Task 5 - Machine Learning 2

Kelvin Dang - 104776732 - Swinburne University of Technology

Overview

Having completed Task 4 and moved to v0.4, my current program is ready to be improved with the capability to solve more advanced stock price prediction problems, including multivariate prediction and multistep prediction.

The multistep prediction problem applies to time series forecasting requiring a prediction of multiple time steps into the future (e.g., the closing price of a company in multiple days into the future). This week, I was tasked to implement a function capable to solve the multistep prediction problem allowing the prediction to be made for a sequence of closing prices of k days into the future.

The multivariate prediction problem would instead take multiple time series as the input for the machine learning model to make the prediction. The other task I had been assigned earlier this week, along with the multistep prediction task, is to implement a function able to solve the simplest form of multivariate prediction problem, taking into account the other features for the same company (i.e., opening price, closing price, highest price, lowest price, and trading volume) as the input for predicting the closing price of the company for a specified day in the future.

The ultimate task for me this week is to combine the two required functions above to solve the multivariate and multistep prediction problem. In this report, I'm going to describe the implementation of the required functionalities of this week in detail.

Implementation

Some parts of this week's functional requirements had already been fulfilled in the previous tasks, so I will go over the respective parts of it in this report, along with others. First of all, the figure below shows the header of the `load_data()` function, constructed for data preprocessing, where the capabilities of solving multivariate and multistep prediction problems were included.

```
def load_data(ticker, start_date, end_date, test_size=0.25,
              store_locally=False, load_locally=False, local_data_path=None,
              scale_data=True, split_by_date=True,
              history_days=50, predict_days=1,
              feature_columns=["Close", "High", "Low", "Open", "Volume"]):
```

Figure 1: The header of the `load_data()` function.

Multistep Prediction

To accommodate the functionality of predicting a sequence of closing prices of the following k days, I introduced a parameter named `predict_days`, specifying how many days in the future we have to predict. By default, it is set to 1, which means the program would make prediction for the following day, but for the multistep problem, it can be modified.

```
predict_days (int): The number of days into the future to predict (default is 1).
```

Figure 2: The description of the `predict_days` parameter.

Multivariate Prediction

For the requirement of predicting the closing price of a company with multiple stock features (e.g., opening price, closing price, highest price, lowest price, and trading volume) as inputs, a parameter named `feature_columns` is included, a list specifying all feature columns we will include as inputs for our model to make predictions. By default, the list includes all possible input features.

```
feature_columns (list): The list of feature columns to use from the DataFrame (default includes common stock features).
```

Figure 3: The description of the `feature_columns` parameter.

Function Implementation

Data Preprocessing

As previously stated, the ultimate goal is to combining the capabilities of solving multistep and multivariate prediction problems together. Here, I will explain how I achieved this.

With reference to P1, I decided to follow the direction of flexibly using price features from not only one date, not the whole date range of the downloaded data, but from a pre-specified number of consecutive days to predict prices for a number of following consecutive days into the future, which is also flexible. This means two separated windows of days, one for historical data, one for

future, need to be set up. With this decision, a fair amount of feature engineering work needed to be done.

Along with the `predict_days` parameter, another parameter named `history_days` was also included, specifying the number of days in the past to look into before making predictions. By default, it is set to 50 days.

```
history_days (int): The number of past days to use for predicting the future (default is 50).
```

Figure 4: The description of the `history_days` parameter.

With that done, I proceeded to do some feature engineering based on the previously mentioned idea, starting from the point when the data is already downloaded and scaled. First, I duplicated the index column indicating the date for easier access and utilization later down the way, and shifted the target column, “Close”, upwards for a number of `predict_days` days. The reason for this is that each historical sequence will designate its ending date as the representative, so the target price of one date, which is the price of `predict_days` days later, would be the target price of the whole window having that date as the closing representative.

The problem of this shifting process is that it would make the last `predict_days` rows of the data have NaN target values. These would be retrieved for later, as they would be the testing features, before getting dropped, which is an earlier accomplished part of Task 2’s requirements.

```
# Duplicate the "Date" index into a column for easier access later.
df["Date"] = df.index
# Create the target column by shifting "Close" upwards.
df["Target"] = df["Close"].shift(-predict_days)
# The last 'predict_days' rows will have NaN target values, so get them before dropping.
last_sequence = np.array(df[feature_columns].tail(predict_days))
# Drop NaN values.
df.dropna(inplace=True)
```

Figure 5: Duplication of the “Date” column, shifting of the target column, and retrieval and deletion of the ending rows.

Then, I create the historical data sequence, which is the data collected over `history_days` days and used to predict the target closing price.

```
# Create the sequences, which is the data collected over "history_days", used to predict the "Target".
sequence_data = []
sequences = deque(maxlen=history_days)
for entry, target in zip(df[feature_columns + ["Date"]].values, df["Target"].values):
    sequences.append(entry)
    if len(sequences) == history_days:
        sequence_data.append([np.array(sequences), target])
```

Figure 6: The creation of historical data sequences.

The idea behind this is that from the data retrieved from day 1 to d, and value selections of h for `history_days` and p for `predict_days`, we would have a total of (d - h + 1) historical sequences, from [1, 2, 3, ..., h], [2, 3, 4, ..., h + 1], [3, 4, 5, ..., h + 2], up until [d - h + 1, d - h + 2, d - h + 3, ..., d]. Furthermore, the corresponding targets for (a part of) the above sequences would be [h + p, h + p + 1, h + p + 2, ..., d], omitting the NaNs we had dropped. This is an iterative process, and as a result, I initialized a double-ended queue variable named `sequences` with the maximum length of exactly `history_days`, and an empty list named `sequence_data`. Then, the program would run over all of the entries and targets available, and append each `sequence` to `sequence_data` once it has reach a total of `history_days` entries and targets. This implementation ensures the output similar to what has been explained right above.

Note that the closing `predict_days` rows of the data were previously retrieved and dropped. They would be needed to predict the future days, as the requirements stated. Therefore, they were concatenated with their previous historical sequence, forming the last data sequence consisting of `history_days` days of historical data and `predict_days` days of future data for the multistep prediction problem.

```
# Get the last sequence for prediction, by appending the last "history_days" sequence with the "predict_days" sequence.
# For instance, if history_days=50 and predict_days=1, then we want the last 51 days.
# This last_sequence will be used for predicting future stock prices that are not available in the dataset.
last_sequence = list([s[:len(feature_columns)] for s in sequences]) + list(last_sequence)
last_sequence = np.array(last_sequence).astype(np.float32)
# Add this to the results.
result["last_sequence"] = last_sequence
```

Figure 7: The retrieval and creation of the `last_sequence`.

Right after this, the feature sequences and targets would start to be constructed, by looping over the sequence data list and append each sequence and target value. At the end of this process, they would be converted to NumPy arrays, for future input to the model.

```
# Construct the feature sequences and targets.
X, y = [], []
for seq, target in sequence_data:
    X.append(seq)
    y.append(target)
# Convert to numpy arrays.
X = np.array(X)
y = np.array(y)
```

Figure 8: Collecting all sequences and target values to X and y.

To finish, I implemented the optional train-test splitting (as stated in the requirements of Task 2), retrieved the list of test dates, cleaned the duplicated dates, remove the earlier copied “Date” column, and convert the data values to `float32`.

```
if split_by_date:
    # Split the data into training and testing sets based on date, not randomly splitting.
    train_samples = int((1 - test_size) * len(X))
    result["X_train"] = X[:train_samples]
    result["y_train"] = y[:train_samples]
    result["X_test"] = X[train_samples:]
    result["y_test"] = y[train_samples:]
else:
    # Randomly split the data into training and testing sets.
    result["X_train"], result["X_test"], result["y_train"], result["y_test"] = train_test_split(X, y,
                                                                                                test_size=test_size)

# Get the list of test dates.
dates = result["X_test"][:, -1, -1]
# Retrieve test features from the original DataFrame.
result["test_df"] = result["df"].loc[dates]
# Remove duplicated dates in the testing DataFrame.
result["test_df"] = result["test_df"][~result["test_df"].index.duplicated(keep="first")]
# Remove dates from the training and testing DataFrames, and convert to float32.
result["X_train"] = result["X_train"][:, :, :len(feature_columns)].astype(np.float32)
result["X_test"] = result["X_test"][:, :, :len(feature_columns)].astype(np.float32)
```

Figure 9: Final preparations of the data.

Finally, I took a peek into the `X_train` and `y_train` data. They are shown in the figure below.

```
[[0.13491544 0.11728908 0.13700378 0.12678973 0.11430572]
 [0.14286122 0.124743   0.14229898 0.13294534 0.13144593]
 [0.14119442 0.12502551 0.14146301 0.13520448 0.08991365]
 ...
 [0.05028979 0.06317788 0.0551221  0.06866979 0.597112  ]
 [0.0687578  0.04943354 0.0212052  0.02575839 1.        ]
 [0.03102628 0.04377409 0.03579804 0.03136056 0.62713534]]

...

[[0.4327431  0.42512888 0.43877736 0.4311959  0.07125609]
 [0.43797314 0.42755097 0.43738273 0.42830262 0.08187255]
 [0.4322135  0.42317775 0.42549524 0.42776433 0.1945789 ]
 ...
 [0.5432355  0.5319697  0.5421789  0.5385167  0.11213928]
 [0.5317824  0.52800024 0.536368   0.5353543  0.10069438]
 [0.5271482  0.5152843  0.52119315 0.5175909  0.10791454]]

[[0.43797314 0.42755097 0.43738273 0.42830262 0.08187255]
 [0.4322135  0.42317775 0.42549524 0.42776433 0.1945789 ]
 [0.43896616 0.43104956 0.44176587 0.43745348 0.10163338]
 ...
 [0.5317824  0.52800024 0.536368   0.5353543  0.10069438]
 [0.5271482  0.5152843  0.52119315 0.5175909  0.10791454]
 [0.5349601  0.52477086 0.533944   0.531048   0.1179508 ]]

[[0.4322135  0.42317775 0.42549524 0.42776433 0.1945789 ]
 [0.43896616 0.43104956 0.44176587 0.43745348 0.10163338]
 [0.45022064 0.43710476 0.45046565 0.44357648 0.05891209]
 ...
 [0.5271482  0.5152843  0.52119315 0.5175909  0.10791454]
 [0.5349601  0.52477086 0.533944   0.531048   0.1179508 ]
 [0.5271482  0.5175046  0.52836555 0.52728003 0.08194391]]]
(1088, 50, 5)
```

Figure 10: The `X_train` data.

```
[0.07603135 0.05506304 0.03801569 ... 0.52357334 0.52171955 0.53813791]
(1088,)
```

Figure 11: The `y_train` data.

As expected, `X_train` was in the format of a three-dimensional NumPy array, each dimension representing the feature prices (multivariate), the dates in each sequence, and the list of all sequences, while `y_train` was a one-dimensional NumPy array flocked with target “Close” price values.

By these additions to the data loading and preprocessing function `load_data()`, the program is now well-equipped to tackle multivariate and multistep prediction problems.

Making Predictions

Still inside the `stock_prediction.py` module, a new function named `predict()` was created in order to allow the model to make predictions for a series of days into the future, using the last sequence. The figure below shows the full implementation of this function, right before going into detail.

```python
def predict(model, data):
    """
    Using the trained model and the last sequence of data to predict the stock prices of the next "PREDICT_DAYS" days.
    Parameters:
        model (tf.keras.Model): The trained RNN model.
        data (dict): The dictionary returned by the load_data function, containing the last sequence and scalers.
    """
    # Initialize a list to store predicted prices.
    result = []
    # Retrieve the last sequence from the data dictionary.
    last_sequence = data["last_sequence"]
    # Loop over the sequence to predict multiple days into the future. (1 day forward to PREDICT_DAYS days forward.)
    for i in range(1, PREDICT_DAYS + 1):
        # Retrieve the sequence to predict that day (i.e., the sequence targeted at day i in the future).
        predict_sequence = last_sequence[i:(i + HISTORY_DAYS)]
        # Expand dimensions to match the model's expected input shape.
        predict_sequence = np.expand_dims(predict_sequence, axis=0)
        # Get the prediction.
        prediction = model.predict(predict_sequence)
        # Get the price, by inverting the scaling if previously scaled.
        if SCALE_DATA:
            predicted_price = data["column_scaler"]["Close"].inverse_transform(prediction)
        else:
            predicted_price = prediction
        # Append the predicted price to the list.
        result.append(predicted_price[0][0])
    # Convert the list to a numpy array.
    result = np.array(result)
    return result
```

Figure 12: The `predict()` function, used for predicting future stock prices.

As mentioned earlier, a sequence named `last_sequence` is gathered by concatenating the last historical sequence of `HISTORY_DAYS` days and the following sequence of `PREDICT_DAYS` days, allowing us to solve the multistep problem by predicting prices for a series of `PREDICT_DAYS` days beyond the downloaded data. This sequence has a total of `(HISTORY_DAYS + PREDICT_DAYS)` days with price information, allowing us to retrieve a total of `(PREDICT_DAYS + 1)` sub-

sequences with the window of `HISTORY_DAYS` days. Because each sequence of HISTORY_DAYS days is targeted to the closing price of `PREDICT_DAYS` later, we can see that the first sub-sequence of those mentioned just earlier is not targeted into the future, but the last day of the downloaded data. Therefore, we can disregard the first one, and use `PREDICT_DAYS` others to predict the closing price of a series of `PREDICT_DAYS` days into the future.

After being retrieved, each sub-sequence would have to be expanded from 2D to 3D before being supplied to the model as input. Each prediction made is targeted to the sub-sequence's respective date into the future, and is going to be appended to the result list. Finally, the result list is converted to a NumPy array, representing the result for a multistep problem.

Testing

With all capabilities of solving multivariate and multistep prediction problems implemented, I would kick off the testing part by trying to solve a combination of these problems, which is predicting the closing prices of 3 days ahead using all common stock details (i.e., opening price, closing price, highest price, lowest price, and total amount). The figure below shows my parameter configurations.

```python
# Stock data information.
TICKER = "CBA.AX"
START_DATE = "2020-01-01"
END_DATE = "2025-08-01"

# User options.
# Local storage and loading options.
STORE_LOCALLY = True
LOAD_LOCALLY = False
STOCK_DATA_PATH = os.path.join("data", f"{TICKER}_{START_DATE}_{END_DATE}.csv")
LOCAL_DATA_PATH = None
# Feature scaling option.
SCALE_DATA = True
# Data preprocessing options.
SPLIT_BY_DATE = True
HISTORY_DAYS = 50
PREDICT_DAYS = 3
FEATURE_COLUMNS = ["Close", "High", "Low", "Open", "Volume"]
TEST_SIZE = 0.2
# Visualization options.
PRICE_COLUMNS = ["Close", "High", "Low", "Open"]

# Model options.
LOSS_FUNCTION = "huber"  # Options: "mse", "mae", "huber"
UNITS = 256
LAYER = LSTM  # Options: "LSTM", "GRU", "SimpleRNN"
N_LAYERS = 2
DROPOUT = 0.4
OPTIMIZER = "adam"  # Options: "adam", "rmsprop", "sgd"
BIDIRECTIONAL = False

# Training options.
BATCH_SIZE = 64
EPOCHS = 100
```

Figure 13: My parameter and option configurations.

I used the data for CBA.AX (the ticker of Commonwealth Bank of Australia), retrieved from 02/01/2020 to 31/07/2025, excluding the start and end dates specified above. As a result, the three future dates to have their closing prices predicted is going to be 01/08/2025, 04/08/2025, and 05/08/2025. The dates of 02/08/2025 and 03/08/2025 are respectively Saturday and Sunday, so there are no price details recorded for them two.

The figure below shows the training and predicted results for the aforementioned dates, using the pre-specified parameter configurations.

```
17/17 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step - loss: 1.9189e-04 - mae: 0.0151 - val_loss: 4.4465e-04 - val_mae: 0.0239
Epoch 98/100
17/17 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - loss: 1.8898e-04 - mae: 0.0147 - val_loss: 4.1623e-04 - val_mae: 0.0227
Epoch 99/100
17/17 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - loss: 2.0179e-04 - mae: 0.0154 - val_loss: 6.6668e-04 - val_mae: 0.0317
Epoch 100/100
17/17 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step - loss: 1.8904e-04 - mae: 0.0150 - val_loss: 6.0193e-04 - val_mae: 0.0297
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 123ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
[168.07872 168.78171 169.95796]
```

Figure 14: Training and predicted results of the model.

With that, the prediction results are as follows:

Date	Close
2025-08-01	168.07872
2025-08-04	168.78171
2025-08-05	169.95796

I had also checked the above predictions against the real-time prices, by retrieving the data of those dates on `yfinance`, and garner the below results.

```
import yfinance as yf
yf.download("CBA.AX", start="2025-08-01", end="2025-08-06", multi_level_index=False)

/tmp/ipython-input-1012930427.py:2: FutureWarning: YF.download() has changed argument auto_adjust default to True
  yf.download("CBA.AX", start="2025-08-01", end="2025-08-06", multi_level_index=False)
[*********************100%***********************]  1 of 1 completed
```

Date	Close	High	Low	Open	Volume
2025-08-01	172.399048	174.171690	171.965734	173.324762	1725512
2025-08-04	172.271011	172.271011	170.793812	171.355155	1178894
2025-08-05	174.644379	174.934893	172.399037	173.009608	1356835

Figure 15: The actual closing prices of the predicted dates.

The multivariate-and-multistep predictions turned out to be only around 2-2.5% lower than the actual prices.