ChatGPT

# Preprocessing Pipeline for Jane Street Data (MOIRAI Transformer)

## Feature Selection (Reducing Multicollinearity and Noise)

The Jane Street market dataset contains 130 anonymized features (mostly continuous, plus one binary feature) [1]. Many features are **highly correlated**, forming clear clusters [2], which can introduce multicollinearity and hinder learning [3]. We first eliminate or consolidate such redundant features and drop those with little variance or predictive signal:

- **Remove constant/near-constant features:** Drop any feature with essentially no variation. These add no information.
- **Correlation-based pruning:** Compute the feature correlation matrix and remove features in each high-correlation pair (e.g. Pearson $|r| > 0.95$) to avoid multicollinearity [2]. Alternatively, use hierarchical clustering on the correlation matrix and pick one representative per cluster (per López de Prado's approach) to preserve diversity. This mitigates the *"low signal-to-noise ratio and multicollinearity"* problem noted in the data [3].
- **Low importance filter:** Optionally, evaluate each feature's predictive power (e.g. correlation with the target or feature importance from a quick model). Features with consistently negligible contribution (no predictive signal) can be dropped to simplify the model. *(Indeed, none of the raw features have a stable strong correlation with the response on their own [2], so we rely on combinations or more complex patterns.)*

We also address **missing values**: certain features are systematically missing at specific times (e.g. early each day) [4]. We can impute these using forward-fill or a rolling window mean. For example, one Kaggle solution filled missing entries with the mean of the previous 100 values (within the same trade side category) [5]. This preserves continuity without leakage across days.

## Stationarity with Fractional Differentiation

Financial features often exhibit trends or non-stationarity (e.g. some feature groups show trends [6] [7]). Following **Marcos López de Prado's** framework, we apply **fractional differentiation** to make each time series feature stationary while retaining maximum memory. Fractional differencing allows us to remove unit-root behavior with a fractional order $d$ instead of an integer order, thus preserving long-term correlations [8].

**Procedure:** For each feature, we determine the minimum differentiation order *d* that makes it stationary:

1. **Augmented Dickey-Fuller test (ADF):** Compute the ADF p-value for the feature. If p > 0.05 (non-stationary), proceed.
2. **Iterate over fractional orders:** Gradually increase $d$ in [0, 1] until the ADF test indicates stationarity (p < 0.05). López de Prado suggests: *"First, compute a cumulative sum of the series... Second, compute the FFD($d$) series for various $d \in [0,1]$. Third, find the minimum $d$ such that*

*the p-value of the ADF statistic falls below 5%. Fourth, use that FFD($d$) series as your feature."* [9] . We use *fractional differenced (FD)* returns at that $d$ as the new feature.

3. **Verify memory retention:** Ensure the chosen $d$ is as low as possible. Typically, $d<0.5$ suffices for financial data [10] . At this $d$, the FD series should be stationary **and** highly correlated with the original (e.g. >99% correlation [8] ), indicating minimal information loss. In contrast, the usual 1st-differencing ($d=1$) often "over-differentiates" and nearly wipes out memory [11] .

After fractional differencing, the features are approximately stationary (e.g. no unit root) and suitable for modeling. We also consider stationarizing the target *resp* if needed (though *resp* is a short-horizon return and may already be stationary). If the target has a trend component, one could similarly apply fractional differencing to the target series (or use de-trended returns).

## Event-Based Sampling (Informative Sample Selection)

Rather than using every timestamp, **event-based sampling** picks only points where meaningful market moves occur, as per López de Prado's recommendations [12] . This can reduce noise by focusing the model on significant events: for example, large price changes or volatility bursts. We implement a **symmetric CUSUM filter** to detect such events:

- **CUSUM filter:** We accumulate price changes (or returns) and trigger an event when the cumulative move exceeds a threshold $\theta$ in either direction [12] . The threshold can be set based on volatility (e.g. $\theta = k \cdot \sigma$ of returns). This avoids multiple triggers on tiny fluctuations and ensures each event reflects a substantial deviation.
- **Result:** The output is a set of event indices (times) when an upward or downward price move of magnitude $\theta$ occurred. These indices mark **informative samples**. For example, we might filter the dataset to include only these event points (and perhaps a window around them for context). This is especially useful if we plan to **meta-label** or focus on sizable moves.

*Note:* In the Jane Street data, each row is an independent trade opportunity rather than a continuous price series [13] . If treating it as time-series, event sampling could mean selecting trades where the *resp* (return) exceeds a threshold (signifying a big profit/loss event). This step is optional for regression – if small moves add mostly noise, we could down-sample them. Otherwise, one may retain all data but consider event labels for auxiliary tasks (like classification of large vs small moves).

## Structural Break Detection (Regime Changes)

Financial data often undergo regime shifts (e.g. volatility regime changes or market regime changes). We incorporate **structural break detection** to identify if and when the data-generating process changes significantly. López de Prado classifies structural break tests into: **CUSUM tests** for mean shifts, and **explosiveness tests** for bubble-like behavior [14] .

For our preprocessing:

- **CUSUM-based break test:** We can apply a **Brown-Durbin-Evans CUSUM test** on a rolling residual series [15] . For example, fit a simple OLS on the target (or a key feature) over time and use `breaks_cusumolsresid` to test for breaks in the mean. A significant test ($p < 0.05$) indicates a structural break [14] . We then segment the data at that point (e.g. treat pre- and post-break as distinct regimes for modeling or perform retraining).

- **Volatility or explosiveness tests:** We could also run tests like the **supremum ADF** for explosive root detection [16], to catch rapid changes (bubbles/crashes). In practice, a large spike in the *resp* distribution or a CUSUM of squared returns can signal a volatility regime change.
- **Handling breaks:** If a break is detected, we may reset certain transformations (e.g. recompute fractional differencing or re-standardize per regime) and avoid training a single model across a regime boundary. We might also include a **regime indicator feature** so the model can account for regime context.

By detecting structural breaks, we ensure our preprocessing (and model training) is **regime-aware**, aligning with López de Prado's guidance that the best opportunities come from spotting structural breaks early [17].

## Meta-Labeling for Regression Targets (Optional)

**Meta-labeling** is a technique (from López de Prado) where a secondary model learns to **predict the correctness of the primary model's signals** [18]. While originally devised for classification (trades) [19], we can adapt it in a regression setting if needed:

- **Primary model and signal:** The primary model could be our regression that forecasts *resp*. To create binary signals, we define an action threshold (e.g. predict *action*=1 if forecasted *resp* > 0). The primary model thus emits a signal (trade or not).
- **Meta-label:** We label each trade signal as 1 if it was profitable (e.g. actual *resp* > 0) or met some profit threshold, otherwise 0. This meta-label indicates whether the primary model's decision was correct.
- **Secondary model:** We train a second classifier on the same feature set (or extended set including the primary prediction) to **predict this meta-label** – essentially, to predict if the primary model's bet will pay off [18]. This model can use additional information (e.g. confidence, volatility regime) to filter out false positives.

In practice, meta-labeling can improve precision by allowing only high-confidence bets. For example, if the primary regressor predicts a small positive return under high volatility (sideways market), the meta-model might learn to flag that as unreliable (label 0) [20]. For our pipeline, meta-labeling is an **optional enhancement** – it adds complexity and is most beneficial if we are making discrete decisions (like trade vs no-trade). If the goal is purely to forecast *resp* accurately, one might omit meta-labeling. Otherwise, following Chapter 3 of *Advances in Financial ML*, we can implement this two-stage approach for more robust signal extraction [18].

## Feature Engineering & Transformation (Jansen's Recommendations)

In addition to the above, we apply **Stefan Jansen's feature engineering best practices** to refine the data (ensuring they don't conflict with de Prado's methods):

- **Outlier handling (Winsorization):** Financial data can have extreme outlier values. We cap (winsorize) feature distributions at a reasonable percentile (e.g. 1% and 99%) [21] to prevent outliers from skewing the model. This retains all data but limits the influence of anomalies. For example, if *feature_5* spikes on one day, we replace values beyond the 99th percentile with that 99% value.
- **Scaling/Normalization:** It's crucial to **standardize features** for a transformer model. We scale each feature to zero-mean, unit-variance (z-score) after winsorization. Jansen's pipeline often

clips outliers then normalizes data [21]. In fact, the MOIRAI transformer itself applies instance normalization to inputs [22], so providing pre-standardized features aligns with that practice and avoids numerical instability [23]. *(If using neural networks, scaling is typically necessary to speed up convergence and prevent any feature from dominating due to scale [23].)*

- **Temporal features:** If not already present, we incorporate any time-related features. The dataset has a *date* (day index) and possibly an intra-day sequence. We ensure data is sorted chronologically by date and time. We can also add explicit time-of-day or day-of-week features if useful (though some anonymous features, like feature_64, appear to encode time-of-day effects [24]). These time features can help the model learn periodic patterns.
- **Feature interactions:** Jansen suggests creating additional features if domain knowledge permits (technical indicators, ratios, etc.). However, with anonymized features, direct domain features are hard. One could generate interaction terms between highly predictive features (as one solution did by multiplying certain feature pairs) [25]. We must be cautious: new features should add unique information. Given de Prado's emphasis on avoiding spurious complexity, we only add interactions that show clear value (e.g. identified via model explanations as in TabNet) [25].
- **PCA/Dimensionality reduction (if needed):** If feature count remains high even after selection, we might use PCA or autoencoders to compress features. Jansen demonstrates using PCA to capture the variance with fewer components. In the Jane Street case, a PCA was tested and found to not significantly improve results [25], possibly because the model (like a transformer or neural net) can handle the raw features. We, therefore, prioritize retaining interpretability and the fractional differentiated features rather than aggressively reducing dimensions, unless computational constraints demand it.

All features and the target *resp* may also be **lagged** if we were building autoregressive features (common in time-series modeling). Since MOIRAI can ingest sequence data, we rely on the transformer to learn temporal patterns from the ordered sequence of trades, rather than manually creating lag features for short-term memory.

## Optimizing for the MOIRAI Transformer

The Salesforce **MOIRAI** model is a large transformer designed for universal time-series forecasting [26] [27]. To leverage it effectively, we prepare the data in a format and scale the model expects:

- **Sequence structuring:** We arrange the data into **chronological sequences**. Each trading day can be treated as one sequence (since days are independent in this data [13]). We group the preprocessed DataFrame by `date` and form sequences of feature vectors in time order. This yields, for example, ~500 sequences (each sequence is one day's trades) for training. This segmentation prevents the model from learning across the artificial overnight gaps.
- **Feature encoding:** MOIRAI uses an "any-variate" attention that flattens all time and feature dimensions into one sequence with learned variate identifiers [28]. We thus **retain all selected features as separate input variates**. No need to one-hot the binary feature (it can be left as -1/+1 or 0/1 normalized). The transformer will get each feature as its own series, and MOIRAI's architecture will handle interactions across features [28].
- **Normalization:** As mentioned, ensure features are normalized (which we do in preprocessing). MOIRAI applies internal instance normalization [22], but providing already standardized inputs (and similarly scaling outputs if needed) helps stable training. We may also normalize *resp* if the magnitude varies widely, or simply ensure *resp* is in a reasonable range (the returns here are relatively small).

- **Padding and masking:** If sequences (days) have variable lengths, we pad them to a fixed length for batch training and use MOIRAI's masking capabilities so that padding tokens are ignored. This is a detail handled during data loader preparation rather than preprocessing per se.
- **High compute considerations:** MOIRAI is a large model; our preprocessing choices (fractional differencing, grouping by day, etc.) are feasible given a high-compute environment. Fractional differentiation and correlation clustering can be computationally heavy but can be parallelized or done offline. The prepared sequences should be fed in batches that fit memory, and one can leverage GPUs for the model.

By following these steps, we ensure the data is **clean, stationary, and appropriately scaled**, focusing the model on meaningful signals (via feature selection and event filters) and respecting the time-series structure (via sequencing and regime awareness). This alignment with both **López de Prado's** advanced techniques and **Jansen's** practical guidelines sets a strong foundation for training the MOIRAI transformer on this financial dataset.

## Example Python Preprocessing Pipeline Code

Below is an example implementation of the above steps in Python. It processes the Jane Street data for regression modeling with a transformer:

```
import pandas as pd, numpy as np
from statsmodels.tsa.stattools import adfuller
import statsmodels.api as sm
import statsmodels.stats.diagnostic as sm_diag

# Assume `df` is a DataFrame with columns: ['date', 'ts_id',
'feature_0', ..., 'feature_N', 'resp']

# 1. Sort by time (date and trade id) to preserve chronological order
df = df.sort_values(['date', 'ts_id']).reset_index(drop=True)

# 2. Feature Selection: drop constant or redundant features
# Drop near-constant features
for col in df.filter(like='feature'):
    if df[col].nunique() <= 1:
        df.drop(columns=col, inplace=True)

# Drop highly correlated features (Pearson |r| > 0.95)
corr = df.filter(like='feature').corr().abs()
# Take upper triangle of correlation matrix
upper = corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))
high_corr_cols = [col for col in upper.columns if any(upper[col] > 0.95)]
df.drop(columns=high_corr_cols, inplace=True)

# 3. Handle missing values: fill forward then backward (or use rolling mean
per day)
df.fillna(method='ffill', inplace=True)
df.fillna(method='bfill', inplace=True)

# 4. Fractional Differentiation for stationarity
```

```
def frac_diff(series: pd.Series, d: float, thresh: float = 1e-5) ->
pd.Series:
    """Fractional differencing (fixed-width window) of a series."""
    # Compute fractional weights for difference
    w = [1.0]
    k = 1
    # Generate weights until they become very small
    while abs(w[-1]) > thresh:
        w.append(-w[-1] * (d - k + 1) / k)
        k += 1
    w = np.array(w[::-1])  # reverse for convolution

    # Apply convolution of weights with series values
    diff_series = np.convolve(series, w, mode='valid')
    # Pad the beginning with NAs for initial period lost
    result = pd.Series(data=np.concatenate([[np.nan]*(len(w)-1),
diff_series]), index=series.index)
    return result

# Find minimum d for each feature to pass ADF test
selected_feats = [f for f in df.columns if f.startswith('feature_')]
min_d = {}  # store chosen d for each feature
for col in selected_feats:
    series = df[col].astype(float)
    # Skip if already stationary
    pval0 = adfuller(series.dropna(), maxlag=1, regression='c', autolag=None)
[1]
    if pval0 < 0.05:
        min_d[col] = 0  # no differencing needed
        continue
    # Try fractional differences from 0 to 1
    for d in np.linspace(0, 1, 11):  # increments of 0.1
        fd_series = frac_diff(series, d)
        fd_series = fd_series.dropna()
        pval = adfuller(fd_series, maxlag=1, regression='c', autolag=None)[1]
        if pval < 0.05:
            min_d[col] = d
            # replace original series with fractionally differenced version
            df[col] = frac_diff(series, d)
            break

# 5. Event-based sampling (using CUSUM filter on resp to identify significant
moves)
resp = df['resp'].astype(float)
# Set threshold = 3 * std of resp (for example)
threshold = 3 * resp.std()
s_pos = s_neg = 0.0
event_index = []
for i in range(1, len(resp)):
    x = resp.iat[i]
    # CUSUM accumulation
```

```
    s_pos = max(0, s_pos + x)
    s_neg = min(0, s_neg + x)
    if s_pos > threshold or s_neg < -threshold:
        event_index.append(i)
        s_pos = s_neg = 0.0
# `event_index` now holds indices of events; we could filter or mark them
df['event'] = 0
df.loc[event_index, 'event'] = 1  # flag events (optional use in model or
meta-labeling)

# 6. Structural Break Test (CUSUM test on residuals of resp)
# Fit OLS on a simple time trend (or constant) to detect a break in mean
X = sm.add_constant(np.arange(len(resp)))
ols_res = sm.OLS(resp, X).fit().resid
# Apply CUSUM test on OLS residuals
cusum_stat, p_value, _ = sm_diag.breaks_cusumolsresid(ols_res,
ddof=ols_res.index.nlevels)
if p_value < 0.05:
    print(f"Structural break detected (p={p_value:.3f}). Consider splitting
data at break.")

# 7. Winsorization (cap outliers at 1st and 99th percentiles for each
feature)
for col in selected_feats + ['resp']:
    lo, hi = df[col].quantile(0.01), df[col].quantile(0.99)
    df[col] = df[col].clip(lower=lo, upper=hi)

# 8. Standardization (z-score scaling) of features
feat_cols = selected_feats  # (after any drops/differencing)
for col in feat_cols:
    mean, std = df[col].mean(), df[col].std()
    df[col] = (df[col] - mean) / std

# Target scaling (optional): If needed, scale resp as well (e.g., standardize
or just clip outliers as done).

# 9. Prepare sequences for Transformer (group by day)
sequences = []  # will hold arrays of shape [time_steps, feature_dim] for
each day
for day, group in df.groupby('date'):
    seq = group.sort_values('ts_id')  # ensure intraday order
    seq_features = seq[feat_cols].values  # feature matrix for this day
    sequences.append(seq_features)
# `sequences` can now be fed to the MOIRAI model (with appropriate padding
and masking if needed).
```

This code performs the preprocessing in a logical order: feature reduction, filling missing data, fractional differencing for stationarity, event flagging, structural break testing, outlier capping, scaling, and finally organizing the data into sequences for the transformer. Each step corresponds to the best practices drawn from López de Prado's techniques and Jansen's guidelines, setting up the Jane Street market data for effective regression modeling with the MOIRAI transformer.

**Sources:** Best practices and methods are based on López de Prado's *Advances in Financial Machine Learning* [8] [9] and Jansen's *Machine Learning for Algorithmic Trading* (feature engineering & scaling) [21], applied to the Jane Street Kaggle dataset [3] [2].

---


[1] [2] [3] [4] [5] [6] [7] [13] [24] [25] GitHub - codefluence/jane_street: kaggle competition
https://github.com/codefluence/jane_street

[8] [10] [11] Advances in Financial Machine Learning
https://agorism.dev/book/finance/ml/Marcos%20Lopez%20de%20Prado%20-%20Advances%20in%20Financial%20Machine%20Learning-Wiley%20%282018%29.pdf

[9] [23] Do we need to fractionally differentiated all features in ML prediction for finance time series? - Quantitative Finance Stack Exchange
https://quant.stackexchange.com/questions/69040/do-we-need-to-fractionally-differentiated-all-features-in-ml-prediction-for-fina

[12] [17] Advances in Financial Machine Learning · Reasonable Deviations
https://reasonabledeviations.com/notes/adv_fin_ml/

[14] [15] [16] [18] [19] [20] Extract market features to decide when to deploy or stop strategies - Quantitative Finance Stack Exchange
https://quant.stackexchange.com/questions/45317/extract-market-features-to-decide-when-to-deploy-or-stop-strategies

[21] 05 Machine Learning Algorithmic Trading - Notes - Junfan Zhu's home
https://junfanz1.github.io/blog/book%20notes%20series/Machine-Learning-Algorithmic-Trading-Notes/

[22] [26] [27] [28] Moirai: A Time Series Foundation Model for Universal Forecasting
https://www.salesforce.com/blog/moirai/