LSTM Stock Predictor Using Closing Prices

In this notebook, you will build and train a custom LSTM RNN that uses a 10 day window of Bitcoin closing prices to predict the 11th day closing price.

You will need to:

1. Prepare the data for training and testing
2. Build and train a custom LSTM RNN
3. Evaluate the performance of the model

Data Preparation

In this section, you will need to prepare the training and testing data for the model. The model will use a rolling 10 day window to predict the 11th day closing price.

You will need to:

1. Use the `window_data` function to generate the X and y values for the model.
2. Split the data into 70% training and 30% testing
3. Apply the MinMaxScaler to the X and y values
4. Reshape the X_train and X_test data for the model. Note: The required input format for the LSTM is:

`reshape((X_train.shape[0], X_train.shape[1], 1))`

In [1]:
```
import numpy as np
import pandas as pd
import hvplot.pandas
```

In [2]:
```
# Set the random seed for reproducibility
# Note: This is for the homework solution, but it is good practice to comment this out and run multiple experiments to evaluate your model
from numpy.random import seed
seed(1)
from tensorflow import random
random.set_seed(2)
```

In [3]:
```
# Load the fear and greed sentiment data for Bitcoin
df = pd.read_csv('btc_sentiment.csv', index_col="date", infer_datetime_format=True, parse_dates=True)
df = df.drop(columns="fng_classification")
df.head()
```

Out[3]:

	fng_value
date	
2019-07-29	19
2019-07-28	16
2019-07-27	47
2019-07-26	24
2019-07-25	42

In [4]:
```
# Load the historical closing prices for Bitcoin
df2 = pd.read_csv('btc_historic.csv', index_col="Date", infer_datetime_format=True, parse_dates=True)['Close']
df2 = df2.sort_index()
df2.tail()
```

Out[4]:
```
Date
2019-07-25    9882.429688
2019-07-26    9847.450195
2019-07-27    9478.320313
2019-07-28    9531.769531
2019-07-29    9529.889648
Name: Close, dtype: float64
```

In [5]:
```
# Join the data into a single DataFrame
df = df.join(df2, how="inner")
df.tail()
```

Out[5]:

	fng_value	Close
2019-07-25	42	9882.429688
2019-07-26	24	9847.450195
2019-07-27	47	9478.320313
2019-07-28	16	9531.769531
2019-07-29	19	9529.889648

In [6]:
```
df.head()
```

Out[6]:

	fng_value	Close
2018-02-01	30	9114.719727
2018-02-02	15	8870.820313
2018-02-03	40	9251.269531
2018-02-04	24	8218.049805
2018-02-05	11	6937.080078

In [7]:
```
# This function accepts the column number for the features (X) and the target (y)
# It chunks the data up with a rolling window of Xt-n to predict Xt
# It returns a numpy array of X any y
def window_data(df, window, feature_col_number, target_col_number):
    X = []
    y = []
    for i in range(len(df) - window - 1):
        features = df.iloc[i:(i + window), feature_col_number]
        target = df.iloc[(i + window), target_col_number]
        X.append(features)
        y.append(target)
    return np.array(X), np.array(y).reshape(-1, 1)
```

In [8]:
```
# Predict Closing Prices using a 10 day window of previous closing prices
# Then, experiment with window sizes anywhere from 1 to 10 and see how the model performance changes
window_size = 10

# Column index 0 is the 'fng_value' column
# Column index 1 is the `Close` column
feature_column = 1
target_column = 1
X, y = window_data(df, window_size, feature_column, target_column)
```

In [9]:
```
# Use 70% of the data for training and the remaineder for testing
split = int(0.7 * len(X))
X_train = X[: split]
X_test = X[split:]
y_train = y[: split]
y_test = y[split:]
```

In [10]:
```
from sklearn.preprocessing import MinMaxScaler
# Use the MinMaxScaler to scale data between 0 and 1.
scaler = MinMaxScaler()

# scaler.fit(X_train)
# X_train = scaler.transform(X_train)
# X_test = scaler.transform(X_test)

# scaler.fit(y_train)
# y_train = scaler.transform(y_train)
# y_test = scaler.transform(y_test)
```

In [11]:
```
scaler.fit(X_train)
X_train_scaler = scaler.transform(X_train)
X_test_scaler = scaler.transform(X_test)

scaler.fit(y_train)
y_train_scaler = scaler.transform(y_train)
y_test_scaler = scaler.transform(y_test)
```

In [12]:
```
# Reshape the features for the model
X_train_reshape = X_train_scaler.reshape((X_train_scaler.shape[0], X_train_scaler.shape[1], 1))
X_test_reshape = X_test_scaler.reshape((X_test_scaler.shape[0], X_test_scaler.shape[1], 1))
```

In [13]:
```
# Reshape the features for the model
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))
```

Build and Train the LSTM RNN

In this section, you will design a custom LSTM RNN and fit (train) it using the training data.

You will need to:

1. Define the model architecture
2. Compile the model
3. Fit the model to the training data

Hints:

You will want to use the same model architecture and random seed for both notebooks. This is necessary to accurately compare the performance of the FNG model vs the closing price model.

In [14]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

In [15]:
```
# Build the LSTM model.
# The return sequences need to be set to True if you are adding additional LSTM layers, but
# You don't have to do this for the final layer.
# Note: The dropouts help prevent overfitting
# Note: The input shape is the number of time steps and the number of indicators
# Note: Batching inputs has a different input shape of Samples/TimeSteps/Features

model = Sequential()

number_units = 30
dropout_fraction = 0.5

# Layer 1
model.add(LSTM(
    units=number_units,
    return_sequences=True,
    input_shape=(X_train_scaler.shape[1], 1))
    )
model.add(Dropout(dropout_fraction))
# Layer 2
model.add(LSTM(units=number_units, return_sequences=True))
model.add(Dropout(dropout_fraction))
# Layer 3
model.add(LSTM(units=number_units))
model.add(Dropout(dropout_fraction))
# Output layer
model.add(Dense(1))
```

```
2021-11-29 23:14:20.324061: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in per
formance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [16]:
```
# Compile the model
model.compile(optimizer="adam", loss="mean_squared_error")
```

In [17]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 10, 30)            3840

 dropout (Dropout)           (None, 10, 30)            0

 lstm_1 (LSTM)               (None, 10, 30)            7320

 dropout_1 (Dropout)         (None, 10, 30)            0

 lstm_2 (LSTM)               (None, 30)                7320

 dropout_2 (Dropout)         (None, 30)                0

 dense (Dense)               (None, 1)                 31

=================================================================
Total params: 18,511
Trainable params: 18,511
Non-trainable params: 0
_________________________________________________________________
```

In [18]:
```
# Summarize the model
# YOUR CODE HERE!
```

In [19]:
```
# Train the model
# Use at least 10 epochs
# Do not shuffle the data
# Experiement with the batch size, but a smaller batch size is recommended
# %%time
# Train the model
model.fit(X_train_scaler, y_train_scaler, epochs=20, shuffle=False, batch_size=1, verbose=1)
```

```
Epoch 1/20
372/372 [==============================] - 4s 4ms/step - loss: 0.0433
Epoch 2/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0282
Epoch 3/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0341
Epoch 4/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0300
Epoch 5/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0266
Epoch 6/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0249
Epoch 7/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0222
Epoch 8/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0208
Epoch 9/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0228
Epoch 10/20
372/372 [==============================] - 2s 5ms/step - loss: 0.0219
Epoch 11/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0155
Epoch 12/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0170
Epoch 13/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0159
Epoch 14/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0155
Epoch 15/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0142
Epoch 16/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0142
Epoch 17/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0151
Epoch 18/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0142
Epoch 19/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0125
Epoch 20/20
372/372 [==============================] - 2s 4ms/step - loss: 0.0117
```

Out[19]:
```
<keras.callbacks.History at 0x7f97f1f7b750>
```

Model Performance

In this section, you will evaluate the model using the test data.

You will need to:

1. Evaluate the model using the `X_test` and `y_test` data.
2. Use the X_test data to make predictions
3. Create a DataFrame of Real (y_test) vs predicted values.
4. Plot the Real vs predicted values as a line chart

Hints

Remember to apply the `inverse_transform` function to the predicted and y_test values to recover the actual closing prices.

In [20]:
```
model.evaluate(X_test_scaler, y_test_scaler)
```

```
5/5 [==============================] - 1s 3ms/step - loss: 0.0420
```

Out[20]:
```
0.042018868029117584
```

In [26]:
```
# model.evaluate(X_test, y_test)
```

In [27]:
```
# Evaluate the model
# model.evaluate(X_test, y_test)
```

In [23]:
```
# Make some predictions
predicted = model.predict(X_test_scaler)
```

In [24]:
```
# Recover the original prices instead of the scaled version
predicted = scaler.inverse_transform(predicted)
real_prices = scaler.inverse_transform(y_test_scaler.reshape(-1, 1))
```

In [28]:
```
# # Create a DataFrame of Real and Predicted values
# stocks = pd.DataFrame({
#     "Real": real_prices.ravel(),
#     "Predicted": predicted_prices.ravel()
# }, index = df.index[-len(real_prices): ])
# stocks.head()
```

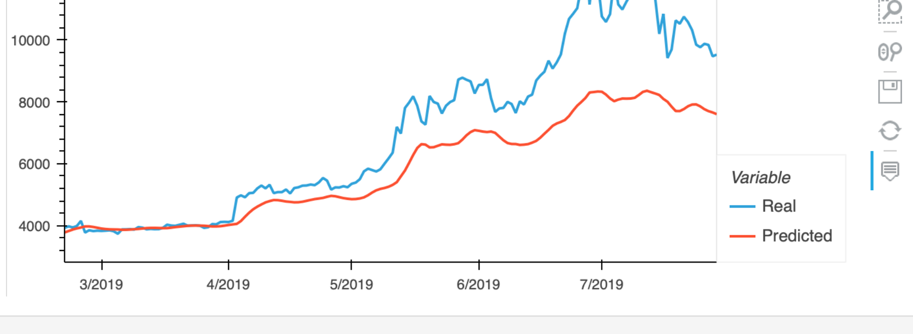
In [29]:
```
stocks = pd.DataFrame({
    "Real": real_prices.ravel(),
    "Predicted": predicted.ravel()
}, index = df.index[-len(real_prices): ])
stocks.head()
```

Out[29]:

	Real	Predicted
2019-02-20	3924.239990	3774.347412
2019-02-21	3974.050049	3820.133545
2019-02-22	3937.040039	3868.391113
2019-02-23	3983.530029	3906.191162
2019-02-24	4149.089844	3934.825195

In [30]:
```
# Plot the real vs predicted values as a line chart
stocks.hvplot()
```

Out[30]:



In [ ]: