Task C.2 – Data Processing 1

COS30018 – Intelligent Systems

Rahul Raju

Student ID: 105143065

Overview of Enhancements to v0.1 Codebase

Building upon the foundation of the original v0.1 stock prediction script, this enhances implementation introduces significant improvements in data handling, feature engineering, and code modularity while maintaining the core LSTM-based prediction methodology. The enhancements address several limitations of the original code, particularly around data processing flexibility and reproducibility.

Requirements

Python Dependencies

- numpy
- matplotlib
- pandas
- pandas-datareader
- tensorflow
- scikit-learn
- yfinance
- pickle
- os

Key Enhancements Implemented

Advanced Data Processing Function

The most substantial enhancement is the *load_process_data()* function, which provides a comprehensive solution for handling financial data with multiple features.

```
def load_process_data(company, start_date, end_date, features=['Open', 'High', 'Low', 'Close', 'Volume'],
                      split_method='date', split_ratio=0.8, split_date=None,
                      scale_features=True, save_local=True, data_dir='./stock_data/'):
```

This function introduces several crucial improvements:

```
# Check if data already exists locally
if save_local and os.path.exists(filepath):
    print("Loading saved data from local directory...")
    with open(filepath, 'rb') as f:
        return pickle.load(f)
```

This mechanism addresses the reproducibility issue from v0.1 by saving processed data locally using Python's *pickle* module. This eliminates redundant API calls to Yahoo Finance and ensures consistent results across multiple runs which is a significant improvement over the original approach that fetched fresh data each execution.

Comprehensive Missing Value Handling:

```
# Processing missing values (NaN)
# Replace NaN with previous value - forward fill
data = data.ffill()
# Update remaining NaN values at beginning - back fill
data = data.bfill()
data = data[features]  # Select features
```

The function implements both forward-fill and backward-fill strategies to handle NaN values, which is more robust than the original code that didn't explicitly address missing data issues.

Flexible Data Splitting Methods:

```
# Split data into train and test data sets
if split_method == 'date':
    # use a specific date to split
    if split_date is None:
        # if no split date provided, calculate based on ratio
        split_index = int(len(data) * split_ratio)
        split_date = data.index[split_index]
    else:
        split_date = pd.to_datetime(split_date)

    train_data = data[data.index < split_date]
    test_data = data[data.index >= split_date]

elif split_method == 'ratio':
    # split ratio (80% train, 20% test)
    split_index = int(len(data) * split_ratio)
    train_data = data.iloc[:split_index]
    test_data = data.iloc[split_index:]
else:
    raise ValueError("split_method must be 'date' or 'ratio'")
```

This function for data splitting methods supports both date-based splitting (maintains temporal order) and ratio-based splitting (percentage division).

Individual Feature Scaling:

```
# Scale the features if requested
if scale_features:
    print("Scaling features...")
    for feature in features:
        # create a scaler for this feature
        scaler = MinMaxScaler(feature_range=(0, 1))

        # fit scaler on training data only
        train_values = train_data[feature].values.reshape(-1, 1)
        scaler.fit(train_values)

        # transform both training and test data
        train_data[feature] = scaler.transform(train_values).flatten()
        test_values = test_data[feature].values.reshape(-1, 1)
        test_data[feature] = scaler.transform(test_values).flatten()

        # store scaler for future access
        feature_scalers[feature] = scaler
```

Each feature is scaled independently, and all scalers are stored for potential inverse transformations which replaces the original single-feature approach which proves to be a significant improvement.

Multi-Feature Support with Backward Compatibility

While the enhanced code processes multiple features (Open, High, Low, Close, Volume), it maintains backward compatibility with the original prediction approach by focusing only on the Close price for the LSTM model.

```
# Load and process the data parameters
processed_data = load_process_data(
    company=COMPANY,
    start_date=TRAIN_START,
    end_date=TRAIN_END,
    features=['Open', 'High', 'Low', 'Close', 'Volume'],
    split_method='date',
    split_date='2023-01-01',
    scale_features=True,
    save_local=True
)
```

This design allows for future expansion to multi-feature models while keeping the current implementation consistent with the original methodology.

Complex Data Reshaping Operations

One of the more technically challenging aspects involves the data reshaping for LSTM compatibility.

```
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
# We now reshape x_train into a 3D array(p, q, 1); Note that x_train
# is an array of p inputs with each input being a 2D array
```

This line transforms a 2D array (samples x time steps) into a 3D array (samples x time steps x features) required by Keras LSTM layers. The -1 parameter is reshape operations allows numpy to automatically calculate the appropriate dimension size, which is particularly useful when dealing with variable length time series data.

Sequence Preparation Logic

The sequence generation for training and testing involves sophisticated array manipulation.

```
# Prepare the data
for x in range(PREDICTION_DAYS, len(scaled_data)):
    x_train.append(scaled_data[x-PREDICTION_DAYS:x])
    y_train.append(scaled_data[x])
```

This creates sliding windows of historical data (x_train) paired with the subsequent value (y_train). For a 60-day prediction window, each training sample contains 60 consecutive days of data, and the target is the 61st day's value.

Test Data Integration Challenge

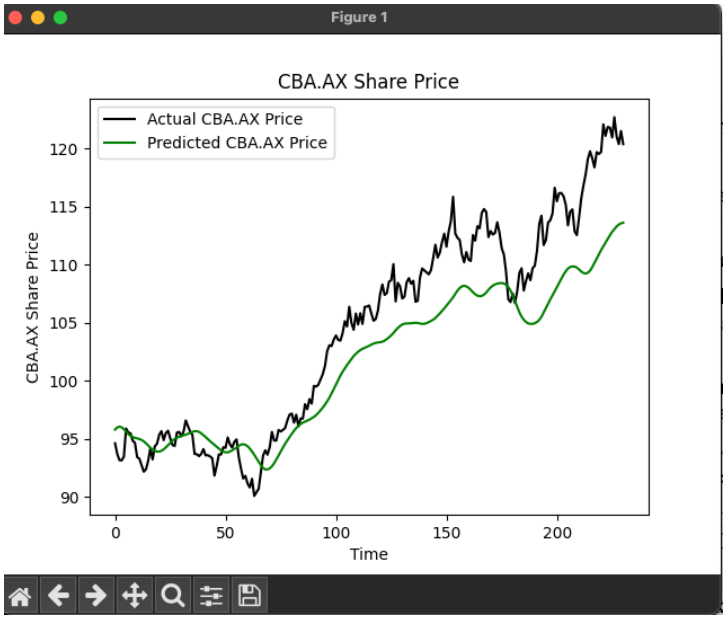
A particularly complex section involves integrating new test data with the existing training data.

```
# Get the full dataset (training + test) for creating sequences
full_data = yf.download(COMPANY, TRAIN_START, TEST_END)
full_prices = full_data[PRICE_VALUE].values

# Scale the full dataset using the same scaler
model_inputs = full_prices[len(full_prices) - len(test_data_new) - PREDICTION_DAYS:]
# We need to do the above because to predict the closing price of the fisrt
# PREDICTION_DAYS of the test period [TEST_START, TEST_END], we'll need the
# data from the training period

model_inputs = model_inputs.reshape(-1, 1)
# TO DO: Explain the above line

model_inputs = close_scaler.transform(model_inputs)
```

This approach ensures that test sequences are prepared using the same scaling parameters as the training data, maintaining consistency in the data processing pipeline.

```
/Users/rahul/venvs/price_prediction/lib/python3.13/site-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do
not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(sh
ape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Epoch 1/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.1164
Epoch 2/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0153
Epoch 3/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0091
Epoch 4/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0079
Epoch 5/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0081
Epoch 6/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0077
Epoch 7/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0067
Epoch 8/25
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0063
Epoch 9/25
```

```
/Users/rahul/Swinburne/year2/sem2/IntelligentSystems/Assignment1/C.2/v0.1/stock_prediction.py:297: FutureWarning:
YF.download() has changed argument auto_adjust default to True
  test_data_new = yf.download(COMPANY, TEST_START, TEST_END)
[*********************100%***********************]  1 of 1 completed
/Users/rahul/Swinburne/year2/sem2/IntelligentSystems/Assignment1/C.2/v0.1/stock_prediction.py:306: FutureWarning:
YF.download() has changed argument auto_adjust default to True
  full_data = yf.download(COMPANY, TRAIN_START, TEST_END)
[*********************100%***********************]  1 of 1 completed
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
Prediction: [[108.82041]]
```



Technical Challenges and Solutions

Dimensionality Matching Issues

During development, I encountered an error regarding the array elements with a sequence. This occurred because the test sequence generation logic was creating arrays of inconsistent lengths. The solution for this was to implement consistent array reshaping.

Model Architecture Insights

The LSTM architecture remains consistent with the original v0.1 implementation.

```
model.add(LSTM(units=50, return_sequences=True, input_shape=(x_train.shape[1], 1)))
```

```
model.add(Dropout(0.2))
# The Dropout layer randomly sets input units to 0 with a frequency of
# rate (= 0.2 above) at each step during training time, which helps
# prevent overfitting (one of the major problems of ML).

model.add(LSTM(units=50, return_sequences=True))
# More on Stacked LSTM:
# https://machinelearningmastery.com/stacked-long-short-term-memory-networks/

model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units=1))
```

The use of return_sequences=True in the first two LSTM layers creates a stacked architecture where each layer passes its full sequence of outputs to the next layer, rather than just the final output. This allows the network to learn patterns at different time scales.

The dropout layers (with rate=0.2) helps prevent overfitting by randomly excluding 20% of neurons during each training iteration, forcing the network to learn more robust features.

Performance Considerations

The enhanced code maintains similar performance characteristics to the original while adding significant functionality. The local caching mechanism actually improves performance on subsequent runs by eliminating the need for data downloading and reprocessing.

The most computationally intensive operations remains:

1. The LSTM training process (25 epochs with batch size 32)
2. The sequence generation for large datasets
3. The prediction phase with multiple forward passes through the trained network

Future Enhancement Implementation

The current implementation lays the groundwork for several advanced features:

- **Multi-feature LSTM Models**: The data processing pipeline now supports multiple features, which could be incorporated into the LSTM input for potentially improved predictions.
- **Hyperparameter Optimization**: The modular design allows for easy experimentation with different LSTM architectures, dropout rates, and training parameters.
- **Advanced Evaluation Metrics**: Beyond visual comparison, the code could be extended to include quantitative metrics like Mean Absolute Error, Mean Squared Error, and trading strategy simulations.
- **Real-time Prediction**: The local caching and modular design could support real-time prediction systems with periodic model retraining.