TASK C.2 REPORT – DATA PROCESSING 1

Unit: COS30018 – Intelligent Systems
Project: Option C – FinTech101 (Stock Price Prediction)
Student: Anh Vu Le – 104653505
Date: 31 August 2025

1. Introduction

The initial codebase (stock_prediction.py, v0.1) has several limitations in its data processing pipeline:

- Manual specification of training and test date ranges.
- Only one feature (Close) is used, while other features (Open, High, Low, Volume, Adj Close) are ignored.
- Missing values (NaN) are not systematically handled.
- Scaling is applied incorrectly, fitting on the entire dataset and re-used for test data, which leads to **ISSUE #2** (test values falling outside [0,1]).

To address these issues, I implemented a new function load_and_process_data in data_processing.py that fulfills all requirements (a) through (e) of Task C.2.

2. Implementation

The function load_and_process_data provides:

1. **(a) Flexible date specification** – User inputs start_date and end_date for the entire dataset.
2. **(b) NaN handling** – Missing values are forward filled, then backward filled, and finally dropped if still present.
3. **(c) Flexible splitting** – Supports:
   - **ratio** (e.g., 80% train / 20% test),
   - **date** (split at a given cutoff date),
   - **random** (train/test split with shuffling and reproducibility).
4. **(d) Local caching** – Downloads are stored as .csv in a cache directory with metadata in .json. Subsequent runs load from cache.
5. **(e) Feature scaling** – Each feature has its own MinMaxScaler. Crucially, scalers are fit **only on training data** to prevent leakage, then applied to test data. The scalers are stored in a dictionary and saved to disk for reuse.

The return object is a dictionary with:

- train_data, test_data: processed DataFrames (scaled if enabled)
- raw_train_data, raw_test_data: unscaled versions
- scalers: per-feature fitted scalers

- metadata: dictionary with all processing parameters and dataset info

3. Explanation of Complex Code

Some lines require deeper understanding and are documented in detail:

- **Date validation**:

```
pd.to_datetime(start_date)
pd.to_datetime(end_date)
```

Ensures correct format and that start < end.

Cache filenames:

```
cache_key = f"{ticker}_{start_date}_{end_date}"
cache_csv_path = os.path.join(cache_dir, f"{cache_key}.csv")
```

The key encodes ticker and date range so different requests don't overwrite each other.

Feature normalization:
Mapping like {'adjclose': 'Adj Close'} allows both lowercase/uppercase variants to be accepted and ensures consistency with Yahoo Finance column names.

NaN handling:

```
feature_data_clean =
feature_data.fillna(method='ffill').fillna(method='bfill').dropna()
```

Forward fill, backward fill, then drop any rows still missing. This prevents errors in training.

R**atio split**:

```
split_index = int(len(feature_data_clean) * split_value)
raw_train_data = feature_data_clean.iloc[:split_index]
raw_test_data  = feature_data_clean.iloc[split_index:]
```

Maintains time order, critical for stock prediction.

Random split:
Uses train_test_split(..., random_state=42, shuffle=True). Randomness can break temporal order but is useful to test robustness.

Scaling:

```
scaler.fit(train_values_2d)
```

```
train_scaled_2d = scaler.transform(train_values_2d)

test_scaled_2d  = scaler.transform(test_values_2d)
```

Fit only on training set, transform both train and test. This solves the leakage problem noted in v0.1.

Reshaping for scalers:

```
train_values_2d = train_values_1d.reshape(-1, 1)
```

Converts a 1D array into a 2D column vector, required by scikit-learn scalers.

4. Evidence

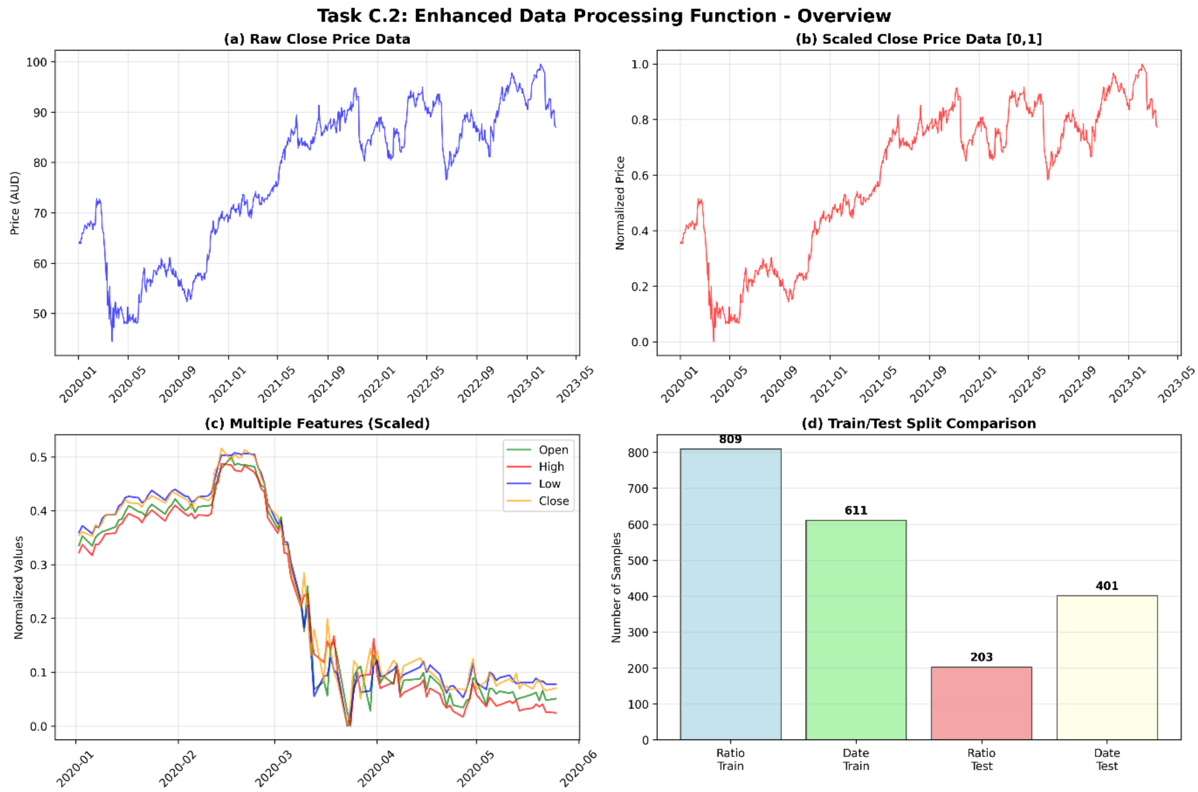

Figure 1 shows the raw and scaled price series (requirement a & e).

Date	Close	High	Low	Open	Volume
1/2/2020	63.94633	64.21851	63.55407	63.87428	1416232
1/3/2020	64.29053	64.995	64.24251	64.81889	1622784
1/6/2020	63.85826	63.94632	63.42598	63.82624	2129260
1/7/2020	65.00301	65.00301	64.18648	64.69881	2417468
1/8/2020	64.76286	65.04305	64.0664	65.01903	1719114
1/9/2020	65.23517	65.51535	64.979	65.24318	3014295
1/10/2020	66.04371	66.04371	65.32724	65.34725	2875353
1/13/2020	66.0277	66.10775	65.4273	65.7235	1434635
1/14/2020	66.58006	66.86025	66.25985	66.41195	2703855
1/15/2020	66.97232	67.06037	66.52402	66.52402	2039328
1/16/2020	67.61275	67.61275	67.14845	67.24451	3058484
1/17/2020	67.28454	68.01302	67.25252	67.8369	2401336
1/20/2020	67.18847	67.56472	67.10842	67.24451	1980399
1/21/2020	66.82822	67.14843	66.60407	67.14843	1923944
1/22/2020	67.60474	67.6928	66.83623	66.84424	3656698
1/23/2020	67.66077	68.005	67.45264	67.59674	2105704
1/24/2020	67.99699	68.36524	67.8609	67.96497	2839291
1/28/2020	67.26051	67.30053	66.87625	67.00434	2273413
1/29/2020	67.73283	67.93296	67.37259	67.59674	1877335
1/30/2020	68.39727	68.39727	67.78887	67.91695	3176161

Figure 2 shows the first few rows of the processed dataset (CBA.AX, 2020). This confirms requirement (a) – date range selection, and requirement (b) – NaN handling

5. Conclusion

The enhanced function load_and_process_data:

- Fixes all limitations of v0.1 data handling.
- Implements requirements (a)–(e) of Task C.2.
- Provides a reusable, well-documented data pipeline.
- Ensures proper NaN handling, flexible splitting, reproducibility via cache and metadata, and safe scaling without leakage.

This function now serves as a solid foundation for later tasks (C.3 onwards), where model improvements will build on this cleaned and structured dataset.