Project B

Task 2 – Data Processing 1

Function to allow specification of start and end date for the whole dataset, and to store and load data on the local machine.

```python
def load_or_save_data(company_ticker, start_date, end_date, base_dir="data"):
    ###
    # Loads the dataset for the input company and range.
    # If the dataset is not available locally, it downloads the data and saves it as a CSV file.
    # Parameters:
    # - company_ticker: ticker of the company (e.g., "AMZN")
    # - start_date: the start date for the dataset in 'YYYY-MM-DD' format
    # - end_date: the end date for the dataset in 'YYYY-MM-DD' format
    # - base_dir: the base directory where the data will be saved, default is set to "data"
    # Returns:
    # - data: pandas DF, the loaded dataset with the specified features
    ###

    # Generate the save path based on ticker and date range
    filename = f"{company_ticker}_{start_date}_to_{end_date}.csv"
    save_path = os.path.join(base_dir, filename)

    # Check if the file already exists
    if os.path.exists(save_path):
        # Load the dataset from the local file
        data = pd.read_csv(save_path, index_col=0, parse_dates=True)
        print(f"Data loaded from local file: {save_path}")
    else:
        # If the file doesn't exist, download the data
        data = yf.download(company_ticker, start=start_date, end=end_date)

        # Make sure the base directory exists
        os.makedirs(base_dir, exist_ok=True)

        # Save the dataset locally
        data.to_csv(save_path)
        print(f"Data downloaded and saved locally to: {save_path}")

    return data
```

Python

Function to deal with the NaN issue in the data.

```python
def handle_nan(data, method='drop'):
    ###
    # Handles NaN values in the dataset based on the specified method.
    # Parameters:
    # - data: pandas Dataframe
    # - method: str, how to handle NaN values. Options are 'drop', 'fill_mean', 'fill_median', 'fill_ffill'.
    # Drop removes all NaN data from the dataset.
    # Mean replaces the NaN data with the mean average of all the data
    # Mediam replaces the NaN data with the median average of all the data
    # FFill sets the NaN data to the most recent valid data
    # Returns:
    # - data: pandas Dataframe, the dataset with NaN values handled
    ###


    if method == 'drop':
        data = data.dropna()
    elif method == 'fill_mean':
        data = data.fillna(data.mean())
    elif method == 'fill_median':
        data = data.fillna(data.median())
    elif method == 'fill_ffill':
        data = data.fillna(method='ffill')
    else:
        raise ValueError("Choose from 'drop', 'fill_mean', 'fill_median', 'fill_ffill'.")

    return data
```

Python

Function to allow different methods of splitting the data. This function splits the data either by date or randomly.

```python
def split_data(data, test_size=0.25, split_by_date=True, date_column='Date'):

    ###
    # Splits the dataset into training and testing sets based on the specified methods.
    # Parameters:
    # - data: pandas datadframe, the dataset to split
    # - test_size: float, the amount of the dataset to include in the test split (default is 0.25)
    # - split_by_date: bool, split the data by date (True) or randomly (False). If false, the data is split using sklearns train_test_split method
    # - date_column: str, the name of the date column to use for date-based splitting (only needed if split_by_date=True)
    # Returns:
    # - train_data: training set as a pandas dataframe
    # - test_data: testing set as a pandas dataframe
    #
    #
    ###

    if split_by_date:
        # Sort data by date
        data = data.sort_values(by=date_column)

        # Determine the split index
        split_index = int(len(data) * (1 - test_size))

        # Split the data
        train_data = data.iloc[:split_index]
        test_data = data.iloc[split_index:]
    else:
        # Randomly split the data using sklearn's train_test_split
        train_data, test_data = train_test_split(data, test_size=test_size, shuffle=True, random_state=180)

    return train_data, test_data
```

Python

Function to give the option of scaling the features columns that then stores the scalers in a data structure.

```python
from sklearn.preprocessing import MinMaxScaler

def scale_data(data, feature_columns):
    ###
    # Scales the specified feature columns in the dataset using MinMaxScaler and stores the scalers.
    # Parameters:
    # - data: pandas dataframe, the dataset to scale
    # - feature_columns: list, a list of feature columns to scale (e.g., ["Adj Close"])
    # Returns:
    # - scaled_data: pandas dataframe, the dataset with scaled feature columns
    # - scalers: dict, a dictionary of scalers used to scale the feature columns
    ###
    scalers = {}
    scaled_data = data.copy()

    for feature in feature_columns:
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled_data[feature] = scaler.fit_transform(scaled_data[[feature]])
        scalers[feature] = scaler  # Store the scaler for future access

    return scaled_data, scalers
```