Report: Quadeye Market Data Prediction Challenge

Saurabh Kaushik(antineutrino7642)

---

1. Executive Summary

This submission is designed for robustness in a low signal-to-noise environment. Instead of relying on complex architectures, the focus is on transforming raw market features into stable, relative signals and enforcing realistic constraints at prediction time. The final pipeline uses:

- Cross-sectional rank-based features
- A volatility regime feature
- A CatBoost regressor with ordered boosting
- Market-neutral post-processing
- Global prediction shrinkage

The goal is to predict idiosyncratic (relative) movements between symbols rather than overall market direction.

---

2. Data Loading and Safety

The dataset is loaded dynamically from the Kaggle input directory to ensure portability:

```
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        if filename == 'train.csv': train_path = os.path.join(dirname, filename)
        if filename == 'test.csv':  test_path  = os.path.join(dirname, filename)

train = pd.read_csv(train_path)
test  = pd.read_csv(test_path)
```

Immediately after loading, a strict submission template is saved:

```
submission_template = test[['Id']].copy()
```

This prevents ID misalignment or duplication errors later during submission.

---

3. Feature Engineering

All feature logic is implemented inside the `process_data` function.

3.1 Sorting for Time Consistency

```
df = df.sort_values(['date_id', 'time_id', 'symbol_id']).reset_index(drop=True)
```

Sorting ensures that all grouping, ranking, and validation logic respects chronological order and that no future information is implicitly leaked into earlier samples.

3.2 Cross-Sectional Ranking

Each numeric feature (`f*`) is converted into a percentile rank at each `(date_id, time_id)` snapshot:

```
df[f'{col}_rank'] = df.groupby(['date_id', 'time_id'])[col].transform(
    lambda x: x.rank(pct=True, method='first')
)
```

This removes absolute scale and preserves only the relative ordering between symbols at the same timestamp. Using `method='first'` ensures deterministic ranking when duplicate values exist.

3.3 Rank Centering

The rank values are centered around zero:

```
df[f'{col}_rank'] = df[f'{col}_rank'] - 0.5
```

Centering produces symmetric features in the range approximately `[-0.5, +0.5]`, which improves tree split balance and reduces bias toward one-sided thresholds.

3.4 Volatility Regime Feature

A global volatility proxy is introduced:

```
df['market_vol'] = df.groupby(['date_id', 'time_id'])
[f_cols[0]].transform('std')
```

This provides the model with regime information indicating whether the market is currently calm or turbulent. The first base feature is used as a stable proxy to avoid excessive dimensionality.

3.5 Feature Selection and Casting

Only rank features and the volatility feature are retained:

```
keep_cols = [c for c in df.columns if '_rank' in c] + ['market_vol']
```

All selected features are cast to `float32` to reduce memory usage and improve training throughput:

```
df[c] = df[c].astype(np.float32)
```

---

4. Target Stabilization

The training target is clipped to reduce the influence of extreme outliers:

```
y_min, y_max = train['y'].quantile(0.005), train['y'].quantile(0.995)
train['y'] = train['y'].clip(y_min, y_max)
```

This suppresses rare tail events that disproportionately inflate RMSE and destabilize gradient updates.

---

5. Validation Methodology

A strict chronological split is used:

```
dates = train['date_id'].unique()
split_date = dates[int(len(dates) * 0.9)]

X_train = train[train['date_id'] <= split_date]
X_val   = train[train['date_id'] >  split_date]
```

- First 90% of dates → training
- Last 10% of dates → validation

This avoids look-ahead bias and mirrors real production conditions where only past data is available.

---

6. Model Choice and Training

6.1 CatBoost Configuration

```
model = CatBoostRegressor(
    iterations=2000,
    learning_rate=0.03,
    depth=6,
    loss_function='RMSE',
    l2_leaf_reg=5.0,
    random_seed=42,
    boosting_type='Ordered'
)
```

Key design choices:

- Shallow depth (`depth=6`) to prevent overfitting
- L2 regularization to smooth predictions
- Ordered boosting to reduce target leakage

6.2 Training Procedure

```
model.fit(
    X_train, y_train,
    eval_set=(X_val, y_val),
    early_stopping_rounds=50,
    use_best_model=True
)
```

Early stopping prevents over-training and automatically selects the optimal number of boosting iterations based on validation RMSE.

---

7. Prediction and Post-Processing

7.1 Raw Predictions

```
preds = model.predict(test[features])
```

7.2 Market-Neutral Centering

```
test['raw_pred'] = preds
market_means = test.groupby(['date_id', 'time_id'])
['raw_pred'].transform('mean')
preds = preds - market_means
```

This ensures that the sum of predictions across all symbols at each time step is zero, removing systematic market direction.

7.3 Global Shrinkage

```
preds = preds * CFG['shrinkage_factor']
```

This reduces prediction magnitude to improve RMSE stability and suppress overconfident outputs.

7.4 Final Clipping

```
preds = np.clip(preds, y_min, y_max)
```

This keeps predictions within the same bounds as the stabilized training target, improving tail-risk control.

---

8. Submission Generation

```
submission_template['y'] = preds
submission_template.to_csv('submission.csv', index=False)
```

Using a pre-saved ID template ensures correct row ordering and prevents duplicate ID issues.

---

9. Summary

This solution focuses on:

- Converting raw features into relative signals using cross-sectional ranks
- Adding a volatility regime indicator
- Training a regularized CatBoost model with ordered boosting
- Enforcing market-neutral predictions

- Applying shrinkage and clipping for stability

The overall design prioritizes generalization and robustness over model complexity.

---

10. Detailed Walkthrough of the Full Pipeline

This section provides a step-by-step explanation of how data flows through the system from raw CSV files to the final submission file.

10.1 Environment Setup and Configuration

A configuration dictionary is defined to centralize all modeling hyperparameters:

```
CFG = {
    'iterations': 2000,
    'learning_rate': 0.03,
    'depth': 6,
    'loss_function': 'RMSE',
    'l2_leaf_reg': 5.0,
    'shrinkage_factor': 0.15,
    'random_seed': 42
}
```

This ensures reproducibility and allows controlled experimentation without modifying core logic.

10.2 Dynamic Dataset Discovery

```
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        if filename == 'train.csv': train_path = os.path.join(dirname, filename)
        if filename == 'test.csv':  test_path  = os.path.join(dirname, filename)
```

Instead of hardcoding file paths, the pipeline automatically locates the dataset inside the Kaggle container. This makes the code portable across environments and prevents file-not-found failures during execution.

10.3 Submission Template Preservation

```
submission_template = test[['Id']].copy()
```

The ID column from the test set is preserved immediately. This design guarantees:

- Row order consistency
- No accidental shuffling
- No duplication of IDs
- Submission format compliance

10.4 Feature Transformation Function (`process_data`)

All feature logic is encapsulated inside a single function to ensure consistent transformations across train and test datasets.

10.4.1 Sorting

```
df = df.sort_values(['date_id', 'time_id', 'symbol_id']).reset_index(drop=True)
```

Sorting enforces a strict temporal and cross-sectional order. This is important for:

- Reproducible ranking
- Stable groupby operations
- Correct time-based splitting

10.4.2 Feature Column Detection

```
f_cols = [c for c in df.columns if c.startswith('f')]
```

This dynamically identifies all numeric signal features without relying on hardcoded names.

10.4.3 Cross-Sectional Ranking

```
df[f'{col}_rank'] = df.groupby(['date_id', 'time_id'])[col].transform(
    lambda x: x.rank(pct=True, method='first')
)
```

At each timestamp, all symbols are ranked against each other for each feature. This removes global scale effects and forces the model to focus on relative dispersion.

10.4.4 Rank Centering

```
df[f'{col}_rank'] = df[f'{col}_rank'] - 0.5
```

Centering shifts all features into a symmetric range around zero, which improves decision boundary balance in tree models.

10.4.5 Volatility Regime Feature

```
df['market_vol'] = df.groupby(['date_id', 'time_id'])
[f_cols[0]].transform('std')
```

This introduces a regime-awareness signal that allows the model to behave differently during calm and turbulent market periods.

10.4.6 Feature Selection

```
keep_cols = [c for c in df.columns if '_rank' in c] + ['market_vol']
```

This design intentionally excludes raw feature values to reduce noise and improve stationarity.

10.4.7 Memory Optimization

```
for c in keep_cols:
    df[c] = df[c].astype(np.float32)
```

Casting to `float32` halves memory usage and improves cache locality during training.

10.5 Target Stabilization

```
y_min, y_max = train['y'].quantile(0.005), train['y'].quantile(0.995)
train['y'] = train['y'].clip(y_min, y_max)
```

This removes extreme tail observations that would otherwise dominate RMSE and distort tree splits.

10.6 Time-Based Train–Validation Split

```
dates = train['date_id'].unique()
split_date = dates[int(len(dates) * 0.9)]
```

Only historical data is used for training, and the final 10% of dates is held out for validation. This prevents future information leakage.

10.7 CatBoost Training Pipeline

10.7.1 Model Initialization

```
model = CatBoostRegressor(
    iterations=CFG['iterations'],
    learning_rate=CFG['learning_rate'],
    depth=CFG['depth'],
    loss_function=CFG['loss_function'],
    l2_leaf_reg=CFG['l2_leaf_reg'],
    random_seed=CFG['random_seed'],
    boosting_type='Ordered'
)
```

Each hyperparameter is chosen to balance bias and variance under noisy conditions.

10.7.2 Model Fitting

```
model.fit(
    X_train, y_train,
    eval_set=(X_val, y_val),
    e
```