Task C.2 – Data Processing 1

Anh Vu Le
104653505

File: data_processing.py

Contents

(a) Start & End Dates

Code (lines ~87–95):

Requirement (a): Write a function to load and process a dataset with multiple features a. This function will allow you to specify the start date and the end date for the whole dataset as inputs

```
78      #-------------------------------------------------------------------------
79      # REQUIREMENT (a): Allow user to specify start date and end date for whole dataset
80      # Input validation for date parameters
81      #-------------------------------------------------------------------------
82
83      try:
84          # Validate date format by attempting to parse them
85          pd.to_datetime(start_date)
86          pd.to_datetime(end_date)
87          print(f"✓ Requirement (a): Date range specified - {start_date} to {end_date}")
88      except:
89          raise ValueError("start_date and end_date must be in 'YYYY-MM-DD' format")
90
91      if pd.to_datetime(start_date) >= pd.to_datetime(end_date):
92          raise ValueError("start_date must be earlier than end_date")
93
```

Explanation

- pd.to_datetime(start_date/end_date) parses the two strings into real dates and will throw if the format is wrong (guards against bad inputs).
- Immediately validating dates prevents downstream silent failures in splitting/scaling that assume a valid window.

(b) NaN Handling

Code (lines ~203–241; key ops at 218, 223, 228):

Requirement (b): This function will allow you to deal with the NaN issue in the data.

```
198     #-------------------------------------------------------------------------
199     # REQUIREMENT (b): Deal with NaN issue in the data
200     # Handle NaN (missing) values properly
201     #-------------------------------------------------------------------------
202
203     print(f"✓ Requirement (b): Handling NaN values in the data")
204
205     # Check for NaN values in our selected features
206     nan_counts = feature_data.isnull().sum()
207     total_nans = nan_counts.sum()
208
209     if total_nans > 0:
210         print(f"Found {total_nans} NaN values:")
211         for col, count in nan_counts.items():
212             if count > 0:
213                 print(f"  {col}: {count} NaN values")
214
215         # Strategy 1: Forward fill (use previous day's value)
216         # This is reasonable for stock prices as they tend to be continuous
217         # method='ffill' means forward fill - propagate last valid observation forward
218         feature_data_clean = feature_data.fillna(method='ffill')
219
220         # Strategy 2: Backward fill for any remaining NaN at the beginning
221         # This handles cases where the first few rows have NaN values
222         # method='bfill' means backward fill - use next valid observation to fill gap
223         feature_data_clean = feature_data_clean.fillna(method='bfill')
224
225         # Strategy 3: Drop any remaining rows with NaN (as last resort)
226         # If there are still NaN values after forward and backward fill, remove those rows
227         rows_before_drop = len(feature_data_clean)
228         feature_data_clean = feature_data_clean.dropna()
229         rows_after_drop = len(feature_data_clean)
230
```

Explanation

- Logs per-column NaN counts, then applies **forward fill** → **backward fill** → **final drop** (only if something is still missing).
- This ordering is a common, conservative pattern for financial time series: carry the last valid price forward, patch early gaps, and finally drop any stubborn rows.
- The prints (“dropped X rows”, “After cleaning: 0 NaN”) are good, concrete evidence for your report.

(c) Train/Test Split Methods

Code (lines ~247–316):

Requirement (c): This function will also allow you to use different methods to split the data into train/test data; e.g. you can split it according to some specified ratio of train/test and you can specify to split it by date or randomly.

```
242     #-----------------------------------------------------------------------
243     # REQUIREMENT (c): Use different methods to split data into train/test
244     # Split data into training and testing sets using flexible methods
245     #-----------------------------------------------------------------------
246
247     print(f"✓ Requirement (c): Splitting data using '{split_method}' method")
248
249     if split_method == 'ratio':
250         # Method 1: Split by ratio - first X% for training, remaining for testing
251         # This maintains temporal order which is important for time series data
252         # We don't shuffle the data because time order matters in stock prediction
253         split_index = int(len(feature_data_clean) * split_value)
254
255         raw_train_data = feature_data_clean.iloc[:split_index].copy()
256         raw_test_data = feature_data_clean.iloc[split_index:].copy()
257
258         print(f"Ratio split ({split_value:.1%}):")
259         print(f"  Training: {len(raw_train_data)} samples ({raw_train_data.index[0]} to {raw_train_data.index[-1]})")
260         print(f"  Testing: {len(raw_test_data)} samples ({raw_test_data.index[0]} to {raw_test_data.index[-1]})")
261
262     elif split_method == 'date':
263         # Method 2: Split by specific date - all data before split_value for training
264         # This is useful when you want to test on a specific time period
265         # For example, train on 2020-2023 data, test on 2023-2024 data
266         split_date = pd.to_datetime(split_value)
267
268         raw_train_data = feature_data_clean[feature_data_clean.index < split_date].copy()
269         raw_test_data = feature_data_clean[feature_data_clean.index >= split_date].copy()
270
```

Explanation

- **Ratio**: splits by proportion using .iloc[:k] (keeps temporal order intact).
- **Date**: converts the cutoff using pd.to_datetime(split_value) and filters by index < / ≥ that date (clear temporal separation).
- **Random**: samples indices with train_test_split(..., random_state=42) (reproducible), then **sorts** inside each subset so sequences remain chronological for sequence models.
- Immediately after this block your code prints sizes and date ranges of train/test, which is perfect to paste into the appendix as verification.

(d) Local Storage & Caching

Code (lines ~99–154):

Requirement (d): This function will have the option to allow you to store the downloaded data on your local machine for future uses and to load the data locally to save time

```
94      #---------------------------------------------------------------------------
95      # REQUIREMENT (d): Store downloaded data locally and load from cache
96      # Setup caching system to avoid repeated downloads
97      #---------------------------------------------------------------------------
98
99      print(f"✓ Requirement (d): Setting up local data caching")
100
101     # Create cache directory if it doesn't exist
102     # This allows us to store downloaded data locally for future use
103     if not os.path.exists(cache_dir):
104         os.makedirs(cache_dir)
105         print(f"Created cache directory: {cache_dir}")
106
107     # Generate unique cache key based on ticker and date range
108     # This ensures we can cache different datasets separately
109     cache_key = f"{ticker}_{start_date}_{end_date}"
110     cache_csv_path = os.path.join(cache_dir, f"{cache_key}.csv")
111     cache_meta_path = os.path.join(cache_dir, f"{cache_key}_meta.json")
112
113     # Check if cached data exists and load it
114     if os.path.exists(cache_csv_path) and os.path.exists(cache_meta_path):
115         print(f"✓ Loading from cache: {cache_csv_path}")
116         # Load the CSV file with proper date parsing
117         # index_col=0 means first column (Date) becomes the index
118         # parse_dates=True converts the index to datetime objects
119         data = pd.read_csv(cache_csv_path, index_col=0, parse_dates=True)
120
121         # Load metadata to verify cache validity
122         with open(cache_meta_path, 'r') as f:
123             cache_metadata = json.load(f)
124             print(f"Cache created: {cache_metadata['cache_date']}")
```

```
125     else:
126         print(f"✓ Downloading fresh data for {ticker} from {start_date} to {end_date}")
127
128         # Download data using yfinance
129         # yfinance is more reliable than pandas_datareader for Yahoo Finance data
130         data = yf.download(ticker, start=start_date, end=end_date)
131
132         # Handle potential multi-level column structure from yfinance
133         # Sometimes yfinance returns MultiIndex columns, we want simple column names
134         if isinstance(data.columns, pd.MultiIndex):
135             # Get the first level of column names (the actual feature names)
136             data.columns = data.columns.get_level_values(0)
137
138         # Save to cache for future use
139         data.to_csv(cache_csv_path)
140         print(f"✓ Data cached to: {cache_csv_path}")
141
```

Explanation

- Checks for a cached CSV + metadata JSON; if present, loads them with parse_dates=True so the index is datetime.
- If missing, downloads via yfinance (line ~130), flattens MultiIndex columns when needed, then writes both the CSV and a metadata JSON (ticker, range, columns, shape, cache time).
- This meets the “store & load locally” requirement, speeds up experiments, and gives provenance via JSON.

(e) Feature Scaling & Scaler Storage

Code (lines ~324–395+):

Requirement (e): This function will also allow you to have an option to scale your feature columns and store the scalers in a data structure to allow future access to these scalers

```
315     #--------------------------------------------------------------------------
316     # REQUIREMENT (e): Scale feature columns and store scalers
317     # Apply feature scaling with proper handling to prevent data leakage
318     # This addresses ISSUE #2 mentioned in v0.1 comments
319     #--------------------------------------------------------------------------
320
321     scalers = {}
322
323     if scale_features:
324         print(f"✓ Requirement (e): Applying MinMax scaling and storing scalers")
325         print(f"Scaling mode: {scale_mode}")
326         print("IMPORTANT: Fitting scalers on training data only to prevent data leakage")
327
328         # Initialize scaled datasets as copies of raw data
329         train_data = raw_train_data.copy()
330         test_data = raw_test_data.copy()
331
332         if scale_mode == 'all_features':
333             # Task C.5: Scale all features together using one scaler
334             scaler = MinMaxScaler(feature_range=(0, 1))
335
336             # Fit on the entire training dataframe
337             scaler.fit(raw_train_data[normalized_features])
338
339             # Transform both train and test data
340             train_data[normalized_features] = scaler.transform(raw_train_data[normalized_features])
341             test_data[normalized_features] = scaler.transform(raw_test_data[normalized_features])
342
343             # Store the single scaler
344             scalers['all'] = scaler
345             print("✓ Applied a single scaler to all features.")
```

Explanation

- Supports **two modes**:
    - **all_features**: one MinMaxScaler across the full feature matrix.
    - **per_feature**: one scaler per column (each column reshaped to 2D with .values.reshape(-1,1) because scikit-learn scalers expect (n, 1)).
- Every scaler is **fit on training data only** (no leakage), then applied to both train and test.
- Fitted scalers are stored in the scalers dict and also persisted to …_scalers.pkl (a few lines below this block in your file) so later stages can reuse them (e.g., inverse transform during inference).

Conclusion

The enhanced data processing function:

Fixes all major issues in v0.1,

Implements requirements (a)–(e),

Provides a reproducible and extensible pipeline for later tasks.

This lays the groundwork for improved model training in Task C.3 and beyond.