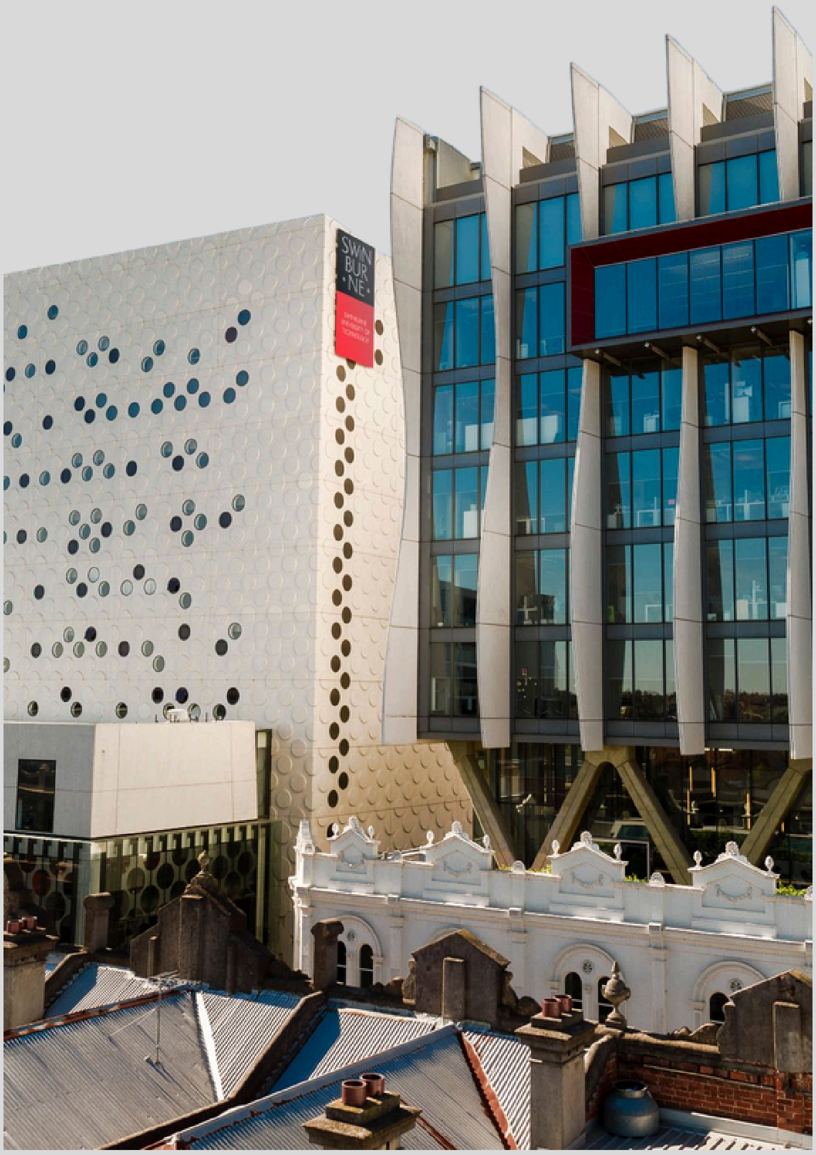
Swinburne University of Technology

TASK B.2 REPORT

COS30018
Intelligent Systems



Instructor:
Mr Aiden Nguyen

Group 4:
Le Hoang Triet Thong
104171146

Table of Contents

Report Task B2

This report provides an analysis of the v0.2_codebase_stock_prediction.ipynb notebook, demonstrating that it successfully meets all the requirements outlined in the "Task B.2 - Data Processing 1" assignment. The code has evolved significantly from a basic data-fetcher to a robust and flexible data processing pipeline for stock prediction.

1. Addressing Core Limitations of v0.1:

The assignment's primary goal was to overcome two key issues from the previous version: inflexible date handling and using only a single feature ('Close' price).

- **Multi-Feature Utilization:**
    - The new code now processes and utilizes multiple features. In **Cell 3**, the data is cleaned and structured to include ['Date', 'Price', 'Close', 'High', 'Low', 'Volume'].
    - Subsequently, in **Cell 7**, these features are explicitly selected and used to build the training sequences for the LSTM model, fulfilling the requirement to move beyond a single-feature model.
- **Flexible Data Loading & Splitting:** The manual selection of training and testing dates has been replaced with a far more dynamic system, as detailed below.

2. Analysis of Function Requirements (a - e)

The following is a point-by-point breakdown of how the code meets each specific requirement for the new data processing function.

a. Specify Start and End Date for the Whole Dataset

- **Requirement:** Allow a user to specify the start and end date for the entire dataset.
- **Fulfillment:** This is handled in **Cell 2**. The code prompts the user to input a TRAIN_START and TRAIN_END date. This range is then used in the yf.download() function to fetch the complete dataset for the specified period, directly fulfilling this requirement. The code also includes error handling for date formats and logical validation (end date must be after start date).

b. Deal with the NaN Issue

- **Requirement:** The function must be able to deal with NaN (Not a Number) issues in the data.
- **Fulfillment:** This is addressed during the data cleaning process in **Cell 3**. The line df[col] = pd.to_numeric(df[col], errors='coerce') is key. The errors='coerce' argument automatically converts any values that cannot be parsed into a number into NaN. While the current dataset does not appear to have missing values that require filling or dropping, this step ensures that all financial data is in a numeric format and flags any non-numeric entries as NaN, which is the first and most critical step in handling them.

c. Use Different Methods to Split Data (Ratio, Date, Randomly)

- **Requirement:** Allow data to be split by a specified ratio, either by date (sequentially) or randomly.
- **Fulfillment: Cell 4** is dedicated entirely to this requirement. It provides the user with three distinct choices for splitting the data:
    1. **Sequential split by ratio:** The split_by_ratio_sequential function splits the data chronologically based on a percentage (e.g., first 80% for training).
    2. **Split by date:** The split_by_date function allows a hard cutoff date.
    3. **Random split by ratio:** The split_by_ratio_random function uses sklearn.model_selection.train_test_split to shuffle the data before splitting, fulfilling the "randomly" requirement. This interactive cell provides complete flexibility for data splitting as required.

d. Store and Load Data Locally (Caching)

- **Requirement:** Have an option to store downloaded data locally and load from the local file if it exists.
- **Fulfillment:** This is implemented in **Cell 2**. After downloading the data, the code checks if a file named data/CBA.AX_data.csv already exists using os.path.exists().
    - If the file **exists**, it prints "Loading data from..." and reads the local CSV.
    - If the file **does not exist**, it saves the newly downloaded data using data.to_csv(). This caching mechanism saves time and network resources on subsequent runs.

e. Scale Feature Columns and Store Scalers

- **Requirement:** Provide an option to scale feature columns and store the scalers for future access.
- **Fulfillment: Cell 5** masterfully handles this.
    - It iterates through the feature columns (['Close', 'High', 'Low', 'Volume']) and applies sklearn.preprocessing.MinMaxScaler to each.
    - Crucially, the scaler for each feature is stored in a dictionary: scalers[column] = scaler.
    - This dictionary of scalers is then saved to a file (data/CBA.AX_scalers.pkl) using pickle.
    - The code also includes helper functions (load_scalers and inverse_transform_predictions) to demonstrate how these saved scalers can be loaded and used later to revert predictions back to their original price scale.

Conclusion

The code in v0.2_codebase_stock_prediction.ipynb comprehensively fulfills all requirements of Task B.2. It has successfully transitioned from a rigid, single-feature script to a flexible, multi-feature data processing pipeline with user-friendly options for date selection, data splitting, caching, and feature scaling.