Task 2 - Data Processing 1

Kelvin Dang - 104776732 - Swinburne University of Technology

Overview

In this week's task, I was challenged with writing a function to load and process a dataset with multiple features, with the following requirements:

- This function will allow me to specify the start date and the end date for the whole dataset as inputs.
- This function will allow me to deal with the NaN issue in the data.
- This function will also allow me to use different methods to split the data into train/test data; e.g., I can split it according to some specified ratio of train/test and I can specify to split it by date or randomly.
- This function will have the option to allow me to store the downloaded data on my local machine for future uses and to load the data locally to save time.
- This function will also allow me to have an option to scale my feature columns and store the scalers in a data structure to allow future access to these scalers.

My process of implementing these features for my function will be explained in detail starting from right below, in chronological order.

Setup

I created a new project folder, with the name of "FinTech101", as the project document stated. Inside that, I decided for an initial file structure of the project as follows:

- `stock_prediction.py`: Provides methods for data loading, data preprocessing, and model creation—for future purposes.
- `train.py` and `test.py`: For model training and testing purposes.

- `parameters.py`: To store all model and user parameters (e.g., ticker, start-end dates).
- `requirements.txt`: Includes all modules to be installed for a properly working environment.

Moreover, I made a decision to work with a virtual environment, so that it would be convenient for testing and debugging. Along the process, I was able to properly recreate the environment by using VSCode's Command Palette after every changes made to `requirements.txt` (to accommodate new packages that need to be installed).

Below is the initial structure of the FinTech101 project.

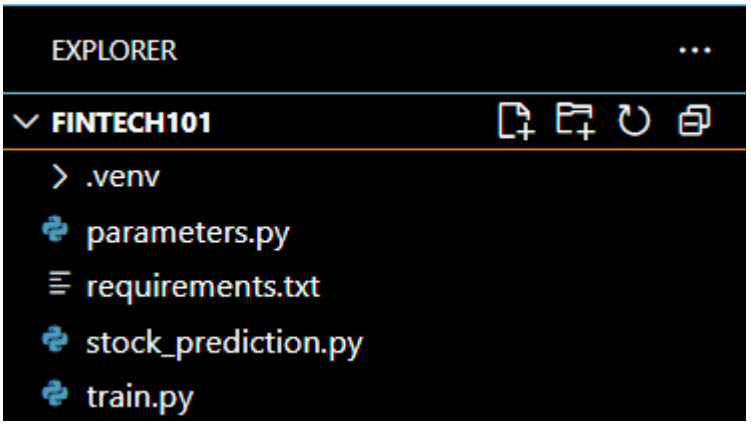


Figure 1: Initial file structure of FinTech101.

Data Loading and Processing Function

I defined a new function for data loading and preprocessing with the name of `load_data()` inside `stock_prediction.py`, and started implementing the required functionalities one-by-one.

Custom Start and End Dates

The first required functionality I would like to implement is custom start and end dates for the whole dataset as inputs, as I thought this is the simplest to fulfill.

I began by adding START_DATE and END_DATE as two parameters for this requirement, along with the TICKER parameter, which is the user option to specify the stock code to download data of. The TICKER parameter is set to

“CBA.AX” as the v0.1 example did, while START_DATE and END_DATE are set to “2020-01-01” and “2025-08-01”, respectively.

```
3   # Stock data information.
4   TICKER = "CBA.AX"
5   START_DATE = "2020-01-01"
6   END_DATE = "2025-08-01"
```

Figure 2: Parameters for custom start and end dates, along with stock ticker.

I also added `ticker`, `start_date` and `end_date` as arguments for the `load_data()` function, and added a single Yahoo Finance (`yfinance`) API call for stock data downloading.

```
# Download stock data from Yahoo Finance.
df = yf.download(ticker, start=start_date, end=end_date)
```

Figure 3: Data downloading from Yahoo Finance, with ticker, start and end dates used as inputs.

After downloading, I took a little peek at the downloaded DataFrame to see how it looks like.

```
  df = yf.download(ticker, start=start_date, end=end_date)
[**********************100%**************************]  1 of 1 completed
Price           Close       High        Low       Open   Volume
Ticker         CBA.AX     CBA.AX     CBA.AX     CBA.AX   CBA.AX
Date
2020-01-02  63.946331  64.218514  63.554073  63.874286  1416232
2020-01-03  64.290543  64.995013  64.242513  64.818895  1622784
2020-01-06  63.858269  63.946328  63.425988  63.826253  2129260
2020-01-07  65.003029  65.003029  64.186491  64.698830  2417468
2020-01-08  64.762863  65.043047  64.066400  65.019032  1719114
```

Figure 4: The DataFrame format as downloaded from Yahoo Finance API.

With this information, I added a new argument called `feature_columns`, which is a list of column names we would use as features, to the `load_data()` function. In default, it will be a list of all columns in the downloaded DataFrame format, including Close, High, Low, Open, and Volume.

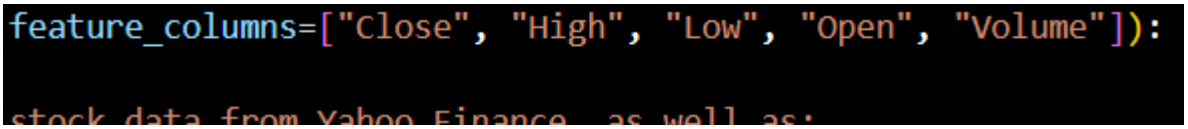


Figure 5: The `feature_columns` argument.

Optional Local Storage and Loading

In accordance with the requirements, the function must have an option for local saving and loading of the downloaded data. I started by including STORE_LOCALLY and LOAD_LOCALLY as two Boolean parameters denoting those options, as well as STOCK_DATA_PATH and LOCAL_DATA_PATH, which is the two data paths for local saving and loading, respectively. The path for saving, STOCK_DATA_PATH, will be generated using the ticker, start and end dates of the downloaded dataset.

```
# User options.
STORE_LOCALLY = True
LOAD_LOCALLY = False
STOCK_DATA_PATH = os.path.join("data", f"{TICKER}_{START_DATE}_{END_DATE}.csv")
LOCAL_DATA_PATH = ""
```

Figure 6: Parameters for local saving and loading, along with the data path for each functionality.

The function was also updated with new parameters to accommodate these new options, with new arguments `load_locally` (False by default), `store_locally` (True by default), and `local_data_path` (None by default).

```
store_locally=False, load_locally=False, local_data_path=None,
```

Figure 7: Arguments for optional local storage and loading.

Then, I implemented data loading controlled by these options as follows.

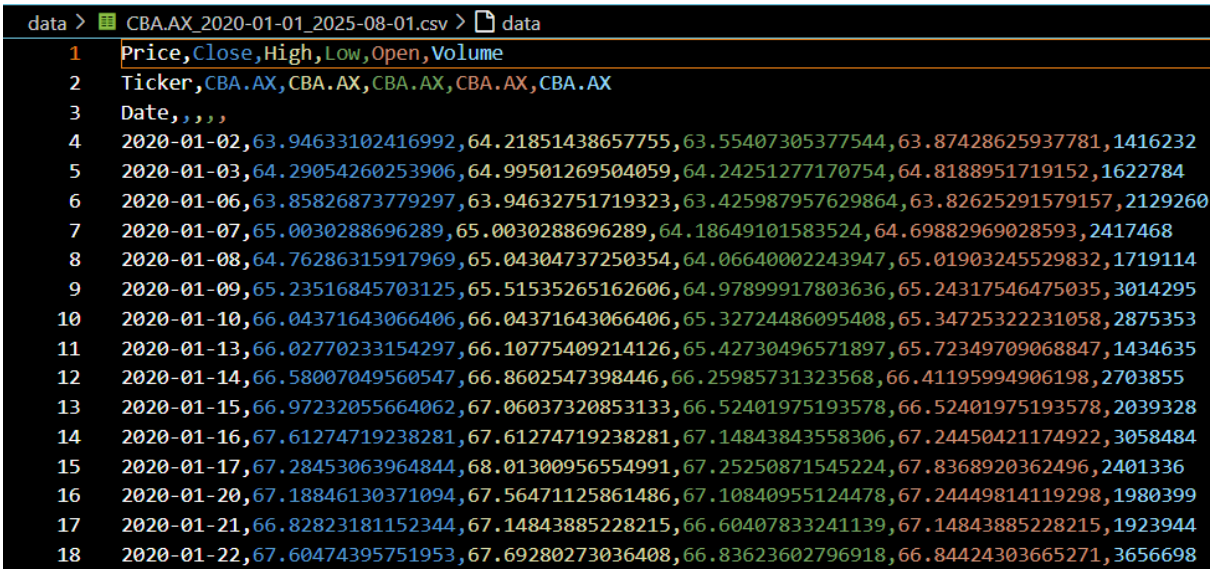
```python
# Check if the user wants to load data from a local CSV file.
if load_locally:
    assert os.path.isfile(local_data_path), f"Local data file not found: {local_data_path}"
    # Load the data from the specified CSV file.
    df = pd.read_csv(local_data_path, index_col="Date", parse_dates=True)
    # Ensure that the data is valid, by checking the columns.
    for col in feature_columns:
        assert col in df.columns, f"Column \"{col}\" not found in local data."
else:
    # Download stock data from Yahoo Finance.
    df = yf.download(ticker, start=start_date, end=end_date)
# Convert the data to a CSV file, and store it locally if requested.
if store_locally:
    if not os.path.isdir("data"):
        os.mkdir("data")
    df.to_csv(STOCK_DATA_PATH)
# Prepare the dictionary for things we want to return.
result = {}
# Include the DataFrame for returning.
result["df"] = df.copy()
```

Figure 8: Data loading, with respect to the local loading and saving options for the user.

First, the program checks if the user chose to load data from their local machine. After being sure that the input local path is valid, we will try to read the local file as CSV, and confirm whether its format is usable (i.e., same as being downloaded from Yahoo Finance) by going through its columns, and looking up in the `feature_columns` list. If the option is not used, we will download stock data from Yahoo Finance by a single API call.

Then, we check if the user opted to store the downloaded data locally. If that is the case, we will create a new folder called “data” if it hasn’t been created, and store the requested data as a CSV file there. The file name is generated by the STOCK_DATA_PATH parameter, as mentioned above.

This figure shown below is a testing example, showing the downloaded data can be successfully stored locally.



data > CBA.AX_2020-01-01_2025-08-01.csv > data

```
Price,Close,High,Low,Open,Volume
Ticker,CBA.AX,CBA.AX,CBA.AX,CBA.AX,CBA.AX
Date,,,,,
2020-01-02,63.94633102416992,64.21851438657755,63.55407305377544,63.87428625937781,1416232
2020-01-03,64.29054260253906,64.99501269504059,64.24251277170754,64.8188951719152,1622784
2020-01-06,63.85826873779297,63.94632751719323,63.425987957629864,63.82625291579157,2129260
2020-01-07,65.0030288696289,65.0030288696289,64.18649101583524,64.69882969028593,2417468
2020-01-08,64.76286315917969,65.04304737250354,64.06640002243947,65.01903245529832,1719114
2020-01-09,65.23516845703125,65.51535265162606,64.97899917803636,65.24317546475035,3014295
2020-01-10,66.04371643066406,66.04371643066406,65.32724486095408,65.34725322231058,2875353
2020-01-13,66.02770233154297,66.10775409214126,65.42730496571897,65.72349709068847,1434635
2020-01-14,66.58007049560547,66.8602547398446,66.25985731323568,66.41195994906198,2703855
2020-01-15,66.97232055664062,67.06037320853133,66.52401975193578,66.52401975193578,2039328
2020-01-16,67.61274719238281,67.61274719238281,67.14843843558306,67.24450421174922,3058484
2020-01-17,67.28453063964844,68.01300956554991,67.25250871545224,67.8368920362496,2401336
2020-01-20,67.18846130371094,67.56471125861486,67.10840955124478,67.24449814119298,1980399
2020-01-21,66.82823181152344,67.14843885228215,66.60407833241139,67.14843885228215,1923944
2020-01-22,67.60474395751953,67.69280273036408,66.83623602796918,66.84424303665271,3656698
```

Figure 9: Downloaded data successfully stored locally.

Finally, the downloaded or loaded DataFrame will be included for return at the end of this function's execution. As we have more things to return according to the following requirements, I decided to design this function to return a dictionary, in which the DataFrame will have the "df" key.

Optional Feature Scaling

Next up, I move forward to the requirement of having an option to scale the feature columns, and store the scalers in a data structure to allow future access. The first step is adding a new user option parameter for data scaling with the name of SCALE_DATA, as well as accommodate this option by including a new argument for the load_data() function called scale_data (True by default).

```
# Feature scaling option.
SCALE_DATA = True
```

Figure 10: The user option parameter for optional feature scaling.

```
scale_data=True,
```

Figure 11: The new argument of the `load_data()` function, used for optional feature scaling.

With that done, the optional feature scaling functionality is designed and implemented as follows.

```python
# Scale the data if requested.
if scale_data:
    column_scaler = {}
    # Scale the data from 0 to 1.
    for col in feature_columns:
        scaler = MinMaxScaler()
        # Reshape the data to be a 2D array for the scaler.
        # The parameter -1 allows the function to determine the size of that dimension automatically.
        df[col] = scaler.fit_transform(df[col].values.reshape(-1, 1))
        # Save the scaler for this column.
        column_scaler[col] = scaler
    # Save the column scalers for returning.
    result["column_scaler"] = column_scaler
```

Figure 12: The implementation of optional feature scaling.

I used a `MinMaxScaler` to scale the values of each feature column to a range of 0 to 1, as by default. Each scaler will be saved to a dictionary named `column_scaler` for future access, as per required. This scalers dictionary will also eventually be returned by the `load_data()` function.

The `fit_transform()` call above requires significant attention. `sklearn.preprocessing.fit_transform()` requires a 2D array input, so I had to reshape each column's values array, which is an 1D array, to a 2D one. Here, I passed 1 as the number of columns, and passed (-1) as the number of rows to `np.reshape()`. The reason I passed (-1) is to tell numpy to automatically figure out the number of rows, as long as the new shape is compatible with the original shape, which means the number of elements must be maintained. Therefore, each column's values array will be reshaped from (n) to (n, 1), ready to be passed to `fit_transform()`.

Dealing with NaNs

As we can see, the downloaded data does not have a target column yet. The reason is because the prices included in the data is recorded at their respective dates, i.e., at "present" time. What we want our model to predict is the prices at a "future" time, which would not be present without some tweaks to the data. It is indeed those tweaks that uncovered our problem with NaNs.

Below is how the problem appeared, and how I dealt with it.

```python
# 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 13: Dealing with the NaNs.

The reason why I duplicated the Date column will be explained later down the way. On the other hand, the “tweaks” I mentioned above is actually shifting the “Close” column, which is the column we want to predict for the future, upwards for a custom number of days, named `predict_days`. This custom number lets the user to choose a time in the future to predict the prices, by specifying how many days in advance. This argument was added to the function with a default value of 1 (which means predicting the value of tomorrow), and its respective parameter named PREDICT_DAYS was also added to the `parameters.py` file for user customizations.

This is the bottom values of the DataFrame after shifting. We can definitely see that the NaN value at the bottom needs to be dropped.

Price	Close	High	Low	Open	Volume	Date	Target
Ticker	CBA.AX	CBA.AX	CBA.AX	CBA.AX	CBA.AX		
Date							
2025-07-25	0.873337	0.872248	0.867727	0.881199	0.084381	2025-07-25	0.887213
2025-07-28	0.887213	0.881209	0.872047	0.874947	0.061967	2025-07-28	0.883043
2025-07-29	0.883043	0.879403	0.878424	0.884534	0.065814	2025-07-29	0.901499
2025-07-30	0.901499	0.899966	0.883361	0.885159	0.095783	2025-07-30	0.907788
2025-07-31	0.907788	0.904064	0.902972	0.906974	0.101781	2025-07-31	NaN

Figure 14: The NaN problem we have to resolve.

However, it is not simple as just drop the last entry. It can be used to predict the prices of the dates which is not present in the dataset, i.e., after the end date of it. Therefore, I had to retrieve them before dropping.

After dropping all NaN values, I can confirm that the NaN problem is now well resolved.

Options Regarding Train/Test Splitting

With the target in mind, I decided that the feature values will come from all five original columns, but not from only one date. They will come from a continuous

window of date. The reason for this decision is because by observing the prices for a prolonged period of time, the model would be able to capture important patterns from there, which I thought data from only one date would be far from enough. Some feature engineering would be done, according to that decision of mine.

Here is what I had done, with reference to P1, which has the same idea as above. The number of days used as features is added to the `load_data()` function as an argument called `history_days` (50 by default), and its respective parameter is also added with the name of HISTORY_DAYS.

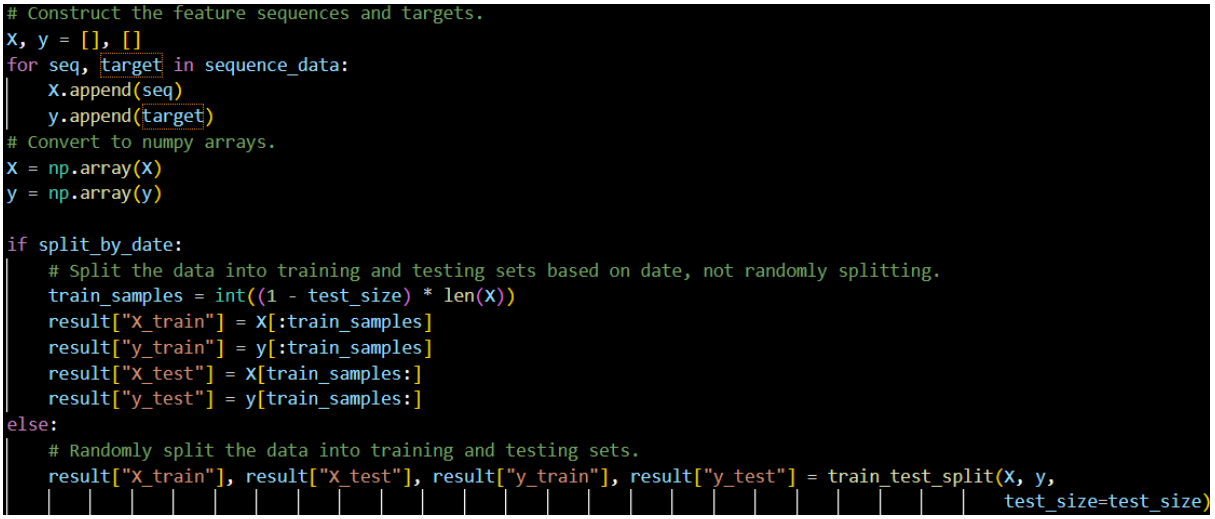
```
# Create the sequences, which is the data collected over "history_days", used to predict the "Target".
sequence_data = []
sequences = deque(maxlen=history_days)
for entry, target in zip(df[feature_columns + ["Date"]].values, df["Target"].values):
    sequences.append(entry)
    if len(sequences)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

# 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 15: Feature engineering to include more dates as features.

Now, I started to construct the features and targets for the model as inputs, as well as implement custom train/test splitting, as required.

A new argument was added for the `load_data()` function with the name of `split_by_date`, as well as its respective parameter SPLIT_BY_DATE. This option allows the user to freely choose whether to do train/test splitting based on date, or randomly.

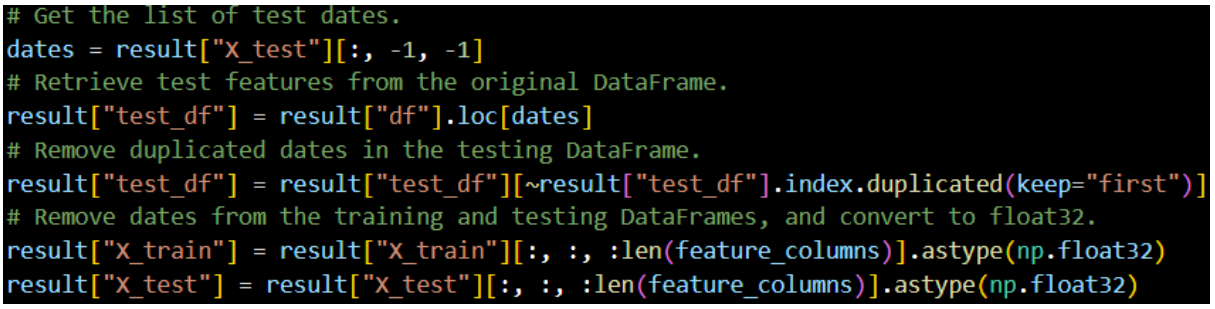
Another new argument was added for the function with the name of `test_size`, as well as its corresponding parameter TEST_SIZE. This custom value will decide the size of the testing data.



```
# 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)

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)
```

Figure 16: Constructing and custom splitting features and targets.

As mentioned a long way above, I duplicated the Date column. The purpose of that is to retrieve the testing DataFrame, which can be useful in the long run. Here is how I did that, as well as deleting the duplicated column when it completed its work.



```
# 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 17: Retrieving the testing DataFrame.

Testing

I included a single `load_data()` function call inside `train.py`, and ran that to test it, as well as showing the training DataFrame for confirmation.

```python
from stock_prediction import load_data
from parameters import *

# Load the data.
data = load_data(ticker=TICKER, start_date=START_DATE, end_date=END_DATE,
                 test_size=TEST_SIZE,
                 store_locally=STORE_LOCALLY, load_locally=LOAD_LOCALLY,
                 local_data_path=LOCAL_DATA_PATH,
                 scale_data=SCALE_DATA,
                 split_by_date=SPLIT_BY_DATE,
                 history_days=HISTORY_DAYS, predict_days=PREDICT_DAYS,
                 feature_columns=FEATURE_COLUMNS)

print(data["df"].head())
```

Figure 18: The `train.py` function.

This test turned out to be successful, concluding the function implementation, with all required functionalities successfully implemented.

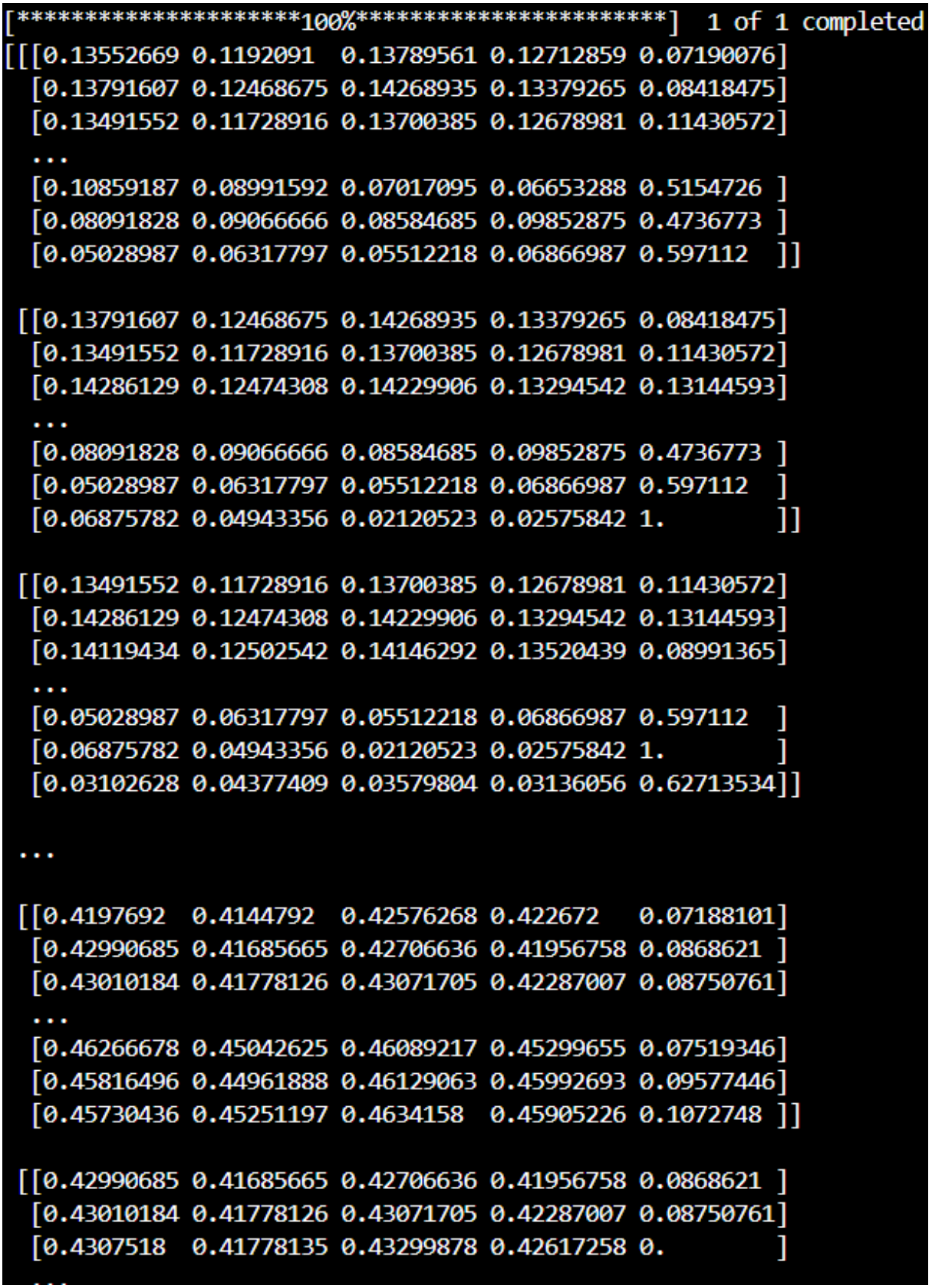


Figure 19: Successful validation of `load_data()`.