Task 2: Data Processing 1

Project leader: Dr Ru Jia

Name: Min Khant Aung
ID: 103833225

Task 2 Report: Data Processing 1

Summary: This report details the process of improving data handling and feature utilization in the existing project code (v0.1) for stock price prediction. The improvements focus on handling multiple features, dealing with NaN values, splitting data, and scaling features. This report also includes explanations of the more complex lines of code to ensure clarity and understanding.

Improvements Overview:

1. **Handling Multiple Features:** The original code only utilized the "Close" feature from the dataset. This has been expanded to allow for the use of multiple features, enhancing the predictive model's ability to capture more complex patterns in the data.
2. **NaN Value Handling:** A new method for dealing with NaN values has been implemented. The user can now choose to either drop rows with NaN values or fill them using various strategies (e.g., forward fill, backward fill, interpolation).
3. **Flexible Data Splitting:** The code now allows for flexible data splitting. Users can choose to split the data by a specific ratio, by date, or randomly, giving more control over how the training and test datasets are defined.
4. **Feature Scaling:** The function now includes an option to scale the feature columns and save the scalers for future use. This ensures consistency in data processing, especially when using the model with new data.
5. **Data Storage and Loading:** To optimize processing time, the function includes an option to save the processed data locally and load it in future runs.

New Function: load_and_process_data

```
def load_and_process_data(
    file_path=None,
    company='CBA.AX',
    start_date='2020-01-01',
    end_date='2023-08-01',
    features=['Open', 'High', 'Low', 'Close', 'Volume'],
    nan_handling='fill',  # 'drop' or 'fill'
    fill_method='ffill',  # 'ffill', 'bfill', 'mean', 'interpolate'
    split_ratio=0.8,
    split_method='random',  # 'date' or 'random'
    scale_data=True,
    save_scalers=True,
    scalers_dir='scalers',
    save_processed_data=True,
    processed_data_dir='processed_data'
```

```
):
    """
    Load and process the stock data with multiple features, handling NaN
values, splitting data,
    and scaling features as specified by the user.
    """

    # Load data from file or download if not provided
    if file_path and os.path.exists(file_path):
        data = pd.read_csv(file_path)
    else:
        data = yf.download(company, start=start_date, end=end_date)[features]
        if save_processed_data:
            os.makedirs(processed_data_dir, exist_ok=True)
            data.to_csv(os.path.join(processed_data_dir,
f'{company}_data.csv'))

    # Handle NaN values
    if nan_handling == 'drop':
        data = data.dropna()
    elif nan_handling == 'fill':
        if fill_method == 'mean':
            data = data.fillna(data.mean())
        elif fill_method == 'interpolate':
            data = data.interpolate()
        else:
            data = data.fillna(method=fill_method)

    # Scale the data
    scalers = {}
    if scale_data:
        for feature in features:
            scaler = MinMaxScaler(feature_range=(0, 1))
            data[feature] =
scaler.fit_transform(data[feature].values.reshape(-1, 1))
            scalers[feature] = scaler
        if save_scalers:
            os.makedirs(scalers_dir, exist_ok=True)
            for feature, scaler in scalers.items():
                scaler_path = os.path.join(scalers_dir,
f'{feature}_scaler.pkl')
                joblib.dump(scaler, scaler_path)

    # Split data
    if split_method == 'date':
        split_point = int(len(data) * split_ratio)
        train_data, test_data = data.iloc[:split_point],
data.iloc[split_point:]
```

```
    else:
        train_data, test_data = train_test_split(data, test_size=1-
split_ratio, shuffle=True)

    x_train, y_train = train_data[features[:-1]].values, train_data[features[-
1]].values
    x_test, y_test = test_data[features[:-1]].values, test_data[features[-
1]].values

    return x_train, x_test, y_train, y_test
```

Code Explanation:

1. Loading Data

- **Line: data = yf.download(company, start=start_date, end=end_date)[features]**
    - **Explanation**: This line uses the yfinance library to download historical stock data for the specified company (company), within the date range defined by start_date and end_date. The features argument filters the data to include only specific columns (like 'Open', 'High', 'Low', 'Close', 'Volume'). The square brackets ([features]) select these columns from the downloaded data.
    - **Research**: The yfinance library wraps Yahoo Finance's API, making it easier to pull financial data directly into Python. Understanding how to correctly filter and select data using Pandas indexing (square brackets) is essential.

2. Handling NaN Values

- **Line: data = data.fillna(method=fill_method)**
    - **Explanation**: This line fills missing values in the data (NaN values) using the method specified by the fill_method variable. Common methods include forward fill (ffill), backward fill (bfill), or filling with a specific value like the mean (mean).
    - **Research**: The fillna() method in Pandas provides various strategies for dealing with missing data. Understanding the implications of each method—like how forward fill works by propagating the last valid observation forward to the next—was key here.

3. Scaling Features

- **Line: scaler = MinMaxScaler(feature_range=(0, 1))**

- **Explanation**: This line initializes a MinMaxScaler object from Scikit-learn, which scales each feature in the dataset to a specified range, here between 0 and 1. This scaling is crucial for models like LSTM, which are sensitive to the scale of the input data.
- **Research**: Scaling is a fundamental preprocessing step, especially for machine learning models. The MinMaxScaler transforms features by scaling them to a given range, which helps prevent the model from becoming biased toward features with larger values.

4. Data Splitting

- **Line: train_data, test_data = train_test_split(data, test_size=1-split_ratio, shuffle=True)**
  - **Explanation**: This line splits the data into training and testing sets using Scikit-learn's train_test_split function. The test_size parameter defines the proportion of the dataset to include in the test split, and shuffle=True ensures that the data is shuffled before splitting to prevent any time-based bias.
  - **Research**: Understanding the importance of splitting data into training and testing sets is essential in machine learning. The train_test_split function is widely used, but it's important to know the effect of shuffling and how the split ratio impacts model evaluation.

5. Saving Data and Scalers

- **Line: joblib.dump(scaler, scaler_path)**
  - **Explanation**: This line saves the scaler object to a file using the joblib library. Joblib is efficient for saving large Python objects, like machine learning models or transformers (e.g., scalers), which can then be reloaded for future use without retraining.
  - **Research**: The choice of joblib over Python's standard pickle module comes from its efficiency with large data structures. Knowing when and why to use joblib for serialization is crucial for this kind of work.

6. LSTM Model Creation

- **Line: model.add(LSTM(units=50, return_sequences=True, input_shape=input_shape))**
  - **Explanation**: This line adds an LSTM layer to the model with 50 units (neurons). The return_sequences=True argument ensures that the layer returns the full sequence of outputs to the next layer.

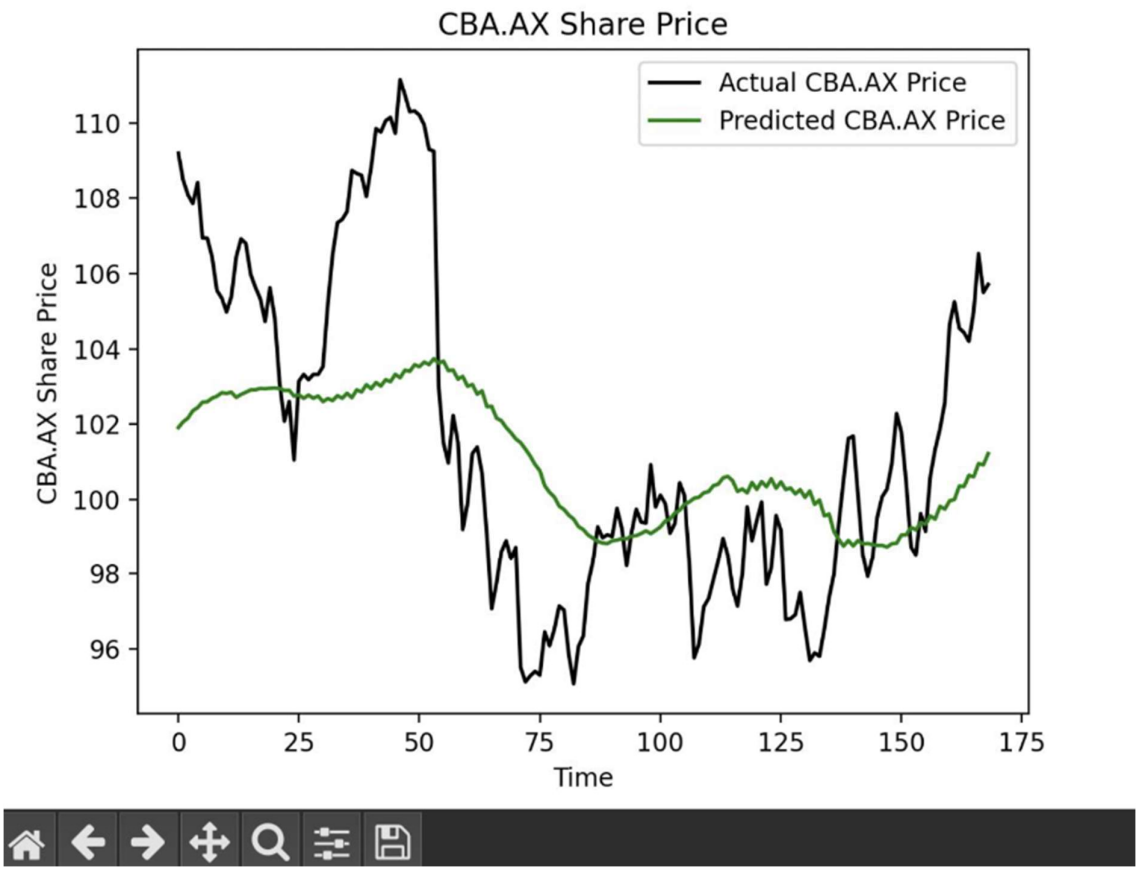
input_shape=input_shape defines the shape of the input data for this layer.

- **Research**: LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) that is particularly good at learning from sequences, like time-series data. The return_sequences parameter is essential when stacking LSTM layers, as it determines whether the full sequence or just the last output is passed to the next layer.

7. Plotting Results

- **Line: plt.plot(predictions, color="green", label="Predicted Price")**
  - **Explanation**: This line uses Matplotlib to plot the predicted stock prices in green. The label argument adds a legend entry, which helps differentiate between actual and predicted prices on the graph.
  - **Research**: Plotting results is crucial for visually assessing model performance. Understanding how to use Matplotlib to create clear, informative plots is an important skill in data science.

Screenshots



Stock price [actual vs prediction] display

This figure displays the model capturing the general trend of stock prices. There are discrepancies during periods of high volatility, and they highlight areas where further tuning and enhancements could improve prediction accuracy.

Conclusion

This task allowed me to dive deeper into the complexities of data processing and model development for stock price prediction. By expanding the functionality of the code, I assumed I've enhanced its ability to handle multiple features, manage missing data more effectively, and scale features appropriately, all crucial steps for building a more robust predictive model. I gained a better understanding of tools like yfinance, MinMaxScaler, and LSTM networks, which will come in handy for upcoming weekly tasks.