COS30049 - Intelligent System

DATA PROCESSING

TRAN DUC ANH DANG

Student ID: 103995439

Requirements

Virtual Environment

- Google Colab
- Jupyter Notebook

Decencies

- numpy
- matplotlib
- mplfinance
- pandas
- scikit-learn
- pandas-datareader
- yfinance
- pandas_ta

Installation

*Note: Anaconda is required unless Google Collab is being used

Anaconda/Virtual Environment

1. Download Anaconda: Go to the Anaconda website (https://www.anaconda.com/products/distribution) and download the appropriate version for your operating system.
2. Install Anaconda: Follow the installation instructions for the OS from the Anaconda website.
3. Open Anaconda Navigator: Launch Anaconda Navigator from your installed applications.
4. Create a New Environment (Required): Create a new environment to isolate Jupyter installation on each project. Click on "Environments" in Navigator and then "Create" to make a new environment.
5. Install Jupyter Notebook: In the selected environment, click on the environment name and select "Open Terminal". In the terminal, type: conda install jupyter.

Dependencies

In Google Colab or Jupyter Notebook, it can directly install the required dependencies using the !pip command in code cells. Here's an example of how to install the dependencies:

!pip install <package> or !pip install -r <text file>

Data Processing 1

ensure_directory_exist

```
# Double check directory
def ensure_directory_exists(dir_path):
    # If directory not exist => create
    if not os.path.isdir(dir_path):
        os.mkdir(dir_path)
```

The function `ensure_directory_exists` takes the following parameter:

- `dir_path`: This parameter is a string representing the path of the directory you want to ensure exists.

This function has the following features:

- Checking and Creating Directory: The primary purpose of this function is to ensure that a specified directory exists. It checks if the directory at the provided `dir_path` exists using the `os.path.isdir` function. If the directory does not exist, it creates the directory using the `os.mkdir` function.

load_data

```
# Load Raw Data
def load_data(start, end, ticker, source='yahoo'):
    ensure_directory_exists(DATA_DIR)

    # Check if CSV file exists
    # If exist => load
    # If not exist => download
    if os.path.exists(CSV_FILE):
        print('Loading Existing Data')
        data = pd.read_csv(CSV_FILE)
    else:
        print('Downloading Data')
        data = yf.download(ticker, start, end, progress=False)
        data.to_csv(CSV_FILE)

    return data
```

The function `load_data` takes several parameters:

- `start` and `end`: These are date values in the format 'YYYY-MM-DD' that define the time range for the financial data you want to load.
- `ticker`: This parameter is a string representing the stock ticker symbol (e.g., 'AAPL' for Apple Inc.) for which you want to fetch the financial data.
- `source`: This parameter is optional and specifies the data source. The default value is 'yahoo', which refers to Yahoo Finance.

This function has the following features:

- Creating a Directory: The function first ensures that a directory exists to hold the financial data. If the directory doesn't exist, it creates one using the path defined in the DATA_DIR variable.
- Checking for Existing Data: The function then checks if the financial data already exists by looking for a CSV file at the path specified in the CSV_FILE variable. If the file is found, the function assumes the data has already been downloaded or loaded and reads it from the CSV using the Pandas library.
- Downloading and Saving Data: If the CSV file containing the financial data doesn't exist, the function assumes that the data needs to be fetched. It uses the yf.download function from Yahoo Finance (based on the specified source) to download the financial data for the given stock ticker and time range. The progress=False argument suppresses progress messages during the download. The downloaded data is then saved to a new CSV file using the to_csv method.

data_validation

```python
# Data Validation
def data_validation(start, end, ticker):
    ensure_directory_exists(PREPARED_DATA_DIR)


    if os.path.exists(PREPARED_DATA_FILE):
        print('Loading Prepared Data')
        df = pd.read_csv(PREPARED_DATA_FILE)
    else:
        print('Processing Raw Data')

        # Read Raw Data File
        df = pd.read_csv(CSV_FILE)

        df['Date'] = pd.to_datetime(df['Date'])

        df.set_index('Date', inplace=True)

        # Adding indicators
        df['RSI']=ta.rsi(df.Close, length=15)
        df['EMAF']=ta.ema(df.Close, length=20)
        df['EMAM']=ta.ema(df.Close, length=100)
        df['EMAS']=ta.ema(df.Close, length=150)

        df['Target'] = df['Adj Close']-df.Open
        df['Target'] = df['Target'].shift(-1)

        df['TargetClass'] = [1 if df.Target[i]>0 else 0 for i in range(len(df))]

        df['TargetNextClose'] = df['Adj Close'].shift(-1)

        # Drop NaN issue in data
        df.dropna(inplace=True)

        # Drop Columns
        # df.drop(['Volume','Close', 'Date'], axis=1, inplace=True)

        # Export Prepared Data
        df.to_csv(PREPARED_DATA_FILE, index=False)

    return df
```

The function `data_validation` takes several parameters:

- `start` and `end`: These are date values in the format 'YYYY-MM-DD' that define the time range for the financial data you want to validate and process.
- `ticker`: This parameter is a string representing the stock ticker symbol (e.g., 'AAPL' for Apple Inc.) for which you intend to validate and preprocess the financial data.

This function has the following features:

- Creating a Directory: The function first ensures that a directory exists to hold the prepared data. If the directory doesn't exist, it creates one using the path defined in the `PREPARED_DATA_DIR` variable.
- Checking for Existing Data: The function then checks if prepared data already exists by looking for a CSV file at the path specified in the `PREPARED_DATA_FILE` variable. If the file is found, the function assumes the data has already been processed and loads it from the CSV using the Pandas library.
- Processing Raw Data: If the prepared data CSV file doesn't exist, the function assumes that the raw data needs to be processed. It reads the raw financial data from a CSV file located at the path specified in the `CSV_FILE` variable. Then, the function applies several preprocessing steps to this raw data:
    - Adding Indicators: The function adds indicators to the data, such as the Relative Strength Index (RSI) and various Exponential Moving Averages (EMAF, EMAM, EMAS), calculated using the `ta` library.
    - Calculating Targets: The function calculates the 'Target' column, which represents the difference between the adjusted closing price and the opening price. It also shifts this target one step back to represent the future movement.
    - Creating Target Class: The function generates a binary 'TargetClass' column based on whether the 'Target' is greater than zero, indicating a positive change.
    - TargetNextClose: The function creates a 'TargetNextClose' column by shifting the 'Adj Close' column one step back.
    - Handling Missing Data: The function removes any rows that contain NaN (missing) values.
    - Dropping Columns: Optionally, there are commented-out lines to drop certain columns like 'Volume', 'Close', and 'Date'. You can uncomment these lines if you want to remove these columns from the final processed data.
    - Exporting Prepared Data: Once all the preprocessing steps are complete, the function saves the processed data to a new CSV file using the `to_csv` method. This ensures that the next time the function is called, it can load the already processed data directly from the CSV file without repeating the preprocessing steps.

split_data

```python
# Split Data by Date or Randomly
def split_data(df, split_ratio, split_by_date=True):
    if split_by_date:
        # Split by date
        train_size = int(len(df) * split_ratio)
        train_data = df.iloc[:train_size]
        test_data = df.iloc[train_size:]
    else:
        # Split Randomly
        train_data, test_data = train_test_split(df, test_size=1-split_ratio, random_state=42)

    print(f"Train Data Shape: {train_data.shape}")
    print(f"Test Data Shape: {test_data.shape}")

    return train_data, test_data
```

The function `split_data` takes the following parameters:

- `df`: This parameter is a DataFrame containing the financial data that you want to split.
- `split_ratio`: This is the ratio of data to be used for training. The rest will be used for testing. For example, a `split_ratio` of 0.8 would mean 80% of the data is used for training and 20% for testing.
- `split_by_date`: This is an optional boolean parameter (defaulting to `True`) that indicates whether you want to split the data by date or randomly.

This function has the following features:

- Splitting Data by Date: If `split_by_date` is set to `True`, the function calculates the index at which the split should occur based on the ratio of data for training. It then splits the DataFrame into two parts: the first part for training (`train_data`) and the remaining part for testing (`test_data`).
- Splitting Data Randomly: If `split_by_date` is set to `False`, the function uses the `train_test_split` function from the scikit-learn library to randomly split the DataFrame into training and testing sets according to the specified `split_ratio`. The `random_state=42` ensures reproducibility of the random split.
- Printing Shapes: After splitting, the function prints the shapes of the training and testing data, indicating how many rows and columns each set contains.

scaler_features

```python
# Scaler
def scaler_features(input_data, scale=True):
    if scale:
        scaler = MinMaxScaler(feature_range=(0, 1))

        # Reshaping if input_data is a Series or 1D numpy array
        if len(input_data.shape) == 1:
            input_data = input_data.values.reshape(-1, 1)

        scaled_data = scaler.fit_transform(input_data)
        return scaled_data, scaler
    else:
        return input_data, None
```

The function `scaler_features` takes the following parameters:

- `input_data`: This parameter represents the data that you want to scale. It could be a pandas Series, a 1D numpy array, or a 2D numpy array.
- `scale`: This is an optional boolean parameter (defaulting to `True`) that indicates whether you want to scale the data.

This function has the following features:

- Scaling Data: If the `scale` parameter is set to `True`, the function creates an instance of the `MinMaxScaler` from scikit-learn. The `feature_range` parameter sets the range to which the data will be scaled (between 0 and 1).
- Reshaping Data: Before scaling, the function checks if the input data has a shape of 1 dimension (i.e., it's a Series or 1D numpy array). If so, it reshapes the data into a 2D array with one column using the `.reshape(-1, 1)` method. This is necessary because scikit-learn's scaler expects a 2D input.
- Scaling and Transforming Data: The function then uses the scaler to fit and transform the input data, resulting in scaled data. This scaled data is returned along with the scaler instance.
- Not Scaling Data: If the `scale` parameter is set to `False`, the function simply returns the original input data as-is, without any scaling. In this case, the scaler instance returned is `None`.

create_datasets

```python
def create_datasets(start, end, ticker):
    # Download or Load Raw Data
    data = load_data(start, end, ticker)

    # Data Validation
    df = data_validation(start, end, ticker)

    if os.path.exists(TRAIN_DATA_FILE) and os.path.exists(TEST_DATA_FILE):
        print('Loading Existing Train and Test Data')
        train_data = pd.read_csv(TRAIN_DATA_FILE)
        test_data = pd.read_csv(TEST_DATA_FILE)

        print(f"Train Data Shape: {train_data.shape}")
        print(f"Test Data Shape: {test_data.shape}")

        # Load feature and target scalers
        train_feature_scaler = load_object(SCALER_FEATURE_FILE)
        train_target_scaler = load_object(SCALER_TARGET_FILE)

        # Load x_train, y_train, x_test, y_test
        train_arrays = np.load(TRAIN_ARRAY_FILE)
        x_train = train_arrays['x_train']
        y_train = train_arrays['y_train']

        test_arrays = np.load(TEST_ARRAY_FILE)
        x_test = test_arrays['x_test']
        y_test = test_arrays['y_test']

    else:
        print('Processing Train and Test Data')
        # Split Data
        train_data, test_data = split_data(df, split_ratio)

        # Define features and target
        feature_columns = ['Open', 'High', 'Low', 'RSI', 'EMAF', 'EMAM', 'EMAS']
        target_column = 'TargetNextClose'

        # Preparing Train Datasets
        # Scaler for features
        scaled_data_train, train_feature_scaler = scaler_features(train_data[feature_columns])
        # Scaler for target
        scaled_target_train, train_target_scaler = scaler_features(train_data[target_column].values.reshape(-1, 1))

        x_train, y_train = [], []
        for i in range(step_size, len(scaled_data_train)):
            x_train.append(scaled_data_train[i-step_size:i])
            y_train.append(scaled_target_train[i])

        x_train, y_train = np.array(x_train), np.array(y_train)

        # Preparing Test Datasets
        # Use the feature scaler to scale the test data
        scaled_data_test = train_feature_scaler.transform(test_data[feature_columns])
        # Use the target scaler to scale the test target
        scaled_target_test = train_target_scaler.transform(test_data[target_column].values.reshape(-1, 1))

        x_test, y_test = [], []
        for i in range(step_size, len(scaled_data_test)):
            x_test.append(scaled_data_test[i-step_size:i])
            y_test.append(scaled_target_test[i])

        x_test, y_test = np.array(x_test), np.array(y_test)

        # Save train_data and test_data
        train_data.to_csv(TRAIN_DATA_FILE, index=False)
        test_data.to_csv(TEST_DATA_FILE, index=False)

        # Save feature and target scalers
        save_object(train_feature_scaler, SCALER_FEATURE_FILE)
        save_object(train_target_scaler, SCALER_TARGET_FILE)

        # Save x_train, y_train, x_test, y_test
        np.savez(TRAIN_ARRAY_FILE, x_train=x_train, y_train=y_train)
        np.savez(TEST_ARRAY_FILE, x_test=x_test, y_test=y_test)

    return data, df, train_data, test_data, train_feature_scaler, train_target_scaler, x_train, x_test, y_train, y_test
```

The function `create_datasets` takes the following parameters:

- `start` and `end`: These are date strings in the format 'YYYY-MM-DD'. They define the time range for downloading the financial data.
- `ticker`: A string that represents the stock ticker symbol for which the datasets are to be created. For example, 'AAPL' for Apple Inc.

This function has the following features:

- Downloading or Loading Raw Data: The function initially calls the `load_data` function to either download or load existing raw financial data based on the provided `start`, `end`, and `ticker` parameters.
- Data Validation: Once the raw data is loaded, the `data_validation` function is called to preprocess and validate the data. The cleaned and processed data is stored in a DataFrame (`df`).
- Splitting Data: The `split_data` function is invoked to partition the cleaned DataFrame (`df`) into training (`train_data`) and testing (`test_data`) datasets. The split is determined by a predefined `split_ratio`.
- Defining Features and Target: The function specifies which columns in the DataFrame will be treated as features (`feature_columns`) and which one will be used as the target (`target_column`).
- Preparing Train Datasets: The steps are carried out:
    - Feature scaling is performed using `train_feature_scaler`, which scales the feature columns of the training data.
    - Target scaling is done using `train_target_scaler`, which scales the target values in the training data.
    - Sequences of scaled features and corresponding target values (`x_train` and `y_train`) are generated. These sequences are designed for training time-series models like LSTMs.
- Preparing Test Datasets:
    - Applies the same feature scaler (`train_feature_scaler`) that was used on the training data to scale the feature columns.
    - Uses the same target scaler (`train_target_scaler`) that was used on the training data to scale the target values.
    - Creates sequences of scaled features and corresponding target values (`x_test` and `y_test`) for testing.
- Saving Prepared Train Data: The prepared training sequences (`x_train` and `y_train`) and testing sequences (`x_test` and `y_test`) are saved to `.npz` files for future use. This is done using the `np.savez` function.
- Returning Prepared Data and Information: The function returns various data and objects:
    - Raw data (`data`)
    - Processed DataFrame (`df`)
    - Training and testing datasets (`train_data` and `test_data`)
    - Feature and target scalers (`train_feature_scaler` and `train_target_scaler`)
    - Prepared training and testing sequences (`x_train`, `y_train`, `x_test`, `y_test`)

Data Processing 1 - Output

Set Up:

```
start='2015-01-01'
end='2023-08-25'
ticker='TSLA'

# Price Value
price_value = 'Close' # This can be change to 'Open', 'Close', 'Adj Close , 'High', 'Low'

# Split Dataset for Training/Testing
split_ratio=0.8

# Number of look back days to base the prediction
step_size = 30 # Can be changed

# Directory
DATA_DIR = os.path.join(SKELETON_DIR, "data")
PREPARED_DATA_DIR = os.path.join(SKELETON_DIR, "prepared-data")

# File Path
CSV_FILE = os.path.join(DATA_DIR, f"RawData-from-{start}to-{end}-{ticker}_stock_data.csv")
PREPARED_DATA_FILE = os.path.join(PREPARED_DATA_DIR, f"PreparedData-from-{start}to-{end}-{ticker}_stock_data.csv")
PREPARED_TRAIN = os.path.join(PREPARED_DATA_DIR, f"{ticker}_xytrain-from-{start}to-{end}-{ticker}_prepared_data.npz")
TRAIN_DATA_FILE = os.path.join(PREPARED_DATA_DIR, f"TrainData-from-{start}to-{end}-{ticker}_stock_data.csv")
TEST_DATA_FILE = os.path.join(PREPARED_DATA_DIR, f"TestData-from-{start}to-{end}-{ticker}_stock_data.csv")
SCALER_FEATURE_FILE = os.path.join(PREPARED_DATA_DIR, f"FeatureScaler-from-{start}to-{end}-{ticker}.pkl")
SCALER_TARGET_FILE = os.path.join(PREPARED_DATA_DIR, f"TargetScaler-from-{start}to-{end}-{ticker}.pkl")
TRAIN_ARRAY_FILE = os.path.join(PREPARED_DATA_DIR, f"{ticker}_xytrain-from-{start}to-{end}_train_arrays.npz")
TEST_ARRAY_FILE = os.path.join(PREPARED_DATA_DIR, f"{ticker}_xytrain-from-{start}to-{end}_test_arrays.npz")
```

Scenario 1 - First Time Running Output, split_by_date=True

Checking Data Type

```
data, df, train_data, test_data, train_feature_scaler, train_target_scaler, x_train, x_test, y_trai
```

```
Downloading Data
Processing Raw Data
Processing Train and Test Data
Train Data Shape: (1620, 13)
Test Data Shape: (406, 13)
```

```
print("Data shapes/types:")
print("data:", type(data))
print("df:", type(df))
print("train_data:", train_data.shape)
print("test_data:", test_data.shape)
print("train_feature_scaler:", type(train_feature_scaler))
print("train_target_scaler:", type(train_target_scaler))
print("x_train:", x_train.shape)
print("x_test:", x_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
Data shapes/types:
data: <class 'pandas.core.frame.DataFrame'>
df: <class 'pandas.core.frame.DataFrame'>
train_data: (1620, 13)
test_data: (406, 13)
train_feature_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
train_target_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
x_train: (1590, 30, 7)
x_test: (376, 30, 7)
y_train: (1590, 1)
y_test: (376, 1)
```

Raw Data

```
# Raw Data
print(len(data))
data.head(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-01-02	14.858000	14.883333	14.217333	14.620667	14.620667	71466000
1	2015-01-05	14.303333	14.433333	13.810667	14.006000	14.006000	80527500
2	2015-01-06	14.004000	14.280000	13.614000	14.085333	14.085333	93928500

```
[16] # Raw Data
print(len(data))
data.tail(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
2173	2023-08-22	240.250000	240.820007	229.550003	233.19(	[illegible]	[illegible]
2174	2023-08-23	229.339996	238.979996	229.289993	236.86(	[illegible]	[illegible]
2175	2023-08-24	238.660004	238.919998	228.179993	230.03(	[illegible]	[illegible]

Clear output

executed by Tran Duc Anh Dang
2:10 PM (31 minutes ago)
executed in 0.022s

Validated Data

```
[17] # Valiadted Data
print(len(df))
df.head(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS	
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760	-
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562	
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985	

```
[18] # Valiadted Data
print(len(df))
df.tail(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960	2
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426	2
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546	2

Train Data

+ Code + Text

[19]
```
# Train Data
print(len(train_data))
train_data.head(3)
```

1620

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS	
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760	-
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562	
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985	

[20]
```
# Train Data
print(len(train_data))
train_data.tail(3)
```

1620

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
1617	359.000000	362.666656	340.166656	354.899994	354.899994	90336600	51.079304	354.262754	316.084408	29
1618	360.123322	360.309998	336.666656	342.320007	342.320007	84164700	47.672453	353.125350	316.603925	29
1619	333.333344	353.033325	326.666656	352.706665	352.706665	91815000	50.587870	353.085475	317.318831	29

Test Data

```
# Test Data
print(len(test_data))
test_data.head(3)
```

406

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
1620	351.223328	358.616669	346.273346	354.799988	354.799988	66063300	51.175269	353.248762	318.061032	29
1621	359.616669	371.613342	357.529999	368.739990	368.739990	83739000	54.992715	354.724117	319.064576	29
1622	369.690002	371.866669	342.179993	343.853333	343.853333	97209900	47.838278	353.688804	319.555442	29

+ Code + Text

[22]
```
# Test Data
print(len(test_data))
test_data.tail(3)
```

406

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960	2
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426	2
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546	2

Checking Ratio

```
# Checking Ratio
print(f'Actual Ratio: {split_ratio}')
print(f'Train Ratio: {len(train_data)/len(df)}')
print(f'Test Ratio: {len(test_data)/len(df)}')
```

```
Actual Ratio: 0.8
Train Ratio: 0.7996051332675223
Test Ratio: 0.20039486673247778
```

Scenario 2 - Loaded Data, split_by_date=True

Checking Data Type

```
data, df, train_data, test_data, train_feature_scaler, train_target_scaler, x_train, x_test, y_trai
```

```
Loading Existing Data
Loading Prepared Data
Loading Existing Train and Test Data
Train Data Shape: (1620, 13)
Test Data Shape: (406, 13)
```

```
[36] print("Data shapes/types:")
print("data:", type(data))
print("df:", type(df))
print("train_data:", train_data.shape)
print("test_data:", test_data.shape)
print("train_feature_scaler:", type(train_feature_scaler))
print("train_target_scaler:", type(train_target_scaler))
print("x_train:", x_train.shape)
print("x_test:", x_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
Data shapes/types:
data: <class 'pandas.core.frame.DataFrame'>
df: <class 'pandas.core.frame.DataFrame'>
train_data: (1620, 13)
test_data: (406, 13)
train_feature_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
train_target_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
x_train: (1590, 30, 7)
x_test: (376, 30, 7)
y_train: (1590, 1)
y_test: (376, 1)
```

Raw Data

```python
# Raw Data
print(len(data))
data.head(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-01-02	14.858000	14.883333	14.217333	14.620667	14.620667	71466000
1	2015-01-05	14.303333	14.433333	13.810667	14.006000	14.006000	80527500
2	2015-01-06	14.004000	14.280000	13.614000	14.085333	14.085333	93928500

```python
[16] # Raw Data
print(len(data))
data.tail(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
2173	2023-08-22	240.250000	240.820007	229.550003	233.190		
2174	2023-08-23	229.339996	238.979996	229.289993	236.860		
2175	2023-08-24	238.660004	238.919998	228.179993	230.039		

Clear output

executed by Tran Duc Anh Dang
2:10 PM (31 minutes ago)
executed in 0.022s

Validated Data

```python
[17] # Valiadted Data
print(len(df))
df.head(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS	
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760	-
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562	
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985	

```python
[18] # Valiadted Data
print(len(df))
df.tail(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960	2
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426	2
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546	2

Train Data

```
[19] # Train Data
print(len(train_data))
train_data.head(3)
```

1620

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS	
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760	-
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562	
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985	

```
[20] # Train Data
print(len(train_data))
train_data.tail(3)
```

1620

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
1617	359.000000	362.666656	340.166656	354.899994	354.899994	90336600	51.079304	354.262754	316.084408	29
1618	360.123322	360.309998	336.666656	342.320007	342.320007	84164700	47.672453	353.125350	316.603925	29
1619	333.333344	353.033325	326.666656	352.706665	352.706665	91815000	50.587870	353.085475	317.318831	29

Test Data

```
# Test Data
print(len(test_data))
test_data.head(3)
```

406

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
1620	351.223328	358.616669	346.273346	354.799988	354.799988	66063300	51.175269	353.248762	318.061032	29
1621	359.616669	371.613342	357.529999	368.739990	368.739990	83739000	54.992715	354.724117	319.064576	29
1622	369.690002	371.866669	342.179993	343.853333	343.853333	97209900	47.838278	353.688804	319.555442	29

```
[22] # Test Data
print(len(test_data))
test_data.tail(3)
```

406

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960	2
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426	2
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546	2

Checking Ratio

```
# Checking Ratio
print(f'Actual Ratio: {split_ratio}')
print(f'Train Ratio: {len(train_data)/len(df)}')
print(f'Test Ratio: {len(test_data)/len(df)}')
```

```
Actual Ratio: 0.8
Train Ratio: 0.7996051332675223
Test Ratio: 0.20039486673247778
```

Scenario 3 - First Time Running Output, split_by_date=False

Checking Data Type

```
data, df, train_data, test_data, train_feature_scaler, train_target_scaler, x_train, x_test, y_trai
```

```
Downloading Data
Processing Raw Data
Processing Train and Test Data
Train Data Shape: (1620, 13)
Test Data Shape: (406, 13)
```

```
print("Data shapes/types:")
print("data:", type(data))
print("df:", type(df))
print("train_data:", train_data.shape)
print("test_data:", test_data.shape)
print("train_feature_scaler:", type(train_feature_scaler))
print("train_target_scaler:", type(train_target_scaler))
print("x_train:", x_train.shape)
print("x_test:", x_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
Data shapes/types:
data: <class 'pandas.core.frame.DataFrame'>
df: <class 'pandas.core.frame.DataFrame'>
train_data: (1620, 13)
test_data: (406, 13)
train_feature_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
train_target_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
x_train: (1590, 30, 7)
x_test: (376, 30, 7)
y_train: (1590, 1)
y_test: (376, 1)
```

Raw Data

```
# Raw Data
print(len(data))
data.head(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-01-02	14.858000	14.883333	14.217333	14.620667	14.620667	71466000
1	2015-01-05	14.303333	14.433333	13.810667	14.006000	14.006000	80527500
2	2015-01-06	14.004000	14.280000	13.614000	14.085333	14.085333	93928500

```
[16] # Raw Data
print(len(data))
data.tail(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
2173	2023-08-22	240.250000	240.820007	229.550003	233.190		
2174	2023-08-23	229.339996	238.979996	229.289993	236.860		
2175	2023-08-24	238.660004	238.919998	228.179993	230.039		

Clear output

executed by Tran Duc Anh Dang
2:10 PM (31 minutes ago)
executed in 0.022s

Validated Data

```
[17] # Valiadted Data
print(len(df))
df.head(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985

```
[18] # Valiadted Data
print(len(df))
df.tail(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546

Train Data

[48]

```
# Train Data
print(len(train_data))
train_data.head(3)
```

```
1620
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2022-04-06	357.823334	359.666656	342.566681	348.586670	348.586670	89348400	57.529061	335.575872	313.754059
2020-01-27	36.132668	37.629333	35.952000	37.201332	37.201332	204121500	74.095858	33.542984	24.789140
2018-07-17	20.587334	21.649332	20.566668	21.512667	21.512667	104943000	50.028587	21.559629	21.094551

[49]

```
# Train Data
print(len(train_data))
train_data.tail(3)
```

```
1620
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2019-01-07	21.448000	22.449333	21.183332	22.330667	22.330667	113268000	50.986928	21.976227	21.600074
2021-05-24	193.866669	204.826660	191.216660	202.146667	202.146667	103674300	43.505263	207.619472	219.036009
2020-01-28	37.899334	38.453999	37.205334	37.793331	37.793331	176827500	75.401047	33.947779	25.046649

Test Data

```python
# Test Data
print(len(test_data))
test_data.head(3)
```

```
406
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2020-09-11	127.313332	127.500000	120.166664	124.239998	124.239998	182152500	49.882747	129.044430	93.262145
2019-07-02	15.259333	15.276667	14.814667	14.970000	14.970000	138885000	54.576751	14.602292	16.470878
2021-09-21	244.929993	248.246674	243.479996	246.460007	246.460007	48992100	54.883551	244.924270	229.422722

```python
[51] # Test Data
print(len(test_data))
test_data.tail(3)
```

```
406
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2018-04-11	20.049334	20.598667	19.977333	20.062000	20.062000	112243500	48.754785	19.924297	21.401880
2021-10-01	259.466675	260.260010	254.529999	258.406677	258.406677	51094200	61.053445	252.158224	233.596625
2021-12-10	336.250000	340.326660	327.510010	339.010010	339.010010	59664300	46.204247	353.844977	303.041365

Checking Ratio

```python
# Checking Ratio
print(f'Actual Ratio: {split_ratio}')
print(f'Train Ratio: {len(train_data)/len(df)}')
print(f'Test Ratio: {len(test_data)/len(df)}')
```

```
Actual Ratio: 0.8
Train Ratio: 0.7996051332675223
Test Ratio: 0.20039486673247778
```

Scenario 4 - Loaded Data, split_by_date=False

Checking Data Type

```
data, df, train_data, test_data, train_feature_scaler, train_target_scaler, x_train, x_test, y_trai
```

```
Loading Existing Data
Loading Prepared Data
Loading Existing Train and Test Data
Train Data Shape: (1620, 13)
Test Data Shape: (406, 13)
```

```
[36] print("Data shapes/types:")
print("data:", type(data))
print("df:", type(df))
print("train_data:", train_data.shape)
print("test_data:", test_data.shape)
print("train_feature_scaler:", type(train_feature_scaler))
print("train_target_scaler:", type(train_target_scaler))
print("x_train:", x_train.shape)
print("x_test:", x_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
Data shapes/types:
data: <class 'pandas.core.frame.DataFrame'>
df: <class 'pandas.core.frame.DataFrame'>
train_data: (1620, 13)
test_data: (406, 13)
train_feature_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
train_target_scaler: <class 'sklearn.preprocessing._data.MinMaxScaler'>
x_train: (1590, 30, 7)
x_test: (376, 30, 7)
y_train: (1590, 1)
y_test: (376, 1)
```

Raw Data

```
# Raw Data
print(len(data))
data.head(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-01-02	14.858000	14.883333	14.217333	14.620667	14.620667	71466000
1	2015-01-05	14.303333	14.433333	13.810667	14.006000	14.006000	80527500
2	2015-01-06	14.004000	14.280000	13.614000	14.085333	14.085333	93928500

```
[16] # Raw Data
print(len(data))
data.tail(3)
```

2176

	Date	Open	High	Low	Close	Adj Close	Volume
2173	2023-08-22	240.250000	240.820007	229.550003	233.190		
2174	2023-08-23	229.339996	238.979996	229.289993	236.860		
2175	2023-08-24	238.660004	238.919998	228.179993	230.039		

Clear output

executed by Tran Duc Anh Dang
2:10 PM (31 minutes ago)
executed in 0.022s

Validated Data

```
[17] # Valiadted Data
print(len(df))
df.head(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	EMAS	
0	16.636000	17.000000	15.741333	16.408667	16.408667	219357000	40.237745	17.562829	16.245100	15.200760	-
1	16.238667	16.248667	15.892667	16.167334	16.167334	76101000	38.598500	17.429924	16.243561	15.213562	
2	15.876667	16.198000	15.736667	16.076000	16.076000	62788500	37.971249	17.300979	16.240243	15.224985	

```
[18] # Valiadted Data
print(len(df))
df.tail(3)
```

2026

	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM	
2023	221.550003	232.130005	220.580002	231.279999	231.279999	135702700	41.132038	243.068776	230.590960	2
2024	240.250000	240.820007	229.550003	233.190002	233.190002	130597900	42.417038	242.127941	230.642426	2
2025	229.339996	238.979996	229.289993	236.860001	236.860001	101077600	44.893445	241.626232	230.765546	2

Train Data

```
[48] # Train Data
print(len(train_data))
train_data.head(3)
```

1620

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2022-04-06	357.823334	359.666656	342.566681	348.586670	348.586670	89348400	57.529061	335.575872	313.754059
2020-01-27	36.132668	37.629333	35.952000	37.201332	37.201332	204121500	74.095858	33.542984	24.789140
2018-07-17	20.587334	21.649332	20.566668	21.512667	21.512667	104943000	50.028587	21.559629	21.094551

```
[49] # Train Data
print(len(train_data))
train_data.tail(3)
```

1620

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2019-01-07	21.448000	22.449333	21.183332	22.330667	22.330667	113268000	50.986928	21.976227	21.600074
2021-05-24	193.866669	204.826660	191.216660	202.146667	202.146667	103674300	43.505263	207.619472	219.036009
2020-01-28	37.899334	38.453999	37.205334	37.793331	37.793331	176827500	75.401047	33.947779	25.046649

Test Data

```
# Test Data
print(len(test_data))
test_data.head(3)
```

```
406
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2020-09-11	127.313332	127.500000	120.166664	124.239998	124.239998	182152500	49.882747	129.044430	93.262145
2019-07-02	15.259333	15.276667	14.814667	14.970000	14.970000	138885000	54.576751	14.602292	16.470878
2021-09-21	244.929993	248.246674	243.479996	246.460007	246.460007	48992100	54.883551	244.924270	229.422722

```
[51] # Test Data
print(len(test_data))
test_data.tail(3)
```

```
406
```

Date	Open	High	Low	Close	Adj Close	Volume	RSI	EMAF	EMAM
2018-04-11	20.049334	20.598667	19.977333	20.062000	20.062000	112243500	48.754785	19.924297	21.401880
2021-10-01	259.466675	260.260010	254.529999	258.406677	258.406677	51094200	61.053445	252.158224	233.596625
2021-12-10	336.250000	340.326660	327.510010	339.010010	339.010010	59664300	46.204247	353.844977	303.041365

Checking Ratio

```
# Checking Ratio
print(f'Actual Ratio: {split_ratio}')
print(f'Train Ratio: {len(train_data)/len(df)}')
print(f'Test Ratio: {len(test_data)/len(df)}')
```

```
Actual Ratio: 0.8
Train Ratio: 0.7996051332675223
Test Ratio: 0.20039486673247778
```

Testing Summary:

1. Training and Test Data: Prepared datasets were either loaded from existing files or freshly processed and then saved for future use.

1. Split by Date:
    a. Train Data Shape: (Shape when split by date)
    b. Test Data Shape: (Shape when split by date)
2. Random Split:
    a. Train Data Shape: (Shape when split randomly)
    b. Test Data Shape: (Shape when split randomly)

3. Feature and Target Scaling: Scalers were used to normalize the features and target columns. These scalers were saved for future use.
4. Prepared Sequences: For both training and test datasets, sequences of scaled features and target values were prepared and saved in .npz files.

Testing Conclusion:

The create_datasets function successfully prepared the data for machine learning applications, with options for both chronological and random data splitting. The prepared data and associated utilities like scalers are saved for easy retrieval, making the pipeline efficient and robust.