**Task 2: Data Processing 1**

**Muhammad Ali, 103960437, COS30018, 23/08/2024.**

**Requirements / Deliverables –**

* Write a function that improves the data processing for the initial v0.1 stock prediction code.
* The function should load and process a dataset with the following requirements:
  + Allows you to specify the start date and the end date for the whole dataset as inputs.
  + Allows you to deal with the NaN issue in the data.
  + Allows you to use different methods to split the data into train/test data. E.g. You can split it according to some specified ratio of train/test and you can specify to split it by date or randomly.
  + Allows you to store the downloaded data on your local machine for future uses and to load the data locally to save time.
  + The function will also allow you to have an option to scale your feature columns and store the scalers in a data structure to allow future access to these scalers.

**Result –**

The provided function fulfils the requirements and uses an advanced data processing technique that utilizes Simple Moving Averages (SMA) and Relative Strength Index (RSI) for increased accuracy when training the model.

You must install the provided *requirements.txt in repository folder Task 2.*

*A graph of loss and training loss

Description automatically generated*

The model now provides a graph that displays the model training loss and its validation loss (gradual decline is *good*).

A graph with green lines and numbers

Description automatically generated

Improve accuracy compared to the previous model. Accuracy can be further improved using more advanced techniques for architecture optimisation, data handling, and training techniques.

***Load\_and\_process.py***

import os  
import pandas as pd  
import numpy as np  
from sklearn.preprocessing import MinMaxScaler  
import yfinance as yf  
from sklearn.model\_selection import train\_test\_split  
  
def add\_technical\_indicators(df):  
 *"""  
 Add technical indicators to the stock data DataFrame.  
 This includes Simple Moving Average (SMA) and Relative Strength Index (RSI).  
  
 Parameters:  
 - df (pd.DataFrame): DataFrame containing stock data with at least a 'Close' column.  
  
 Returns:  
 - df (pd.DataFrame): DataFrame with added technical indicators.  
 """* # Simple Moving Average (SMA)  
 df['SMA\_20'] = df['Close'].rolling(window=20).mean()  
 df['SMA\_50'] = df['Close'].rolling(window=50).mean()  
  
 # Relative Strength Index (RSI)  
 delta = df['Close'].diff(1)  
 gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()  
 loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()  
 rs = gain / loss  
 df['RSI'] = 100 - (100 / (1 + rs))  
  
 df.fillna(0, inplace=True) # Fill NaNs resulting from rolling operations  
  
 return df  
  
def load\_and\_process\_data\_with\_gap(ticker, start\_date, end\_date, handle\_nan='drop',  
 split\_method='date', test\_ratio=0.2, scale\_data=True,  
 save\_local=False, load\_local=False, local\_dir='stock\_data',  
 feature\_columns=None, gap\_period='30D'):  
 *"""  
 Load and process stock market data with a gap between training and testing periods.  
 This function fetches the stock data, processes it (e.g., handling NaNs, adding technical indicators),  
 splits it into training and testing sets, and optionally scales the data.  
  
 Parameters:  
 - ticker (str): Stock ticker symbol.  
 - start\_date (str): Start date for data in 'YYYY-MM-DD' format.  
 - end\_date (str): End date for data in 'YYYY-MM-DD' format.  
 - handle\_nan (str): Method to handle NaN values ('drop', 'fill', 'none').  
 - split\_method (str): How to split the data ('date', 'random').  
 - test\_ratio (float): Ratio of the test set, if split\_method is 'random'.  
 - scale\_data (bool): Whether to scale the data.  
 - save\_local (bool): Whether to save the data locally.  
 - load\_local (bool): Whether to load data from a local file if available.  
 - local\_dir (str): Directory to save/load the data.  
 - feature\_columns (list): List of columns to use as features.  
 - gap\_period (str): Gap period (e.g., '30D' for 30 days) between the training and testing periods.  
  
 Returns:  
 - X\_train, X\_test (np.ndarray): Training and testing feature matrices.  
 - y\_train, y\_test (np.ndarray): Training and testing target vectors.  
 - scaler (MinMaxScaler or None): The scaler object used for data normalization, if scaling is enabled.  
 """* # Load data from Yahoo Finance or local  
 if load\_local and os.path.exists(os.path.join(local\_dir, f"{ticker}.csv")):  
 df = pd.read\_csv(os.path.join(local\_dir, f"{ticker}.csv"), index\_col=0, parse\_dates=True)  
 else:  
 df = yf.download(ticker, start=start\_date, end=end\_date)  
 if save\_local:  
 if not os.path.exists(local\_dir):  
 os.makedirs(local\_dir)  
 df.to\_csv(os.path.join(local\_dir, f"{ticker}.csv"))  
   
 # Handle NaN values  
 if handle\_nan == 'drop':  
 df.dropna(inplace=True)  
 elif handle\_nan == 'fill':  
 df.fillna(method='ffill', inplace=True)  
   
 # Add technical indicators to the DataFrame  
 df = add\_technical\_indicators(df)  
   
 # Determine features and target  
 if feature\_columns is None:  
 feature\_columns = df.columns.tolist()  
 feature\_columns.remove('Close') # Assuming 'Close' is the target by default  
   
 X = df[feature\_columns].values  
 y = df['Close'].values  
  
 # Data splitting logic  
 if split\_method == 'date':  
 gap = pd.to\_timedelta(gap\_period).days  
 test\_start\_idx = int(len(df) \* (1 - test\_ratio)) + gap  
 X\_train, X\_test = X[:test\_start\_idx], X[test\_start\_idx:]  
 y\_train, y\_test = y[:test\_start\_idx], y[test\_start\_idx:]  
 else:  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_ratio, shuffle=True, random\_state=42)  
  
 # Data scaling  
 scaler = None  
 if scale\_data:  
 scaler = MinMaxScaler()  
 X\_train = scaler.fit\_transform(X\_train)  
 X\_test = scaler.transform(X\_test)  
 y\_scaler = MinMaxScaler()  
 y\_train = y\_scaler.fit\_transform(y\_train.reshape(-1, 1)).flatten()  
 y\_test = y\_scaler.transform(y\_test.reshape(-1, 1)).flatten()  
  
 return X\_train, X\_test, y\_train, y\_test, scaler

***stock\_prediction0.2***

import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
import tensorflow as tf  
  
from sklearn.preprocessing import MinMaxScaler  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Dropout, LSTM, GRU  
from tensorflow.keras.regularizers import l2, l1\_l2  
from tensorflow.keras.losses import Huber  
from tensorflow.keras.callbacks import EarlyStopping  
from load\_and\_process import load\_and\_process\_data\_with\_gap  
  
#------------------------------------------------------------------------------  
# Load Data with Additional Features  
#------------------------------------------------------------------------------  
COMPANY = 'CBA.AX'  
TRAIN\_START = '2022-08-01'  
TRAIN\_END = '2024-08-23'  
  
# Load and process data  
X\_train, X\_test, y\_train, y\_test, scalers = load\_and\_process\_data\_with\_gap(  
 ticker=COMPANY,  
 start\_date=TRAIN\_START,  
 end\_date=TRAIN\_END,  
 handle\_nan='drop',  
 split\_method='date',  
 test\_ratio=0.2,  
 scale\_data=True,  
 save\_local=True,  
 load\_local=True,  
 local\_dir='stock\_data',  
 feature\_columns=None # Use all available features  
)  
  
#------------------------------------------------------------------------------  
# Prepare Data  
#------------------------------------------------------------------------------  
PREDICTION\_DAYS = 60  
  
# Print the original shape of the training data  
print("Original shape of X\_train:", X\_train.shape)  
  
# Number of features (columns)  
n\_features = X\_train.shape[1]  
  
# Determine the number of samples and time steps  
n\_samples = X\_train.shape[0]  
  
# Calculate possible time steps based on the total number of elements  
n\_timesteps = X\_train.size // (n\_samples \* n\_features)  
  
print(f"Calculated time steps: {n\_timesteps}")  
  
# Reshape data for LSTM model  
X\_train = np.reshape(X\_train, (n\_samples, n\_timesteps, n\_features))  
  
print("Reshaped X\_train shape:", X\_train.shape)  
  
# Repeat the process for X\_test  
print("Original shape of X\_test:", X\_test.shape)  
n\_samples\_test = X\_test.shape[0]  
X\_test = np.reshape(X\_test, (n\_samples\_test, n\_timesteps, n\_features))  
print("Reshaped X\_test shape:", X\_test.shape)  
  
#------------------------------------------------------------------------------  
# Build the Model  
#------------------------------------------------------------------------------  
# Initialize the Sequential model  
model = Sequential()  
  
# Add the first LSTM layer with dropout  
model.add(LSTM(units=150, return\_sequences=True, input\_shape=(n\_timesteps, n\_features)))  
model.add(Dropout(0.2))  
  
# Add the second LSTM layer with dropout  
model.add(LSTM(units=100, return\_sequences=False))  
model.add(Dropout(0.2))  
  
# Add the output layer  
model.add(Dense(units=1))  
  
# Compile the model using Adam optimizer and Huber loss  
model.compile(optimizer='adam', loss=Huber())  
  
# Early stopping to prevent overfitting  
early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)  
  
# Train the model with validation and early stopping  
history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, verbose=1, validation\_split=0.2, callbacks=[early\_stopping])  
  
#------------------------------------------------------------------------------  
# Plot the Training and Validation Loss  
#------------------------------------------------------------------------------  
plt.plot(history.history['loss'], label='Training Loss')  
plt.plot(history.history['val\_loss'], label='Validation Loss')  
plt.title('Model Loss')  
plt.xlabel('Epoch')  
plt.ylabel('Loss')  
plt.legend()  
plt.show()  
  
#------------------------------------------------------------------------------  
# Test the Model Accuracy on Existing Data  
#------------------------------------------------------------------------------  
# Predict prices using the test set  
predicted\_prices = model.predict(X\_test)  
  
# Print raw predictions before inverse\_transform  
print("Raw predictions before inverse\_transform:", predicted\_prices[:5])  
  
# Assuming y\_train/y\_test were scaled separately with a specific scaler for y  
y\_scaler = MinMaxScaler()  
y\_train = y\_train.reshape(-1, 1)  
y\_scaler.fit(y\_train)  
  
# Inverse transform the predictions to get actual predicted prices  
predicted\_prices = y\_scaler.inverse\_transform(predicted\_prices)  
  
# Print inverse-transformed predictions  
print("Inverse-transformed predictions:", predicted\_prices[:5])  
  
# Display the actual and predicted prices for each day  
for i in range(len(predicted\_prices)):  
 print(f"Day {i + 1}: Actual Price = {y\_test[i]}, Predicted Price = {predicted\_prices[i][0]}")  
  
#------------------------------------------------------------------------------  
# Plot the Test Predictions  
#------------------------------------------------------------------------------  
actual\_prices = y\_test.reshape(-1, 1)  
actual\_prices = y\_scaler.inverse\_transform(actual\_prices)  
  
plt.plot(actual\_prices, color="black", label=f"Actual {COMPANY} Price")  
plt.plot(predicted\_prices, color="green", label=f"Predicted {COMPANY} Price")  
plt.title(f"{COMPANY} Share Price")  
plt.xlabel("Time")  
plt.ylabel(f"{COMPANY} Share Price")  
plt.legend()  
plt.show()  
  
#------------------------------------------------------------------------------  
# Predict Next Day  
#------------------------------------------------------------------------------  
# Use the last PREDICTION\_DAYS days from the test set to predict the next day's price  
real\_data = X\_test[-PREDICTION\_DAYS:]  
real\_data = np.reshape(real\_data, (real\_data.shape[0], real\_data.shape[1], n\_features))  
  
# Predict the next day's price  
prediction = model.predict(real\_data)  
prediction = y\_scaler.inverse\_transform(prediction)  
print(f"Next Day Prediction: {prediction[0][0]}")