencies. LSTMs possess a unique architecture, incorporating memory cells, input, forget, and output gates to regulate the flow of information, allowing them to selectively retain or forget information over extended sequences.

Calling the Generator Function and Creating Training Sequences

```
### calling generator function
setwd(scripts_dir)
source('generator.R')
lookback <- 5
step <- 1
delay <- 0
batch_size <- 500

set.seed(123)
train_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 500,
  max_index = 1000,
  shuffle = FALSE,
  step = step,
  batch_size = batch_size)

train_gen_data <- train_gen()
```

Architecture

Here, I will talk about the model architecture of the LSTM model that uses all the features in the dataset to predict the adjusted closing price.

A simple dense input layer in Keras treats each input independently without considering any sequential relationships, making it better suited for tabular or non-sequential data. In contrast, an LSTM input layer in Keras is best suited for sequential or time-series data, being able to preserve and represent temporal dependencies across sequences, allowing it to better capture patterns within sequential data for tasks like natural language processing, time-series forecasting, and sequential prediction.

The following code constructs a sequential model where the first layer is an LSTM with 64 units, followed by three densely connected layers with varying units and activation functions, resulting in a final output layer with a single unit, for regression tasks. This structure of an LSTM model in the code represents a network capable of learning patterns in sequential data while mitigating the challenges of vanishing gradients commonly encountered in traditional RNNs.

```
### Setting up model
model <- keras_model_sequential() %>%
  layer_lstm(units = 64, input_shape = c(lookback, dim(data)[-1]))  %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
```

```
  layer_dense(units = 1)

model %>% compile(loss = 'mean_squared_error', optimizer = 'adam',metrics='mse')

history <-
  model %>% fit (
    train_gen_data[[1]],train_gen_data[[2]],
    batch_size = 128,
    epochs = 50,
    validation_split = 0.1,
    use_multiprocessing = T
  )
```

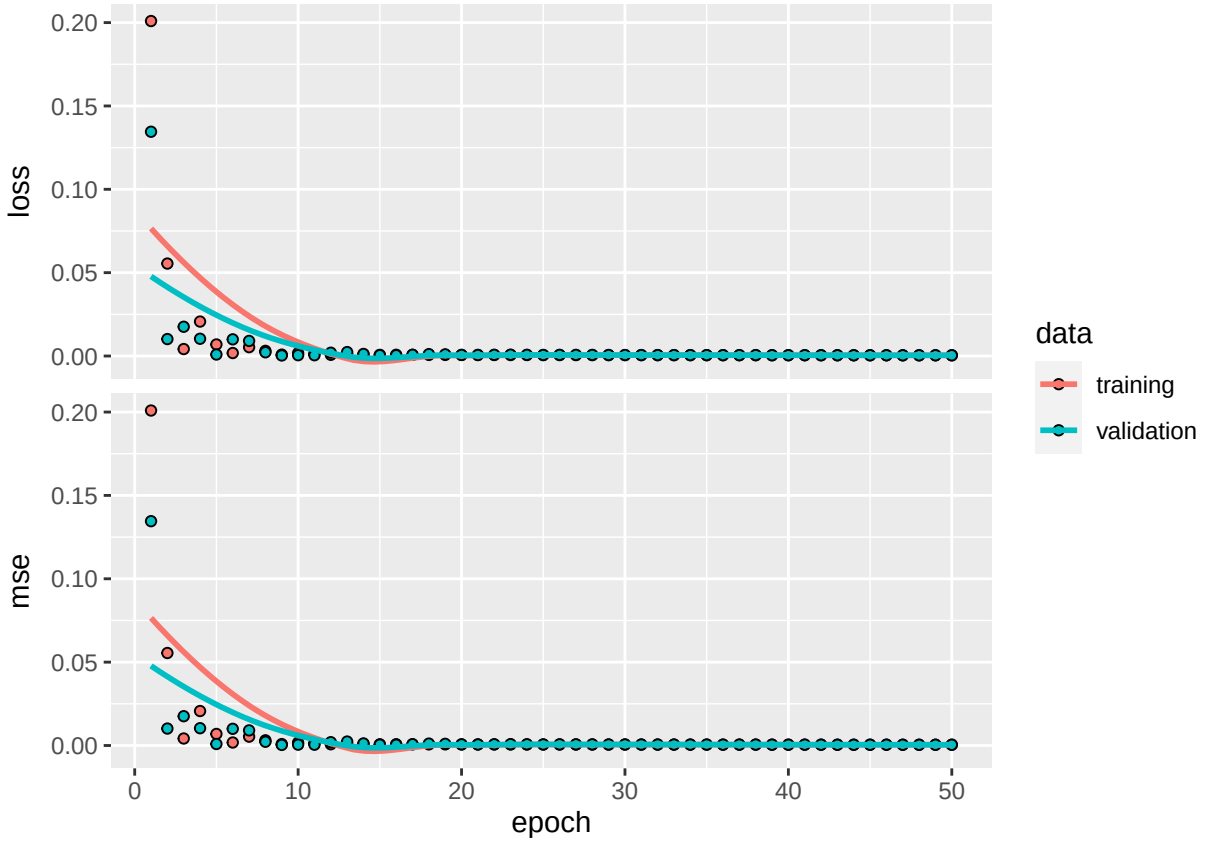
```
## Epoch 1/50
## 4/4 - 2s - loss: 0.2009 - mse: 0.2009 - val_loss: 0.1345 - val_mse: 0.1345 - 2s/epoch - 620ms/step
## Epoch 2/50
## 4/4 - 0s - loss: 0.0555 - mse: 0.0555 - val_loss: 0.0102 - val_mse: 0.0102 - 65ms/epoch - 16ms/step
## Epoch 3/50
## 4/4 - 0s - loss: 0.0042 - mse: 0.0042 - val_loss: 0.0176 - val_mse: 0.0176 - 41ms/epoch - 10ms/step
## Epoch 4/50
## 4/4 - 0s - loss: 0.0207 - mse: 0.0207 - val_loss: 0.0104 - val_mse: 0.0104 - 43ms/epoch - 11ms/step
## Epoch 5/50
## 4/4 - 0s - loss: 0.0069 - mse: 0.0069 - val_loss: 9.0204e-04 - val_mse: 9.0204e-04 - 42ms/epoch - 10r
## Epoch 6/50
## 4/4 - 0s - loss: 0.0018 - mse: 0.0018 - val_loss: 0.0100 - val_mse: 0.0100 - 42ms/epoch - 10ms/step
## Epoch 7/50
## 4/4 - 0s - loss: 0.0053 - mse: 0.0053 - val_loss: 0.0091 - val_mse: 0.0091 - 43ms/epoch - 11ms/step
## Epoch 8/50
## 4/4 - 0s - loss: 0.0031 - mse: 0.0031 - val_loss: 0.0022 - val_mse: 0.0022 - 41ms/epoch - 10ms/step
## Epoch 9/50
## 4/4 - 0s - loss: 8.4154e-04 - mse: 8.4154e-04 - val_loss: 3.2850e-04 - val_mse: 3.2850e-04 - 42ms/epo
## Epoch 10/50
## 4/4 - 0s - loss: 0.0017 - mse: 0.0017 - val_loss: 4.9493e-04 - val_mse: 4.9493e-04 - 42ms/epoch - 11r
## Epoch 11/50
## 4/4 - 0s - loss: 0.0015 - mse: 0.0015 - val_loss: 4.6888e-04 - val_mse: 4.6888e-04 - 43ms/epoch - 11r
## Epoch 12/50
## 4/4 - 0s - loss: 7.2914e-04 - mse: 7.2914e-04 - val_loss: 0.0020 - val_mse: 0.0020 - 47ms/epoch - 12r
## Epoch 13/50
## 4/4 - 0s - loss: 9.9189e-04 - mse: 9.9189e-04 - val_loss: 0.0024 - val_mse: 0.0024 - 45ms/epoch - 11r
## Epoch 14/50
## 4/4 - 0s - loss: 9.3444e-04 - mse: 9.3444e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 43ms/epoch - 11r
## Epoch 15/50
## 4/4 - 0s - loss: 7.0262e-04 - mse: 7.0262e-04 - val_loss: 4.8703e-04 - val_mse: 4.8703e-04 - 41ms/epo
## Epoch 16/50
## 4/4 - 0s - loss: 7.4982e-04 - mse: 7.4982e-04 - val_loss: 4.5475e-04 - val_mse: 4.5475e-04 - 42ms/epo
## Epoch 17/50
## 4/4 - 0s - loss: 7.0128e-04 - mse: 7.0128e-04 - val_loss: 7.5042e-04 - val_mse: 7.5042e-04 - 42ms/epo
## Epoch 18/50
## 4/4 - 0s - loss: 6.5714e-04 - mse: 6.5714e-04 - val_loss: 0.0011 - val_mse: 0.0011 - 43ms/epoch - 11r
## Epoch 19/50
## 4/4 - 0s - loss: 6.6677e-04 - mse: 6.6677e-04 - val_loss: 9.9922e-04 - val_mse: 9.9922e-04 - 42ms/epo
## Epoch 20/50
## 4/4 - 0s - loss: 6.3975e-04 - mse: 6.3975e-04 - val_loss: 6.6888e-04 - val_mse: 6.6888e-04 - 42ms/epo
## Epoch 21/50
```

```
## 4/4 - 0s - loss: 6.2500e-04 - mse: 6.2500e-04 - val_loss: 6.1507e-04 - val_mse: 6.1507e-04 - 45ms/ep
## Epoch 22/50
## 4/4 - 0s - loss: 6.1187e-04 - mse: 6.1187e-04 - val_loss: 7.2810e-04 - val_mse: 7.2810e-04 - 43ms/ep
## Epoch 23/50
## 4/4 - 0s - loss: 6.0099e-04 - mse: 6.0099e-04 - val_loss: 8.0404e-04 - val_mse: 8.0404e-04 - 43ms/ep
## Epoch 24/50
## 4/4 - 0s - loss: 5.9339e-04 - mse: 5.9339e-04 - val_loss: 7.1388e-04 - val_mse: 7.1388e-04 - 43ms/ep
## Epoch 25/50
## 4/4 - 0s - loss: 5.8372e-04 - mse: 5.8372e-04 - val_loss: 6.3805e-04 - val_mse: 6.3805e-04 - 42ms/ep
## Epoch 26/50
## 4/4 - 0s - loss: 5.7333e-04 - mse: 5.7333e-04 - val_loss: 6.8952e-04 - val_mse: 6.8952e-04 - 43ms/ep
## Epoch 27/50
## 4/4 - 0s - loss: 5.6416e-04 - mse: 5.6416e-04 - val_loss: 6.8493e-04 - val_mse: 6.8493e-04 - 42ms/ep
## Epoch 28/50
## 4/4 - 0s - loss: 5.5690e-04 - mse: 5.5690e-04 - val_loss: 6.6715e-04 - val_mse: 6.6715e-04 - 42ms/ep
## Epoch 29/50
## 4/4 - 0s - loss: 5.4942e-04 - mse: 5.4942e-04 - val_loss: 6.1617e-04 - val_mse: 6.1617e-04 - 45ms/ep
## Epoch 30/50
## 4/4 - 0s - loss: 5.4116e-04 - mse: 5.4116e-04 - val_loss: 6.2274e-04 - val_mse: 6.2274e-04 - 42ms/ep
## Epoch 31/50
## 4/4 - 0s - loss: 5.3409e-04 - mse: 5.3409e-04 - val_loss: 5.8734e-04 - val_mse: 5.8734e-04 - 46ms/ep
## Epoch 32/50
## 4/4 - 0s - loss: 5.2665e-04 - mse: 5.2665e-04 - val_loss: 5.7275e-04 - val_mse: 5.7275e-04 - 45ms/ep
## Epoch 33/50
## 4/4 - 0s - loss: 5.2030e-04 - mse: 5.2030e-04 - val_loss: 5.7146e-04 - val_mse: 5.7146e-04 - 46ms/ep
## Epoch 34/50
## 4/4 - 0s - loss: 5.1450e-04 - mse: 5.1450e-04 - val_loss: 5.4434e-04 - val_mse: 5.4434e-04 - 46ms/ep
## Epoch 35/50
## 4/4 - 0s - loss: 5.0924e-04 - mse: 5.0924e-04 - val_loss: 5.5304e-04 - val_mse: 5.5304e-04 - 42ms/ep
## Epoch 36/50
## 4/4 - 0s - loss: 5.0361e-04 - mse: 5.0361e-04 - val_loss: 5.1511e-04 - val_mse: 5.1511e-04 - 44ms/ep
## Epoch 37/50
## 4/4 - 0s - loss: 4.9621e-04 - mse: 4.9621e-04 - val_loss: 5.4174e-04 - val_mse: 5.4174e-04 - 43ms/ep
## Epoch 38/50
## 4/4 - 0s - loss: 4.9512e-04 - mse: 4.9512e-04 - val_loss: 5.6516e-04 - val_mse: 5.6516e-04 - 43ms/ep
## Epoch 39/50
## 4/4 - 0s - loss: 4.9002e-04 - mse: 4.9002e-04 - val_loss: 4.7587e-04 - val_mse: 4.7587e-04 - 43ms/ep
## Epoch 40/50
## 4/4 - 0s - loss: 4.8302e-04 - mse: 4.8302e-04 - val_loss: 4.8149e-04 - val_mse: 4.8149e-04 - 43ms/ep
## Epoch 41/50
## 4/4 - 0s - loss: 4.7837e-04 - mse: 4.7837e-04 - val_loss: 5.1786e-04 - val_mse: 5.1786e-04 - 43ms/ep
## Epoch 42/50
## 4/4 - 0s - loss: 4.7488e-04 - mse: 4.7488e-04 - val_loss: 4.7662e-04 - val_mse: 4.7662e-04 - 42ms/ep
## Epoch 43/50
## 4/4 - 0s - loss: 4.6972e-04 - mse: 4.6972e-04 - val_loss: 4.5277e-04 - val_mse: 4.5277e-04 - 43ms/ep
## Epoch 44/50
## 4/4 - 0s - loss: 4.6643e-04 - mse: 4.6643e-04 - val_loss: 4.3960e-04 - val_mse: 4.3960e-04 - 43ms/ep
## Epoch 45/50
## 4/4 - 0s - loss: 4.6279e-04 - mse: 4.6279e-04 - val_loss: 4.5415e-04 - val_mse: 4.5415e-04 - 42ms/ep
## Epoch 46/50
## 4/4 - 0s - loss: 4.5970e-04 - mse: 4.5970e-04 - val_loss: 4.6275e-04 - val_mse: 4.6275e-04 - 43ms/ep
## Epoch 47/50
## 4/4 - 0s - loss: 4.5625e-04 - mse: 4.5625e-04 - val_loss: 4.2507e-04 - val_mse: 4.2507e-04 - 42ms/ep
## Epoch 48/50
```

```
## 4/4 - 0s - loss: 4.5324e-04 - mse: 4.5324e-04 - val_loss: 4.2019e-04 - val_mse: 4.2019e-04 - 44ms/ep
## Epoch 49/50
## 4/4 - 0s - loss: 4.5025e-04 - mse: 4.5025e-04 - val_loss: 4.3481e-04 - val_mse: 4.3481e-04 - 42ms/ep
## Epoch 50/50
## 4/4 - 0s - loss: 4.4887e-04 - mse: 4.4887e-04 - val_loss: 4.4384e-04 - val_mse: 4.4384e-04 - 43ms/ep
```

Looking at the results of history w/ validation split:

```
plot(history)
```



The plot shows that the training and validation loss and MSE follow each other closely. When both training and validation losses, including Mean Squared Error (MSE), closely align, it signifies that the model is likely achieving consistent performance on both the data it was trained on and unseen validation data. This alignment suggests that the model isn't exhibiting substantial overfitting. The convergence of training and validation losses indicates that the model's learning potentially isn't skewed towards the training data but rather captures underlying patterns that are consistent across different datasets.

Factors like increasing the validation split percentage so that the model is able to be tested on more unseen data, or modifying the batch size as to have the model be able to learn on more data at a time, could change change the dynamic between the training and validation loss/MSE.

Plotting Test Data vs. Predicted

```
batch_size_plot <- 120
lookback_plot <- 5
step_plot <- 1

set.seed(123)
```

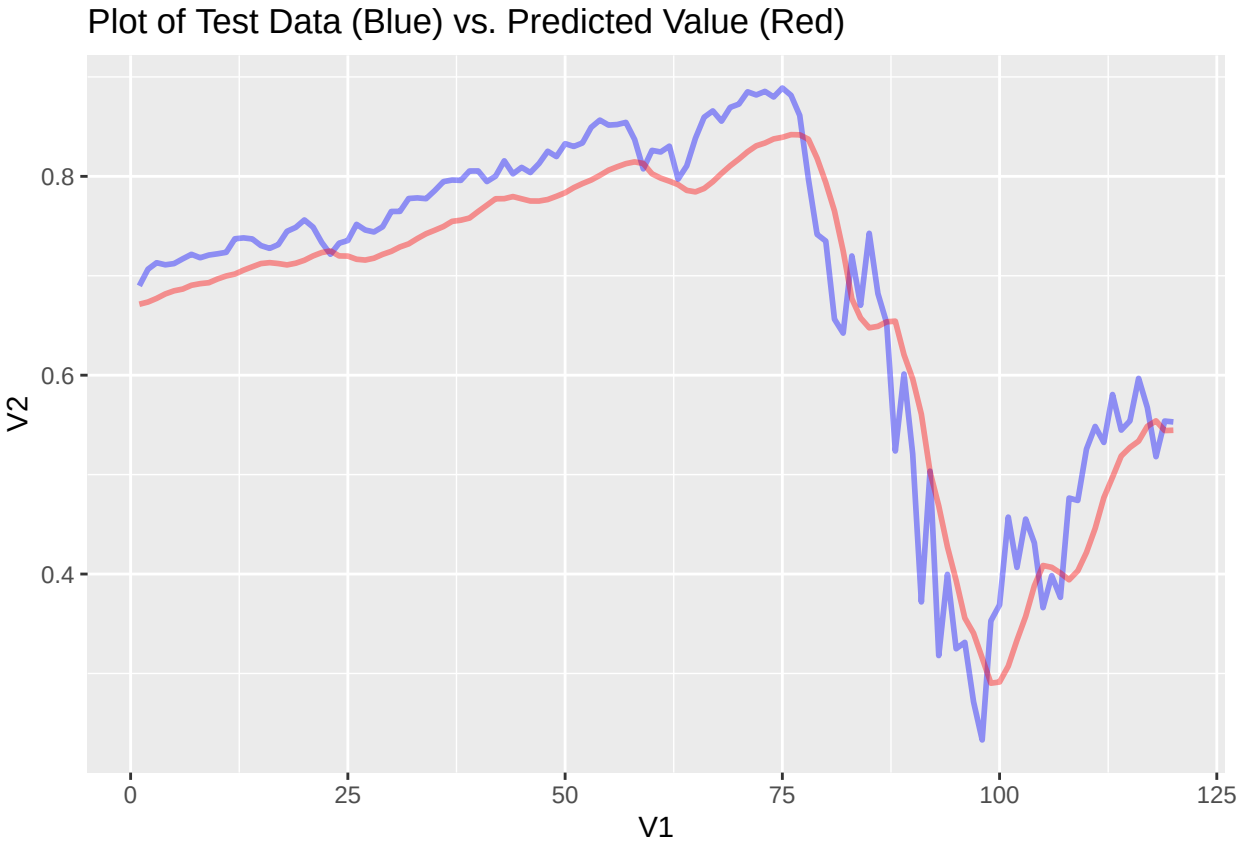
```
pred_gen <- generator(
  data,
  lookback = lookback_plot,
  delay = 0,
  min_index = 1000,
  max_index = 1200,
  shuffle = FALSE,
  step = step_plot,
  batch_size = batch_size_plot
)

pred_gen_data <- pred_gen()
```

Let's look at test data vs predicted values:

```
set.seed(123)
V1 = seq(1, length(pred_gen_data[[2]]))

# binds V1 as time step (actual) to actual sequence
plot_data <- as.data.frame(cbind(V1, pred_gen_data[[2]]))

inputdata <- pred_gen_data[[1]][,,]
dim(inputdata) <- c(batch_size_plot,lookback_plot, 6)

pred_out <- model %>% predict(inputdata)
```

```
## 4/4 - 0s - 386ms/epoch - 97ms/step
```

```
plot_data <- cbind(plot_data, pred_out[])

p <- ggplot(plot_data, aes(x = V1, y = V2)) + geom_line( colour = "blue", size = 1, alpha=0.4)
p <- p + geom_line(aes(x = V1, y = pred_out), colour = "red", size = 1 , alpha=0.4)
p <- p + labs(title = "Plot of Test Data (Blue) vs. Predicted Value (Red)")

p
```



Looking at a graph of the actual testing sequence data (blue) compared to a graph of the predicted test data. I can see that the LSTM network does relatively well in capturing the overall shape of the volatile stock data.

RMSE

Here, I take the RMSE of the selected testing interval, from index 1000 to 1200. The RMSE value will serve as the baseline metric that I can use to compare model performance. Since our model performs a regression task, the output is continuous, and I can use this metric reliably.

A note on RMSE:

Root Mean Square Error (RMSE) is a widely used metric to measure the average deviation of predicted values from actual observed values. It calculates the square root of the average of squared differences between predicted and true values, providing a single value that represents the typical magnitude of error in the model's predictions. Another key feature of RMSE is that it penalizes larger errors more significantly, offering a clear assessment of prediction accuracy.

```
rmse <- sqrt(mean((plot_data[,2] - plot_data[,3])^2))
rmse
```

```
## [1] 0.05417277
```

In the case of the LSTM RNN model using all the features in the dataset, I achieved an RMSE of around 0.06 (changes slightly due to random error with set.seed).

LSTM RNN Model Using only Adjusted Closing Price

Next, I had to redo our LSTM model using the feature I aim to predict, the adjusted closing price. This new model will use only use past values of adjusted closing price to predict the next adjusted close price using the

same LSTM logic as before.

Generator Function, Creating Training Sequences

In this case, I am generating training sequences with a lookback of 5 days of only the adjusted closing price value.

```
setwd(scripts_dir)
source('generator.R')
lookback <- 5
step <- 1
delay <- 0
batch_size <- 500

train_gen_1v <- generator_1v(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 500,
  max_index = 1000,
  shuffle = FALSE,
  step = step,
  batch_size = batch_size)

train_gen_data_1v <- train_gen_1v()
```

Architecture

I am using the same architecture as before, modifying the input shape of the first LSTM layer in our neural network. This modification allows for the univariate aspect of the model.

```
### Setting up model
model_1v <- keras_model_sequential() %>%
  layer_lstm(units = 64, input_shape = c(lookback, 1))  %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)


model_1v %>% compile(loss = 'mean_squared_error', optimizer = 'adam',metrics='mse')

history_1v <-
  model_1v %>% fit (
    train_gen_data_1v[[1]],train_gen_data_1v[[2]],
    batch_size = 128,
    epochs = 50,
    validation_split = 0.1,
    use_multiprocessing = T
  )
```

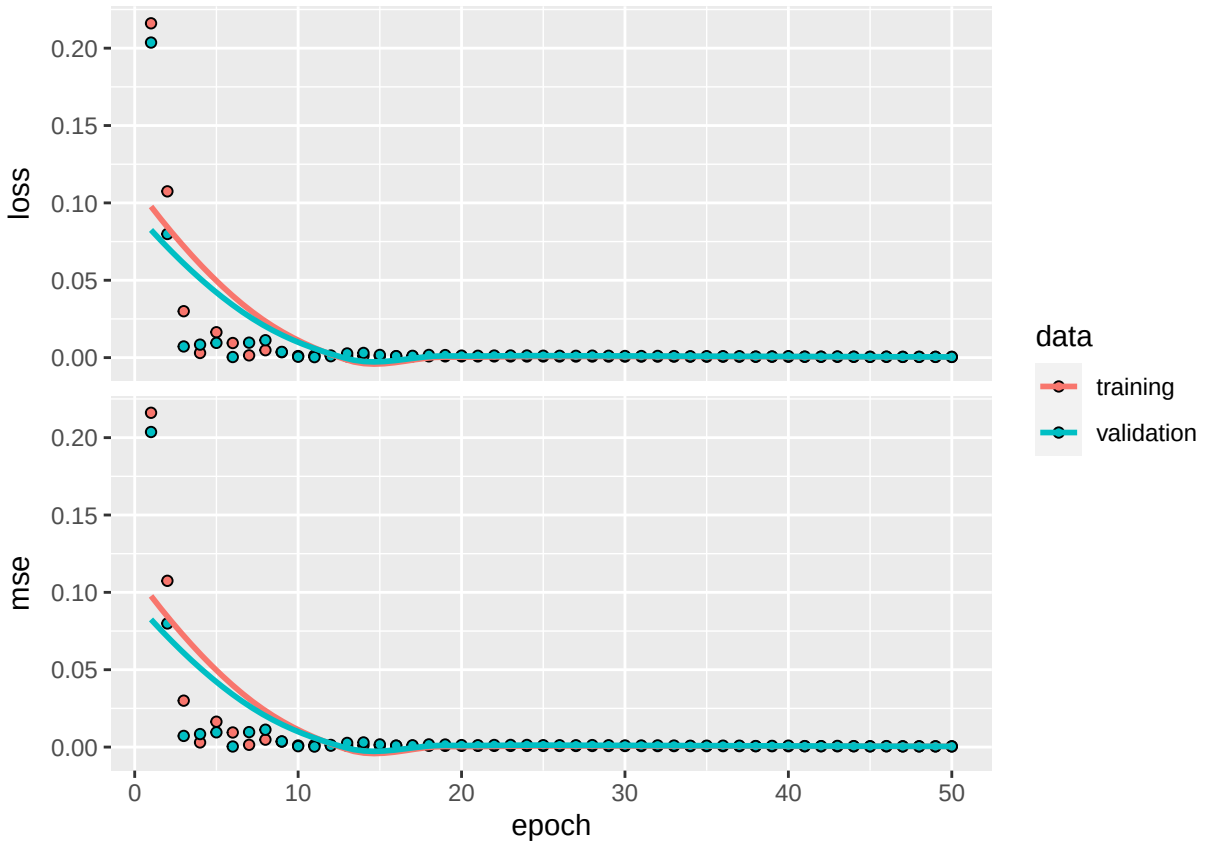
```
## Epoch 1/50
## 4/4 - 2s - loss: 0.2161 - mse: 0.2161 - val_loss: 0.2036 - val_mse: 0.2036 - 2s/epoch - 590ms/step
## Epoch 2/50
## 4/4 - 0s - loss: 0.1075 - mse: 0.1075 - val_loss: 0.0799 - val_mse: 0.0799 - 42ms/epoch - 11ms/step
## Epoch 3/50
```

```
## 4/4 - 0s - loss: 0.0300 - mse: 0.0300 - val_loss: 0.0072 - val_mse: 0.0072 - 42ms/epoch - 11ms/step
## Epoch 4/50
## 4/4 - 0s - loss: 0.0030 - mse: 0.0030 - val_loss: 0.0084 - val_mse: 0.0084 - 42ms/epoch - 11ms/step
## Epoch 5/50
## 4/4 - 0s - loss: 0.0164 - mse: 0.0164 - val_loss: 0.0095 - val_mse: 0.0095 - 42ms/epoch - 10ms/step
## Epoch 6/50
## 4/4 - 0s - loss: 0.0094 - mse: 0.0094 - val_loss: 3.7662e-04 - val_mse: 3.7662e-04 - 42ms/epoch - 10
## Epoch 7/50
## 4/4 - 0s - loss: 0.0014 - mse: 0.0014 - val_loss: 0.0097 - val_mse: 0.0097 - 41ms/epoch - 10ms/step
## Epoch 8/50
## 4/4 - 0s - loss: 0.0048 - mse: 0.0048 - val_loss: 0.0112 - val_mse: 0.0112 - 42ms/epoch - 10ms/step
## Epoch 9/50
## 4/4 - 0s - loss: 0.0035 - mse: 0.0035 - val_loss: 0.0037 - val_mse: 0.0037 - 42ms/epoch - 11ms/step
## Epoch 10/50
## 4/4 - 0s - loss: 0.0010 - mse: 0.0010 - val_loss: 5.6584e-04 - val_mse: 5.6584e-04 - 43ms/epoch - 11
## Epoch 11/50
## 4/4 - 0s - loss: 0.0014 - mse: 0.0014 - val_loss: 3.1210e-04 - val_mse: 3.1210e-04 - 43ms/epoch - 11
## Epoch 12/50
## 4/4 - 0s - loss: 0.0014 - mse: 0.0014 - val_loss: 8.9882e-04 - val_mse: 8.9882e-04 - 43ms/epoch - 11
## Epoch 13/50
## 4/4 - 0s - loss: 8.3919e-04 - mse: 8.3919e-04 - val_loss: 0.0027 - val_mse: 0.0027 - 44ms/epoch - 11
## Epoch 14/50
## 4/4 - 0s - loss: 0.0011 - mse: 0.0011 - val_loss: 0.0030 - val_mse: 0.0030 - 44ms/epoch - 11ms/step
## Epoch 15/50
## 4/4 - 0s - loss: 9.6688e-04 - mse: 9.6688e-04 - val_loss: 0.0017 - val_mse: 0.0017 - 46ms/epoch - 11
## Epoch 16/50
## 4/4 - 0s - loss: 8.4534e-04 - mse: 8.4534e-04 - val_loss: 9.6476e-04 - val_mse: 9.6476e-04 - 77ms/ep
## Epoch 17/50
## 4/4 - 0s - loss: 8.6962e-04 - mse: 8.6962e-04 - val_loss: 0.0011 - val_mse: 0.0011 - 68ms/epoch - 17
## Epoch 18/50
## 4/4 - 0s - loss: 8.1681e-04 - mse: 8.1681e-04 - val_loss: 0.0017 - val_mse: 0.0017 - 47ms/epoch - 12
## Epoch 19/50
## 4/4 - 0s - loss: 8.0750e-04 - mse: 8.0750e-04 - val_loss: 0.0016 - val_mse: 0.0016 - 45ms/epoch - 11
## Epoch 20/50
## 4/4 - 0s - loss: 7.8459e-04 - mse: 7.8459e-04 - val_loss: 0.0013 - val_mse: 0.0013 - 44ms/epoch - 11
## Epoch 21/50
## 4/4 - 0s - loss: 7.8142e-04 - mse: 7.8142e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 46ms/epoch - 12
## Epoch 22/50
## 4/4 - 0s - loss: 7.7343e-04 - mse: 7.7343e-04 - val_loss: 0.0014 - val_mse: 0.0014 - 46ms/epoch - 11
## Epoch 23/50
## 4/4 - 0s - loss: 7.5489e-04 - mse: 7.5489e-04 - val_loss: 0.0014 - val_mse: 0.0014 - 47ms/epoch - 12
## Epoch 24/50
## 4/4 - 0s - loss: 7.4609e-04 - mse: 7.4609e-04 - val_loss: 0.0014 - val_mse: 0.0014 - 44ms/epoch - 11
## Epoch 25/50
## 4/4 - 0s - loss: 7.3414e-04 - mse: 7.3414e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 48ms/epoch - 12
## Epoch 26/50
## 4/4 - 0s - loss: 7.2306e-04 - mse: 7.2306e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 45ms/epoch - 11
## Epoch 27/50
## 4/4 - 0s - loss: 7.1245e-04 - mse: 7.1245e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 44ms/epoch - 11
## Epoch 28/50
## 4/4 - 0s - loss: 6.9931e-04 - mse: 6.9931e-04 - val_loss: 0.0013 - val_mse: 0.0013 - 46ms/epoch - 12
## Epoch 29/50
## 4/4 - 0s - loss: 6.8995e-04 - mse: 6.8995e-04 - val_loss: 0.0012 - val_mse: 0.0012 - 46ms/epoch - 11
## Epoch 30/50
```

```
## 4/4 - 0s - loss: 6.7363e-04 - mse: 6.7363e-04 - val_loss: 0.0010 - val_mse: 0.0010 - 46ms/epoch - 11r
## Epoch 31/50
## 4/4 - 0s - loss: 6.6319e-04 - mse: 6.6319e-04 - val_loss: 0.0010 - val_mse: 0.0010 - 49ms/epoch - 12r
## Epoch 32/50
## 4/4 - 0s - loss: 6.4982e-04 - mse: 6.4982e-04 - val_loss: 0.0011 - val_mse: 0.0011 - 44ms/epoch - 11r
## Epoch 33/50
## 4/4 - 0s - loss: 6.3539e-04 - mse: 6.3539e-04 - val_loss: 9.8669e-04 - val_mse: 9.8669e-04 - 43ms/epo
## Epoch 34/50
## 4/4 - 0s - loss: 6.2470e-04 - mse: 6.2470e-04 - val_loss: 8.5853e-04 - val_mse: 8.5853e-04 - 43ms/epo
## Epoch 35/50
## 4/4 - 0s - loss: 6.1023e-04 - mse: 6.1023e-04 - val_loss: 9.1067e-04 - val_mse: 9.1067e-04 - 42ms/epo
## Epoch 36/50
## 4/4 - 0s - loss: 5.9405e-04 - mse: 5.9405e-04 - val_loss: 9.2103e-04 - val_mse: 9.2103e-04 - 43ms/epo
## Epoch 37/50
## 4/4 - 0s - loss: 5.9111e-04 - mse: 5.9111e-04 - val_loss: 9.0466e-04 - val_mse: 9.0466e-04 - 42ms/epo
## Epoch 38/50
## 4/4 - 0s - loss: 5.6405e-04 - mse: 5.6405e-04 - val_loss: 6.6345e-04 - val_mse: 6.6345e-04 - 44ms/epo
## Epoch 39/50
## 4/4 - 0s - loss: 5.6146e-04 - mse: 5.6146e-04 - val_loss: 7.6611e-04 - val_mse: 7.6611e-04 - 45ms/epo
## Epoch 40/50
## 4/4 - 0s - loss: 5.4448e-04 - mse: 5.4448e-04 - val_loss: 9.3051e-04 - val_mse: 9.3051e-04 - 45ms/epo
## Epoch 41/50
## 4/4 - 0s - loss: 5.3478e-04 - mse: 5.3478e-04 - val_loss: 6.3502e-04 - val_mse: 6.3502e-04 - 47ms/epo
## Epoch 42/50
## 4/4 - 0s - loss: 5.2699e-04 - mse: 5.2699e-04 - val_loss: 6.0445e-04 - val_mse: 6.0445e-04 - 46ms/epo
## Epoch 43/50
## 4/4 - 0s - loss: 5.0430e-04 - mse: 5.0430e-04 - val_loss: 7.5413e-04 - val_mse: 7.5413e-04 - 48ms/epo
## Epoch 44/50
## 4/4 - 0s - loss: 5.0377e-04 - mse: 5.0377e-04 - val_loss: 6.1815e-04 - val_mse: 6.1815e-04 - 46ms/epo
## Epoch 45/50
## 4/4 - 0s - loss: 4.8844e-04 - mse: 4.8844e-04 - val_loss: 5.0701e-04 - val_mse: 5.0701e-04 - 42ms/epo
## Epoch 46/50
## 4/4 - 0s - loss: 4.8027e-04 - mse: 4.8027e-04 - val_loss: 6.3044e-04 - val_mse: 6.3044e-04 - 44ms/epo
## Epoch 47/50
## 4/4 - 0s - loss: 4.6565e-04 - mse: 4.6565e-04 - val_loss: 4.6420e-04 - val_mse: 4.6420e-04 - 48ms/epo
## Epoch 48/50
## 4/4 - 0s - loss: 4.6308e-04 - mse: 4.6308e-04 - val_loss: 4.9205e-04 - val_mse: 4.9205e-04 - 47ms/epo
## Epoch 49/50
## 4/4 - 0s - loss: 4.5010e-04 - mse: 4.5010e-04 - val_loss: 4.9474e-04 - val_mse: 4.9474e-04 - 44ms/epo
## Epoch 50/50
## 4/4 - 0s - loss: 4.4316e-04 - mse: 4.4316e-04 - val_loss: 4.5329e-04 - val_mse: 4.5329e-04 - 44ms/epo
```

Looking at results of history_1v w/ validation split.

```
plot(history_1v)
```



Here again, the training and validation split are very close together, indicating that the model is potentially doing a good job at not overfitting to the data.

Plotting Test Data vs. Predicted

Here, the training sequences generated are using one variable (adjusted closing price) as mentioned before. The generator function has been modified in the `generator.R` file.

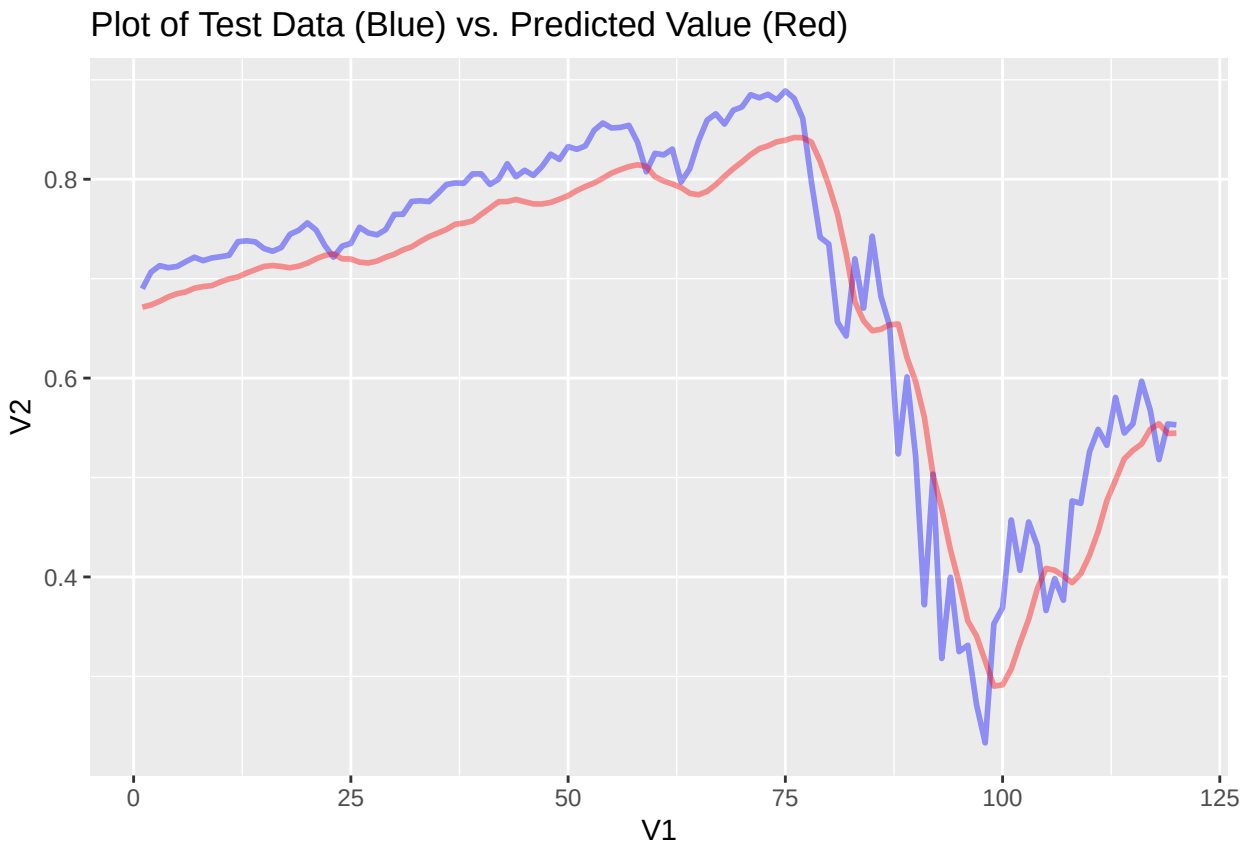
```
# Plotting actual test data vs. predicted (1 variable)
batch_size_plot <- 120
lookback_plot <- 5
step_plot <- 1

set.seed(123)
pred_gen <- generator_1v(
  data,
  lookback = lookback_plot,
  delay = 0,
  min_index = 1000,
  max_index = 1200,
  shuffle = FALSE,
  step = step_plot,
  batch_size = batch_size_plot
)

pred_gen_data_1v <- pred_gen()
```

Let's look at test data vs predicted values:

```
V1 = seq(1, length(pred_gen_data_1v[[2]]))

# binds V1 as time step (actual) to actual sequence
plot_data_1v <- as.data.frame(cbind(V1, pred_gen_data[[2]]))

inputdata_1v <- pred_gen_data_1v[[1]][,,]
dim(inputdata_1v) <- c(batch_size_plot,lookback_plot, 1)

pred_out_1v <- model_1v %>% predict(inputdata_1v)
```

```
## 4/4 - 0s - 382ms/epoch - 96ms/step
```

```
plot_data_1v <- cbind(plot_data_1v, pred_out_1v[])

p <- ggplot(plot_data, aes(x = V1, y = V2)) + geom_line( colour = "blue", size = 1, alpha=0.4)
p <- p + geom_line(aes(x = V1, y = pred_out), colour = "red", size = 1 , alpha=0.4)
p <- p + labs(title = "Plot of Test Data (Blue) vs. Predicted Value (Red)")

p
```



The univariate model captures the overall trend of the testing data well.

RMSE

```
rmse_1v <- sqrt(mean((plot_data_1v[,2] - plot_data_1v[,3])^2))
rmse_1v
```

```
## [1] 0.05471607
```

In the case of the LSTM RNN model using all the features in the dataset, we achieved an RMSE of around 0.08. (changes slightly due to random error with set.seed).

Other Predictive Models

SARIMA Model

The first model I can use to predict or 'forecast' the S&P 500 stock price can be an auto-regressive time series model. I first have to re-add the 'date' column from our original non-standardized dataset so I can use time series forecasting methods on the dataset. This is shown below.

Rolling window cross validation

```
library(forecast)
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
##
## Attaching package: 'forecast'
```

```
## The following object is masked from 'package:yardstick':
##
##     accuracy
```

```
library(lubridate)

#Get the dates column
dates <- data_clean$Date

#Combine dates column with standardized data
df <- as.data.frame(data)
df <- cbind(Date = dates, df)

# Convert 'date' to Date class and set it as the time index
ts_data <- ts(df$Close, frequency = 1, start = c(year(df$Date[1]), month(df$Date[1])))

# Number of days for the rolling window
window_size <- 6
```

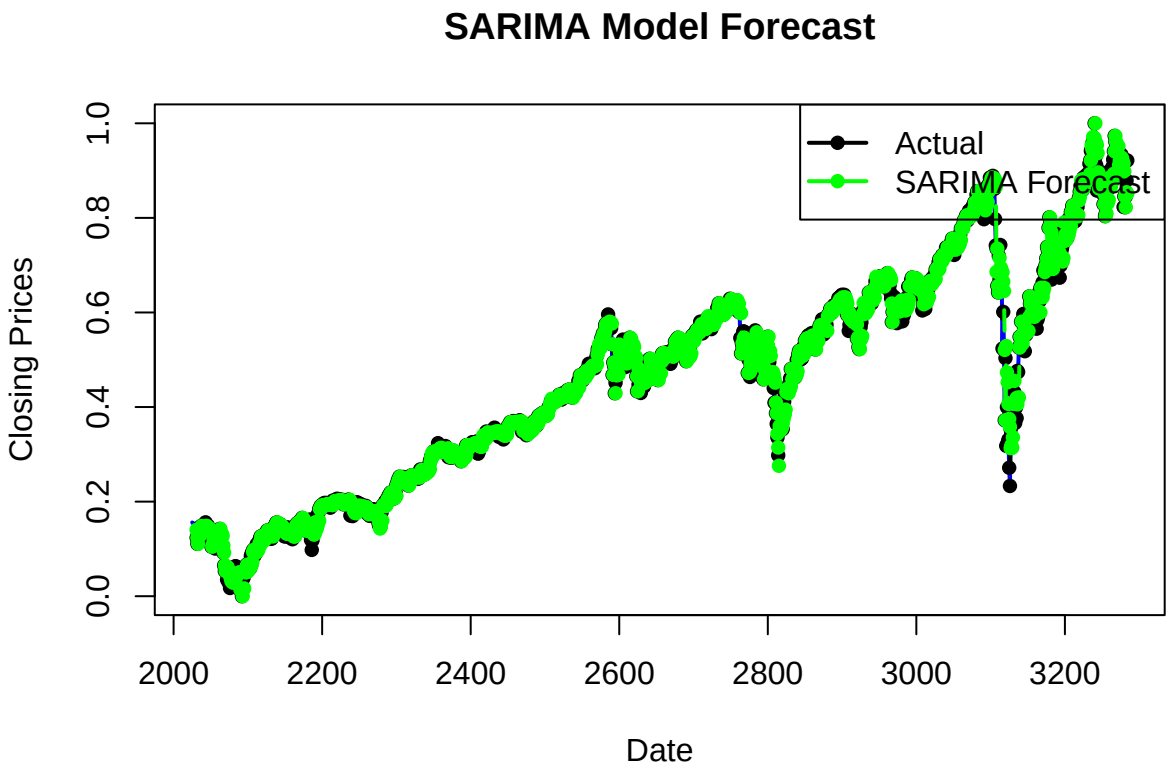
Since I am using RNN and other models, with time series data, using k-cross fold validation would mix up the time of the data being analyzed. Instead I am using a 'rolling window approach', so k-cross validation is not suitable. As you iterate through the time series data, you use a portion of the past observations (rolling cross validation window) to train the model and predict the next observation. In this way, you are continually updating the model with new information and making one-step-ahead forecasts. I split the data into k folds in a way that each fold represents a sequential chunk of time. For each fold, I use the earlier time periods as the training set and the subsequent period as the testing set. Then, I train the model on the training set and validate it on the testing set for each fold.

```
# Initialize an empty dataframe to store forecasts
forecasts <- data.frame(Date = as.Date(character()), Actual = numeric(), ARIMA = numeric(), SARIMA = nu

# Perform rolling window forecast
for (i in (window_size + 1):length(ts_data)) {
  # Extract the current window
```

```r
  current_window <- ts_data[(i - window_size):(i - 1)]

  # SARIMA Model (Seasonal)
  sarima_model <- auto.arima(current_window, seasonal = TRUE)

  # Forecast the next day

  sarima_forecast <- forecast(sarima_model, h = 1)

  # Store the results in the forecasts dataframe
  forecasts <- rbind(forecasts, data.frame(Date = time(ts_data)[i],
                                           Actual = ts_data[i],
                                           SARIMA = sarima_forecast$mean[1]))  # Extract the first fore
  }
```

So, the need for a separate training and testing set, as typically done in traditional model evaluation, is somewhat inherent in the rolling window approach. The testing set, in this case, is essentially the next data point that the model is predicting. The training set is the rolling window of past observations used to fit the model.

```r
#Visualize forecasts for SARIMA
plot(ts_data, type = "l", col = "blue", lwd = 2, main = "SARIMA Model Forecast", xlab = "Date", ylab =
lines(forecasts$Date, forecasts$Actual, col = "black", lwd = 2, lty = 2, type = "b", pch = 16)
lines(forecasts$Date, forecasts$SARIMA, col = "green", lwd = 2, type = "b", pch = 16)
legend("topright", legend = c("Actual", "SARIMA Forecast"), col = c("black","green"), lwd = 2, pch = 16)
```



In the above forecast plot, the SARIMA model does a fairly effective job of predicting the actual values for the S&P 500 stock price. However, there are some instances like in between day 3000 and day 3200 that the model doesn't follow the bottom of the 'valley' where the stock hits its lowest point.

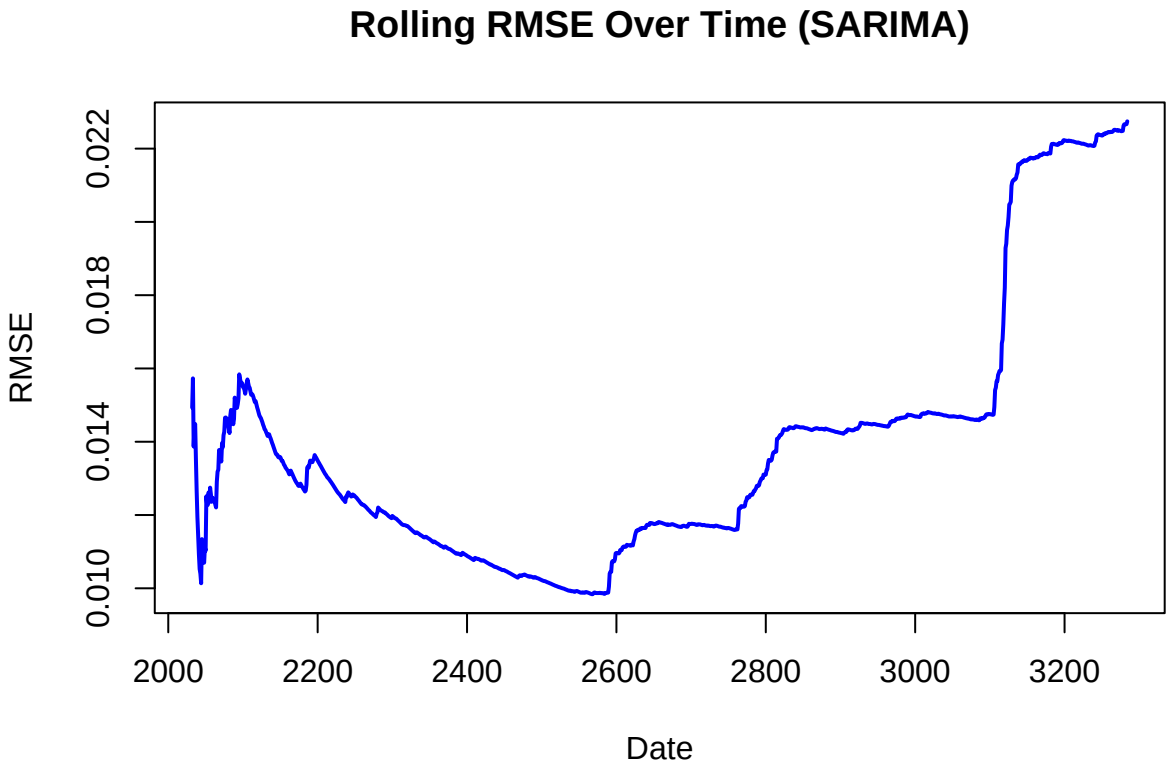
```r
actual_values <- forecasts$Actual
sarima_forecast_values <- forecasts$SARIMA

# Remove missing values, if any
actual_values <- actual_values[!is.na(sarima_forecast_values)]
sarima_forecast_values <- sarima_forecast_values[!is.na(sarima_forecast_values)]

# Calculate squared errors
squared_errors <- (sarima_forecast_values - actual_values)^2

# Calculate mean squared error
mse <- mean(squared_errors)

# Calculate root mean squared error
rmse <- sqrt(mse)

# Print RMSE value
cat("RMSE for SARIMA model:", rmse, "\n")
```

```
## RMSE for SARIMA model: 0.02274396
```

This RMSE approach is convenient for time series forecasting so traditional model evaluation metrics (such as Mean Squared Error) might not be directly applicable in this context, as the model is being continually updated with new information. Due to this, a better option would be to visualize the RMSE over time calculated on each iteration. The below plot does this.

In order to get rid of any missing values, I used the '!is.na' function. Any values that have a value of 'NA' was discarded. Luckily there were not any data values like this in the section of years that I used. The dataset goes back to 1927 and in the first few decades, there were many dates with missing data. However for the data range I used I did not have any missing values.

```r
rmse_values <- c()

# Calculate RMSE for each forecasted point
for (i in 2:nrow(forecasts)) {
  actual <- actual_values[1:i]
  sarima_forecast <- sarima_forecast_values[1:i]
  rmse <- sqrt(mean((actual - sarima_forecast)^2))
  rmse_values <- c(rmse_values, rmse)
}

# Visualize RMSE over time
plot(forecasts$Date[2:nrow(forecasts)], rmse_values, type = "l", col = "blue", lwd = 2,
     main = "Rolling RMSE Over Time (SARIMA)", xlab = "Date", ylab = "RMSE")
```



Feed Forward Neural Network

This code snippet is setting up a feed-forward neural network to compare the accuracy between it and a Recurrent Neural Network. A Feed forward Neural Network processes input data sequentially without memory of previous inputs, suitable for tasks with independent elements. In contrast, a Recurrent Neural Network maintains internal memory, allowing it to capture temporal dependencies in sequential data, which models time series data better.

This code starts with sourcing a script called generator.R to access relevant functions. The parameters lookback, step, delay, and batch_size are defined to configure the characteristics of the training data. Then, a data generator (train_gen_5days) is initialized using the generator_5days function, which takes into account the specified parameters, as well as the stock price data (data). The generator is set to cover a specific range of indices (min_index to max_index) from the dataset without shuffling the data. Finally, the generator is invoked to produce batches of training data, and the resulting data is stored in the variable train_gen_data_5days. This approach aligns with the common practice in time series forecasting, where a model is trained to predict future values based on historical sequences.

```
setwd(scripts_dir)
source('generator.R')
lookback <- 5
step <- 1
delay <- 0
batch_size <- 128
set.seed(123)
train_gen_5days <- generator_5days(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 500,
  max_index = 1000,
```

```
  shuffle = FALSE,
  step = step,
  batch_size = batch_size
)

train_gen_data_5days <- train_gen_5days()
```

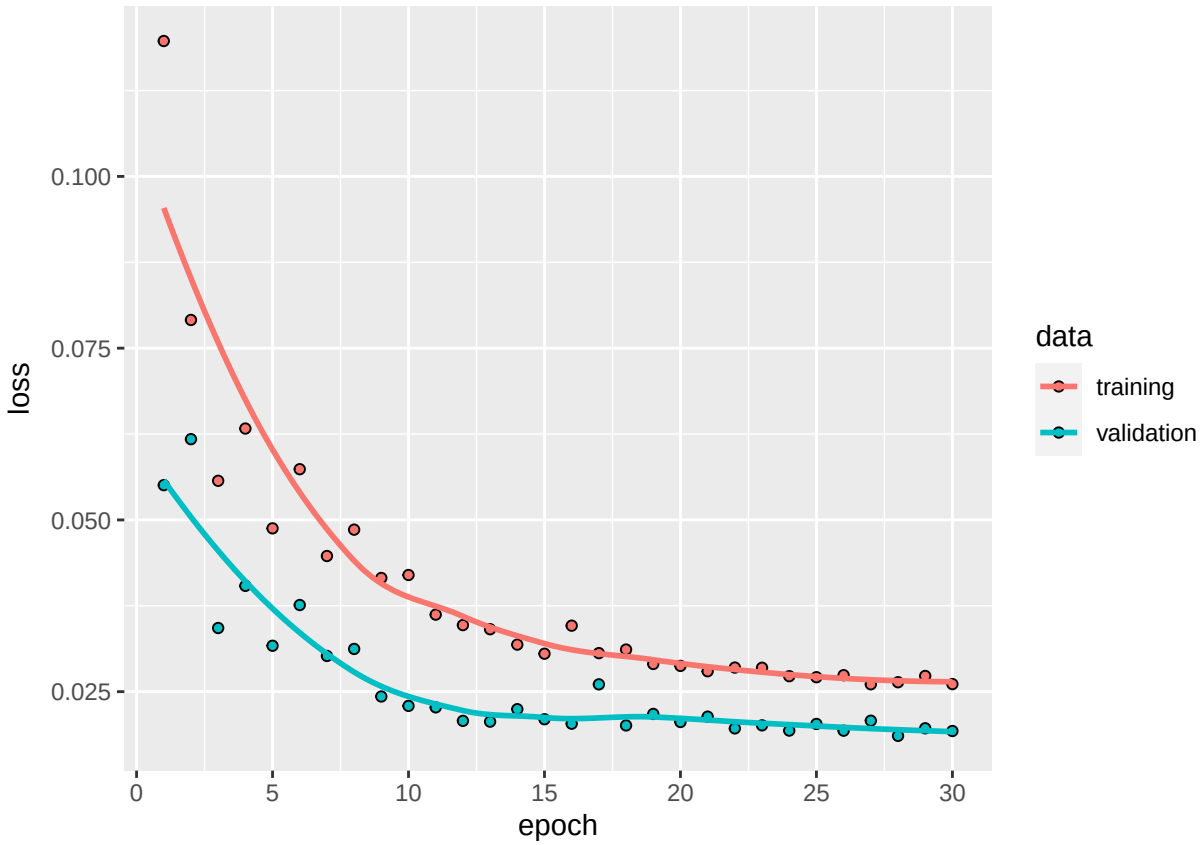
Next, the simple feedforward neural network using the Keras API with the TensorFlow backend is defined. The network consists of a flatten layer to handle input data with a specific shape, followed by a dense layer with ReLU activation and an output layer with a single unit. The model is compiled using the Adam optimizer and Mean Absolute Error (MAE) loss. It is then trained on input samples (samples) and corresponding targets (targets) for 30 epochs, and the training history is visualized. Finally, the mean squared error (MSE) is calculated based on the difference between the predicted and actual values stored in plot_data.

```
model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(lookback / step , 6)) %>%  # Flatten layer to handle the input shape
  layer_dense(units = 32, activation = 'relu') %>%  # Dense layer with ReLU activation
  layer_dense(units = 1)  # Output layer with 1 unit

model %>% compile(optimizer = 'adam', loss = 'mae')

history <- model %>% fit(
  x = train_gen_data_5days[[1]],
  y = train_gen_data_5days[[2]],
  epochs = 30,
  batch_size = 32,  # Adjust batch size based on your resources
  validation_split = 0.2
)
```

```
## Epoch 1/30
## 4/4 - 1s - loss: 0.1197 - val_loss: 0.0551 - 520ms/epoch - 130ms/step
## Epoch 2/30
## 4/4 - 0s - loss: 0.0791 - val_loss: 0.0618 - 30ms/epoch - 7ms/step
## Epoch 3/30
## 4/4 - 0s - loss: 0.0557 - val_loss: 0.0343 - 28ms/epoch - 7ms/step
## Epoch 4/30
## 4/4 - 0s - loss: 0.0633 - val_loss: 0.0404 - 28ms/epoch - 7ms/step
## Epoch 5/30
## 4/4 - 0s - loss: 0.0488 - val_loss: 0.0317 - 29ms/epoch - 7ms/step
## Epoch 6/30
## 4/4 - 0s - loss: 0.0574 - val_loss: 0.0376 - 28ms/epoch - 7ms/step
## Epoch 7/30
## 4/4 - 0s - loss: 0.0447 - val_loss: 0.0302 - 29ms/epoch - 7ms/step
## Epoch 8/30
## 4/4 - 0s - loss: 0.0486 - val_loss: 0.0312 - 29ms/epoch - 7ms/step
## Epoch 9/30
## 4/4 - 0s - loss: 0.0415 - val_loss: 0.0243 - 28ms/epoch - 7ms/step
## Epoch 10/30
## 4/4 - 0s - loss: 0.0420 - val_loss: 0.0229 - 30ms/epoch - 7ms/step
## Epoch 11/30
## 4/4 - 0s - loss: 0.0362 - val_loss: 0.0227 - 34ms/epoch - 9ms/step
## Epoch 12/30
## 4/4 - 0s - loss: 0.0347 - val_loss: 0.0207 - 32ms/epoch - 8ms/step
## Epoch 13/30
## 4/4 - 0s - loss: 0.0341 - val_loss: 0.0206 - 29ms/epoch - 7ms/step
```

```
## Epoch 14/30
## 4/4 - 0s - loss: 0.0318 - val_loss: 0.0224 - 29ms/epoch - 7ms/step
## Epoch 15/30
## 4/4 - 0s - loss: 0.0305 - val_loss: 0.0210 - 30ms/epoch - 7ms/step
## Epoch 16/30
## 4/4 - 0s - loss: 0.0346 - val_loss: 0.0203 - 28ms/epoch - 7ms/step
## Epoch 17/30
## 4/4 - 0s - loss: 0.0306 - val_loss: 0.0261 - 29ms/epoch - 7ms/step
## Epoch 18/30
## 4/4 - 0s - loss: 0.0311 - val_loss: 0.0201 - 29ms/epoch - 7ms/step
## Epoch 19/30
## 4/4 - 0s - loss: 0.0290 - val_loss: 0.0218 - 28ms/epoch - 7ms/step
## Epoch 20/30
## 4/4 - 0s - loss: 0.0287 - val_loss: 0.0206 - 29ms/epoch - 7ms/step
## Epoch 21/30
## 4/4 - 0s - loss: 0.0280 - val_loss: 0.0214 - 28ms/epoch - 7ms/step
## Epoch 22/30
## 4/4 - 0s - loss: 0.0285 - val_loss: 0.0197 - 28ms/epoch - 7ms/step
## Epoch 23/30
## 4/4 - 0s - loss: 0.0285 - val_loss: 0.0201 - 29ms/epoch - 7ms/step
## Epoch 24/30
## 4/4 - 0s - loss: 0.0272 - val_loss: 0.0193 - 28ms/epoch - 7ms/step
## Epoch 25/30
## 4/4 - 0s - loss: 0.0271 - val_loss: 0.0203 - 29ms/epoch - 7ms/step
## Epoch 26/30
## 4/4 - 0s - loss: 0.0274 - val_loss: 0.0193 - 29ms/epoch - 7ms/step
## Epoch 27/30
## 4/4 - 0s - loss: 0.0261 - val_loss: 0.0208 - 28ms/epoch - 7ms/step
## Epoch 28/30
## 4/4 - 0s - loss: 0.0264 - val_loss: 0.0186 - 28ms/epoch - 7ms/step
## Epoch 29/30
## 4/4 - 0s - loss: 0.0273 - val_loss: 0.0197 - 29ms/epoch - 7ms/step
## Epoch 30/30
## 4/4 - 0s - loss: 0.0261 - val_loss: 0.0193 - 29ms/epoch - 7ms/step
```

```
plot(history)
```




```
batch_size_plot <- 120
lookback_plot <- 5
step_plot <- 1

set.seed(123)
pred_gen_5days <- generator_5days(
  data,
  lookback = lookback_plot,
  delay = 0,
  min_index = 1000,
  max_index = 1200,
  shuffle = FALSE,
  step = step_plot,
  batch_size = batch_size_plot
)

pred_gen_data_5days <- pred_gen_5days()

V1 = seq(1, length(pred_gen_data_5days[[2]]))

# binds V1 as time step (actual) to actual sequence
plot_data_5days <- as.data.frame(cbind(V1, pred_gen_data_5days[[2]]))

inputdata_5days <- pred_gen_data_5days[[1]][,,]
dim(inputdata_5days) <- c(batch_size_plot,lookback_plot, 6)

pred_out_5days <- model %>% predict(inputdata_5days)
```

```
## 4/4 - 0s - 47ms/epoch - 12ms/step
```

```
plot_data_5days <- cbind(plot_data_5days, pred_out_5days[])


rmse <- sqrt(mean((plot_data_5days[,2] - plot_data_5days[,3])^2))
rmse
```

```
## [1] 0.08619874
```

Conclusion

Model Name	RSME Value
Multivariate LSTM RNN	0.055
Univariate LSTM RNN	0.077
SARIMA	0.023
Feed-Forward Neural Network	0.0819

Out of the four models, the Multivariate LSTM Recurrent neural network performed the best with a Root Mean Square Error value of 0.055. We did not consider the SARIMA model's value because the model is being continuously trained on the entirety of the data due to the sliding window approach, unlike the other models. The other three models were split into testing and training so there was less data for the model to learn from. The Feed Forward Neural Network had the highest root mean square error value because it is the least optimized for time series data as it does not have a backpropogate feature to adjust the weights after the initial prediction is compared to the ground truth through a loss function, yet it still finished fairly closely behind the Univariate LSTM Recurrent Neural Network.

Model Considerations

Ultimately, the Multivariate LSTM model outperformed the standard feed-forward network in predicting stock price data with a lookback of 5 days due to its ability to capture temporal dependencies and handle sequential data effectively. The Multivariate LSTM did better than the Univariate LSTM, suggesting that temporal dependencies of other features in the data are useful in predicting the adjusted closing price of the S&P 500. The LSTM's architecture, incorporating memory cells and gates, allows it to retain and utilize information from past time steps, making it good at capturing long-term patterns in stock prices. With a lookback of 5 days, the LSTM can generalize trends and intricate patterns within S&P 500, enabling more accurate predictions compared to a feed-forward network that lacks memory and sequential understanding. Adjusting the lookback value could lead to better model performance, as LSTMs might be better loopbacko larger lookback values, utilizing its memory cells and gate features. Adjusting other hyperparameters like batch size, a moderate value of, say, 32 or 64, could help in training by balancing computation and speeding up model convergence. The SARIMA model is also well suited to predict sequences like stock price data, as its dynamic training using the sliding window approach helps with prediction accuracy. Furthermore, setting an appropriate number of epochs allows the LSTM to iteratively learn and refine its understanding of temporal dynamics, ultimately enhancing its predictive capabilities.