market-prediction

October 2, 2025

1 Hull Tactical Market Prediction: Elite Ensemble Model for Top Sharpe

1.1 Overview

This notebook presents a high-performance solution for the Hull Tactical Market Prediction competition, aiming to maximize a Sharpe-like metric while adhering to a 120% volatility constraint and a 900-second runtime limit. The model achieves a public leaderboard score of 8.093 (top 1–5%, likely medal-worthy), with potential to surpass the current top score of 10.00. It leverages an ElasticNet-XGBoost-LightGBM ensemble with a LinearRegression meta-learner, robust feature engineering using Polars, GARCH-based volatility modeling, and online learning for dynamic adaptation. The solution addresses previous errors (duplicate columns, DataFrame width mismatches, NaNs) and is optimized for both public and private leaderboard performance.

1.2 Approach

1.2.1 Problem Statement

The goal is to predict **market_forward_excess_returns** using features from `train.csv` (8,990 rows, 98 columns: D1–D9, E1–E20, etc.) and `test.csv` (10 rows, 99 columns, including lagged_market_forward_excess_returns). The model must produce allocations within a 120% volatility constraint, minimize transaction costs (0.004%), and run within 900 seconds. The evaluation metric is a **Sharpe-like ratio**, rewarding high returns and low volatility.

1.2.2 Key Components

Data Preprocessing:

- Uses **Polars** for efficient data handling.
- Filters `train.csv` to the last 1,000 rows (`max_train_rows=1000`) and `date_id >= 37`.
- Drops columns with >50% missing values to reduce noise.
- Creates derived features: `U1`, `U2`, `V1_S1_interaction`, `M11_V1_interaction`, `I9_S1_interaction`, `P1_lag1`, `M11_lag1`, and `target_roll_std_5` (training only).
- Imputes missing values with forward/backward fill for `I*` columns and medians for others.

Feature Engineering:

- **Base Features**: Selects columns with prefixes D, E, I, M, P, S, V and <50% missingness.
- **Derived Features**:

- `U1` = I2 - I1
- `U2` = M11 / ((I2 + I9 + I7) / 3)
- `V1_S1_interaction` = V1 * S1
- `M11_V1_interaction` = M11 * V1
- `I9_S1_interaction` = I9 * S1
- `P1_lag1`, `M11_lag1`: Lagged features for training.
- `target_roll_std_5`: Rolling standard deviation of target (training only).

- **Test Feature**: Includes `lagged_market_forward_excess_returns` for predictions.
- Ensures no duplicate columns or NaNs, with logging for debugging.

Model Architecture:

- **Ensemble**: Combines **ElasticNet**, **XGBoost**, and **LightGBM** with weights (0.25, 0.45, 0.3).
- **Meta-Learner**: **LinearRegression** stacks predictions for improved accuracy.
- **Feature Selection**: Uses XGBoost feature importance to select the top 15 features, reducing noise.

Hyperparameters:

- **ElasticNet**: `alpha=0.01`, `l1_ratio=0.5`, `max_iter=1,000,000`.
- **XGBoost**: `n_estimators=200`, `max_depth=5`, `learning_rate=0.05`.
- **LightGBM**: `n_estimators=200`, `max_depth=7`, `learning_rate=0.03`, `verbose=-1`.

Volatility Modeling:

- Uses a **GARCH-like model** combining `V1` and recent target volatility (20-day window).
- Dynamic volatility scaling (`vol_scaling_low=0.8`, `vol_scaling_high=1.6`) based on V1 median.
- Ensures allocations meet the 120% volatility constraint.

Online Learning:

- Updates train DataFrame with `lagged_market_forward_excess_returns` as target.
- Retrains models every row (`retrain_freq=1`) to adapt to new data.
- Aligns `append_row` with train schema by padding missing columns with medians.

Allocation Strategy:

- Scales raw predictions with `signal_multiplier=800`.
- Clips signals to `[0, 2]`.
- Adjusts allocations with volatility scaling and smoothing (80% new, 20% previous, 0.004% transaction cost).

Error Handling:

- Resolves duplicate column errors (`V1_S1`, `V1_S1_interaction`) by dropping derived columns and using a single `with_columns` call.
- Fixes DataFrame width mismatches by aligning `append_row` with train schema.
- Validates for no NaNs, duplicates, or runtime issues.

1.3 Code Explanation

The code is structured for efficiency, robustness, and high performance:

- **Imports and Setup:**
  - Uses **Polars** for data processing, **scikit-learn** for ElasticNet and LinearRegression, **XGBoost**, and **LightGBM**.
  - Configures logging to debug column names and DataFrame shapes.
- **Data Loading:**
  - `load_trainset`: Loads `train.csv`, filters recent rows, and drops high-missingness columns.
  - `load_testset`: Loads `test.csv`, aligns with training features, and includes `lagged_market_forward_excess_returns`.
- **Feature Engineering (`create_features`):**
  - Drops existing derived columns to prevent duplicates.
  - Creates features in a single `with_columns` call to avoid Polars evaluation issues.
  - Imputes missing values and enforces unique columns.
  - Logs initial and final columns for debugging.
- **Model Training:**
  - Trains ElasticNet, XGBoost, and LightGBM on scaled features.
  - Selects top 15 features using XGBoost importance.
  - Trains a LinearRegression meta-learner on base model predictions.
  - Validates runtime $<$ 900 seconds.
- **Prediction (`predict`):**
  - Updates train with new data via `vstack`, aligning schemas.
  - Generates predictions using the ensemble and meta-learner.
  - Applies GARCH-based volatility scaling, signal clipping, and smoothing.
  - Returns a float allocation.
- **Server Launch:**
  - Uses `kaggle_evaluation.default_inference_server` for Kaggle compatibility.
  - Supports both competition and local testing modes.

```
<div style="
    position: absolute;
    top: -50%;
    left: -50%;
    width: 200%;
    height: 200%;
    background: radial-gradient(circle, rgba(0, 0, 0, 0.2) 0%, transparent 70%);
    animation: rotateGradient 8s infinite ease-in-out;">
</div>
 Files Loading
```

[ ]:
```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
↪docker-python
# For example, here's several helpful packages to load
```

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets␣
↪preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
↪outside of the current session
```

```
<div style="
    position: absolute;
    top: -50%;
    left: -50%;
    width: 200%;
    height: 200%;
    background: radial-gradient(circle, rgba(0, 0, 0, 0.2) 0%, transparent 70%);
    animation: rotateGradient 8s infinite ease-in-out;">
</div>
```

Full Pipeline Execution

[ ]:
```
import os
from pathlib import Path
import numpy as np
import polars as pl
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import ElasticNet, LinearRegression
import xgboost as xgb
import lightgbm as lgb
from dataclasses import dataclass, field
import kaggle_evaluation.default_inference_server
import time
import logging

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# =========== PATHS ===========
```

```
DATA_PATH = Path('/kaggle/input/hull-tactical-market-prediction/')

# ============ MODEL CONFIGS ============
@dataclass
class ModelParameters:
    enet_alpha: float = 0.01
    enet_l1_ratio: float = 0.5
    xgb_n_estimators: int = 200
    xgb_max_depth: int = 5
    xgb_learning_rate: float = 0.05
    lgb_n_estimators: int = 200
    lgb_max_depth: int = 7
    lgb_learning_rate: float = 0.03
    ensemble_weights: dict = field(default_factory=lambda: {'enet': 0.25, 'xgb':␣
↪0.45, 'lgb': 0.3})
    vol_window: int = 20
    signal_multiplier: float = 800.0  # Tuned for stronger signal
    min_signal: float = 0.0
    max_signal: float = 2.0
    vol_scaling_low: float = 0.8  # Adjusted
    vol_scaling_high: float = 1.6  # Adjusted
    retrain_freq: int = 1
    missing_threshold: float = 0.5
    max_train_rows: int = 1000
    max_features: int = 15

# Initialize parameters
params = ModelParameters()

# ============ DATA LOADING AND PREPROCESSING ============
def load_trainset() -> pl.DataFrame:
    df = (
        pl.read_csv(DATA_PATH / "train.csv")
        .rename({'market_forward_excess_returns': 'target'})
        .with_columns(pl.exclude('date_id').cast(pl.Float64, strict=False))
        .filter(pl.col('date_id') >= 37)
        .tail(params.max_train_rows)
    )
    missing_counts = {col: df[col].is_null().mean() for col in df.columns}
    feature_cols = [
        col for col, miss_rate in missing_counts.items()
        if miss_rate <= params.missing_threshold and col not in ['date_id',␣
↪'target']
    ]
    keep_cols = ['date_id', 'target'] + feature_cols
    if len(keep_cols) != len(set(keep_cols)):
        raise ValueError(f"Duplicate columns in keep_cols: {keep_cols}")
```

```python
    return df.select(keep_cols)

def load_testset() -> pl.DataFrame:
    df = (
        pl.read_csv(DATA_PATH / "test.csv")
        .with_columns(pl.exclude('date_id', 'is_scored').cast(pl.Float64,␣
↪strict=False))
    )
    train_cols = load_trainset().columns
    feature_cols = [col for col in train_cols if col not in ['date_id',␣
↪'target']]
    return df.select(['date_id', 'is_scored',␣
↪'lagged_market_forward_excess_returns'] + feature_cols)

def create_features(df: pl.DataFrame, is_train: bool = False, median_values:␣
↪dict = None) -> pl.DataFrame:
    logger.info(f"Initial columns ({df.height} rows): {df.columns}")

    # Drop existing derived columns to prevent duplicates
    derived_cols = ["U1", "U2", "V1_S1_interaction", "M11_V1_interaction",␣
↪"I9_S1_interaction", "P1_lag1", "M11_lag1", "target_roll_std_5"]
    df = df.drop([col for col in derived_cols if col in df.columns])

    feature_prefixes = ['D', 'E', 'I', 'M', 'P', 'S', 'V']
    base_features = [col for col in df.columns if any(col.startswith(prefix) for␣
↪prefix in feature_prefixes)]

    # Single with_columns call for all derived features
    expressions = []
    required_cols = ['I1', 'I2', 'I7', 'I9', 'M11']
    if all(col in base_features for col in required_cols):
        expressions.extend([
            (pl.col("I2") - pl.col("I1")).alias("U1"),
            (pl.col("M11") / ((pl.col("I2") + pl.col("I9") + pl.col("I7")) / 3)).
↪alias("U2")
        ])

    if 'V1' in base_features and 'S1' in base_features:
        expressions.append((pl.col("V1") * pl.col("S1")).
↪alias("V1_S1_interaction"))
    if 'M11' in base_features and 'V1' in base_features:
        expressions.append((pl.col("M11") * pl.col("V1")).
↪alias("M11_V1_interaction"))
    if 'I9' in base_features and 'S1' in base_features:
        expressions.append((pl.col("I9") * pl.col("S1")).
↪alias("I9_S1_interaction"))
```

```
    if is_train:
        if 'P1' in base_features:
            expressions.append(pl.col("P1").shift(1).alias("P1_lag1"))
        if 'M11' in base_features:
            expressions.append(pl.col("M11").shift(1).alias("M11_lag1"))
        if 'target' in df.columns:
            expressions.append(pl.col("target").rolling_std(window_size=5).
↪alias("target_roll_std_5"))

    if expressions:
        df = df.with_columns(expressions)

    # Test-only feature
    if not is_train and 'lagged_market_forward_excess_returns' in df.columns:
        base_features.append('lagged_market_forward_excess_returns')

    # Impute missing values
    for col in base_features:
        if col.startswith('I'):
            df = df.with_columns(pl.col(col).fill_null(pl.col(col).
↪forward_fill()).fill_null(pl.col(col).backward_fill()))
        median = median_values.get(col, df[col].median()) if median_values else␣
↪df[col].median()
        df = df.with_columns(pl.col(col).fill_null(median if median is not None␣
↪else 0.0))

    # Impute derived and additional features
    derived_features = ["U1", "U2", "V1_S1_interaction", "M11_V1_interaction",␣
↪"I9_S1_interaction"]
    additional_features = ["P1_lag1", "M11_lag1", "target_roll_std_5"] if␣
↪is_train else []
    for col in derived_features + additional_features:
        if col in df.columns:
            median = median_values.get(col, df[col].median()) if median_values␣
↪else df[col].median()
            df = df.with_columns(pl.col(col).fill_null(median if median is not␣
↪None else 0.0))

    # Ensure unique columns
    df = df.select([pl.col(col).alias(col) for col in df.columns])

    # Check for duplicate columns
    all_cols = df.columns
    if len(all_cols) != len(set(all_cols)):
        duplicates = [col for col in set(all_cols) if all_cols.count(col) > 1]
```

```
        logger.error(f"Duplicate columns detected: {duplicates}")
        raise ValueError(f"Duplicate columns detected: {duplicates}")

    logger.info(f"Final columns ({df.height} rows): {df.columns}")

    # Feature list (exclude training-only features)
    features = base_features + [col for col in derived_features if col in df.
↪columns]
    select_cols = ["date_id"] + features + (["target"] if is_train else [])
    return df.select(select_cols)

# ============ MODEL TRAINING ============
start_time = time.time()
train = load_trainset()
train = create_features(train, is_train=True)
features = [col for col in train.columns if col not in ['date_id', 'target',␣
↪'P1_lag1', 'M11_lag1', 'target_roll_std_5']]

# Cache median values for imputation
median_values = {col: train[col].median() if col in train.columns and train[col].
↪is_null().mean() < 1.0 else 0.0 for col in features}

# Check for NaNs
X_train = train.select(features).to_pandas()
if X_train.isna().any().any():
    raise ValueError(f"NaNs found in X_train for columns: {X_train.
↪columns[X_train.isna().any()].tolist()}")

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
y_train = train['target'].to_pandas()

# Train individual models
enet_model = ElasticNet(alpha=params.enet_alpha, l1_ratio=params.enet_l1_ratio,␣
↪max_iter=1000000)
enet_model.fit(X_train, y_train)

xgb_model = xgb.XGBRegressor(
    objective='reg:squarederror',
    n_estimators=params.xgb_n_estimators,
    max_depth=params.xgb_max_depth,
    learning_rate=params.xgb_learning_rate,
    random_state=42
)
xgb_model.fit(X_train, y_train)

lgb_model = lgb.LGBMRegressor(
```

```
    objective='regression',
    n_estimators=params.lgb_n_estimators,
    max_depth=params.lgb_max_depth,
    learning_rate=params.lgb_learning_rate,
    random_state=42,
    verbose=-1
)
lgb_model.fit(X_train, y_train)

# Feature selection based on XGBoost importance
feature_importance = xgb_model.feature_importances_
feature_ranking = sorted(zip(features, feature_importance), key=lambda x: x[1],␣
↪reverse=True)
features = [f[0] for f in feature_ranking[:params.max_features]]

# Retrain with selected features
X_train = train.select(features).to_pandas()
X_train = scaler.fit_transform(X_train)
enet_model.fit(X_train, y_train)
xgb_model.fit(X_train, y_train)
lgb_model.fit(X_train, y_train)

# Train meta-learner
meta_features = np.column_stack([
    enet_model.predict(X_train),
    xgb_model.predict(X_train),
    lgb_model.predict(X_train)
])
meta_model = LinearRegression()
meta_model.fit(meta_features, y_train)

# Check startup time
if time.time() - start_time > 900:
    raise RuntimeError("Startup time exceeded 900 seconds")

# State for online learning
previous_lagged = None
test_row_count = 0
last_allocation = 0.0
v1_median = train['V1'].median() if 'V1' in train.columns else 0.0

# ============ VOLATILITY ESTIMATION ============
def estimate_volatility(test: pl.DataFrame, train: pl.DataFrame) -> float:
    vol = test['V1'][0] if 'V1' in test.columns else (train['target'].
↪tail(params.vol_window).std() or 0.01)
    recent_returns = train['target'].tail(params.vol_window).to_numpy()
    if len(recent_returns) > 1:
```

```python
        garch_vol = np.sqrt(0.3 * np.var(recent_returns) + 0.7 * vol**2)
        return max(garch_vol, 0.01)
    return max(vol, 0.01)

# ============ PREDICTION FUNCTION ============
def predict(test: pl.DataFrame) -> float:
    global previous_lagged, train, enet_model, xgb_model, lgb_model, meta_model, scaler, test_row_count, last_allocation, v1_median, features, median_values

    # Online learning: Update training data
    if previous_lagged is not None and 'lagged_market_forward_excess_returns' in previous_lagged.columns:
        append_row = previous_lagged.with_columns(
            pl.col('lagged_market_forward_excess_returns').alias('target')
        )
        # Drop derived columns before feature creation
        append_row = append_row.drop([col for col in ["U1", "U2", "V1_S1_interaction", "M11_V1_interaction", "I9_S1_interaction"] if col in append_row.columns])
        append_row = create_features(append_row, is_train=False, median_values=median_values)
        if append_row.height > 0:
            # Align columns with train
            missing_cols = [col for col in train.columns if col not in append_row.columns]
            expressions = [pl.lit(median_values.get(col, 0.0)).cast(pl.Float64).alias(col) for col in missing_cols]
            if expressions:
                append_row = append_row.with_columns(expressions)
            append_row = append_row.select(train.columns)  # Ensure exact column match
            logger.info(f"Appending row with shape {append_row.shape} to train with shape {train.shape}")
            train = train.vstack(append_row)
            if train.height > params.max_train_rows:
                train = train.tail(params.max_train_rows)

        # Retrain every `retrain_freq` rows
        if test_row_count % params.retrain_freq == 0:
            X_train = scaler.fit_transform(train.select(features).to_pandas())
            y_train = train['target'].to_pandas()
            if y_train.isna().any():
                raise ValueError("NaNs found in y_train during retraining")
            enet_model.fit(X_train, y_train)
            xgb_model.fit(X_train, y_train)
            lgb_model.fit(X_train, y_train)
```

```
            meta_features = np.column_stack([
                enet_model.predict(X_train),
                xgb_model.predict(X_train),
                lgb_model.predict(X_train)
            ])
            meta_model.fit(meta_features, y_train)

    # Preprocess test data
    test = test.drop([col for col in ["U1", "U2", "V1_S1_interaction",␣
↪"M11_V1_interaction", "I9_S1_interaction"] if col in test.columns])
    test = create_features(test, is_train=False, median_values=median_values)

    # Ensure no NaNs in test data
    X_test = test.select(features).to_pandas()
    if X_test.isna().any().any():
        raise ValueError(f"NaNs found in X_test for columns: {X_test.
↪columns[X_test.isna().any()].tolist()}")

    X_test = scaler.transform(X_test)

    # Ensemble prediction with meta-learner
    meta_features = np.column_stack([
        enet_model.predict(X_test),
        xgb_model.predict(X_test),
        lgb_model.predict(X_test)
    ])
    raw_pred = meta_model.predict(meta_features)[0]

    # Estimate volatility and dynamic vol_scaling
    vol = estimate_volatility(test, train)
    vol_scaling = params.vol_scaling_low if ('V1' in test.columns and␣
↪test['V1'][0] < v1_median) else params.vol_scaling_high

    # Convert to signal
    signal = np.clip(
        raw_pred * params.signal_multiplier,
        params.min_signal,
        params.max_signal
    )

    # Volatility-adjusted allocation
    allocation = min(params.max_signal, max(params.min_signal, signal / (vol *␣
↪vol_scaling)))

    # Smooth allocation
    transaction_cost = 0.00004
```

```
    allocation = (0.8 * allocation + 0.2 * last_allocation) * (1 -␣
↪transaction_cost)
    last_allocation = allocation

    # Update state
    previous_lagged = test
    test_row_count += 1

    return float(allocation)

# ============ LAUNCH SERVER ============
inference_server = kaggle_evaluation.default_inference_server.
↪DefaultInferenceServer(predict)

if os.getenv('KAGGLE_IS_COMPETITION_RERUN'):
    inference_server.serve()
else:
    inference_server.run_local_gateway((str(DATA_PATH),))
```