Capstone project report **Kiem Nguyen**
Machine Learning Engineer Nanodegree **February 2021**

Project Overview

Trading for profit is a difficult problem to solve. In an efficient market, buyers and sellers would have all the information needed to make a rational trading decision so the stock is should remain at its fair values. The reality is the financial market is not efficient in real life especially now when automated trading allows for thousands of transactions to occur within a nanosecond.

This project tries to select Alpha signal from trading using data science and machine learning techniques. The data set is a 10-year trading data of SPY from January 2021 to the most recent date February 2021. SPY is a stock market index that measures the stock performance of 500 large companies listed on stock exchanges in the US. This fund is the largest ETF in the world. The SPY movement could signal the direction of the market.

I use pandas-datareader library to extract end-of-day stock pricing data from Yahoo Finance from 2011-2020. This library provides functions to extract data from various internet sources in a pandas data frame. I received an error when trying to collect SPY data from the Quandl data source. It turned out that Quandl only has SPY data for the paid subscribers so the API key in my account doesn't work. Google also retired their stock data source. Fortunately, the Yahoo Finance data source works great so I decided to use this data source.

Problem Statement

Between 2010 and 2020 the S&P 500 has an annual average return of 13.6% in the past 10 years. This project would provide a practical approach to predict future price movements in financial markets based on past returns. The assumption is that certain patterns in financial markets repeat themselves such that past observations can be leveraged to predict future price movements. (Hilpisch, Yves. 2020. Artificial Intelligence in Finance: A Python-Based Guide.)

Metrics

For Neural Network models and ensembling models, I used the RMSE to calculate the score. I would look for the model with the lowest RMSE score to use for the final prediction.

Data Exploration

The data has both stock's closing price (Close) and adjusted closing price (Adj Close). The closing price is the last transaction price before the market closes. The adjusted closing price factors in corporate actions, such as stock splits, dividends, and rights offerings. I decided to use the adjusted closing price for the models.

Date	High	Low	Open	Close	Volume	Adj Close
2021-02-12	392.899994	389.769989	389.850006	392.640015	50505700.0	392.640015
2021-02-16	394.170013	391.529999	393.959991	392.299988	50700800.0	392.299988
2021-02-17	392.660004	389.329987	390.420013	392.390015	52290600.0	392.390015
2021-02-18	391.519989	387.739990	389.589996	390.720001	59552200.0	390.720001
2021-02-19	392.380005	389.549988	392.070007	390.029999	83142800.0	390.029999

The SPY data from 2010-01-01 to the most recent date 2021-02-19 has 2802 rows and 6 columns

```
df.shape # get the number of rows and columns
```

```
(2802, 6)
```

```
df.describe() # generate descriptive statistics
```

	High	Low	Open	Close	Volume	Adj Close
count	2802.000000	2802.000000	2802.000000	2802.000000	2.802000e+03	2802.000000
mean	210.783105	208.517659	209.704789	209.734333	1.237649e+08	192.583213
std	69.809154	69.075299	69.468840	69.454677	7.465493e+07	75.348641
min	103.419998	101.129997	103.110001	102.199997	2.027000e+07	82.872505
25%	142.442497	141.355003	141.982498	141.982498	7.239802e+07	120.491968
50%	206.800003	204.629997	205.614998	205.614998	1.038480e+08	184.856987
75%	268.587502	265.575005	267.577507	267.287506	1.530642e+08	254.170013
max	394.170013	391.529999	393.959991	392.640015	7.178287e+08	392.640015

We can see there is no null value in the data, and the data index is the Date column:

```
df.info()
```

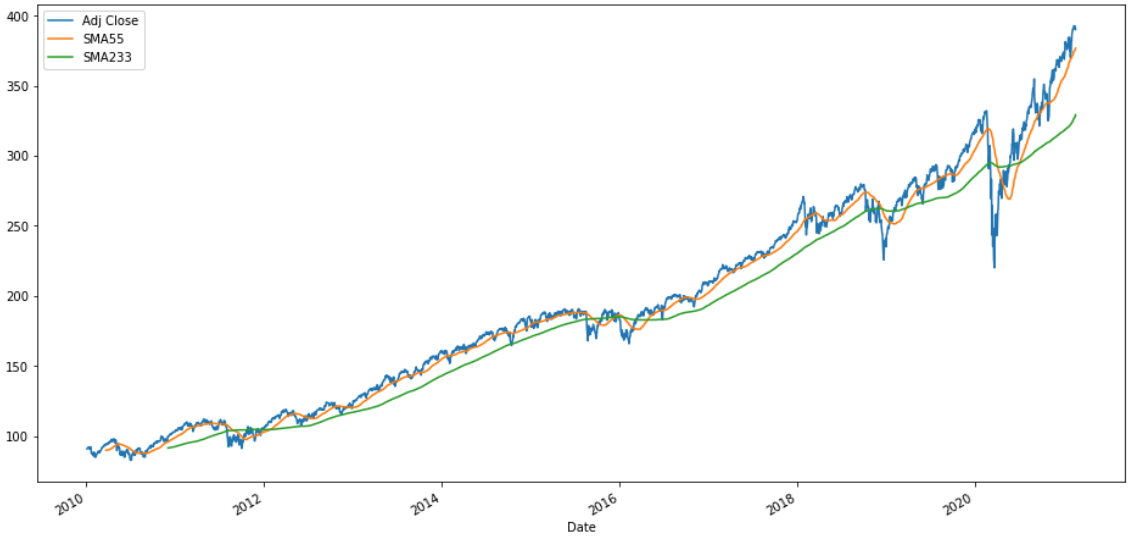
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2802 entries, 2010-01-04 to 2021-02-19
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   High       2802 non-null   float64
 1   Low        2802 non-null   float64
 2   Open       2802 non-null   float64
 3   Close      2802 non-null   float64
 4   Volume     2802 non-null   float64
 5   Adj Close  2802 non-null   float64
dtypes: float64(6)
memory usage: 153.2 KB
```

```
df.index
```

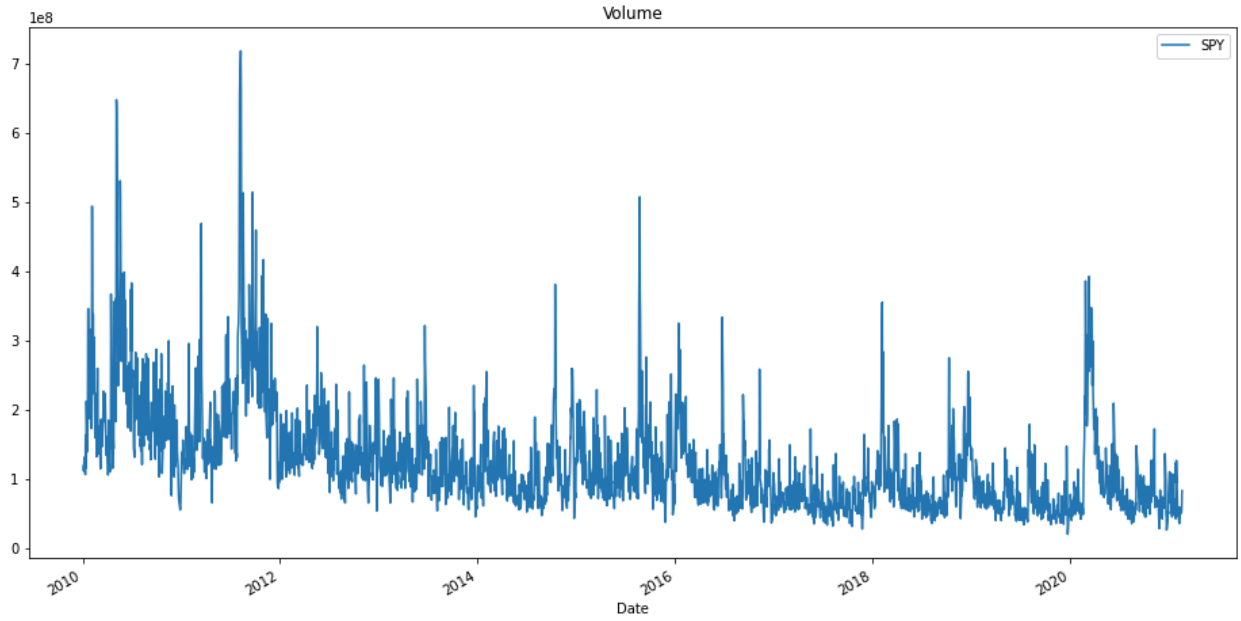
```
DatetimeIndex(['2010-01-04', '2010-01-05', '2010-01-06', '2010-01-07',
               '2010-01-08', '2010-01-11', '2010-01-12', '2010-01-13',
               '2010-01-14', '2010-01-15',
               ...
               '2021-02-05', '2021-02-08', '2021-02-09', '2021-02-10',
               '2021-02-11', '2021-02-12', '2021-02-16', '2021-02-17',
               '2021-02-18', '2021-02-19'],
              dtype='datetime64[ns]', name='Date', length=2802, freq=None)
```

Exploratory Visualization

The adjusted closing price of SPY since 2010-01-01 with 55 days and 233 days simple moving average (SMA). A simple moving average (SMA) calculates the average of a selected range of closing prices, by the number of periods in that range. It is a technical indicator that can aid in determining if an asset price will continue or if it will reverse a bull or bear trend.



The volume of SPY since 2010-01-01:



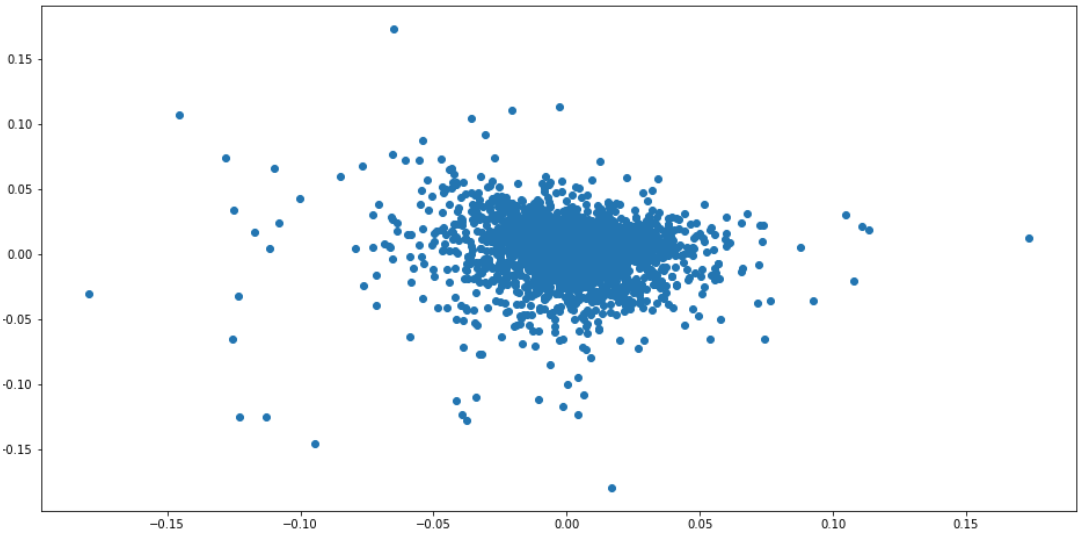
SPY has a huge volume in 2011. Let's find out the exact date of this event to see what happend that date

```
df['Volume'].idxmax()
```

```
Timestamp('2011-08-09 00:00:00')
```

The event was on Black Monday when US and global stock markets crashed.

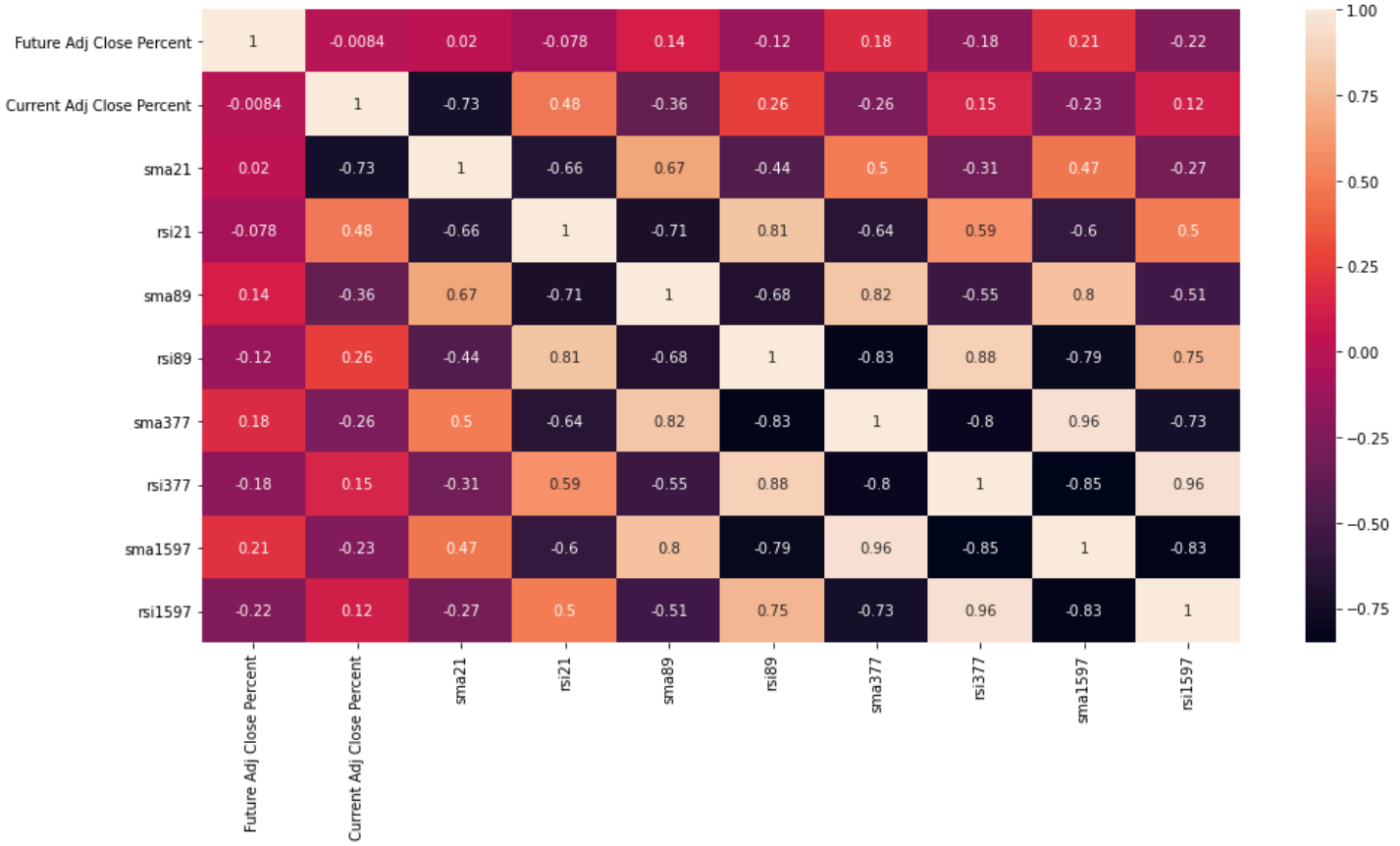
The correlation coefficient would tell us how strong the correlation of previous price changes vs future price changes. If it's highly correlated, then the stock price is a trend following. Otherwise, the stock price is mean reverting.



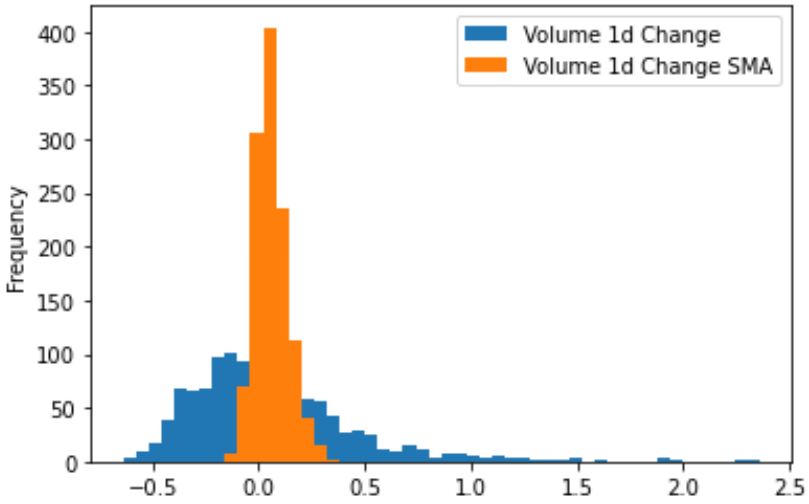
We can see the future price change is negatively correlated to the previous price change for a 5 days trading period. This tells us that a mean reversion trading would be a good trading strategy.

Future Engineering:
Before building the models, I want to add more data to make a better prediction by doing feature engineering. The new features I add in the models are:

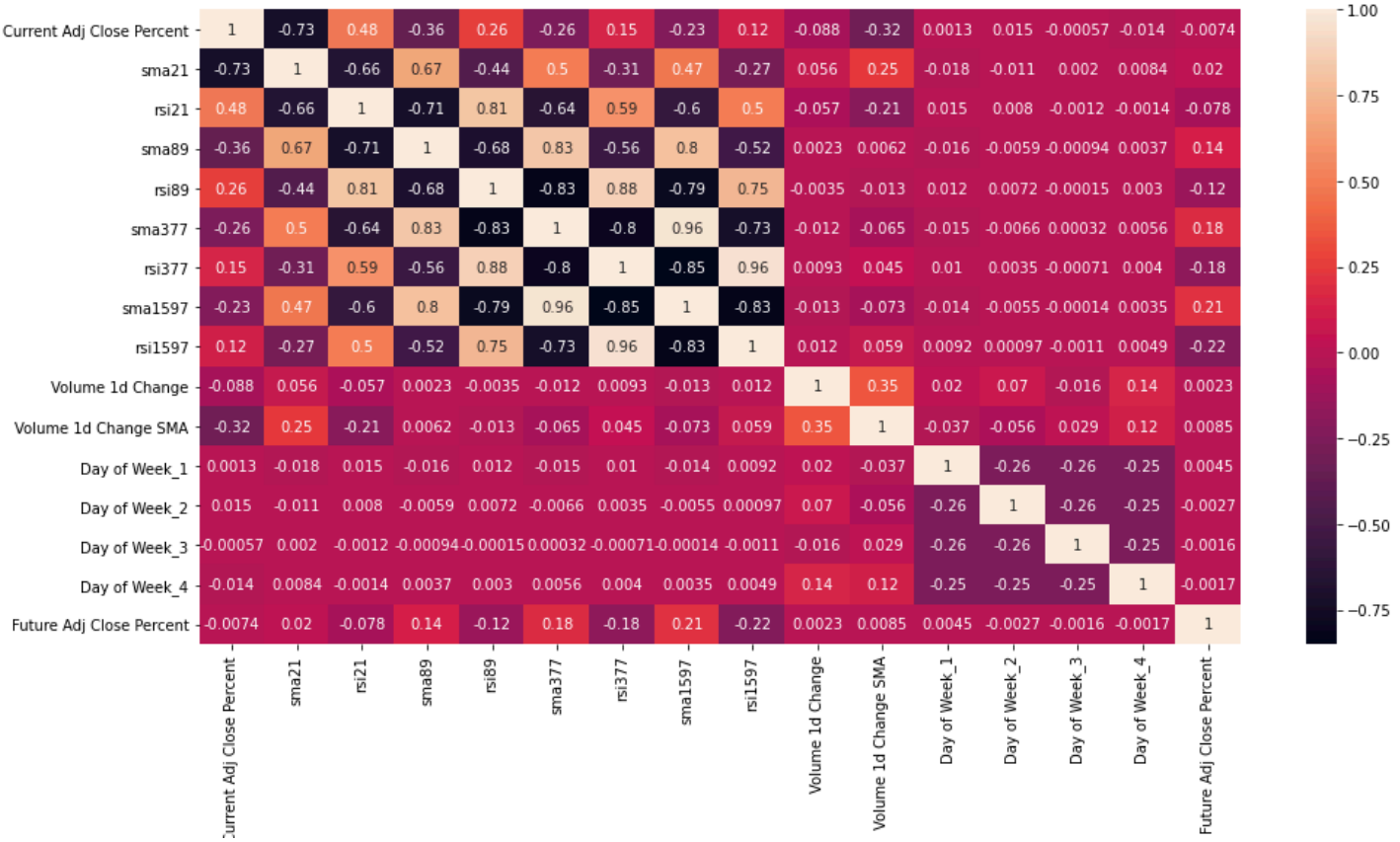
1. An RSI (Relative Strength Index) for different time period. RSI provides signals about bullish and bearish price momentum. A stock is usually considered overbought when the RSI is above 70% and oversold when it is below 30%. Below is the heatmap to see the correlation between the new features and the target (Future Adj Close Percent):



2. A simple moving average (SMA) for different time period which is one of the most common indicators.



3. Day of week feature: As for these new features, I used numbers in the Fibonacci Sequence for different time periods. First, I tried 55, 89, 144, 233 time period, but then switched to 21, 89, 377, 1597 because this give better correlations between the features and targets which could help the model.



We can see the correlations are very weak so the new features won't help the model. I decided to use the default features with the Adj Close instead.

Algorithms and Techniques

I use MinMaxScaler to scale the train and test data to build two Neural Network models with different settings using keras library. Neural nets can capture the interaction between different hyperparameters very well.

```
# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(data)
```

```
# Create the train data set
# With financial time series data, we want to use previous day data to make prediction on the next day.
# This would help us to validate the model performance on the most recent stock price.

train_data = scaled_data[0:train_data_len, :]
# Split to X_train and y_train dataset
X_train = []
y_train = []

for i in range (60, len(train_data)):
    X_train.append(train_data[i-60:i, 0])
    y_train.append(train_data[i, 0])

# Convert x_train and y_train to numpy arrays
X_train, y_train = np.array(X_train), np.array(y_train)
```

```
# Create the test data set
test_data = scaled_data[train_data_len - 60:, :]
# Split to X_test and y_test dataset
X_test = []
y_test = data[train_data_len:, :]

for i in range (60, len(test_data)):
    X_test.append(test_data[i-60:i, 0])

# Convert X_test to a numpy array
X_test = np.array(X_test)
```

The train and test data need to be re-shaped to 3-dimensional data shape for the LSTM network:

```
# Reshaping the train data to 3 dimensional data shape
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
print(X_train.shape)

# Reshaping the test data to 3 dimensional data shape
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
print(X_test.shape)
```

```
(2042, 60, 1)
(700, 60, 1)
```

Neural Networks:
The first Neural Network model has 4 layers. The first two layers are Long Short-Term Memory (LSTM). The third Dense has a relu activation function and the last Dense layer has a linear activation function.

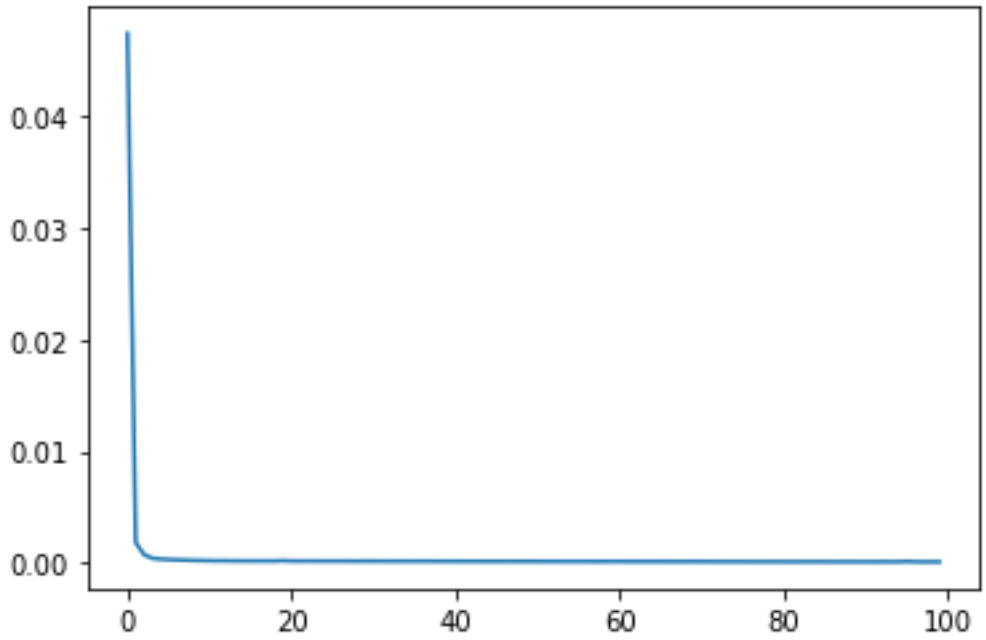
```
optimizer = Adam(learning_rate=0.0001)

def set_seeds(seed=100):
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(100)

set_seeds()

model_1 = Sequential()
model_1.add(LSTM(50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
#model.add(Dropout(0.2)) # add dropout to prevent overfitting
model_1.add(LSTM(50, return_sequences=False))
model_1.add(Dense(20, activation='relu'))
model_1.add(Dense(1, activation='linear'))
```

Next, I fit the model with a mean squared error loss function and train with 100 epochs:

```
# fit the neural network model with a mean squared error loss function
model_1.compile(optimizer=optimizer, loss='mse', metrics=['accuracy'])
history = model_1.fit(X_train, y_train, epochs=100, batch_size=30)
```

```
Epoch 1/100
2042/2042 [==============================] - 19s 9ms/step - loss: 0.0475 - accuracy: 4.8972e-04
Epoch 2/100
2042/2042 [==============================] - 18s 9ms/step - loss: 0.0018 - accuracy: 4.8972e-04
Epoch 3/100
2042/2042 [==============================] - 19s 9ms/step - loss: 7.6210e-04 - accuracy: 4.8972e-04
Epoch 4/100
2042/2042 [==============================] - 20s 10ms/step - loss: 4.3112e-04 - accuracy: 4.8972e-04
Epoch 5/100
2042/2042 [==============================] - 19s 9ms/step - loss: 3.5037e-04 - accuracy: 4.8972e-04
Epoch 6/100
2042/2042 [==============================] - 19s 9ms/step - loss: 3.1397e-04 - accuracy: 4.8972e-04
Epoch 7/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.8413e-04 - accuracy: 4.8972e-04
Epoch 8/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.5873e-04 - accuracy: 4.8972e-04
Epoch 9/100
2042/2042 [==============================] - 20s 10ms/step - loss: 2.3373e-04 - accuracy: 4.8972e-04
Epoch 10/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.1991e-04 - accuracy: 4.8972e-04
```

We can see the loss curve is flattened out, so the neural net was sufficiently trained:



After fitting the model, I use the RMSE to check the model performance, the lower score is better:

```
predict_1 = model_1.predict(X_test)
predict_1 = scaler.inverse_transform(predict_1)

rmse = np.sqrt(np.mean(predict_1 - y_test)**2)
rmse
```

```
3.371237291608538
```

Next, I use keras.losses and tensorflow to create a custom loss function for the second Neural Network model instead of using the previous mean squared error loss function. This custom loss function would give more penalty weight for predicting the wrong stock's closing price.

```
import keras.losses
import tensorflow as tf

def custom_loss(true_val, predict_val):
    penalty = 500
    loss = tf.where(tf.less(true_val * predict_val, 0), penalty * tf.square(true_val - predict_val), tf.square(true_val

    return tf.reduce_mean(loss, axis=-1)

keras.losses.custom_loss = custom_loss
```

A second Neural Network model is fit with a custom loss function and trained with 100 epochs:

```
def set_seeds(seed=100):
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(100)

set_seeds()

model_2 = Sequential()
model_2.add(LSTM(50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model_2.add(Dropout(0.2)) # add dropout to prevent overfitting
model_2.add(LSTM(50, return_sequences=False))
model_2.add(Dense(20, activation='relu'))
model_2.add(Dense(1, activation='linear'))

# fit the neural network model with a custom loss function
model_2.compile(optimizer=optimizer, loss=custom_loss, metrics=['accuracy']) # use the custom loss f
m the previous model
history = model_2.fit(X_train, y_train, epochs=100, batch_size=30)
```

```
Epoch 1/100
2042/2042 [==============================] - 21s 10ms/step - loss: 0.0094 - accuracy: 4.8972e-04
Epoch 2/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.6896e-04 - accuracy: 4.8972e-04
Epoch 3/100
2042/2042 [==============================] - 20s 10ms/step - loss: 2.4683e-04 - accuracy: 4.8972e-04
Epoch 4/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.5496e-04 - accuracy: 4.8972e-04
Epoch 5/100
2042/2042 [==============================] - 19s 9ms/step - loss: 2.3861e-04 - accuracy: 4.8972e-04
Epoch 6/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.3428e-04 - accuracy: 4.8972e-04
Epoch 7/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.3525e-04 - accuracy: 4.8972e-04
Epoch 8/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.2835e-04 - accuracy: 4.8972e-04
Epoch 9/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.2653e-04 - accuracy: 4.8972e-04
Epoch 10/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.2066e-04 - accuracy: 4.8972e-04
Epoch 11/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.3081e-04 - accuracy: 4.8972e-04
Epoch 12/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.1071e-04 - accuracy: 4.8972e-04
Epoch 13/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.1440e-04 - accuracy: 4.8972e-04
Epoch 14/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.2091e-04 - accuracy: 4.8972e-04
Epoch 15/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.0286e-04 - accuracy: 4.8972e-04
Epoch 16/100
2042/2042 [==============================] - 18s 9ms/step - loss: 2.0839e-04 - accuracy: 4.8972e-04
```

The 2nd Neural Network model with a custom loss function performs much better than the 1st neural network model:

```
predict_2 = model_2.predict(X_test)
predict_2 = scaler.inverse_transform(predict_2)

rmse = np.sqrt(np.mean(predict_2 - y_test)**2)
rmse
```

```
0.8854498073032924
```

Finally, I stack both neural network models and take the average prediction scores to improve the prediction.

```
ensembling_predict = np.mean(np.hstack((predict_1, predict_2)), axis=1)
```

```
from sklearn.metrics import mean_squared_error

rmse = np.sqrt(np.mean(ensembling_predict - y_test)**2)
rmse
```

```
2.1283434186662946
```

For the final result, we can see the RMSE values are about the average of both models. I decided to use the Neural Network model with a custom loss function to compare the prediction with the real stock price and a 20 days SME (Simple Moving Average) as a baseline. We can see the 2nd neural network model performs very well with only 0.885 RMSE. It also fits closely to the 20 days Simple Moving Average (SMA).