Name: Ngô Lê Ngọc Quý . **Student ID:** 21040003

Course: Artificial Intelligence

Lecturer: Trần Thế Vũ

CHALLENGE 2 REPORT

I. Overview

The primary objective of this challenge is to train a Recurrent Neural Network (RNN) model to develop a time series forecasting model that predicts the future values of cryptocurrencies and deploy this model as a web application. The project consists of three main phases: data collection and preprocessing, model building and predicting, and deployment. The model will be trained to predict Bitcoin (BTC) and Ethereum (ETH) and users will be able to see the prediction of cryptocurrencies through a web-based interface.

Github repository of the challenge: [https://github.com/ngocquyxs/-AI-Challenge-2]

II. Phases

1. Data Collection and Preprocessing

1.1. Downloading Datasets

The code sets up a date range for downloading a dataset spanning the last 2 years.

```
# Date range specification – set the current day for the last day of dataset
d1 = today.strftime("%Y-%m-%d")
end_date = d1
# Calculate Start Date – choose a starting day of 4 years ago for data
d2 = date.today() - timedelta(days=365*4)
d2 = d2.strftime("%Y-%m-%d")
start_date = d2
```

This code is a function for downloading historical financial data for a cryptocurrency from Yahoo Finance.

```
# "download_data" function
def download_data(crypto_name, start_date, end_date):
    data = yf.download(cryto_name,
                       start=start_date,
                       end=end_date,
                       progress=False)
    data["Date"] = data.index
```

```
    data = data[["Date", "Open", "High", "Low", "Close", "Adj Close", "Volume"]]

    data.reset_index(drop=True, inplace=True)

    return data
```

1.2. Exploratory Data Analyst

This code is a function designed to print key information about a given DataFrame.

```
# "information" function

def information(data):

  print(data.info())

  print(data.describe())

  print("Has Null values?:", data.isnull().values.any())

  print("Shape:", data.shape)
```

The code defines three functions (plot, high_low_plot, and open_close_plot) for creating different types of plots using the Plotly library.

```
# "plot" function

def plot(y):

  // 'cycle' to assign names to each trace.

  names = cycle(['Open Price','Close Price','High Price','Low Price'])

  // creates a line plot

  fig = px.line(y, x=y.Date, y=[y['Open'], y['Close'], y['High'], y['Low']], labels={'Date':
'Date','value':'Stock value'})

  fig.update_layout(title_text='Performance Graph', font_size=15, font_color='black',
legend_title_text='Stock Parameters')

  fig.for_each_trace(lambda t:  t.update(name = next(names)))

  fig.update_xaxes(showgrid=False)

  fig.update_yaxes(showgrid=False)

  fig.show()
```

```
    return fig


# "high_low_plot" function
def high_low_plot(df, y, new_order):
    monthvise_high = y.groupby(df['Date'].dt.strftime('%B'))['High'].max()
    monthvise_high = monthvise_high.reindex(new_order, axis=0)
    // 'groupby' to aggregate maximum high and minimum low prices for each month.
    monthvise_low = y.groupby(df['Date'].dt.strftime('%B'))['Low'].min()
    monthvise_low = monthvise_low.reindex(new_order, axis=0)
    // 'go.Figure' constructor to initialize the plot.
    fig = go.Figure()
    fig.add_trace(go.Bar(
        x = monthvise_high.index,
        y = monthvise_high,
        name='High Price',
        marker_color='rgb(0, 153, 204)'
    ))
    fig.add_trace(go.Bar(
        x = monthvise_low.index,
        y = monthvise_low,
        name='Low Price',
        marker_color='rgb(255, 128, 0)'
    ))

    fig.update_layout(barmode='group', title=' Monthly High Price and Low Price')
    fig.show()
    return fig
```

```
# "open_close_plot" function

def open_close_plot(data):

    // 'go.Figure' constructor to initialize the plot.

    fig = go.Figure()


    fig.add_trace(go.Bar(

        x = data.index,

        y = data['Open'],

        name ='Open Price',

        marker_color='crimson'

    ))

    fig.add_trace(go.Bar(

        x=data.index,

        y=data['Close'],

        name='Close Price',

        marker_color='lightsalmon'

    ))


    fig.update_layout(barmode='group', xaxis_tickangle=-45, title='Monthly Open-Close Price Analysis')

    fig.show()

    return fig
```

This function appears to be designed for analyzing stock price data on a yearly basis.

```
# "data_yearly" function

def data_yearly(df, start_year, end_year):

    // Date conversion
```

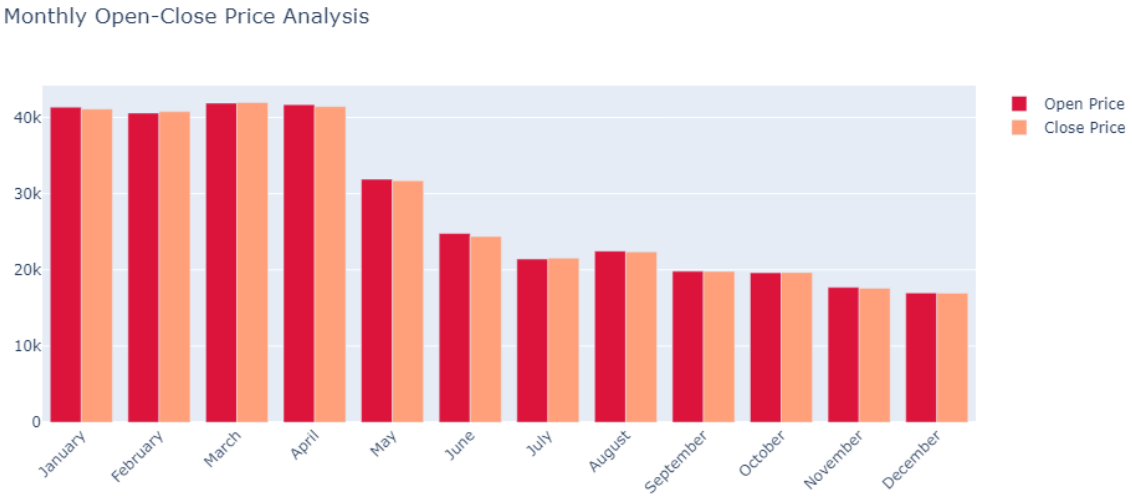
```
    df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
   // Filter data by year
    y = df.loc[(df['Date'] >= f'{str(start_year)}-01-01')
                & (df['Date'] < f'{str(end_year)}-01-01')]
   // Drop columns
    y.drop(y[['Adj Close','Volume']],axis=1)
   // Monthly aggregation
    monthvise= y.groupby(y['Date'].dt.strftime('%B'))[['Open','Close']].mean()
   // Reorder months
    new_order = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September',
'October', 'November', 'December']
    monthvise = monthvise.reindex(new_order, axis=0)
    print(monthvise)
    // Visualization
    open_close_plot(monthvise)
    high_low_plot(df, y, new_order)
    plot(y)
```

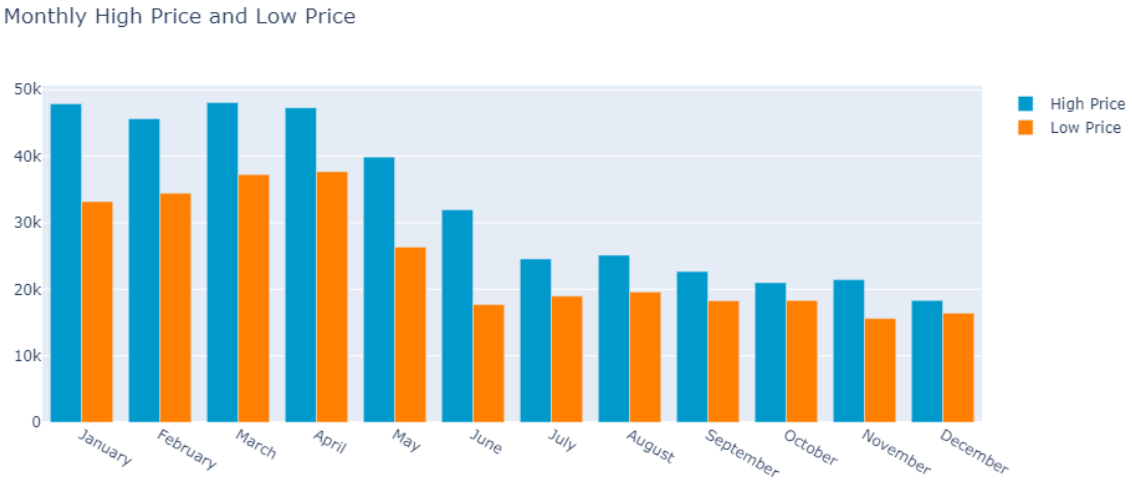
This function appears to be designed to display yearly data for a given dataset using the **"data_yearly"** function.

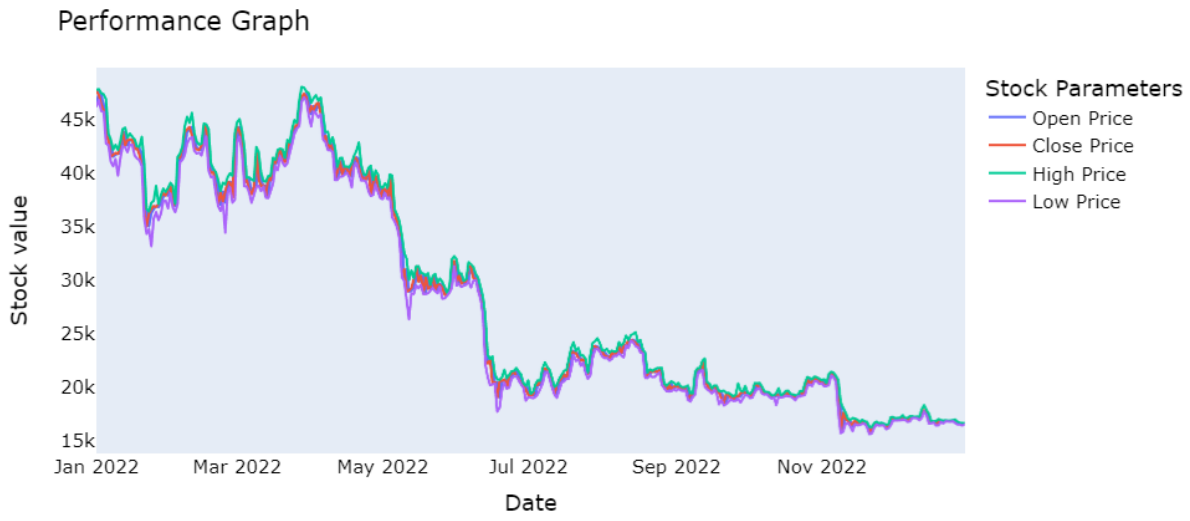
```
# "show_data" function
def show_data(data):
    for year in range(2019, 2024):
        print(f"In {year}")
        print(data_yearly(data, year, year + 1))
```

BTC data in 2022

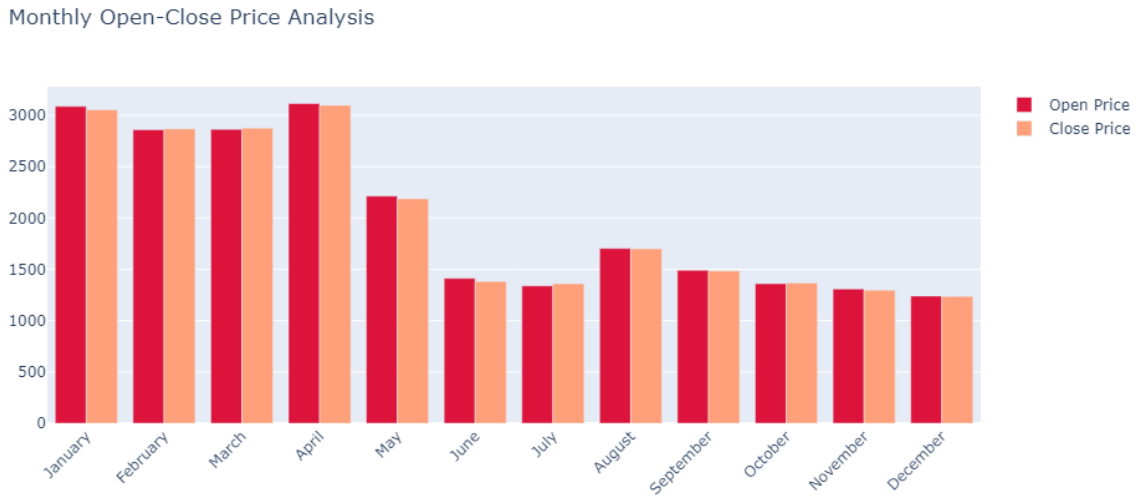
Date	Open	Close
January	41368.073463	41114.422379
February	40591.103934	40763.474051
March	41889.148438	41966.237525
April	41694.653646	41435.319661
May	31900.711127	31706.105217
June	24783.338477	24383.685482
July	21424.733052	21539.253843
August	22471.866557	22366.266318
September	19821.353711	19804.779232
October	19616.090285	19650.525643
November	17711.480599	17600.814323
December	16969.578818	16949.608808

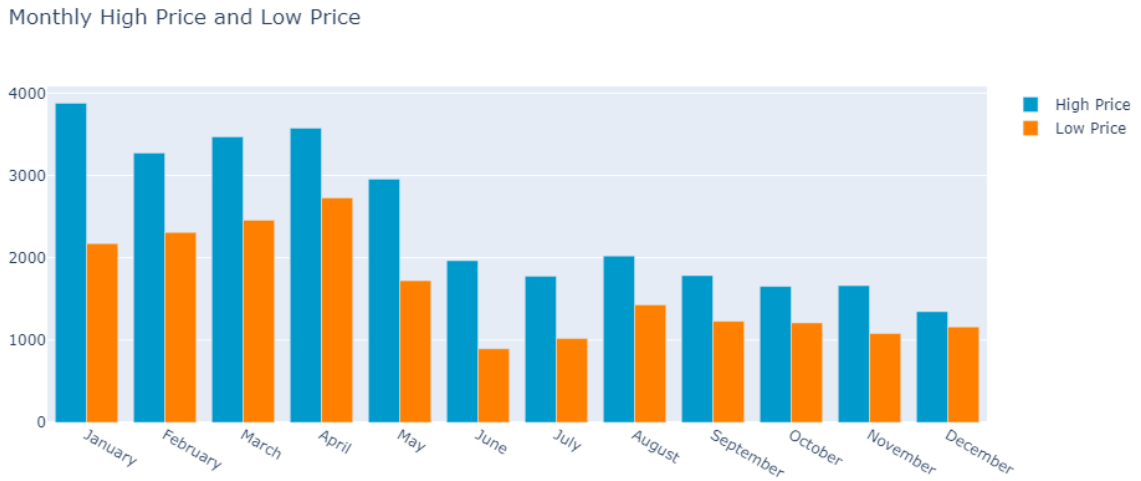


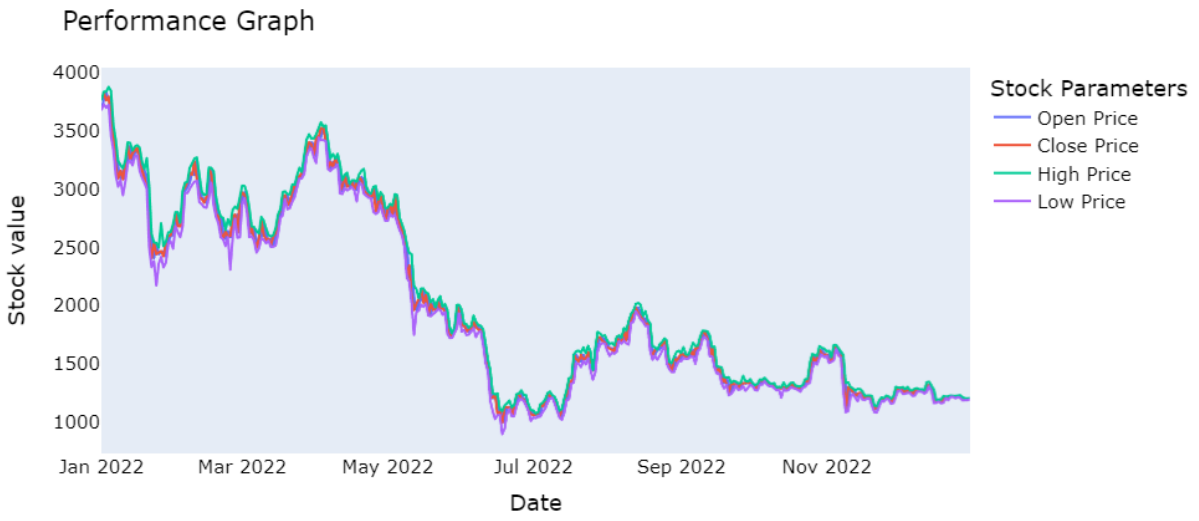




ETH data in 2022







1.3. Preprocessing

This function appears to be designed for creating input-output pairs for a time series forecasting problem.

```
# "create_dataset" function

def create_dataset(data, time_step):
    dataX, dataY = [], []
    for i in range(len(data)-time_step-1):
        a = data[i:(i+time_step), 0]
        dataX.append(a)
        dataY.append(data[i + time_step, 0])
    return np.array(dataX), np.array(dataY)
```

This function provides a comprehensive preprocessing pipeline for time series data, including visualization, filtering, normalization, and splitting into training and testing sets.

```
# "preprocessing" function

def preprocessing(data):
    // Extract columns
    close_df = data[["Date", "Close"]]
```

```
    print("Shape of close dataframe:", close_df.shape)


    // Visualize historical performance

    fig = px.line(close_df, x= close_df.Date, y= close_df.Close,
labels={'date':'Date','close':'Close Price'})

    fig.update_traces(marker_line_width=2, opacity=0.8, marker_line_color='pink')

    fig.update_layout(title_text='Historical Price Performance Overview (2019-2023)',
plot_bgcolor='white',

                font_size=15, font_color='black')

    fig.update_xaxes(showgrid=False)

    fig.update_yaxes(showgrid=False)

    fig.show()


    // Filter data

    close_df = close_df[close_df['Date'] > '2022-11-23']

    close_stock = close_df.copy()

    print("Total data for prediction: ",close_df.shape[0])


    // Visualize predictive analysis

    fig = px.line(close_df, x=close_df.Date,
y=close_df.Close,labels={'date':'Date','close':'Close Stock'})

    fig.update_traces(marker_line_width=2, opacity=0.8, marker_line_color='orange')

    fig.update_layout(title_text='Predictive Analysis for Future Close Prices',

                plot_bgcolor='white', font_size=15, font_color='black')

    fig.update_xaxes(showgrid=False)

    fig.update_yaxes(showgrid=False)

    fig.show()
```

```
// Delete 'Date' column and normalize Data
del close_df['Date']
scaler=MinMaxScaler(feature_range=(0,1))
close_df=scaler.fit_transform(np.array(close_df).reshape(-1,1))
print("Dataset:", close_df.shape)

// Split dataset into Training and Testing sets
training_size=int(len(close_df)*0.80)
test_size=len(close_df)-training_size
train_data,test_data=close_df[0:training_size,:],close_df[training_size:len(close_df),:1]
print("train_data: ", train_data.shape)
print("test_data: ", test_data.shape)

return close_stock, close_df, train_data, test_data, scaler
```

2. Model Building and Predicting

2.1. Preparing

This code is necessary for preparing price data for machine learning.

```
# Prepare data
close_stock, close_df, train_data, test_data, scaler = preprocessing(data)
close_stock.to_csv("training_data.csv")

time_step = 1
X_train, y_train = create_dataset(train_data, time_step)
```

```
X_test, y_test = create_dataset(test_data, time_step)

# Display the shape

print("X_train: ", X_train.shape)

print("y_train: ", y_train.shape)

print("X_test: ", X_test.shape)

print("y_test", y_test.shape)


# Reshape input for LSTM

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)

print("X_train: ", X_train.shape)

print("X_test: ", X_test.shape)
```

2.2. Building Model

This code assumes that the data has been preprocessed and formatted appropriately for time series prediction with an LSTM network.

```
# Create a Sequential Model

model=Sequential()

# Add an LSTM Layer

model.add(LSTM(10,input_shape=(None,1),activation="relu"))

# Add a Dense Output Layer

model.add(Dense(1))

# Compile the Model

model.compile(loss="mean_squared_error",optimizer="adam")
```

```
# Train the Model

history = model.fit(X_train,y_train,validation_data=(X_test,y_test),epochs=80,batch_size=1,verbose=1)
```

This code visualizes the training and validation of loss values during the training of a machine learning model, specifically an LSTM neural network for time series prediction.

```
# Plot training & validation loss values

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper right')


plt.show()
```

2.3. Making Prediction

This code is making predictions using the trained LSTM model on both the training and testing datasets.

```
# Make predictions

train_predict=model.predict(X_train)

test_predict=model.predict(X_test)

train_predict.shape, test_predict.shape
```

This code is transforming the values back to their original scale.

```
# Inverse transformation
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))

# Calculate and print explained variance regression scores
print("Train data explained variance regression score:",
    explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:",
    explained_variance_score(original_ytest, test_predict))

# Calculate and print R2 scores
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
```

This code generates a plot to visualize the comparison between the original close price of a stock and the predicted close prices, both for the training and testing sets.

```
# Predictions plot
look_back = time_step

# Initialize arrays for predicted values
trainPredictPlot = np.empty_like(close_df)
trainPredictPlot[:, :] = np.nan

# Fill in the predicted values for the training set
```

```
trainPredictPlot[look_back:len(train_predict) + look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# Shift test predictions for plotting
testPredictPlot = np.empty_like(close_df)
testPredictPlot[:, :] = np.nan

# Fill in the predicted values for the testing set
testPredictPlot[len(train_predict) + (look_back * 2) + 1:len(close_df) - 1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

# Create a DataFrame for plotting
plotdf = pd.DataFrame({
    'date': close_stock['Date'],
    'original_close': close_stock['Close'],
    'train_predicted_close': trainPredictPlot.reshape(1, -1)[0].tolist(),
    'test_predicted_close': testPredictPlot.reshape(1, -1)[0].tolist()
})

# Plotting using Plotly Express
fig = px.line(plotdf, x=plotdf['date'], y=[plotdf['original_close'],
plotdf['train_predicted_close'],
                                           plotdf['test_predicted_close']],
              labels={'value': 'Stock price', 'date': 'Date'})
fig.update_layout(
    title_text='Comparison between original close price vs predicted close price',
```

```
    plot_bgcolor='white', font_size=15, font_color='black', legend_title_text='Close Price'
)

# Update trace names for better visualization
names = cycle(['Original close price', 'Train predicted close price', 'Test predicted close price'])
fig.for_each_trace(lambda t: t.update(name=next(names)))

# Update layout to remove gridlines
fig.update_xaxes(showgrid=False)
fig.update_yaxes(showgrid=False)

# Show the plot
fig.show()
```

This code appears to generate predictions for the next 30 days using the trained LSTM model.

```
# Prepare input dataset
x_input = test_data[len(test_data) - time_step:].reshape(1, -1)
temp_input = list(x_input)
temp_input = temp_input[0].tolist()

from numpy import array

lst_output = []
n_steps = time_step
```

```
i = 0
pred_days = 30

while i < pred_days:

    if len(temp_input) > time_step:

        x_input = np.array(temp_input[1:])
        x_input = x_input.reshape(1, -1)
        x_input = x_input.reshape((1, n_steps, 1))

        yhat = model.predict(x_input, verbose=0)
        temp_input.extend(yhat[0].tolist())
        temp_input = temp_input[1:]

        lst_output.extend(yhat.tolist())
        i = i + 1

    else:

        x_input = x_input.reshape((1, n_steps, 1))
        yhat = model.predict(x_input, verbose=0)
        temp_input.extend(yhat[0].tolist())

        lst_output.extend(yhat.tolist())
        i = i + 1
```

```
print("Output of predicted next days: ", len(lst_output))
```

This code concatenates the original closing stock prices with the predicted closing prices and then plots the entire series, including both original and predicted values.

```
# Concatenate original and predicted data

lstmdf = np.concatenate([close_df, np.array(lst_output).reshape(-1, 1)])


# Inverse transform to the original scale

lstmdf = scaler.inverse_transform(lstmdf)


# Flatten the array

lstmdf = lstmdf.flatten()


# Create a DataFrame with the same index as close_df

plot_df = pd.DataFrame(lstmdf, index=np.arange(len(lstmdf)), columns=['Close'])


# Plotting

fig = px.line(x=plot_df.index, y=plot_df['Close'], labels={'y': 'Stock price', 'x':
'Timestamp'})

fig.update_layout(title_text='Plotting whole closing stock price with prediction',

                  plot_bgcolor='white', font_size=15, font_color='black',
legend_title_text='Stock')


# Add original and predicted close prices to the legend

names = cycle(['Original close price', 'Predicted close price'])

fig.for_each_trace(lambda t: t.update(name=next(names)))
```

```
fig.update_xaxes(showgrid=False)
fig.update_yaxes(showgrid=False)
fig.show()
```

This code saves the model.

```
model.save("data_lstm.h5")
```

3. Deployment

3.1. Using Streamlit

This code is a simple implementation of a Streamlit web application for cryptocurrency price prediction.

```
# Load models
btc_model = load_model('btc_lstm.h5')
eth_model = load_model('eth_lstm.h5')

# Load scalers
btc_scaler = MinMaxScaler()
eth_scaler = MinMaxScaler()

# Load BTC training data
btc_data = pd.read_csv('btc_training_data.csv')
btc_scaler.fit(btc_data[['Close']].values.reshape(-1, 1))

# Load ETH training data
eth_data = pd.read_csv('eth_training_data.csv')
```

```
eth_scaler.fit(eth_data[['Close']].values.reshape(-1, 1))


def get_prediction_data(model, scaler, training_data, days_to_predict):
    // Calculate prediction dates
    last_training_date = datetime.strptime(training_data['Date'].max(), '%Y-%m-%d')
    prediction_dates = [(last_training_date + timedelta(days=i)).strftime('%Y-%m-%d') for i in range(1, days_to_predict + 1)]


    // Create input data for the model
    last_sequence = training_data['Close'].tail(15).values.reshape(-1, 1)
    input_sequence = scaler.transform(last_sequence).reshape(1, 15, 1)


    // Predict values
    predicted_values_scaled = []
    for _ in range(days_to_predict):
        prediction = model.predict(input_sequence, verbose=0)
        predicted_values_scaled.append(prediction[0, 0])
        input_sequence = np.roll(input_sequence, -1)
        input_sequence[0, -1, 0] = prediction[0, 0]


    // Convert predicted values back to the original scale
    predicted_values = scaler.inverse_transform(np.array(predicted_values_scaled).reshape(-1, 1)).reshape(-1)


    return prediction_dates, predicted_values


# Streamlit app
```

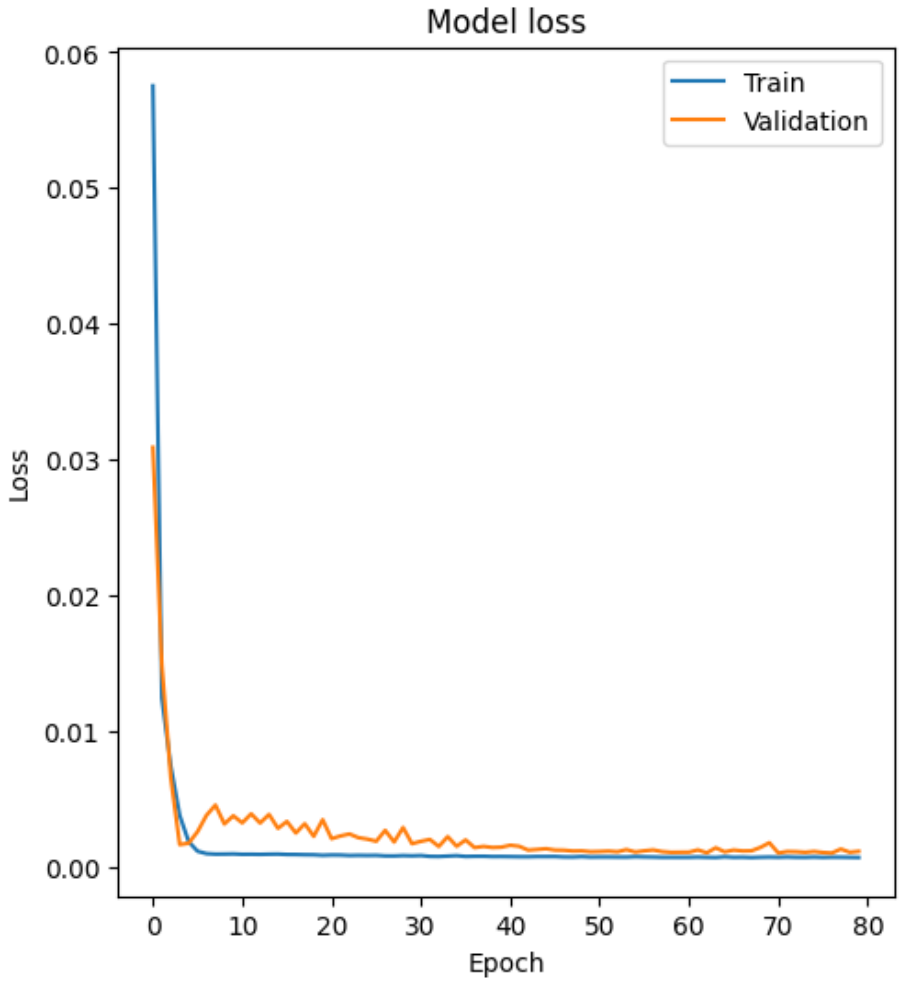
```
st.title('Cryptocurrency Price Prediction')


# Sidebar with user input

crypto_choice = st.sidebar.selectbox('Select Cryptocurrency', ['BTC-USD', 'ETH-USD'])

days_to_predict = st.sidebar.slider('Select Number of Days to Predict', 1, 30, 7)


# Fetch data and models based on user input

if crypto_choice == 'BTC-USD':

    model = btc_model

    scaler = btc_scaler

    training_data = btc_data

elif crypto_choice == 'ETH-USD':

    model = eth_model

    scaler = eth_scaler

    training_data = eth_data


# Get prediction data

prediction_dates, predicted_values = get_prediction_data(model, scaler, training_data,
days_to_predict)


# Display predicted prices

st.subheader(f'Predicted Prices for {crypto_choice} in the Next {days_to_predict} Days')

st.line_chart(pd.Series(predicted_values, index=prediction_dates))
```

III. Conclusion

This challenge has successfully implemented a time series forecasting model that predicts the future values of cryptocurrencies, evaluating its performance on test data, visualizing key training metrics, and integrating it into a Streamlit-based web application.

1. BTC-USD

Plot training & validation loss values



Variance regression score

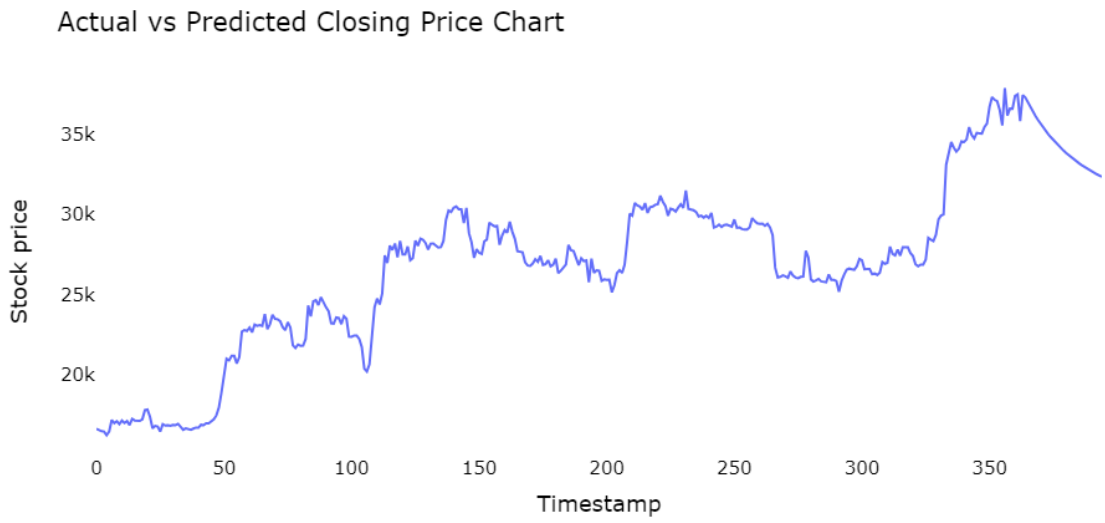
- Train data: 0.9846903797885722
- Test data: 0.972691999337772

R2 score

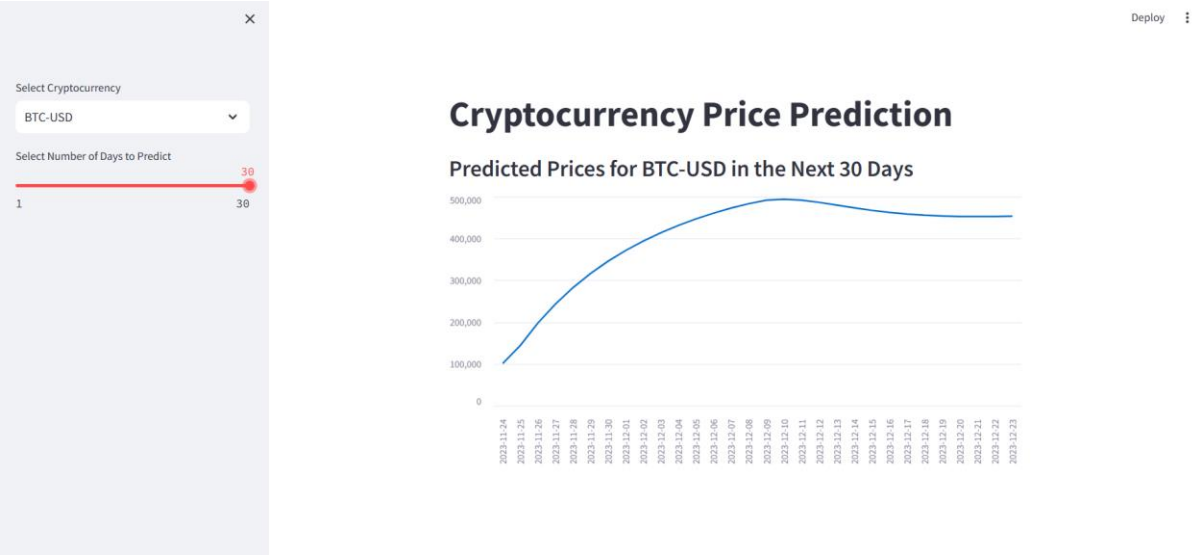
- Train data: 0.9845696500785845
- Test data: 0.9699498230896972

Predictions plot on actual data

Plot of actual and predicted data for 30 days

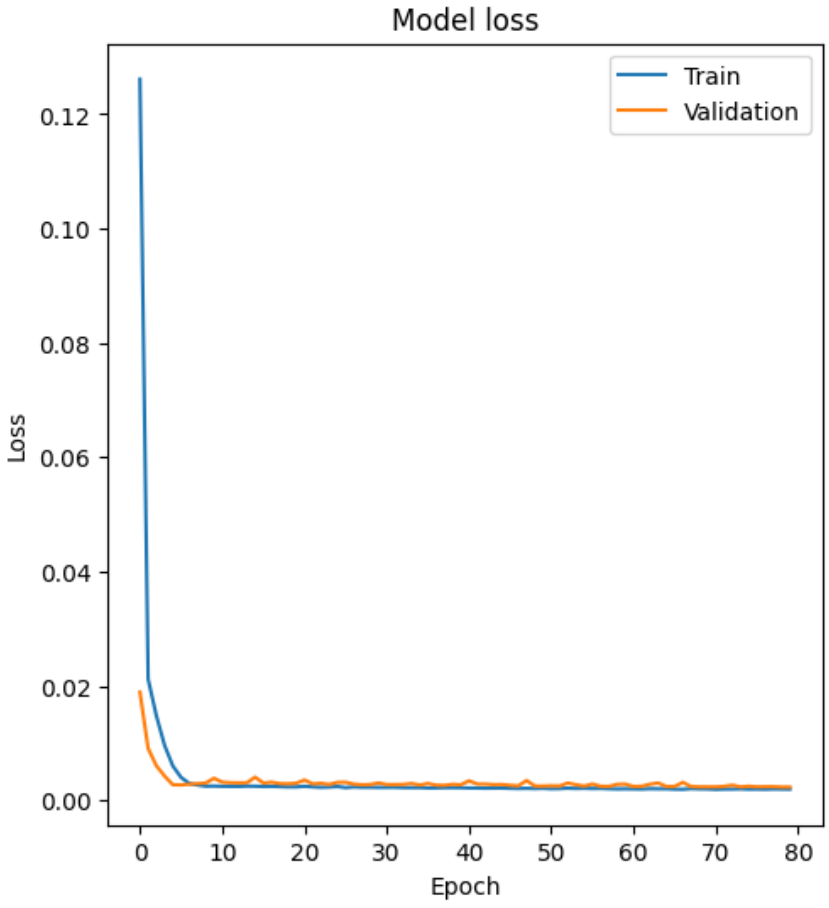


Streamlit interface for prediction



2. *ETH-USD*

Plot training & validation loss values



Variance regression score

- Train data: 0.9691209956327952
- Test data: 0.9265540921618313

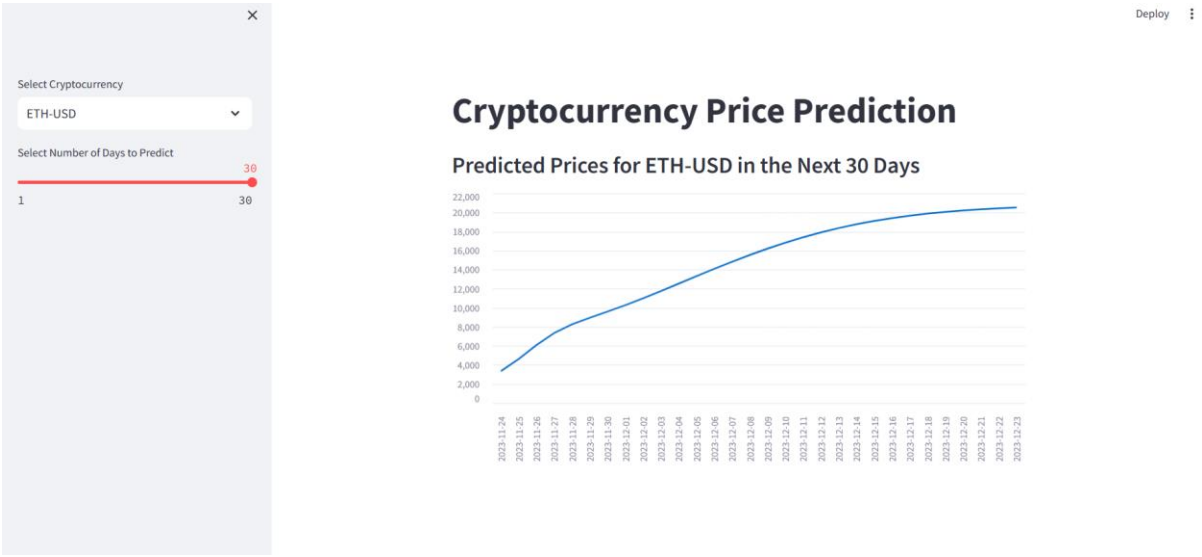
R2 score

- Train data: 0.9685335459279051
- Test data: 0.9265075731088035

Predictions plot on actual data

Plot of actual and predicted data for 30 days

Streamlit interface for prediction



IV. Reference

The above challenge has reference from

- Code and notebooks for model:

[https://www.kaggle.com/code/meetnagadia/bitcoin-price-prediction-using-lstm]

[https://github.com/Ali619/Bitcoin-Price-Prediction-LSTM/blob/master/Bitcoin_Price_Prediction.ipynb]